OWASP
The Open Web Application Security Project
http://www.owasp.org

Securing Enterprise Web Applications at the Source: An Application Security Perspective

Author: Eugene Lebanidze

eugene.lebanidze@gmail.com

EXECUTIVE SUMMARY

Purpose:

This paper considers a variety of application level threats facing enterprise web applications and how those can be mitigated in order to promote security. Evidence shows that perhaps as many as sixty percent of attacks on enterprise web applications are facilitated by exploitable vulnerabilities present in the source code. In this paper we take the approach of examining the various threats specific to the application layer along with their corresponding compensating controls. Threats specific to each of the tiers of the n-tiered enterprise web application are discussed with focus on threat modeling. Compensating controls are addressed at the architecture, design, implementation and deployment levels. The discussion focuses around the core security services, namely confidentiality, integrity, authentication, authorization, availability, non-repudiation and accountability. The paper examines how these core security services are supported in the J2EE and .NET frameworks. Recommendations and standards from various organizations are considered in this paper (e.g. OWASP, NIST, etc.)

We also recognize that development and deployment of a secure enterprise web-based application is dependent upon the organization's enterprise security architecture (ESA) framework. Therefore, we map some elements of our discussion for security architecture and technology issues to the five rings of the enterprise security architecture model, particularly in the context of the Zachman Framework enterprise architecture (EA). While most of the web application security threats and mitigations discussed fall in rings four and five, we tie the local risk to the enterprise risk in ring one of the ESA. In this discussion we draw from the NIST special publications SP800-37 ("Guide for the Security C&A of Federal Information Systems) [20], SP800-64 ("Security Considerations in the Information System Development Life Cycle") [16], SP800-53 ("Recommended Security Controls for Federal Information Systems") [15], and SP800-26 ("Security Self-Assessment Guide for Information Technology Systems") [14].

Security Criteria:

The security criteria for evaluation of web application security revolve around the core security services collectively known as CI4A (Confidentiality, Integrity, Authentication, Authorization, Availability, Accountability). Non-repudiation is another security service commonly tied with accountability. In the context of enterprise web applications, confidentiality is concerned with the privacy of information that passes through or is stored inside the web application. Integrity ensures that the data used is free from deliberate or accidental modification. Authentication addresses verification of identities. Authentication can also be thought of in the context of source integrity. Authorization focuses on access rights to various application subsystems, functionality, and data. Availability, an often ignored aspect of security, is nevertheless an important metric for the security posture of the web application. Many attacks that compromise application availability exploit coding mistakes introduced at the application source level that could have been easily avoided. Non-repudiation addresses the need to prove that a certain action has been taken by an identity without plausible deniability. Accountability, tied with non-repudiation, allows holding people accountable for their actions.

There are a variety of known vulnerabilities that can be exploited in web applications to compromise the core security services outlined above (CI4A). While a provision of a complete taxonomy of attacks would be impractical, if not impossible, especially in light of an ever increasing ingenuity exhibited by hackers (not to mention a growing arsenal of hacking tools), it makes very good sense to consider the classes of vulnerabilities that a web application can contain that could be exploited by attackers to compromise CI4A. Having those classes of vulnerabilities in mind helps structure an assessment of application security and more importantly helps focus security analysis across all stages of application development, including requirement specification, architecture, design, implementation, testing, etc. One organization took the initiative to provide the top ten classes of vulnerabilities that plague modern enterprise web applications that have the greatest impact on CI4A. The Open Web Application Security Project (OWASP) has recommended the top ten list of application level vulnerabilities in web applications that make for a very useful security criteria compliance metrics. The top ten classes of vulnerabilities outlined by OWASP are: unvalidated input, broken access control, broken authentication and session management, cross site scripting (XSS) flaws, buffer overflows, injection flaws, improper error handling, insecure storage, denial of service and insecure configuration management [24]. Each of these is discussed in great detail subsequently in this paper.

We also consider the various security criteria for certification and accreditation (C&A), as outlined in NIST SP800-37 ("Guide for the Security C&A of Federal Information Systems") as they apply to development of enterprise web-based application. Certification analyzes risks local to the web application, such as potential vulnerabilities present in the source code and the corresponding compensating controls (Rings 4-5 of the ESA). On the other hand, accreditation focuses on enterprise level risks that may result from vulnerabilities found in a web-based enterprise application. Additional security criteria can be gathered from SP800-53 ("Recommended Security Controls for Federal Information Systems") and SP800-26 ("Security Self-Assessment Guide for Information Technology Systems"). Secure enterprise web-based applications and their supporting technologies should follow the recommended security controls in SP800-53.

Scope and Limitations:

One of the most prevalent concepts in security is the defense in depth principle. This principle dictates that there should be multiple levels of protection to protect critical assets. The focus of this paper is on security of enterprise web applications from the perspective of application security. There are certainly other perspectives from which web application security could and should be evaluated. For instance, network security and host security also play a critical role. However, this paper focuses on application security because of the special significance that this security layer has come to play. Many attacks on corporate applications come form inside the network, thus rendering such protection mechanisms as firewalls useless. Additionally, intrusion detection techniques will not work when the problem is poor input validation in the application. The point is that network and host security can often help fight the symptoms of the problem where the source of the problem is in the application source. This paper also discusses the architecture, design and deployment considerations that have security implications.

It is often suggested that security is a process and not a product. We certainly believe that to be true since most secure software is the result of security aware software development processes where security is built in and thus software is developed with security in mind. In other words, throughout the various stages of the Software Development Lifecycle (SDLC), software project members are responsible for performing security related activities and producing/consuming security artifacts in order to provide a more structured approach to application security. Automation of source code analysis for security vulnerabilities should also be applied as part of any secure development process. We are omitting the discussion of security process in this paper due to time constraints, but it is implied that developers who apply secure coding techniques that we describe in this paper when developing secure enterprise web-based applications should operate in the context of a secure software development process. To learn more about integrating security in the SDLC please refer to the Comprehensive Lightweight Application Security Process (CLASP) authored by John Viega and available free from Secure Software (www.securesoftware.com) [11]. A secure development process should also incorporate C&A (NIST SP-800-37), various process level security controls (NIST SP-800-53) and contain various metrics for security self-assessment described in NIST SP-800-26. Additionally, a secure process will be part of the enterprise security architecture where both local risk (Rings 4-5) and enterprise risks (Ring 1) are mitigated.

Summary of Conclusions:

We conclude that J2EE and .NET platforms both have fairly comparable security models that provide extensive support for all of the core security services. Throughout this paper, as we discuss many of the specific compensating controls, we very often come back and tie our discussion to the key security principles. We cannot emphasize enough how critical it is for all members of the software project team to be familiar with and follow these principles because only solutions that comply with these principles will have any chance of providing true protection for the core security services. We introduced threat modeling as a critical ingredient for development of secure enterprise web-based applications. It is impossible to defend against the threats without first understanding what the threats are. To this end we recommended identifying all resources that need protection (assets), documenting security assumptions, identifying attack surface as well as input and output attack vectors, combining these vectors into attack trees (scenarios) and ensuring that proper mitigations are put in place at the appropriate places. It is also important to remember that development and deployment of secure web-based applications is contingent upon a secure development process. While we did not provide a detailed discussion of secure development process in this paper due to time limitation, the baseline process that we mentioned is derived from CLASP. Additionally, C&A should be made part of this process, in order to manage both local and enterprise risk, as outlined in NIST SP800-37. In addition to a secure development process that is aware of the various application security issues and makes it an explicit part of the process, an ESA is required to support development and deployment of secure enterprise web-based application. We discuss the ESA rings 1-5 and show that development and deployment of secure enterprise web-based application depends on the enterprise security architecture framework.

Table of Contents

Table of Tables

Table of Figures

1.0 Web Application Security in the Enterprise

1.1 Development of Secure Web Applications within an Enterprise Security Architecture Framework

While most of the threats and mitigations on enterprise web-based applications that are discussed in this paper fall under the functional and data views of the Zachman Framework enterprise architecture (EA), it is important to note that security of these applications also largely depends on the organization's enterprise security architecture (ESA) framework. In section 1.5 we will discuss a key security principle commonly referred to as "defense in depth" that further suggests that a solid ESA is a necessary ingredient for development and deployment of secure web-based enterprise applications. In this section we take a closer look at the EA (Zachman Framework in particular) and show its role in promoting web application security. We further discuss the five rings of the ESA and where the various stages of development and deployment of secure enterprise web-based applications fit in. We also draw some helpful information from several NIST special publications, namely SP800-37 ("Guide for the Security C&A of Federal Information Systems), SP800-64 ("Security Considerations in the Information System Development Life Cycle"), SP800-53 ("Recommended Security Controls for Federal Information Systems"), and SP800-26 ("Security Self-Assessment Guide for Information Technology Systems").

1.1.1 Enterprise Architecture (Zachman Framework)

The Zachman Framework model considers the six layered approach to the EA. The highest level is the motivation view, followed by the time view, people view, network view, function view, and data view. The extended ZF also adds a security view to the mix. The motivation view focuses on the list of business goals and strategies. This is important from an application security standpoint because it raises the visibility of the security risks posed by poorly written code. Additionally, at a high level, a security awareness campaign across the organization may be required to align application security with the business goals and develop an improvement strategy. This is also called the timing definition layer.

The time view is concerned with the master schedule or list of events. We cannot emphasize enough how important it is to allocate sufficient time in the schedule for application security review, as well as the creation of various artifacts and performance of certain activities associated with secure development processes. To learn more about integrating application security into SDLC please refer to CLASP. The people view deals with organizational charts and resource availabilities. For instance, it is critical to make sure that a professional security auditor should be available as a mentor to each of the software project teams in order to conduct threat modeling (chapter three) as well as source level security reviews.

The network view identifies the architecture of the network on which the enterprise web-based application is developed and deployed. We will discuss later in this paper that three layers of security protection exist: network, host, and the application itself. While various compensating controls at the network level are required, the bulk of the discussion in this paper focuses around the threats and mitigation at the application level.

The function and data views are where we focus most of our attention in this paper. The functional view deals with the various business processes performed by the application. It is important to note that as functional requirements are documented for the system, a corresponding set of security requirements should be assembled and later used in other parts of the development process. Historically, the focus has been almost exclusively on functionalities (this is what sells after all) and not security, however, we see that the trends are turning around. The data view focuses on entities and data relationship.

Finally the security view of the extended Zachman Framework model defines a list of business assets that need protection, security approaches, as well as security services and mechanisms. For instance, a threat modeling process (discussed in chapter two) will produce a list of business assets in need of protection. Some possible security approaches may include manual security code reviews as well as usage of static analysis tools for automated code scanning for detection of security vulnerabilities. Products and processes for application security should also be coupled with services that may include security awareness training, security process assessments, coding guideline review, etc.

Horizontally the Zachman Framework model consists of the business model, system model, and the technology model. While we focus most of our attention on the technology model of the secure enterprise web-based application, the other two models are also very important both from the "defense in depth" and from the "securing the weakest link" perspective.

1.1.2 Enterprise Security Architecture: Rings 1-5

While EA focuses on business and reference models, ESA is primarily concerned with the various security technologies for the EA. ZF provides an approximation for the ESA with ZF 1-5. ZF1 considers enterprise IT security policy as a framework that is crucial for the enterprise security architecture, operational, physical, web security and disaster planning criteria. ZF2 deals with enterprise security architecture as a framework for enterprise network infrastructure, integration of physical and web security guidance, as well as network security architecture. ZF3 takes into account the baseline network infrastructure security practices. ZF4 is concerned with certification (system level) and accreditation (enterprise level). ZF5 provides the network architecture and system baseline security checklists.

ZF 1-5 play a very important role in securing enterprise web-based applications. For instance, ZF1 mandates the definition of web security guidance which is an essential step for communicating the best practices and policies when it comes to development and deployment of enterprise web-based applications. For instance, CLASP includes some best practices that promote application security, with some of them mapping directly to ZF1, such as institution of security awareness program, monitoring of security metrics, identification of global security policy, etc. ZF2 dictates integration of web security guidance (SP 800-64) that may include specification of operational environment, documenting security relevant requirements and integrating security analysis into source management process, among other things. ZF3 outlines the baseline network infrastructure security practices that are critical for security in web applications because a secure web application cannot be deployed on an insecure network. ZF4 promotes the C&A (SP 800-37) processes that would evaluate the risks associated with the enterprise web application threats on both the system and enterprise security levels. ZF5 outlines the network architecture and system baseline requirements that among other things may include secure coding standards (SP 800-26). CLASP provides a variety of checklists that would map to the ZF5 that can be used as baseline security standards.

ESA focuses on the various security technologies for the EA. These may consist of top-down sets of identified trust modules defining network infrastructure domains and countermeasures/mitigations. In this paper we focus on the application level mitigations to the various security threats. Most of the time, threats can be eliminated simply by following secure coding guidelines. We consider trust modules by identifying the various relevant trust boundaries in an enterprise web-based application and securing their interaction by ensuring that none of the core security services are compromised. ESA also helps prioritize the various risks, which is what we try to do in chapter five with the discussion of threats and mitigations across the various tiers of an enterprise web application where we suggest severities and likelihoods of exploit. ESA helps promote "defense in depth" which we define later in this chapter that supports the core security services at many levels.

Network Applications Consortium (NAC) outlines a vision for EA consisting of overview, governance, architecture and operations. This helps identify ESA as part of the enterprise security program. This leads us into a discussion of the five ring ESA architecture that differs somewhat from ZF1-5. At the highest level (Ring 1) we have the overall enterprise security program whose drivers are business opportunities, business requirements, compliance and threats. Program management (Ring 2) consists of requirements, risk management, strategy, planning, ongoing program assessment and education (awareness). Governance (Ring 3) consists of principles, policies, standards, guidelines, procedures, enforcement and ongoing assessment. Architecture (Ring 4) consists of conceptual framework, conceptual architecture, logical architecture, physical architecture, design and development. Operations (Ring 5) consist of incident management, vulnerability management, compliance, administration and deployment. Much of this paper focuses on the threats and mitigations that fall within rings four and five, but many of the issues relevant to web application security that we have discussed under ZF 1-5 would also map to rings 1-3.

For instance, ring one outlines security threats that can be used to derive security requirements (Ring 2). Security awareness campaigns also belong in ring two that could raise visibility of web application security issues. Ring three mandates policies, standards, and ongoing assessment, all of which are critical to web application security. Ongoing assessments can be used to monitor the various security metrics in order to control changes to the security processes. All of the architectural, design and implementation issues for application security would fall under rings four and five, including threat modeling, automated source code analysis, etc. An in depth discussion of the five ring architecture is not provided here, since we mostly focus on rings four and five in this paper, but as we have mentioned previously, secure enterprise web-based applications can only be developed and deployed in the context of the enterprise security architecture framework that promotes, security guidelines, policies, metrics and awareness. This approach ensures the mitigations of both local and enterprise risks. Additionally, secure enterprise web-based applications are developed under a security aware software development lifecycle (e.g. CLASP) that promotes many of the same activities and controls that are provided by rings 1-5 and also incorporates C&A.

1.2 Web Based Enterprise Applications: Properties and Security Implications

Prior to delving into the various security threats and mitigations for enterprise web applications, it is important to point out some of the differences between these applications and some of the other types of applications. In other words, the question begs to be answered: what is so special about enterprise web applications? After all, there are many other types of applications, such as stand-alone host applications, applications for mainframe environments, traditional client-server applications, embedded system applications, just to name a few. There are several properties of web based enterprise applications that introduce some unique security challenges. While some of these properties may also be present in other types of applications, the combination of all of these is pretty specific to enterprise web applications. Some of these properties include distributed n-tiered architecture, transparency, interoperability, heterogeneity, openness, scalability, failure handling, concurrency, remote access, real-time operation, content delivery over the web, data transfer over the web and need for thorough input validation [2,5].

Many of these properties have an impact on CI4A core security services. For instance, confidentiality obviously plays an important role in data transfer over the web. Concurrent access has important implications from the integrity perspective. Authentication is very important because of remote access. Authorization can be compromised due to poor input validation (e.g. buffer overflows). Finally, availability is an extremely important security concern in enterprise web applications and can be affected by concurrency, failure handling, real time operation, etc. Since enterprise web applications often consist of a collection of various software components from different vendors, vulnerabilities in each of these components might compromise the security of the whole application. Additionally, enterprise web applications are often distributed across heterogeneous hardware and software platforms, introducing additional security challenges. The weakest link principle states that the system is only as secure as its weakest link and enterprise web applications have many possible places to contain weak links. One of the major ones is attributable to the application layer.

1.3 Web Based Enterprise Applications: N-Tiered Architecture

Most enterprise web applications utilize the n-tiered architecture that generally includes the web server tier, application server tier and persistence (database) tier. There are various application level threats specific to each of the tiers and thus security of these applications should be considered in the context of the n-tiered architecture. A detailed discussion of threats and mitigations across each of the tiers is offered in chapter five of this paper. There are also various application level compensating controls that need to be provided at each tier. Some of those controls are provided by the underlying set of technologies used (J2EE and .NET examples are discussed later), while others must be provided by the programmers, architects and designers involved in building the enterprise web application. Application security must be addressed across all of the tiers and at multiple levels (defense in depth principle). Following the defense in depth and securing the weakest link principles is crucial in development of secure enterprise web applications. Figure 1 below illustrates some of the threats to a typical 3-tiered enterprise web application and categorizes those threats into application, network and host level. Network and host level threats will only be given cursory overview in this paper, while application level threats will be discussed in great detail.

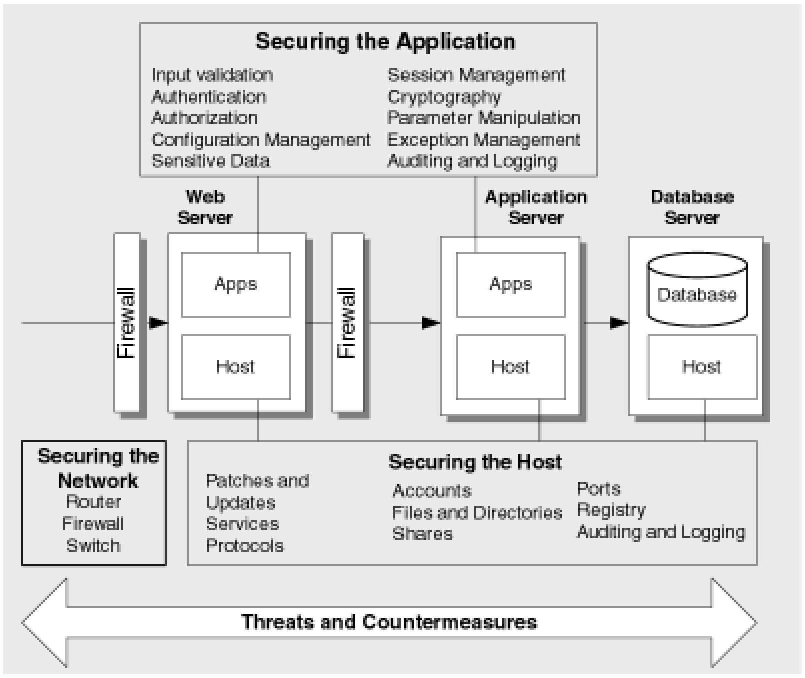


Figure 1: Securing a 3-tiered Enterprise Web Application: At the Application, Network and Host Layers [10]

There are several things that are important to take away from Figure 1 above. It is important to understand where the compensating controls for the various threats are present, at the network, host or application layers. A secure enterprise web application must be deployed on a secure network and reside on secure hosts. There should never be a reason to justify an existing vulnerability at the application level just because the threat might be mitigated by some network security mechanism, as this would violate the defense in depth principle. Analogously, just because the host might be configured in a way as to restrict the privileges of the application, does not mean that it is acceptable for the application to be susceptible to command injection attacks. Configurations might change, or an enterprise web application might be deployed in a different environment, or one level of protection might be circumvented all together. The point is that redundancy is instrumental to ensuring security in enterprise web applications. While some threats might be mitigated on multiple levels, other threats have compensating controls present only at the network, application or host layer. For example, attacks exploiting poor input validation only have compensating controls at the application layer.

There are threats on the web server tier (also called the presentation tier) that must be mitigated. The fact that an attacker can access the web server remotely makes it a very likely target. Additionally, the web server tier of an enterprise web based application is likely to become a front end of any attack on the application. Therefore it is important to anticipate possible attacks on the web server and have countermeasures in place to defend against them. Some common threats to the web server include profiling, denial of service, unauthorized access, arbitrary code execution, elevation of privileges, as well as viruses, worms and Trojan horses.

The application server tier typically contains all of the business logic for the application. Some serious concerns for this tier are network eavesdropping, unauthorized access, as well as viruses, Trojans and worms. A lot of problems with unauthorized access can be attributed to application security issues, such as exploitable buffer overflow conditions, command injection problems, SQL injection, etc. We will discuss in great detail later in the paper the mitigations applicable at the application level.

The database server tier contains the databases to which the application data is persisted. Some of the main threats to the database server tier include SQL injection, network eavesdropping, unauthorized server access and password cracking. Most problems with SQL injection result from poor input validation by the application and can therefore be readily avoided. Some other problems at the database server tier include insecure storage, where data is stored unencrypted in the database. It is important to remember that data typically spends far more time in storage than it does in transit, and consequently protecting data during storage from the prying eyes of attackers is essential.

From the application security perspective, some of the major areas of concern at the web and application tiers are input validation, authentication, authorization, configuration management, sensitive data, session management, cryptography, parameter manipulation, exception management and auditing/logging.

1.4 Application Vulnerability Categories

This section offers a discussion on some of the most common application level vulnerabilities that plague web based enterprise applications. A lot of these will be revisited in chapter five, but they are introduced here. The discussion focuses around the OWASP top ten vulnerabilities [24].

1.4.1 Unvalidated Input

Unvalidated input is a fairly broad vulnerability category that has very serious consequences. All web based applications need to handle input coming from a variety of untrusted sources (most notably the user of the application). If the input is not validated, attackers can use that opportunity to attack the backend components of the applications. In general, validation needs to be performed each time the data crosses a trust boundary. Validation may be performed on the client site, but for performance purposes early. Client site validation should never be relied upon for security. Validation also needs to happen at the web, application and database tiers. A variety of application level attacks could be avoided if input validation is performed properly; those include SQL injection, Cross Site Scripting (XSS), buffer overflows, format string, cookie poisoning, hidden field manipulation, command injections, etc. Unvalidated input could lead to compromise of authorization, integrity, authentication and availability security services. That is discussed in more detail later in the paper. All user input in HTTP requests should always be aggressively validated against white lists (list of allowable input) as opposed to black lists (list of input that is not allowed).

1.4.2 Broken Access Control

Broken access control (authorization) problems result when restrictions on what authenticated users are allowed to do are not properly enforced. Application vulnerabilities that fall in this category could allow attackers to access accounts of other users, view confidential information or use unauthorized functionality. There is a variety of attacks that fall into this category that could allow attackers to escalate privileges. For instance, reliance on hidden fields to establish identity for the purpose of access to web based administrative interfaces will allow an attacker unauthorized access because hidden fields can be easily manipulated. Exploitation of some other vulnerability in the application can cause violation in access control. For instance, crafting an attack that exploits a buffer overflow to modify some flag variable used for an authorization check could result in broken access control. Some key access controls issues include insecure ids, forced browsing past access control checks (URL tampering), path traversal, file permissions and client side caching.

1.4.3 Broken Authentication and Session Management

If proper steps are not taken to protect session tokens and account credentials, such as passwords, keys and session cookies, attackers can use those to defeat authentication checks and assume identities of other users. Authentication mechanisms can be circumvented if authentication credentials and tokens are not properly handled by credential management functions such as those to change password, retrieve forgotten password, account update, etc. Session tokens need to be properly protected against hijacking so that attackers cannot assume identities of authenticated users simply by hijacking the session after the authentication has taken place. Session tokens created should be strong and should be properly protected throughout the lifecycle of the session. Secure Sockets Layer (SSL) technology can go a long way towards creation of a secure session; however SSL is not properly implemented in many instances. Additionally, attacks like cross site scripting can allow an attacker obtain the session tokens even if SSL is used.

1.4.4 Cross Site Scripting (XSS)

Cross site scripting attacks exploits vulnerabilities that fall in the category of poor input validation. Essentially an attacker submits executable scripts as part of the input to the web application and those scripts are then executed on the browsers of other clients. Those attacks often lead to information disclosure of the end user's session tokens, attack the end user's machine or spoof content to fool the end user. Disclosure of session tokens can lead to session hijacking and allow an attacker to assume a valid user's identity (compromise authentication). Spoofing content can also lead to information disclosure if for instance a valid user input his/her login and password information into a form sent to an attacker. XSS attacks can occur at the web tier or at the application tier and aggressive white list input validation should be present in the application to thwart these attacks. There are two types of XSS attacks: stored and reflected. In stored XSS attacks, the malicious script injected by an attacker is permanently stored by the web application for later retrieval by the end user who requests the affected data. Since the malicious script at that point arrived from a trusted server, the client executes the script. In reflected attacks, the malicious script is transferred to the server and then is echoed back to the user either in an error message, search result, or some other response to the end user which includes some of the data fields into which a malicious script has been inserted as part of the request.

1.4.5 Buffer Overflow

Buffer overflow attacks are possible if no proper bounds checking is performed on the buffer to which user input is written. Carefully crafted input that writes data to the buffer past the allocated range can be used to overwrite the return pointer on the stack and point the program counter to a location where malicious shell code has been planted. The attack code is then executed resulting in severe authorization breach on the application (execution of arbitrary code). Arbitrary code can be executed on the host system with the same privileges as those that were granted to the web application. Following the principle of least privilege could help limit the amount of damage an attacker can cause following a successful exploitation of the buffer overflow vulnerability. Buffer overflows can be avoided by proper input validation. Additionally, the likelihood of introducing buffer overflows into the application can be significantly reduced if safe string and memory manipulation function are used. While execution of arbitrary code and taking control of the application process are the more drastic possible consequences of a successful exploitation of the buffer overflow vulnerability, a more frequent impact could be on system availability since buffer overflows will often cause system crashes. Besides the custom application code, components that can be vulnerable to buffer overflows may include CGI components, libraries, drivers and web application server components. Since JAVA is generally considered a safe language in terms of bounds checking and thus fairly protected against buffer overflows, enterprise web applications built with J2EE are generally considered protected against buffer overflows. However, that is not entirely the case. Buffer overflow attacks, while more difficult and rare with JAVA based applications, can still take place. An example may be a buffer overflow in JVM itself. Format string attacks are a subset of buffer overflow attacks. Buffer overflows are discussed in more detail in chapter five.

1.4.6 Command Injections

Enterprise web applications pass parameters when they access external systems, applications, or use local OS resources. Whenever possible, those parameters should not come directly from the user and be defined as constants. Otherwise, they should be rigorously validated prior to usage. If an attacker can embed malicious commands into these parameters, they may be executed by the host system when the access routines are invoked by the application. SQL injection is a particularly common and serious type of injection, where SQL statements are passed to the web application and then without validation are passed to the routine that accesses the database with that SQL statement. Command injections can be used to disclose information, corrupt data and pass malicious code to an external system application via the web application.

1.4.7 Improper Error Handling (Exception Management)

Errors and exceptions occurring during the operation of the enterprise web application should be handled properly. Error information that is echoed to the user in its raw form can cause information disclosure. For instance, letting an attacker know the OS that the host machine is running, the version of the database and the database driver can allow the attacker to exploit existing vulnerabilities for those technologies. Improperly managed exceptions can result in disruption of availability or cause security mechanisms to fail.

1.4.8 Insecure Storage

Data spends far more time in storage than it does in transit and must therefore be stored in a secure manner. Encryption of data is not a bad idea to promote confidentiality and protect application data, passwords, keys, etc. Even when encryption is used, it is not often used properly. Some of the common types of mistakes make that fall into this category include failure to encrypt critical data, insecure storage of keys, certificates and passwords, improper storage of secrets in memory, poor sources of randomness, poor choice of cryptographic algorithms, and homegrown encryption algorithms. While encryption can help protect confidentiality of stored data, hashing can be used to ascertain integrity. A central point here is that as little as possible of sensitive information should be stored by the enterprise web applications. For instance, it might make sense to ask the users to reenter their credit card number each time instead of persisting it.

1.4.9 Denial of Service (DoS)

Attackers can completely consume the resources of the web application as to make it impossible for other users to use the application. Attackers can lock out legitimate users from their accounts. For instance a poorly implemented "forgot password" feature will allow an attacker to continuously change the password for a legitimate user thus effectively locking that user out from their account. DoS can result in failure of various components of the enterprise web application, or even the application as a whole. Many coding problems can compromise the availability of the application and thus facilitate DoS attacks. Some of these include usage of uninitialized variables, null pointer dereferences, poor concurrency management, etc. Concurrent access in enterprise web applications can exacerbate the availability concerns, while load balancing techniques can help provide some mitigation. In chapter five there is a discussion of various application level mistakes that can result in disruption of availability. Lack of or improper error handling can sometimes result in crashes that might enable DoS attacks.

1.4.10 Insecure Configuration Management

Secure configurations at the application, network and host level are critical for provision of security. In fact, most default configuration on many commercial hardware and software products do not provide a sufficient level of security and must be modified. True to the least privilege principle, configuration should always allow for the minimum privilege necessary. Secure configuration is required on the web, application and database server. In order to securely deploy an enterprise web based application, all configurations must be properly performed. Some configuration problems can be due to unpatched security flaws present in the server software, improper file and directory permissions, error messages providing too much information, improper configuration of SSL certificates and encryption settings, use of default certificates, improper server configurations enabling directory listing and directory traversal attacks, among others.

1.5 Key Security Principles

This section introduces some of the guiding principles for software security. Several of those have already been alluded to earlier in the paper, namely the principle of securing the weakest length and defense in depth. These principles play a vital role in application security and should be followed across all stages of the software development lifecycle. These principles also have room in other security areas, including network and host security, and are generally derived from a broader field of engineering. The discussion in this section focuses on those principles as they apply to application security. These security principles are referenced in subsequent discussion [1].

1.5.1 Secure the Weakest Link

This principle dictates that security in a software system is only as strong as the weakest link. It is therefore critical when building software systems to build in all around strong security and make sure that no security holes are present. For instance, if the data is transferred in the system using a strong encryption algorithm, but is stored unencrypted in plaintext, the attacker is more likely to try and go after the data while it is in the repository. "Security practitioners often point out that security is a chain. And just as a chain is only as strong as the weakest link, a software security system is only as secure as its weakest component" [1, p.93]. Attackers will try to compromise the weakest link of the software system and will use all possible techniques, including social engineering attacks. For instance, the weakest link of the system could be a member of technical support falling prey to social engineering. An attacker will not waste time trying to brute force a strong password to get into a system but look for other ways to get in instead. In general, cryptography (if used properly) is rarely the weakest link in the software system, so the attacker is much more likely to attack the endpoints: "Let's say the bad guy in question wants access to secret data being sent from point A to point B over the network protected with SSL. A clever attacker will target one of the end points, try to find a flaw like a buffer overflow, and then look at the data before it gets encrypted, or after it gets decrypted" [1, p.94]. A good example of this is the notorious design flaw from a security standpoint in the gateway server used by wireless application protocol (WAP) to convert data encrypted with WTLS to SSL. An attacker does not have to try and attack the data while it is encrypted, but just wait for the gateway server to decrypt the data. Consequently, any exploitable flaws in the gateway server software (e.g. buffer overflows) could be used as a way to compromise data confidentiality. Consequently, encryption should be used as part of end-to-end solution and both ends should be properly secured.

Similarly, an attacker will not try to compromise a firewall (unless there is a well known vulnerability in firewall software), but instead try to exploit vulnerabilities in applications accessible through the firewall. Since software applications with web interfaces are often the weakest link, it is imperative to make sure that those are secure (hence the reason for writing this paper). Usage of protocols for remote object access, such as SOAP and RMI should also be carefully considered because of the possible security implications. The point is that with all the technologies involved in the enterprise web application, there is a great potential that an attacker will be able to find a hole somewhere, and that is what makes securing those systems such a hard problem. It is therefore critical to be aware of where the weakest links in the system are and do everything possible to protect those until an acceptable level of security is achieved. While 100% security is unattainable, what constitutes "acceptable security" depends on the business use.

1.5.2 Build Defense in Depth

This principle dictates that a variety of overlapping (redundant) defensive strategies is required so that if one defense layer is compromised another layer will be available to thwart an attack and prevent a full breach. A well known design principle of programming language design states that it is necessary to "have a series of defenses so that if an error isn't caught by one, it will probably be caught by another" [27]. For instance, in the context of enterprise web applications, it is not a good idea to justify unencrypted data transfer between the application server and the database server by arguing that those servers are behind a firewall. If an attacker finds a way to get past the firewall (or maybe an attack originates from inside the corporate network), a system should have additional security check guards. For instance, communication between servers should be encrypted and data should be stored encrypted. Additionally, application firewalls should be used to protect against application attacks. However, a caveat about application firewalls is that they should not be used to justify vulnerabilities in the applications themselves. The fact that input filtering might be available in the application firewall does not mean that the applications behind the firewalls do not need to perform input validation, as this would violate the defense in depth principle. Having multi-leveled defense mechanism makes it far more difficult for an attacker to cause full breach of the system and makes it more possible to secure the weakest links of the system.

1.5.3 Secure Failure

While failures cannot always be prevented, the ability of the software system to handle failures gracefully is critical and will allow prevention of many attacks on the enterprise web application, including those related to denial of service and information disclosure. While failure is often unavoidable, security problems related to failure can be avoided. This principle would dictate that the raw system failure log should never be presented directly to the user without filtering as it can provide an attacker with a lot of useful information for subsequent attacks. All errors should also be caught and handled appropriately as this will help prevent denial of service attacks that can exploit system crashes occurring due to improperly handled errors. Exception management capabilities in the Java programming languages go a long way towards handling errors properly, but developers often neglect to make proper use of those capabilities.

Attackers often need to cause the right failure, or wait for the right failure to happen to allow them compromise the system in some way. Consider a situation where a client and server want to communicate via Java's RMI. A server might want to use SSL, but a client may not support it. In this case a client downloads proper socket implementation from the server at runtime. This can be a potentially serious security problem because the server has not yet authenticated itself to the client when the download occurs and thus a client could really be downloading malicious code from a malicious host. In this situation trust is extended where it should not be. The failure occurs when the client fails to establish a secure connection using default libraries, but instead establishes a connection using whatever software it downloads from an unauthenticated and thus untrusted remote host (server). If secure failure principle was followed, the client would first authenticate the server using the default libraries prior agreeing to the download of additional SSL libraries at runtime.

1.5.4 Least Privilege

This principle states that users or processes should only be granted the minimum privilege necessary to perform the task that they are authorized to perform and that the access should be granted for the minimum amount of time necessary. This means that the permissions governing access control should be configured in the minimalist fashion. In the event that user credentials are compromised or application process is taken over by an attacker, the damage that can be done is limited to the rights that the user or the process had. For this reason, it is not a good idea for a user to be logged into a system with administrative privilege or for a process to have more rights than is absolutely necessary.

For instance, if a buffer overflow in the application is exploited to take over a process, arbitrary code execution will be restricted by the rights that the compromised process had. If the permissions were conservative, than the damage could that an attacker can cause will be limited. Additionally, if the amount of time for which the access was authorized is limited, an attacker will have a limited window of opportunity to cause damage. The point is to not give out a full access key to your car when a valet key is sufficient.

Programmers often make the mistake of requesting more privileges than is necessary to get the job done. For instance a process may only need read access to some system object, but a programmer may request full access to the object to make life easier, or in case full access is required later. Some insecure settings might also be a problem, where a default argument requests more privileges than is necessary to perform the task. Windows API can be an example of this where there are some calls for object access that grant full access if 0 is passed an argument (default). In this case, the burden would be on the programmer to restrict the access. It is also important to relinquish privilege after it is no longer necessary, which can be a problem in Java because there is no operating system independent way to relinquish permissions (cannot be done programmatically).

In enterprise web applications, execution of untrusted mobile code can often be problematic. This code should be run in a sandboxed environment with as little privilege as possible. Unfortunately, default system configurations are often insecure and the product vendors advertise very generous security policies because the software was possibly built in a way as to require more privileges than what is required. This makes life easier from functionality standpoint, but is horrible for security.

1.5.5 Compartmentalize

The access control structure should be subdivided into units (compartments) and should not be “all or nothing”. This will also make it easier to conform to the principle of least privilege. This way if an attacker compromises one compartment of the system, he will not have access to the rest of the system. This is similar to having multiple isolated chambers in a submarine, so that if one chamber overflows with water, the whole submarine does not sink. Of course compartmentalization requires a more complex access control structure in the application. Most operating systems do not compartmentalize effectively. Since a single kernel is used, if one part is compromised, then the whole thing is. Trusted operating systems are available, but they are quite complicated and suffer considerable performance degradation because of the overhead associated with increased access control granularity. Compartmentalization can be hard to implement and hard to mange and thus few developers bother with following this principle. Good judgment is the key when following this principle. In the context of enterprise web applications, it is essential to define security roles in a way that promotes compartmentalization. It might also make sense to distribute critical server side application modules to different machines in the event that any of them become compromised.

1.5.6 Simplicity of Design

Unnecessarily complex design and implementation can cause serious security problems. In general, it is a good engineering practice to build systems that do what they are supposed to do and nothing else in the simplest way. Complexity will make it harder to evaluate the security posture of the application and will increase the likelihood of bugs. It is also harder to evaluate any unintended consequences of the code if it is overly complex. Correct code is secure code and it is easy to show how code correctness suffers with increased complexity. Density of functionality and security related bugs increases with code complexity and it can be shown that there is a correlation between the two. The point is to keep code simple which makes it easier to write and review for security. Complex code is also harder to maintain.

It also makes sense to reuse proven software components when possible. Software reuse makes sense from various perspectives including security. If a certain component has been around for a while without any discovered vulnerabilities, chances are that it is safer to use than anything that is homegrown. This definitely holds true for homegrown cryptographic algorithms which should never be used. It is also true that the more functionality is introduced, the more room there is to introduce security related problems. It is important to understand possible security implications of all new functionality prior to making it part of the system. It can be quite complex to have security related code throughout the whole system. Therefore it makes sense to have a few modules which create entry points to the system through which all external input must pass and place security checks in those modules. Special care should be taken to make sure that there is no way around those security check points. Hidden ways to circumvent the checks left for administrator level users can be used by savvy attackers and therefore should not be part of the system. It is important to keep the user in mind as well. For instance, default configurations should provide more than minimal security, and it should not be overly complex to change the security related settings, or the users will not do it.

1.5.7 Privacy

The system should store as little confidential information as possible. For instance, it often makes sense to not store credit card numbers since the users can enter them again. System designers and programmers also need to make sure to protect against information disclosure that could facilitate an attacker launch an attack on the application. Even information disclosure about the operating system used or the version of the database used can help an attacker. In fact, misinformation can even be used to fool a potential adversary. The fewer the number of assets that need to be protected, the easier it will be to protect them. While "one-click-shopping" experience offered by companies like EBay might be convenient, from a security standpoint something like this can be a nightmare and extremely difficult to get right. All data should be stored securely in the database (encrypted). The keys for encrypting and decrypting the data should be kept on a separate machine from the database. In this scenario, if an attacker manages to compromise the machine where the database resides and gain access to it, he or she will still need to obtain the decryption key from another machine. In enterprise web applications, application server software behavior can tell a skilled attacker which software the company runs, be it BEA Weblogic, IBM Websphere, Jakarta Tomcat, or anything else. The same is true for web server behavior, be it Microsoft's IIS, Apache, etc. Even if versioning information is turned off, the behavior of the system can give a skilled attacker enough information.

1.5.8 Be Smart About Hiding Secrets

Hiding secrets is difficult and should be done appropriately. For instance, many programmers make the mistake of hard coding encryption keys in the program code, assuming that an attacker will never have access to the source code. This is a very bad assumption. Hard coding encryption keys and other security tokens is extremely dangerous and should never be done. First, an attacker might have access to the source code. For instance an attacker could even be an insider. Second, the attacker might be skilled in decompiling the binary to get useful information about the source code, including the hard coded key. All security tokens should be stored securely in a database residing on a separate system.

1.5.9 Reluctance to Trust

Trust should be extended only after sufficient authentication and verification that trust is warranted. We can go back to an example where SSL classes are downloaded by a client from a server before the server has authenticated itself. All mobile code should be treated as suspicious before confirmed to be safe and even then should be run only in a sandboxed environment. In an enterprise web application, servers should not trust clients by default and vice versa. Just because both servers are inside a corporate network does not mean that authentication is not necessary. Following this principle also helps with compartmentalization since an attacker compromising an application server will not have automatic access to the database server. Social engineering attacks are a prime example where this principle is not followed which can have potentially disastrous consequences. Since trust is transitive, it is important for trusted processes to be very careful when invoking other processes, and should do so only if those other processes are also trusted. If untrusted code is invoked by a trusted process, this code will have all of the system privileges of the trusted code.

1.5.10 Use Proven Technologies

It has previously been mentioned that software reuse is a good policy that promotes security. Particularly published and well reviewed components that have stood the test of time and scrutiny for security problems are good candidates for reuse. While those components may still contain problems that are undiscovered, it is less likely to run into security problems with those components than with homegrown ones. Homegrown cryptographic solutions should never be used since they are virtually guaranteed to be weaker than publicly available ones. Same is true for various security libraries that are publicly available. Proven technologies should be drawn from communities that pay attention to security and have procedures in place to provide rigorous reviews for security. There is also something to be said for the “many eyeballs” phenomenon, that states that the likelihood of a problem in the piece of code goes down the more people have reviewed the code (although there are some infamous counterexamples to this principle).

2.0 Threat Modeling

2.1 Introduction to Threat Modeling

One of the big questions that architects and designers of enterprise web applications need to answer is: what are the threats on this application? This question is very relevant, because after all, it is impossible to make sure that an application is secure without understanding the types of attacks that adversaries may attempt to launch on the system. The technique for formal evaluation of threats on the application is commonly called threat modeling. Threat modeling process usually starts at the information gathering phase, proceeds to the analysis phase, and culminates with a report that can be used by application architects, designers, developers and security auditors to drive construction and verification.

A typical threat model would first document all of the protected resources of the application, such as data and execution resources. After all, attackers often try to get at those protected resources so it is imperative to know what these are. To get to those resources an attacker would have to use some input or output stream flowing to or from the application. The next step in threat modeling is to isolate the input and output vectors of the application, document the various threats as they apply to each of those input and output vectors and construct attack trees that combine the various input and output vector threats to abstract the various attack scenarios on the application being evaluated. While it is impractical and probably impossible to try and document every single possible attack scenario on the enterprise web application, threat modeling process provides a systematic way of considering the threats thus reducing the chance that some are overlooked and pointing out some of the potential weak or problem areas that need to be given special attention. Threat modeling helps comply with the principle of securing the weakest by facilitating in understanding of what the weak links are. As part of threat modeling it is important to document all of the security related assumptions and decisions made as this information will help understand what affect architectural changes will have on the security posture of the application. After having evaluated the threats, it should be documented what threats are mitigated by the architecture, design and implementation and how that is accomplished. All threats that are not addressed should be clearly documented as well. The threats should also be rated based on severity and potential impact. Section 2.2 provides a sample threat model on the enterprise web application built with J2EE technologies. Section 2.3 introduces Microsoft’s approach to threat modeling with focus on .NET.

2.2 Sample Threat Model: Open Source Enterprise Web Application JPetStore

Since the concept of threat modeling may seem rather abstract at first glance, an example is provided to help the reader understand what is involved. The threat modeling in this example was performed on an open source enterprise web application called JPetStore 4.0. This is J2EE application built on top of the Apache's STRUTS Model View Controller (MVC) architecture. It will become clear that it is important to consider the underlying technologies used in the enterprise web application when performing threat modeling. Chapter three will evaluate these technologies in the J2EE world and chapter four will do the same for the .NET world. In both cases the discussion will focus around the OWASP top ten application security threats as well as many others. This example provides a preview into enterprise web applications built with J2EE and demonstrates how threat modeling can be performed on a sample application.

2.2.1 Overview of JPetStore 4.0

JPetStore 4.0 is a web application based on iBATIS open source persistence layer products, including the SQL Maps 2.0 and Data Access Objects 2.0 frameworks. JPetStore is an example of how these frameworks can be implemented in a typical J2EE web application. JPetStore 4.0 is built on top of the Struts Model View Controller (MVC) framework and uses an experimental BeanAction approach. JPetStore uses a JSP presentation layer.

This section analyzes the architecture of the JPetStore 4.0, a J2EE web application based on the Struts framework. The various security assumptions are documented, including environmental, trust and security functionality. Protected data and execution resources of JPetStore are then enumerated. Input and output vectors (attack surface) for the application are then described, followed by the discussion of the various threats on the architecture of the application through those input and output vectors, and how those threats can affect the protected data and execution resources of the application. The section also discusses some possible mitigations to the threats, and what mitigations (if any) are built into the JPetStore architecture.

2.2.2 Security Assumptions

Environmental Assumptions

Number	Assumption Description
E1	**There are network and host security mechanisms in place, such as firewalls, routers, switches, intrusion detection systems, SSL, etc. The focus here is on security of the web application itself.**
E2	**The network hardware and software are configured to promote maximum security.**

Trust Assumptions

Number	Assumption Description

T1	
T2	

Security Functionality Assumptions

Number	Assumption Description
F1	
F2	

2.2.3 Protected Resources

Data Resources

Number	Name	Type	Description	Roles
D1	**Application Database**		**The data bank for JPetStore**	
D2	**All application code itself (particularly struts-config.xml and web.xml contained in the Web/WEB-INF folder)**		**Application code (at all tiers of the deployed application) must be protected from tampering**	
D3	**Customer accounts**		**Customer accounts store customer specific information.**	

Privileged Execution Resources

Number	Name	OS Priv	Description	Roles
P1	**Registry**	**R/W/C/D**	**Host Based: Registry Access**	
P2	**File System**	**R/W/C/D**	**Host Based: File System Access**	
P3	**Memory**	**R/W**	**Host Based: Memory Access**	
P4	**CPU**	**X**	**Host Based: CPU utilization**	
P5	**Cache**	**R/W**	**Host Based: Cache access**	
P6	**Shares**		**Host Based: Shared resources access**	
P7	**Services**	**R/X**	**Host Based: Services usage**	
P8	**Accounts**		**Host Based: Account access**	
P9	**Auditing**	**R/W/C/D**	**Host Based: Auditing and logging**	

	and Logging		**facilities**	
P10	**Ports**	**R/W**	**Host Based: Enabled ports**	
P11	**Protocols**		**Host Based: Available protocols**	
P12	**Router**		**Network Based:**	
P13	**Firewall**		**Network Based:**	
P14	**Switch**		**Network Based:**	

Users and Roles

Number	Name	Description	Use-Case	Resources
R1	**Customer**	**Uses JPetStore to purchase pets**		
R2				

Supplemental Design Decisions

Number	Element	Design
S1	**Data Access**	**Use Data Access Objects (DAO) 2.0 (iBatis)**
S2	**Data Manipulation**	**Use SQL Maps 2.0 (iBatis)**
S2	**Struts MVC**	**View (JSP), Model (ActionForm), Controller (BeanAction)**

2.2.4 Application Functionality and Data Flow Diagram

JPetStore does not use conventional J2EE architecture in that it utilizes reflection to perform business logic in servlets rather than using Entity Java Beans (EJBs). JPetStore uses reflection based database mapping layer (experimental BeanAction approach) that has some serious implications from a security standpoint, specifically pertaining to parameter manipulation. Reflection is used to access the JavaBean properties in the Struts-based JSP pages. The properties are stored in struts-config.xml and the action is parsed directly out of the URL string. Consequently, input validation would have to be extensively performed in each of the public methods, or the application would be vulnerable to a variety of attacks, including SQL injection, XSS and command injection. We now provide an overview of the way that data flow takes place in JPetStore 4.0.

The Pet Store application feature set will be familiar to most users of the Web. The customer can browse through a catalog of pets that vary by category, product type and individual traits. If the customer sees an item they like, they can add it to their shopping cart. When the customer is done shopping, they can checkout by submitting an order that includes payment, billing and shipping details. Before the customer can checkout, they must sign in or create a new account. The customer's account keeps track of their name, address and contact information. It also keeps track of profile preferences including favorite categories and user interface options (banners etc.) [8]. Below is a site map of the general flow of the application:

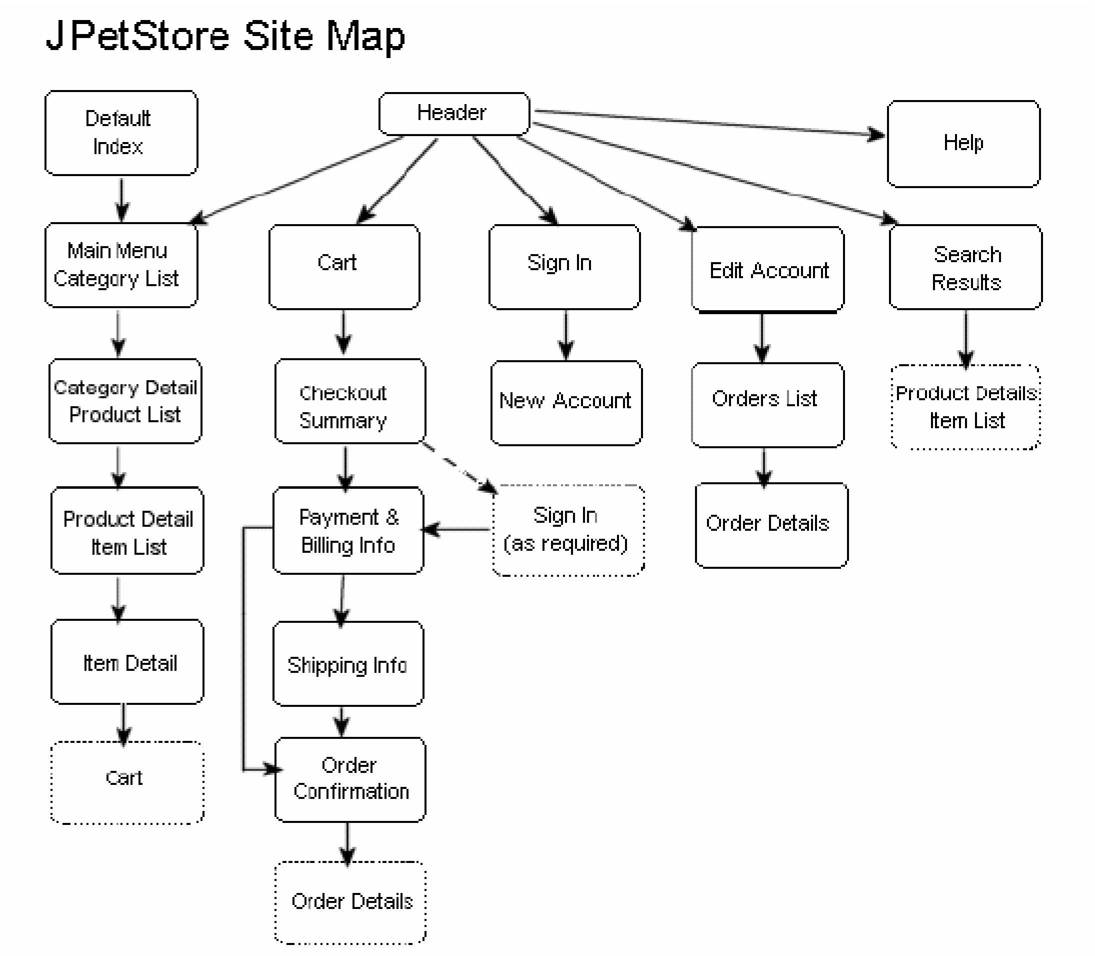


Figure 2: Sitemap for JPetStore 4.0 [8]

-- JPetStore does not use Stored Procedures, nor does it embed SQL in the Java code.
-- JPetStore does not store HTML in the database.
-- JPetStore does not use generated code.
-- JPetStore uses the Model View Controller presentation pattern.
-- JPetStore was implemented using completely open-source freeware, including the development tools, runtime environment and database management system.

JPetStore uses the Struts Model View Controller pattern to improve the maintainability of the presentation layer –the layer which is often the most likely to change. The JPetStore persistence layer uses SQL mapped to Java classes through XML files. JPetStore eliminates stored procedures and eliminates the SQL from the Java source code to improve maintainability in the data tier [8]. The picture below shows a high level view of the architecture of the application.

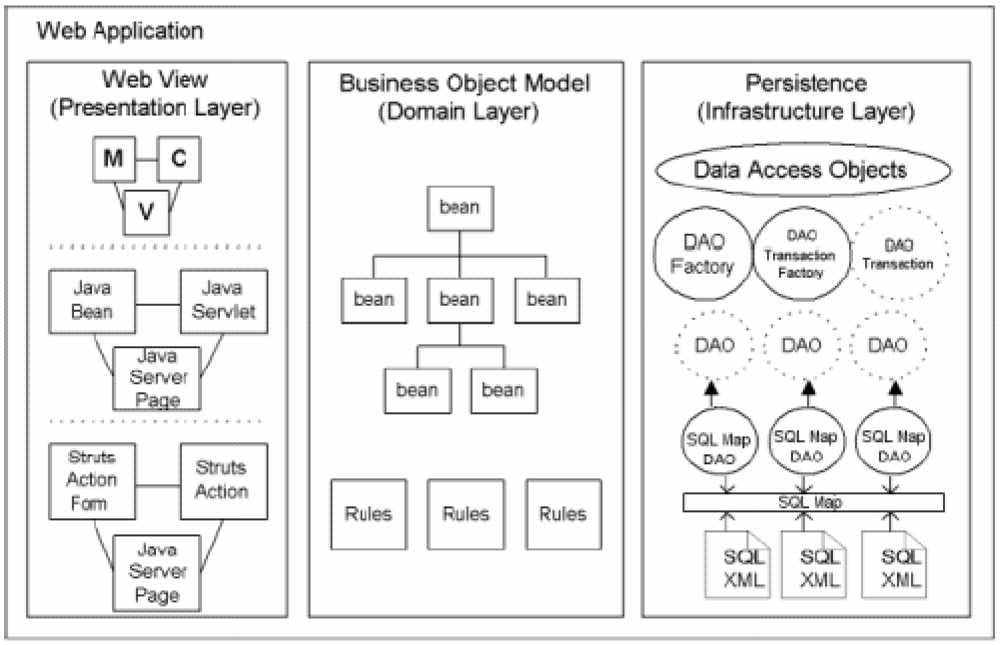


Figure 3: JPetStore 4.0 Application Architecture [8]

JPetStore 4.0 application uses the Model Control View (MVC) based on Struts. Struts is freely available and open-source. The Struts framework played a key role in the design and implementation of the JPetStore presentation layer. Combined with JSP, it helped maintain a consistent look and feel as well as good flow control throughout the application. It did this while helping reduce the overall code size for JPetStore and improving the overall design. [8]

Struts has three main components: the ActionForm (model), the JavaServer Page (view) and the Action (controller). Struts uses an XML descriptor file to connect these three components together, which helps simplify future code maintenance on the presentation layer –a layer prone to change. The diagram below illustrates the three main components and how they interact with each other.

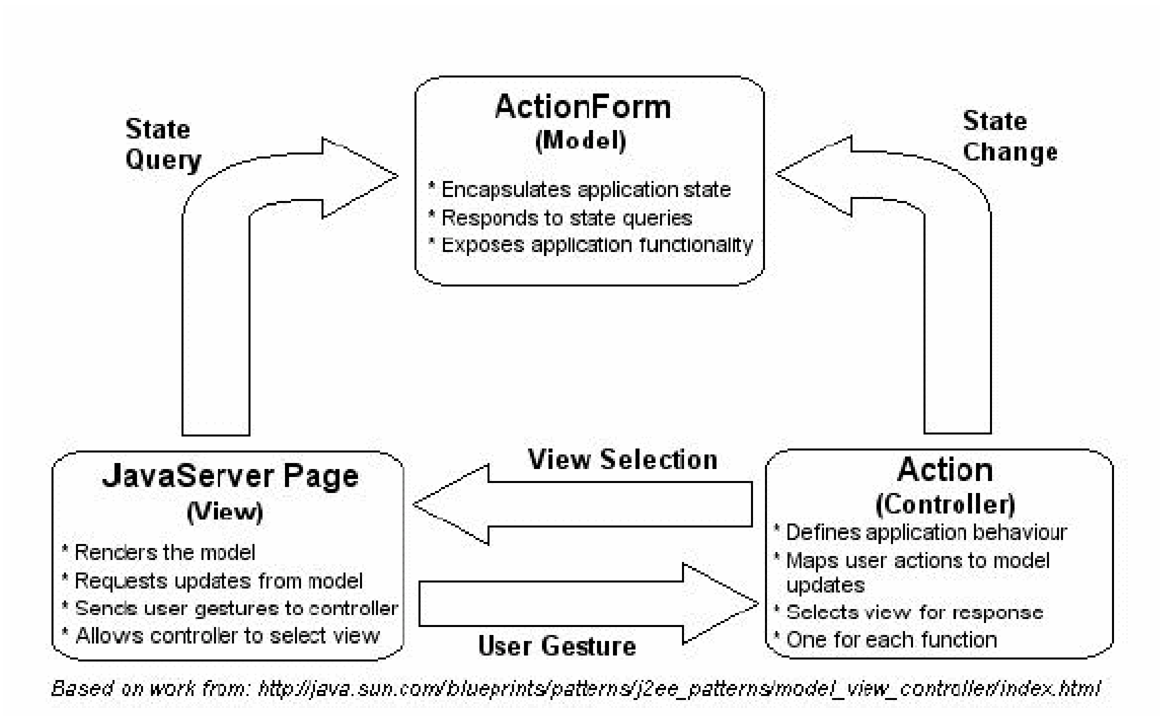

State Query
ActionForm
(Model)
* Encapsulates application state
* Responds to state queries
* Exposes application functionality
State Change
JavaServer Page
(View)
* Renders the model
* Requests updates from model
* Sends user gestures to controller
* Allows controller to select view
View Selection
User Gesture
Action
(Controller)
* Defines application behaviour
* Maps user actions to model updates
* Selects view for response
* One for each function
Based on work from: http://java.sun.com/blueprints/patterns/j2ee_patterns/model_view_controller/index.html

Figure 4: JPetStore 4.0 Application Data Flow [8]

2.2.5 Threats by Application Vulnerability Category (OWASP top ten)

Number	Type	Description	Protocol	Role
C1	**Input Validation**	**Buffer overflows; cross site scripting (XSS); SQL injection; canonicalization**		
C2	**Authentication**	**Network eavesdropping; brute force attacks; dictionary attacks; cookie replay; credential theft**		
C3	**Authorization**	**Elevation of privilege; disclosure of confidential data; data tampering; luring attacks**		
C4	**Configuration Management**	**Unauthorized access to administration interfaces; unauthorized access to configuration stores; retrieval of clear text configuration data; lack of individual accountability; over privileged process and service accounts**		

C5	**Sensitive Data**	**Access sensitive data in storage; network eavesdropping; data tampering**		
C6	**Session Management**	**Session hijacking; session replay; man in the middle**		
C7	**Cryptography**	**Poor key generation or key management; weak or custom encryption**		
C8	**Parameter Manipulation**	**Query string manipulation; form field manipulation; cookie manipulation; HTTP header manipulation**		
C9	**Exception Management**	**Information disclosure; denial of service**		
C10	**Auditing and Logging**	**User denies performing an operation; attacker exploits an application without trace; attacker covers his or her tracks**		

2.2.6 Attack Surface

The diagram below shows the flow of the application attack on a Web based application. Only the application level threats are discussed in this document. This document does not address the network and host level threats. However, network, host and application security are all critical components to building and deploying a secure web application. It is also critical to consider the attack surface across all tiers of the application.

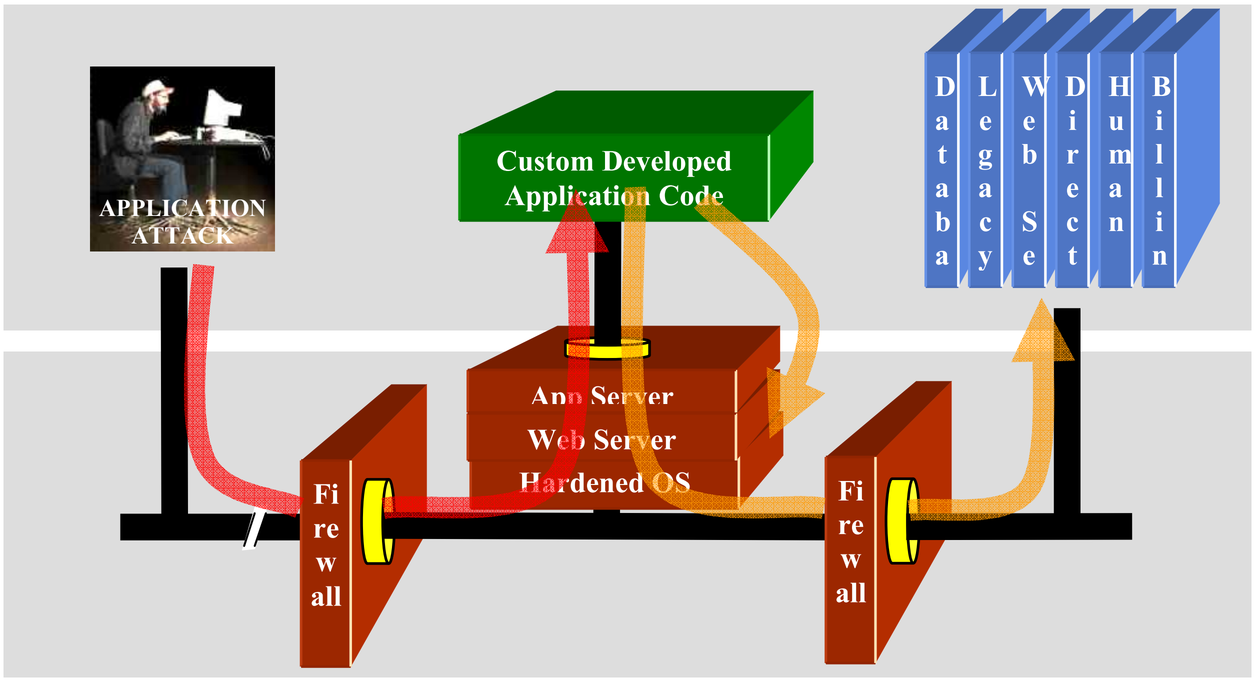


Figure 5: Flow of Application Attack on a Web Based Enterprise Application [7]

2.2.6.1 Input Vectors

Number	Type	Description	Protocol	Role
I1	URL query strings	Pass query string parameters	URL	
I2	Regular Form Fields	Pass information submitted in forms	HTML (GET)	
I3	Hidden Form Fields	Pass information submitted in forms	HTML (POST)	
I4	Java Server Pages	Dynamic HTML pages	HTML and Java	
I5	HTTPRequest	Communication between client/server	HTTP	
I6	Cookies	Used to retrieve user preferences, etc.		

2.2.6.2 Output Vectors

Number	Type	Description	Protocol
O1	Java Server Pages	Dynamic HTML pages	HTML and Java
O2	HTML	Static HTML content	HTML
O3	HTTPResponse	Communication between client/server	HTTP

2.2.7 Possible Threats (By Category)

2.2.7.1 Input Vector Threats

T1-<I1,I2,I3,I4,I5>

Description	SQL Injection
Type	Input Validation
Addressed?	No
Mitigation Techniques	Use input validation. Could use stored parameterized procedures instead of SQL maps. SQL maps are dangerous.
Chosen Mitigation	iBatis SQL Maps and Data Access Objects
Effect Graphs	Violation of data integrity. Loss of data confidentiality.
Risk	High

T2-<I1,I2,I3,I4,I5>

Description	Cross Site Scripting
Type	Input Validation
Addressed?	No
Mitigation Techniques	Use input validation (white lists). Use HTMLEncode and URLEncode methods. Problem exacerbated by reliance on URL strings.
Chosen Mitigation	None
Effect Graphs	Execution of arbitrary code on the browsers of application users.
Risk	High

T3-<I1,I2,I3,I4,I5,I6>

Description	Canonicalization (similar encodings, many different ways to represent the same name, path, etc.)
Type	Input Validation
Addressed?	No
Mitigation Techniques	Encoding should be set properly in the Web config file using requestEncoding and responseEncoding

Chosen Mitigation	**Decisions in JpetStore are constantly made based on the input passed (primarily in URL strings).**
Effect Graphs	**Access to privileged resources, escalation of privilege, etc.**
Risk	**Moderate**

T4-<I1,I2,I3,I4,I5>

Description	**Buffer/Integer Overflow**
Type	**Input Validation**
Addressed?	**No**
Mitigation Techniques	**Validate the size of input. Use safe functions.**
Chosen Mitigation	**Not addressed**
Effect Graphs	**Integer overflows/Buffer overflows could results in execution of arbitrary code on the application server with privileges of the JPetStore application**
Risk	**High. Static code analysis confirmed some Integer Overflow problems in the code.**

T6-<I1,I2,I3,I4,I5>

Description	**Escalation of Privilege**
Type	**Authorization**
Addressed?	**No**
Mitigation Techniques	**Should always use least privilege processes where possible.**
Chosen Mitigation	**Public/private functions are used for security, this is not acceptable**
Effect Graphs	**Access to other user accounts**
Risk	**Medium**

T7-<I1,I2,I3,I4,I5>

Description	**Disclosure of Confidential (Sensitive) Data**
Type	**Authorization**
Addressed?	**No**
Mitigation Techniques	**Perform checks before allowing access to sensitive data. Use strong ACL. Use encryption for sensitive data in configuration file.**
Chosen Mitigation	**Exceptions are displayed to the user as they occur. This can give an attacker too much good information. Also, all configuration files are plaintext.**
Effect Graphs	**Loss of confidentiality. Can also lead to elevation of privilege.**
Risk	**High**

T8-<I1, I2, I3, I4, I5>

Description	Unauthorized Access to Configuration Stores (Web.xml/Struts-config.xml)
Type	Configuration Management
Addressed?	No. All config files are plaintext. No input validation. Used excessively in BeanAction.
Mitigation Techniques	Use ACLs, keep config files outside the Web space.
Chosen Mitigation	None
Effect Graphs	Elevation of privilege, command injection, etc.
Risk	High

T9-<I1, I2, I3, I4, I5>

Description	Retrieval of cleartext Configuration Data
Type	Configuration Management/Secure Storage
Addressed?	No
Mitigation Techniques	
Chosen Mitigation	None
Effect Graphs	
Risk	High

T10-<I1, I2, I3, I4, I5, I6>

Description	Query string manipulation; form field manipulation; cookie manipulation; HTTP header manipulation
Type	Parameter Manipulation
Addressed?	
Mitigation Techniques	Input validation is very important. Never trust any user input. Encrypt query strings. Use Post not Get, avoid query strings that hold sensitive info (especially next action to perform), use data in secure (encrypted) session rather than query strings, etc.
Chosen Mitigation	The architecture of JPetStore makes it more susceptible to this problem than a usual J2EE application
Effect Graphs	
Risk	High

T11-<I1, I2, I3, I4, I5, I6>

Description	Information Disclosure
Type	Exception Management
Addressed?	No
Mitigation Techniques	Don't provide raw exception output to the user.

Chosen Mitigation	
Effect Graphs	**Information disclosure. Facilitates attack.**
Risk	**Medium**

T12-<>

Description	**Denial of Service (attackers can try to crash App server)**
Type	**Exception Management**
Addressed?	**Not everywhere**
Mitigation Techniques	**Always properly handle exceptions.**
Chosen Mitigation	
Effect Graphs	
Risk	**Medium**

T12-<I1, I2, I3, I4, I5>

Description	**Command Injection**
Type	**Input Validation**
Addressed?	**Problem introduced by the experimental BeanAction approach. Unvalidated URL strings can be used to invoke an action for which the user was not authorized and possibly inject arbitrary commands to be executed by the application server.**
Mitigation Techniques	
Chosen Mitigation	**None. Problem introduced by chosen implementation of the Struts MVC architecture.**
Effect Graphs	**This is a very bad problem that could lead to compromise of the whole system through execution of arbitrary system commands that are injected in Java**
Risk	**Very high**

2.2.7.2 Output Vector Threats

Disclosure of Sensitive Information

O1-<I1,I2,I3,I4,I5>

Description	**Cross Site Scripting**
Type	**Input Validation**
Addressed?	**No**
Mitigation Techniques	**Use input validation (white lists). Use HTMLEncode and URLEncode methods. Problem exacerbated by reliance on URL strings.**
Chosen Mitigation	**None**

Effect Graphs	**Execution of arbitrary code on the browsers of application users.**
Risk	**High**

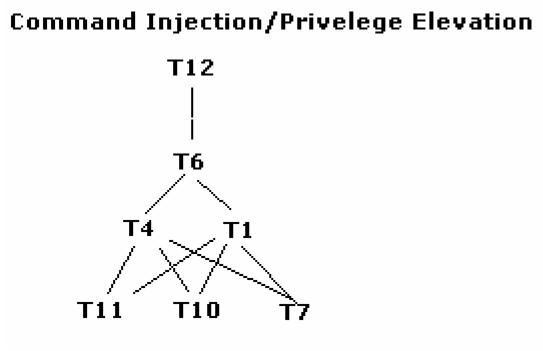
O2-<I1,I2,I3,I4,I5>

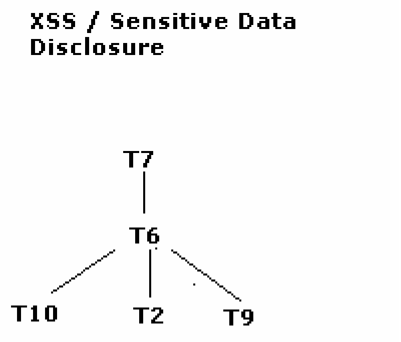
Description	**Disclosure of Confidential (Sensitive) Data**
Type	**Authorization**
Addressed?	**No**
Mitigation Techniques	**Perform checks before allowing access to sensitive data. Use strong ACL. Use encryption for sensitive data in configuration file.**
Chosen Mitigation	**Exceptions are displayed to the user as they occur. This can give an attacker too much good information. Also, all configuration files are plaintext.**
Effect Graphs	**Loss of confidentiality. Can also lead to elevation of privilege.**
Risk	**High**

O3-<I1, I2, I3, I4, I5, I6>

Description	**Information Disclosure**
Type	**Exception Management**
Addressed?	**No**
Mitigation Techniques	**Don't provide raw exception output to the user.**
Chosen Mitigation	
Effect Graphs	**Information disclosure. Facilitates attack.**
Risk	**Medium**

2.2.8 Threat Trees





2.2.9 Conclusions

This section introduced threat modeling by considering security of the JPetStore 4.0 at the architectural, design, as well as code level. The main issues identified are poor input validation and unconventional use of BeanAction for the controller portion of the Struts MVC. Poor input validation can lead to SQL injection, Cross Site Scripting and Canonilization attacks. Additionally, results of automated source code analysis confirmed that the application is susceptible to integer overflows and exception management problems. Finally, the way that MVC is implemented in this application can lead to escalation of privilege and possible command injections.

2.3 Microsoft's Approach to Threat Modeling: STRIDE and DREAD

Section 2.1 outlined the rationale and some of the steps involved in threat modeling. Section 2.2 offered the reader an example threat model for an open source enterprise web based application called JPetStore 4.0 written by Clinton Begin. While previous example focused on the J2EE world, we focus here on the .NET world. This section offers the reader Microsoft's approach to threat modeling and discusses STRIDE and DREAD. STRIDE (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege) is a technique for categorizing application threats each of which has corresponding countermeasures. DREAD (Damage potential, Reproducibility, Exploitability, Affected users, and Discoverability) model is used by Microsoft as a way to help rate various security threats on the application by calculating risk associated with each. That can often be more meaningful than trying to assign severities to the various threats since severities are harder to agree on and are more arbitrary. The idea is to mitigate the security threats that have the greatest impact on the application first. Threat modeling is a fairly new technique and Microsoft's approach to threat modeling is most mature, comprehensive and well documented. Most importantly, those techniques are followed at Microsoft in all of their product development. The remainder of this section presents a detailed overview of the steps in Microsoft's threat modeling process.

Threat modeling provides a systematic methodology to identify and rate the threats that are likely to affect the system. Threat modeling can be done for host and network security as well, but the focus in this section is on the application itself. "By identifying and rating threats based on a solid understanding of the architecture and implementation of your application, you can address threats with appropriate countermeasures in a logical order, starting with the threats that present the greatest risk" [10]. Threat modeling is a more effective alternative (both in terms of results and cost) to indiscriminant application of security features without a thorough understanding of what threats each feature is supposed to address. Additionally, with random approach to security it is impossible to know when the application has attained an acceptable level of security and what areas of the application are still vulnerable. As we mentioned in section 2.1, it is impossible to secure the system before knowing the threats.

Prior to initiating a detailed discussion of threat modeling it is important to understand some key terminology. In the context of threat modeling, an asset is a resource of value, such as the data in the database or in the file system (system resource). There are a variety of assets in an enterprise web based application, including configuration files, encryption/decryption keys, various security related tokens, databases, etc. There are also various file system resources, such as registry, log audits, etc. A threat is a possible occurrence, either malicious or accidental, that might damage or otherwise compromise an asset. A vulnerability is an exploitable weakness in some area of the system that makes a threat possible. Vulnerabilities can occur at the network, host or application levels. An attack, otherwise known as an exploit, is a realization of a threat in which an action is taken by an attacker or some executable code to exploit a vulnerability and compromise an asset. A countermeasure is a built in protection that addresses a threat and mitigates the risk.

Meier offers a useful analogy that helps understand the relation between assets, threats, vulnerabilities, attacks and countermeasures: "Consider a simple house analogy: an item of jewelry in a house is an asset and a burglar is an attacker. A door is a feature of the house and an open door represents a vulnerability. The burglar can exploit the open door to gain access to the house and steal the jewelry. In other words, the attacker exploits a vulnerability to gain access to an asset. The appropriate countermeasure in this case is to close and lock the door" [10]. It is important to remember that a threat model is a living thing and should evolve as time goes by. It only makes sense since threats evolve and new security threats are introduced to which countermeasures must be built into the application. Throughout this process, having a document in place that identifies known security threats and how they have been mitigated provides control over the security of the application.

Threat modeling is not a one time only process. Instead it should be an iterative process that starts during the early stages of the software development lifecycle of the application and continues throughout the rest of the application lifecycle. The reasons for this are that it is impossible to identify all of the security threats on the first pass and the fact that evolution in business requirements will demand reevaluation of threats and countermeasures. There are six steps involved in the threat modeling process, as demonstrated in the figure below. These steps are identification of assets, creation of architectural overview, decomposition of the application, identification of threats, documentation of threats, and rating of the threats.

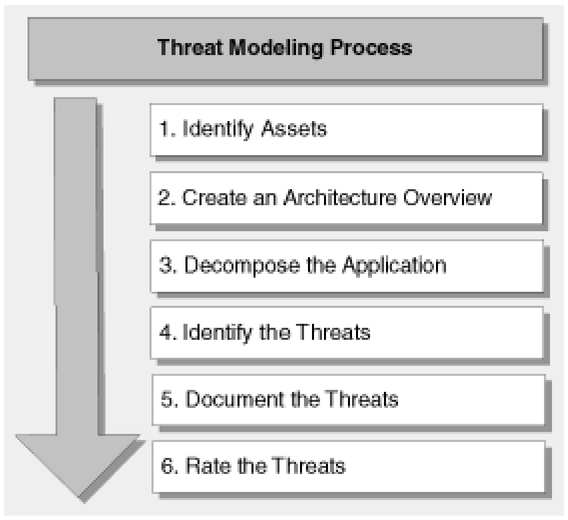


Figure 6: Threat Modeling Process Overview [10]

2.3.1 Identify Assets

Identify the assets that need to be protected. Some assets include configuration files, encryption/decryption keys, various security related tokens, databases, etc. There are also various file system resources, such as registry, log audits, etc. All web pages in the enterprise web based application and web site availability should also be considered assets.

2.3.2 Create an Architecture Overview

At this stage the function of the application is documented, including architecture and physical deployment configuration. It is also important to document all of the technologies used by the application since there may be security threats specific to the set of technologies being utilized. The goal is to look at potential vulnerabilities at the architecture, design or implementation of the application that should be avoided. At this stage, three main tasks are performed: identification of what the application does (functionality), creation of an architecture diagram and identification of technologies.

Identification of what the application does essentially involves the understanding of the business rules of the application and the data flow to and from the various assets. It is helpful at this stage to document the various use cases of the application to put the application functionality in context. It is also important to consider any misuse of business rules and consider what mechanisms exist to protect against such misuse.

The next step is then to create a high-level diagram describing the composition and structure of the application and its subsystems as well as its physical deployment characteristics. Additional supplemental diagrams might be necessary that drill down into the architecture at the middle-tier application server, integration with external systems, etc. A sample application architecture diagram is provided below.

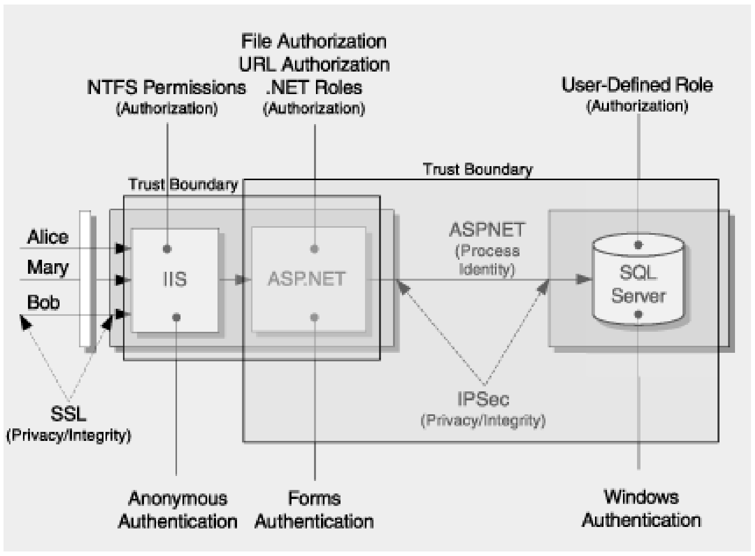


Figure 7: Sample Application Architecture Diagram [10]

As the diagram above shows, a sample application architecture diagram would contain information above the various trust boundaries. There are several trust boundaries illustrated in the diagram. The important thing to remember is that authentication needs to take place each time that data passes through trust boundaries. In the diagram the communication between the client and IIS server takes place over SSL that guarantees privacy and integrity, but not authenticity. Anonymous authentication needs to take place at the IIS server. Then forms authentication needs to take place again at the ASP.NET application server. Finally, windows authentication needs to take place at the database server tier. Each time a trust boundary is crossed re-authentication is required. Additionally, at each tier of the enterprise web application, different authorization needs to happen. In the case of .NET framework it is authorization on NTFS permissions, file/URL/.NET roles and user-defined roles authorization. Just like in the sample application architecture diagram above, it is essential to show the various technologies and security protocols used (SSL, IPSec, etc.). To start drawing the diagram, at first a rough diagram should be constructed that includes the structure of the application and its subsystems, as well as the deployment characteristics. The diagram can then be extended with details about the trust boundaries, authentication and authorization mechanisms. Some of those may not be come clear until the application is decomposed in the next step.

The final step in the creation of architecture overview is to identify the various technologies used in the implementation of the application. Some of the major frameworks for enterprise web applications include J2EE and .NET frameworks. Identifying specific technologies used in the solution helps focus on the technology specific threats that will be looked at later in the process. This analysis will also help determine the appropriate mitigation strategies. In the case of .NET, some of the likely technologies include ASP.NET, Web Services, Enterprise Services, Microsoft .NET Remoting, and ADO.NET. In the case of J2EE, some of the likely parallel technologies might be Entity Java Beans (EJB), Web Services, Java RMI, JDBC, among others.

Detailed discussion of the relevant J2EE and .NET technologies will be presented in chapters three and four. The table below demonstrates a way that implementation technologies can be documented in the threat modeling report.

Technology/Platform	Implementation Details
Microsoft SQL Server on Microsoft Windows Advanced Server 2000	Includes logins, database users, user defined database roles, tables, stored procedures, views, constraints, and triggers.
Microsoft .NET Framework	Used for Forms authentication.
Secure Sockets Layer (SSL)	Used to encrypt HTTP traffic.

Table 1: Documentation of Implementation Technologies [10]

2.3.3 Decompose the Application

In this step in Microsoft's threat modeling process an application is broken down in order to create a security profile for the application that is based on traditional vulnerability areas. In J2EE threat modeling example presented in section 2.2 the traditional vulnerability areas were drawn from the OWASP top ten, but in reality, the list should be far more extensive. In this stage trust boundaries, data flow, entry points, and privilege code is identified. Note that some of that information, such as information about trust boundaries will be needed to update the architecture diagram in the previous step. Gaining insight into the details of the architecture and design of the application will make it easier to analyze security threats. A diagram below shows the focus of the decomposition process.

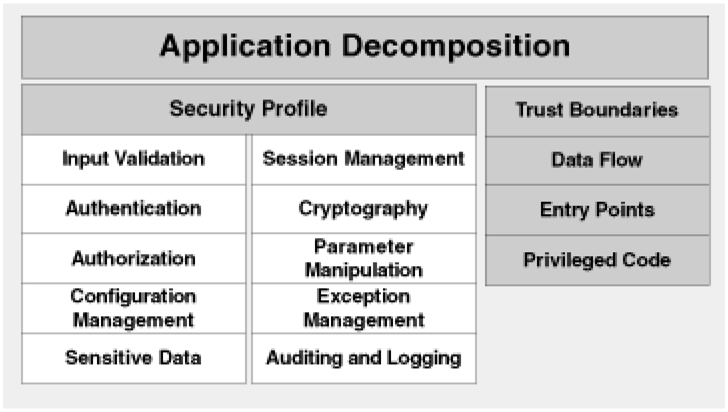


Figure 8: Focus of Application Decomposition Process [10]

The steps for application decomposition include identification of trust boundaries, data flow, entry points, privileged code and security profile documentation. Trust boundaries envelope the tangible assets of the application. The assets of the application would have already been identified in the first step of the threat modeling process. Many of the assets will be determined by the application design. Data flow must be analyzed across each of the subsystems and consideration must be given to the sources of data (i.e. whether the input is trusted or not). Input authentication and authorization must be considered across the trust boundaries. In addition to input, trustworthiness of any mobile code and remote object calls into the application must be evaluated, with consideration again given to authentication and authorization. Data flows across trust boundaries through the entry points. Appropriate gatekeepers must guard all entry points into trust boundaries and recipient entry point needs to validate all data passed across trust boundaries. In evaluation of trust boundaries, consideration must be given both from the code perspective and from the server trust relationship perspective. Code perspective should consider trust on the component/module level, while server trust perspective should focus on whether any given server relies on an upstream server to perform authentication, authorization and input validation. Trusted server relationship can be tricky and so architects and designers should be very careful to not extend too much trust. When in doubt, authentication, authorization and input validation should be performed over again.

The next step in the application decomposition is to identify the data flow. To accomplish this, a person performing threat modeling could start at the highest level and then iteratively decompose the application through analysis of the data flow between the individual subsystems. Particular attention should be paid to data flow across trust boundaries, since authentication, authorization and input validation might be required when that occurs. There are many formal design techniques that could aid with identification of data flow including data flow diagrams and sequence diagrams. Data flow diagrams provide a graphical representation of the data flows, data stores, and the relationships that exist between the data sources and destinations. Sequence diagrams show the chronological interaction between the various system objects and subsystems.

The next step is to identify the entry points to the application by isolating the gateways through which data flows into the application. Some of the data will come from user input, other will come from various client applications, yet additional data will come through integration channels with other enterprise applications. Some data might come from outside the corporate network and other might come from the inside. Regardless of the nature and origin of data, all data enters the application through a set of entry points. It is very important to be aware of all of the entry points since these will also serves as entry points for any attacks on the application. Both internal and external entry points should be well understood, including information on the kind of data that would pass through the entry point and the source of that data. For each of the entry points, the types of mechanisms that provide authentication, authorization and input validation services should be determined and documented. Some of the entry points in an enterprise web application might include user interfaces in web pages, service interfaces in web services, remote object invocation interfaces, message queues, as well as physical and platforms entry points like ports and sockets.

Privileged code is code in the enterprise web application that performs various system privileged operations that require access to secure resources such as DNS servers, directory services, file systems, event logs, environment variables, message queues, the registry, sockets, printers, web services, etc. Untrusted code should never be invoked by trusted code and input that is not trusted and has not been validated should never be passed to the trusted code routines. As we will discuss later in more detail, command injection attacks are made possible because the parameters passed to the trusted code routines come from input that has not be properly validated. The best strategy is to only use constants or trusted data for access to trusted code routines. As part of the threat model, all instances where trusted code is used should be documented along with the parameters passed to the trusted code and any code that might be invoked by trusted code.

Final and perhaps most important step of the application decomposition phase of threat modeling involves documentation of the security profile. In this step architecture, design and implementation approaches are documented in terms of their ability to provide input validation, authentication, authorization, configuration management, auditing and logging, session management, cryptography, among others. The idea is to focus on the common vulnerability areas and ask the right questions to determine whether the common threats within those vulnerability areas are mitigated by the current architecture, design and implementation approaches. A sample table demonstrates the various questions that can be asked across each of the vulnerability categories to facilitate creation and documentation of the security profile. You will notice that the vulnerability categories in this table correspond with the OWASP top ten.

Category	**Considerations**
Input validation	Is all input data validated? Could an attacker inject commands or malicious data into the application? Is data validated as it is passed between separate trust boundaries (by the recipient entry point)? Can data in the database be trusted?
Authentication	Are credentials secured if they are passed over the network? Are strong account policies used? Are strong passwords enforced? Are you using certificates? Are password verifiers (using one-way hashes) used for user passwords?
Authorization	What gatekeepers are used at the entry points of the application? How is authorization enforced at the database? Is a defense in depth strategy used? Do you fail securely and only allow access upon successful confirmation of credentials?

Configuration management	What administration interfaces does the application support? How are they secured? How is remote administration secured? What configuration stores are used and how are they secured?
Sensitive data	What sensitive data is handled by the application? How is it secured over the network and in persistent stores? What type of encryption is used and how are encryption keys secured?
Session management	How are session cookies generated? How are they secured to prevent session hijacking? How is persistent session state secured? How is session state secured as it crosses the network? How does the application authenticate with the session store? Are credentials passed over the wire and are they maintained by the application? If so, how are they secured?
Cryptography	What algorithms and cryptographic techniques are used? How long are encryption keys and how are they secured? Does the application put its own encryption into action? How often are keys recycled?
Parameter manipulation	Does the application detect tampered parameters? Does it validate all parameters in form fields, view state, cookie data, and HTTP headers?
Exception management	How does the application handle error conditions? Are exceptions ever allowed to propagate back to the client? Are generic error messages that do not contain exploitable information used?
Auditing and logging	Does your application audit activity across all tiers on all servers?

	How are log files secured?

Table 2: Building a Security Profile [10, 24]

2.3.4 Identify the Threats

At this point of the threat modeling process we have identified the application assets, created an architecture overview and decomposed the application. It is now time to identify the threats on the enterprise web application. Those threats will have the potential of compromising the assets and affecting the application. While there are a variety of common threats that falls into various vulnerability categories, and those can be easily documented, there are others that may not be as straightforward to identify. Some might be very specific to the way a particular web application is architected. For identifying those threats, a technique that works the best is to get the architects, designers, developers, testers, security professionals, system administrators and other personnel for a brain storming session in front of the whiteboard. The team should be carefully selected and should include top people from each of the roles. At this point there are two techniques that can be applied. The first is taking a list of common threats grouped by network, host and application categories and applying each of those to the architecture of the application under evaluation. Special focus should be given to any of the problem areas identified earlier in the threat modeling process. Some threats will be easy to rule out if they do not apply. The second technique and the one that we focus on in this discussion is STRIDE. As previously mentioned, STRIDE considers the broad categories of threats, such as spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privilege. Those areas correspond to the ways that an attacker might compromise the application and STRIDE helps the threat modeler to systematically identify those threats. This is a goals based approach where the goals of an attacker are considered and the right questions are asked with regards to the architecture and design of the application as they relate to the threat categories identified by STRIDE.

For each of the threat categories identified by STRIDE there are corresponding countermeasures that can be used to mitigate the risk. Going through this part of the threat modeling process, two questions should be asked for which the answered need to be documented. First question is: Does this threat apply to my application? Second question is: Is this threat mitigated? If so, which mitigating technique (countermeasure) was chosen and why? Oftentimes a particular countermeasure chosen will depend on the attack itself. A table below lists some specific threats and countermeasures across the STRIDE threat categories.

Threat	**Countermeasures**
Spoofing user identity	Use strong authentication.

	Do not store secrets (for example, passwords) in plaintext. Do not pass credentials in plaintext over the wire. Protect authentication cookies with Secure Sockets Layer (SSL).
Tampering with data	Use data hashing and signing. Use digital signatures. Use strong authorization. Use tamper-resistant protocols across communication links. Secure communication links with protocols that provide message integrity.
Repudiation	Create secure audit trails. Use digital signatures.
Information disclosure	Use strong authorization. Use strong encryption. Secure communication links with protocols that provide message confidentiality. Do not store secrets (for example, passwords) in plaintext.
Denial of service	Use resource and bandwidth throttling techniques. Validate and filter input.
Elevation of privilege	Follow the principle of least privilege and use least privileged service accounts to run processes and access resources.

Table 3: STRIDE Threats and Countermeasures [10]

The next step is to construct attack trees and attack patterns involving the identified threats. There are two reasons for doing this. The first reason is that attack trees and patterns focus around attack scenarios that can be pursued by an attacker. Each of the attack scenarios comprises the various security threats that were identified. The second reason for doing this is that thinking about attack scenarios may help a threat modeler unveil additional threat that might have been previously overlooked. While there are different definitions for attack trees and attack patterns, Microsoft defines these terms as follows:

"An attack tree is a way of collecting and documenting the potential attacks on your system in a structured and hierarchical manner. The tree structure gives you a descriptive breakdown of various attacks that the attacker uses to compromise the system. By creating attack trees, you create a reusable representation of security issues that helps focus efforts. Your test team can create test plans to validate security design. Developers can make tradeoffs during implementation and architects or developer leads can evaluate the security cost of alternative approaches.

Attack patterns are a formalized approach to capturing attack information in your enterprise. These patterns can help you identify common attack techniques" [MS Paper].

A good way to go about creating attack trees is to identify goals and sub-goals of an attack as well as other actions necessary for a successful attack. An attack tree can be represented as a hierarchical diagram or as an outline. The desired end result of the whole exercise is to have something that portrays an attack profile of the application. This will help in evaluation if likely security risks and guide the choice of mitigations. At this stage, flaws in architecture or design might become apparent and would need to be rectified. Attack trees can be started by first identifying the root nodes, representing the ultimate goals of an attacker, and working towards the leaf nodes, representing the techniques and methodologies used by an attacker to achieve the goals. Each of the leaf nodes might be a separate individual security threat that was previously identified. A diagram representing a sample attack tree is offered below.

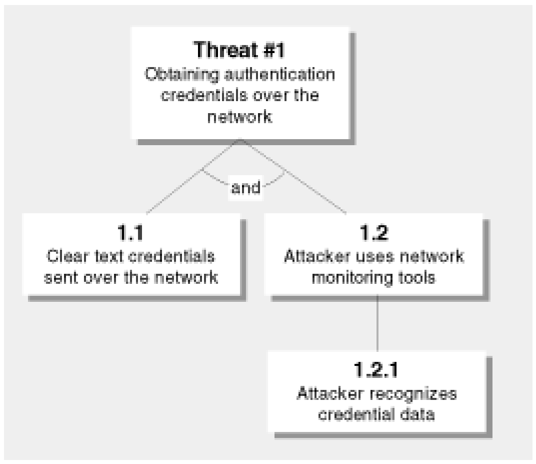

Threat #1
Obtaining authentication credentials over the network
and
1.1
Clear text credentials sent over the network
1.2
Attacker uses network monitoring tools
1.2.1
Attacker recognizes credential data

Figure 9: Sample Attack Tree (Tree Representation) [10]

An approach to documenting an attack tree that takes less space and less time to create is the outline representation. The outline representation of a sample attack tree is shown below.

Threat #1 Attacker obtains authentication credentials by monitoring the network

1.1 Clear text credentials sent over the network **AND**

1.2 Attacker uses network-monitoring tools

1.2.1 Attacker recognizes credential data

Figure 10: Sample Attack Tree (Outline Representation) [10]

Attack patterns provide a way of documenting generic attacks that may occur in many different contexts. The pattern identifies the goal of the attack, preconditions, and the steps to perform the attack. Unlike STRIDE, the focus represented in attack patterns is on attack techniques, and not the goals of the attacker. An attack tree below illustrates a generic attack pattern for injection attacks. Command and SQL injection attacks are specific instances of the generic injection attack.

Pattern	**Code injection attacks**
Attack goals	Command or code execution
Required conditions	Weak input validation Code from the attacker has sufficient privileges on the server.
Attack technique	1.Identify program on target system with an input validation vulnerability. 2.Create code to inject and run using the security context of the target application. 3.Construct input value to insert code into the address space of the target application and force a stack corruption that causes application execution to jump to the injected code.
Attack results	Code from the attacker runs and performs malicious action.

Table 4: Pattern for a Generic Code Injection Attack [10]

2.3.5 Document the Threats

The next step in Microsoft's threat modeling process is to document all of the threats that were identified. Documentation also should have been performed throughout all of the previous stages of the process. In order to effectively document the identified threats, a template should be create that encapsulates various relevant threat attributes, such as target, risk, attack techniques and countermeasures. Risk attribute can be left blank for now. We will discuss how to determine risk in the following section when we rate the threats. A sample template for documenting the threats is presented below.

Threat 1

Threat Description	**Attacker obtains authentication credentials by monitoring the network**
Threat target	Web application user authentication process
Risk	
Attack techniques	Use of network monitoring software
Countermeasures	Use SSL to provide encrypted channel

Threat 2

Threat Description	**Injection of SQL commands**

Threat target	Data access component
Risk	
Attack techniques	Attacker appends SQL commands to user name, which is used to form a SQL query
Countermeasures	Use a regular expression to validate the user name, and use a stored procedure that uses parameters to access the database.

Table 5: Documenting the Threats [10]

2.3.6 Rate the Threats

The final step in the threat modeling process is to rate all of the threats that were identified. That is done by evaluating potential impact of each of the threats on the system. The purpose of this exercise is to help prioritize the threats. It may be unrealistic to expect that under the pressures of a typical software development schedule all of the threats will be mitigated. This may be impossible due to time and money constraints. After all, functionality has to come first. However, having a good way to rate the threats based on the greatest security impact on the application as a whole will help make inform decisions as to what threats must be addressed first. The formula for calculating risk is: **RISK = PROBABILITY * DAMAGE POTENTIAL**. We can now fill in the risk attribute in the template documenting the attributes of each of the threats that we left blank in the previous section. 1-10 scale can be used to represent probability, with higher number meaning higher probability. Same scale can be applied for damage potential.

In order to prioritize the threats, high, medium and low ratings can be used. Threats rated as high pose a significant risk to the application and should be addressed as soon as possible. Medium threats need to be addressed, but are less urgent than high threats. Low threats should only be addressed if the schedule and cost of the project allows. Microsoft has also developed a more sophisticated rating system called DREAD that makes the impact of the security threat more explicit. Adding additional dimensions to consider makes it easier for a team performing threat modeling to agree on the rating. DREAD model is used to calculate risk at Microsoft instead of the simplistic formula above. The following questions must be asked when using DREAD to arrive at the risk for a particular threat:

- Damage potential: How great is the damage if the vulnerability is exploited?
- Reproducibility: How easy is it to reproduce the attack?
- Exploitability: How easy is it to launch an attack?
- Affected users: As a rough percentage, how many users are affected?
- Discoverability: How easy is it to find the vulnerability?

DREAD questions can be extended to meet the particular needs of the application. There might be other dimensions of great importance to a particular application being evaluated. A sample rating table is shown below that can be useful when prioritizing threats.

	Rating	High (3)	Medium (2)	Low (1)
D	Damage potential	The attacker can subvert the security system; get full trust authorization; run as administrator; upload content.	Leaking sensitive information	Leaking trivial information
R	Reproducibility	The attack can be reproduced every time and does not require a timing window.	The attack can be reproduced, but only with a timing window and a particular race situation.	The attack is very difficult to reproduce, even with knowledge of the security hole.
E	Exploitability	A novice programmer could make the attack in a short time.	A skilled programmer could make the attack, then repeat the steps.	The attack requires an extremely skilled person and in-depth knowledge every time to exploit.
A	Affected users	All users, default configuration, key customers	Some users, non-default configuration	Very small percentage of users, obscure feature; affects anonymous users
D	Discoverability	Published information explains the attack. The vulnerability is found in the most commonly used feature and is very noticeable.	The vulnerability is in a seldom-used part of the product, and only a few users should come across it. It would take some thinking to see malicious use.	The bug is obscure, and it is unlikely that users will work out damage potential.

Table 6: Threat Rating Table [10]

After asking the above questions, values (1-3) should be counted for each threat. The result can fall in the range of 5–15. Then values 12-15 are treated as High risk, 8-11 as Medium and 5-7 as Low. Using the sample example we looked at earlier:

- Attacker obtains authentication credentials by monitoring the network.
- SQL commands injected into application.

Threat	D	R	E	A	D	Total	Rating
Attacker obtains authentication credentials by monitoring the network.	3	3	2	2	2	12	High
SQL commands injected into application.	3	3	3	3	2	14	High

Table 7: DREAD Rating [10]

We can now complete the template documenting this threat from the previous section.

Threat Description	**Attacker obtains authentication credentials by monitoring the network**
Threat target	Web application user authentication process
Risk rating	High
Attack techniques	Use of network monitoring software
Countermeasures	Use SSL to provide encrypted channel

2.3.7 Output

The output of the threat modeling process is a document that has multiple audiences. The audiences for this document are the various members of the software development team including the architects, designers, developers, testers and security auditors. The document should facilitate the various project team members understand the security threats on the application that need to be addressed and how they should be mitigated. The threat modeling output document consists of the complete architecture of the application and a list of threats coupled with their mitigations. Figure below defines the structure of the document resulting from the threat modeling process. Section 2.1 and sample threat model offered in 2.2 also outlined some additional parts that can be made part of the output document, such as documentation of assumptions, among others.

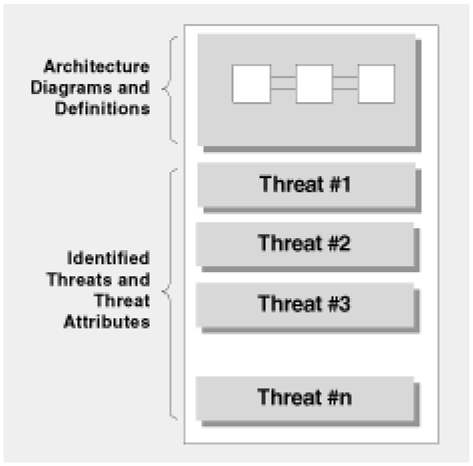

Architecture Diagrams and Definitions
Identified Threats and Threat Attributes
Threat #1
Threat #2
Threat #3
Threat #n

Figure 11: Various Pieces of the Threat Model [10]

Threat modeling process helps mitigate the risk of an attack by considering countermeasures to various threats. The threats themselves are not eliminated with threat modeling, only the risk is managed. Threats will exist despite of the security actions taken or countermeasures applied. The goal is to identify the threats to help manage the risk. Threat modeling provides an effective mechanism for managing and communicating security risks to all members of the software project team. It is also important to remember that threat model is built as a result of an iterative process and it will change over time since new times of threats and attack are discovered. Additionally, changing business requirements may require changes to the system which should prompt revisiting of the threat model.

3.0 J2EE Security

3.1 Architecture Overview

We now introduce an architecture overview for the component technologies of Java 2 Enterprise Edition (J2EE) used in a typical enterprise web application. The discussion is focused on the technologies used across the various tiers, namely the client tier, presentation tier, business logic tier, and the database tier. The discussion in this section presents the architectural overview of some of the J2EE technologies that will be instrumental for further discussion the support for core security services.

The first thing to understand about J2EE which plays a very important from a security standpoint is that J2EE is a specification for an enterprise platform and services. A variety of vendors implement the various parts of the J2EE specification. For instance, Jakarta Tomcat implements the J2EE web container specification and JBoss implements the EJB container specification. The point is that since a wide variety of vendors provide implementation to the J2EE specification, many application level vulnerabilities and compensating controls might in fact be specific to the independent software vendor (ISV) providing the implementation. For this reason, the discussion of security in the context of J2EE will be vendor specific to some extent. Additionally, while J2EE specification mandates certain solutions to address the core security services, the implementation of these solutions will be vendor specific. While we try to keep the discussion generic whenever possible, when implementation level detail needs to be considered, we will be referring to the BEA Weblogic implementation of the J2EE application server and the associated services.

Sun Microsystems offers the following description of the J2EE platform: “The J2EE platform represents a single standard for implementing and deploying enterprise applications. The J2EE platform has been designed through an open process, engaging a range of enterprise computing vendors, to ensure that it meets the widest possible range of enterprise application requirements. As a result, the J2EE platform addresses the core issues that impede organizations' efforts to maintain a competitive pace in the information economy” [23].

J2EE provides a distributed multi-tier application model where individual components can be distributed across different machines and heterogeneous platforms. The client tier of the J2EE enterprise web application is comprised of thin clients (user browsers) that can be located inside or outside the corporate firewall. A web application can also have other types of clients at the client tier, but we focus the discussion in this paper on thin clients. The application server resides on the middle tier of the J2EE enterprise web application and is compromised of the Web and EJB containers that provide services to the client tier. Web container handles the presentation layer of the application via Servlet and JSP technologies. An EJB container provides lifecycle management support for transaction aware business components that implement all of the business logic for the application. In enterprise deployments those two containers would rarely be on the same machine. In fact, the components within each of these containers are likely to be distributed across multiple machines if for no other reason than for fault tolerance and load balancing purposes. The Enterprise Information Systems (EIS) tier is the database or persistence tier of the J2EE based enterprise web application abstracting standardized access via API to variety of data stores. Figure below summarizes a set of technologies used across each of the tiers of the J2EE application.

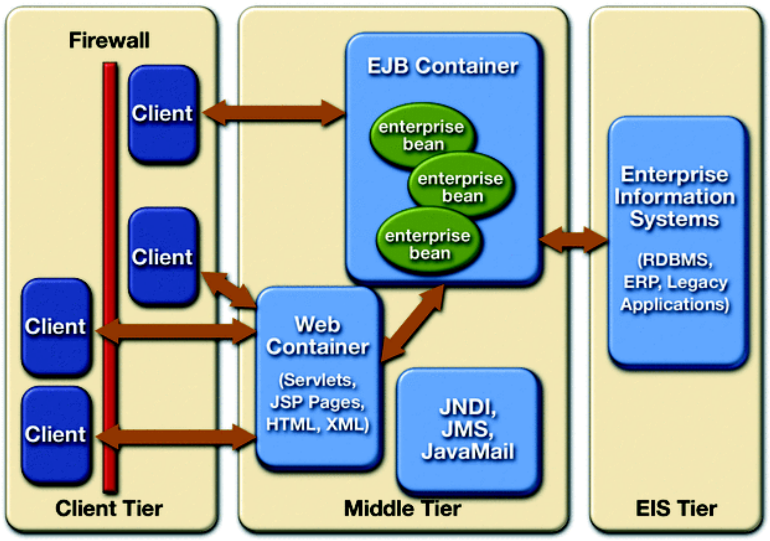


Figure 12: Overview of J2EE Web Application Architecture [23]

The important point to understand about J2EE containers is that they are standardized runtime environments which provide a variety of services to the components that are deployed inside them. Regardless of the vendor implementing the J2EE container, developers writing J2EE components can always rely on certain services to be provided. J2EE application server developers may and usually do choose to make additional features and services available to component developers, but component developers are always guaranteed to have available the set of services mandated by the J2EE specification. Thus if component developers use only vendor independent features available to them by the J2EE containers, the constructed components are guaranteed to work on different implementations of the J2EE application server. Some common services that J2EE containers provide to components include APIs that provide access to database (JDBC), transaction (JTA), naming and directory (JNDI), and messaging services (JMS). The J2EE platform also provides technologies that enable communication between collaborating objects hosted by different servers (RMI). Among other services, security related services are provided by the J2EE containers for the component developers to use. Some of those services are mandated by the J2EE specification and others are application server vendor specific. Figure below presents yet another view of the J2EE architecture with specific emphasis on the various kinds of supported services and the different types of application clients.

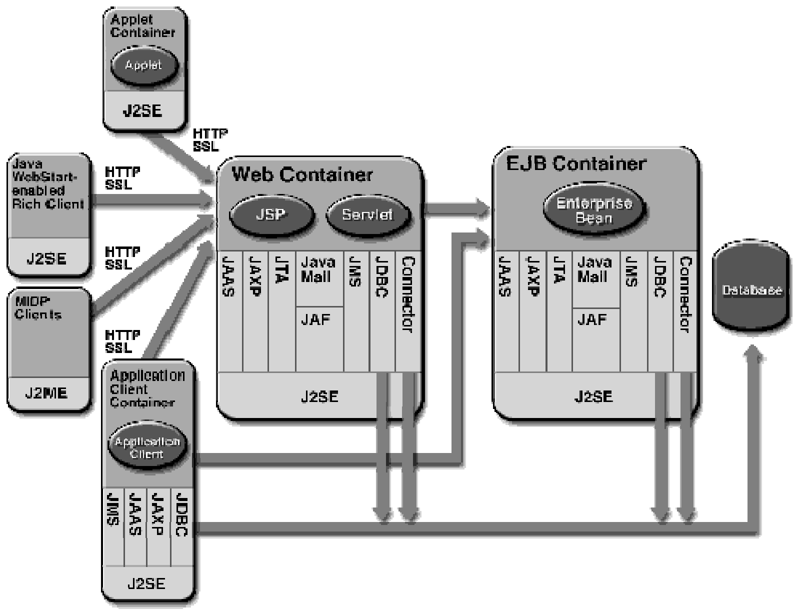


Figure 13: J2EE Components, Containers and Services [23]

The diagram below illustrates the multi-tiered deployment model for a web based enterprise J2EE application. You will notice the each of the tiers lists an associated set of technologies. Client tier can display HTML and XML pages and communicates with the web tier via HTTP or HTTPS. The web tier uses servlet and JSP technologies for presentation layer. Business logic tier (EJB tier) hosts all of the business logic for the application and may make use of container provided services such as the Java Messaging Service (JMS), Java Transaction API (JTA), Java Database Connectivity (JDBC), etc. The EIS tier hosts the databases for the enterprise web application. J2EE security discussion in this paper will focus on this kind of four-tiered deployment model of the web based enterprise application built with J2EE.

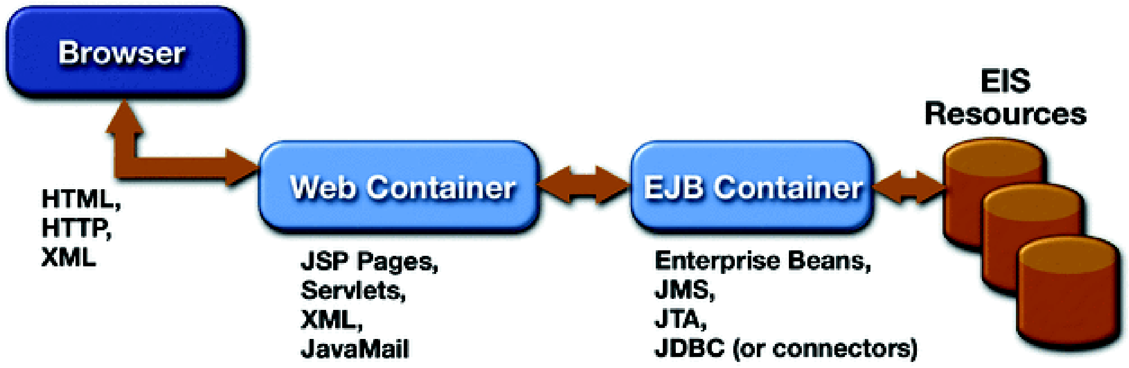


Figure 14: J2EE Four-tiered Deployment Model [23]

It is also important to mention that the data flow in a typical J2EE enterprise web application follows the Model View Controller (MVC) architecture. This architecture attempts to uncouple the business functionality, control logic and presentation code. A model represents application data and business logic dictating access and changes to the data. A controller defines application behavior by ensuring that user actions result in proper model changes. A view is responsible for presenting the content of a model to the user. Understanding of the MVC is very important from the security perspective. Figure below illustrates the inner workings of MVC.

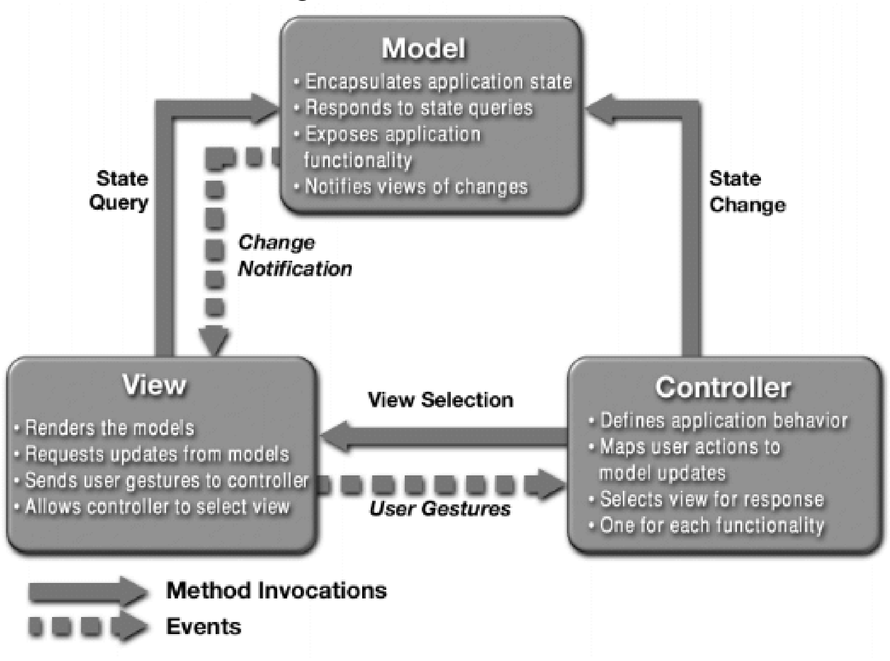

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

Figure 15: J2EE Model View Controller Architecture [23]

Much can be said about J2EE, but the goal of this section was only to provide high level architectural information required for subsequent discussion. A variety of additional information will be provided on J2EE as it becomes necessary. We now move on to look at the way that the J2EE specification addresses the core security services.

3.2 Support for Core Security Services

The discussion in this section focuses on the ways that confidentiality, integrity, authentication, authorization, availability, non-repudiation and accountability are addressed by the J2EE security model. There are two security methodologies provided by J2EE containers, namely declarative and programmatic. Component developer specifies declarative security in the XML deployment descriptor for the component, while programmatic security is specified from within the code (Java) for Web and EJB container components via the security APIs provided by J2EE. We are going to take a look into the security APIs provided by J2EE and see how those can be invoked to support the core security services. For the purposes of this paper, both declarative and programmatic security methodologies fall within the application security category.

A concept of a principal is used for authentication in J2EE. Once an entity has been authenticated by providing an authentication token (such as login/password) it becomes a principal. A principal is an authenticated entity that has certain identity attributes associated with it. Authorization (access control) in J2EE is driven by comparing the principal's identity attributes to the access control rules of the resource. The rights of the principal to access a resource are determined based on the permissions associated with the resource which are defined as part of the application's security policy domain (declarative security). It is important to remember that authentication and authorization are not a one time proposition. As we discussed in chapter two, these services should be invoked each time data crosses a trust boundary at the appropriate entry points.

An application component developer can specify the means by which an entity is authenticated in the system. Authentication to the Web container can take place via authentication data provided by the Web client by means of HTTP basic authentication, digest authentication, form-based authentication, or certificate authentication. Basic authentication involves using login and password combination. Digest authentication is similar to basic encryption, but applies a message digest algorithm on the password prior to transmitting it versus base64 encoding used to transfer the password in basic encryption, which is far more secure and is preferable. Form-based authentication uses a secure form provided by the Web container for authentication in order to log in. Finally, certificate authentication uses PKI techniques, where the client uses a public key certificate to establish its identity and maintains its own security context. No way exists to authenticate to the EJB container in J2EE. The J2EE trust model dictates that authentication would have already taken place by the time an action in an EJB component is invoked on behalf of the principal. However, if an enterprise bean needs to access an external resource, authentication will need to happen again since a trust boundary will have to be crossed. In that case, an enterprise bean can provide authentication data to the external resource directly (programmer handles it) or it can request that the container does this instead (container handles it). In the latter case, authentication data to resource factory reference mapping for the enterprise bean must be specified in the deployment descriptor and that authentication data will be passed to the external resource when connection is obtained by the container. Consequently, in the former case, the passing of authentication data to external resource is programmatic and in the latter case it is declarative.

J2EE platform authorization is based on the concept of security roles where security role is defined as a logical grouping of users defined by an application component provider or assembler. In the deployment environment, a security role is mapped to a principal by the deployer of the application. A security role can use both declarative and/or programmatic security. An application component provider can control access to an enterprise bean's methods in the EJB container by specifying the `method-permission` element in the enterprise bean's deployment descriptor (declarative security). The `method-permission` element contains a list of methods that can be accessed by a given security role. If a principal is in a security role allowed access to a method, the principal may execute the method. That is to say that the capability of a particular security role that is specified in the `method-permission` element drives access control.

Similarly, a principal is allowed access to a Web component in the Web container only if the principal is in the appropriate security role. This access control is provided programmatically by the developer via `EJBContext.isCallerInRole` or `HttpServletRequest.isRemoteUserInRole` methods.

3.2.1 Authentication:

When calls are made from the various application components of the J2EE application, those components usually act on behalf of some user. That is the execution of some web or EJB component is prompted by some user action. In that case, the component is said to impersonate the user thus its identity might be related to that of the user. On the other hand, an application component may call another application component with an identity that is its own, unrelated to the user. The point is that in authentication among application components is sometimes associated with the security role and the security context.

There are often two phases to authentication in J2EE applications: establishment of authentication context and authentication of caller or calling identities. Authentication context is established in a service-independent fashion, utilizing knowledge of some secret. Authentication context encapsulates the identity and is able to provide proofs of identity, otherwise known as authenticators. One can think of the authentication context as an underlying authentication interface for authentication services that can be used after having been instantiated with a specific identity. The authentication context can now be used for authentication of caller or calling identities. Controlling access to the authentication context (i.e. the ability to authenticate an associated identity) forms the foundation for the authentication scheme. Some of the ways to control access to the authentication context are allowing the process to inherit access to the authentication context once the initial authentication is performed by the user, related or trusted components (such as those which are part of the same application) can be granted access to the authentication context once a component is authenticated, and the caller may delegate its authentication context to the called component in cases where the component being called is expected to impersonate the component that called it.

Communication some between some J2EE components may take place without requiring authentication. This may happen for instance when the two components are part of the same protection domain, meaning that they are known to trust each other. A single trust boundary envelops the J2EE protected domain. Special care needs to be taken to ascertain that trust is not being overextended when relying on protection domains. When communication occurs between two components that are members of the same protection domain, no authentication is required, thus no constraints are placed on the identity associated with the call. The caller component in this case may propagate the caller's identity to the called component or select an identity based on its knowledge of the authorization constraints that are in place in the called component. That also means that entities extending trust universally (e.g. some web components) should never belong to any protection domains. A J2EE container is responsible for providing the boundaries between external callers and components inside the container. Boundaries of protection domains do not necessarily align with container boundaries. That is two components that belong to different containers may be part of the same protected domain. While containers enforce the boundaries, various implementations may support protection domain boundaries that span containers. Containers are not necessarily required to host components belonging to different protection domains, but various container implementations may choose to provide support for that. Figure below illustrates a typical J2EE protection domain.

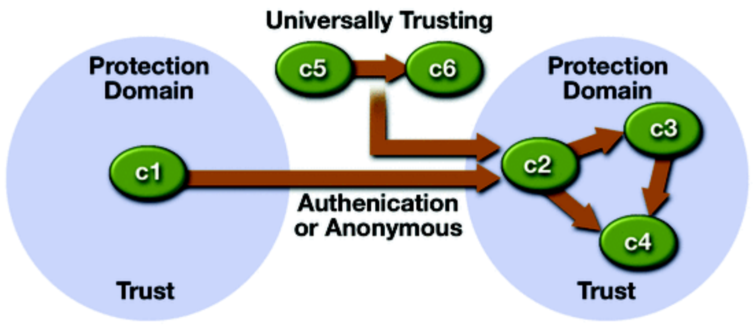


Figure 16: J2EE Protection Domain [23]

In component to component communication, in the case of inbound calls to the component, a container provides the called component with the credential of the calling component that allows the called component to get an authentic representation of the callers' identity. The credential can be a X.509 certificate, kerberos service ticket, etc. For outbound calls, the container of the calling component provides the credentials of that component so that they can be used by the container hosting the called component to establish the caller's identity as described above. The critical point is that J2EE containers provide bi-directional authentication capabilities that can be used to enforce the protection domain boundaries. Application developers need to understand how containers handle authentication and protection domains in order to make sure that trust is never overextended and where additional programmatic authentication is warranted. If it is unacceptable to rely on inter-container trust model, programmers may have to write authentication related code to overwrite the default behavior. If no valid identity proof is presented to the called component by the caller component, and the two components are not part of the same protection domain, the call should be rejected. Figure below illustrates J2EE authentication for two user scenarios: authenticated and unauthenticated users.

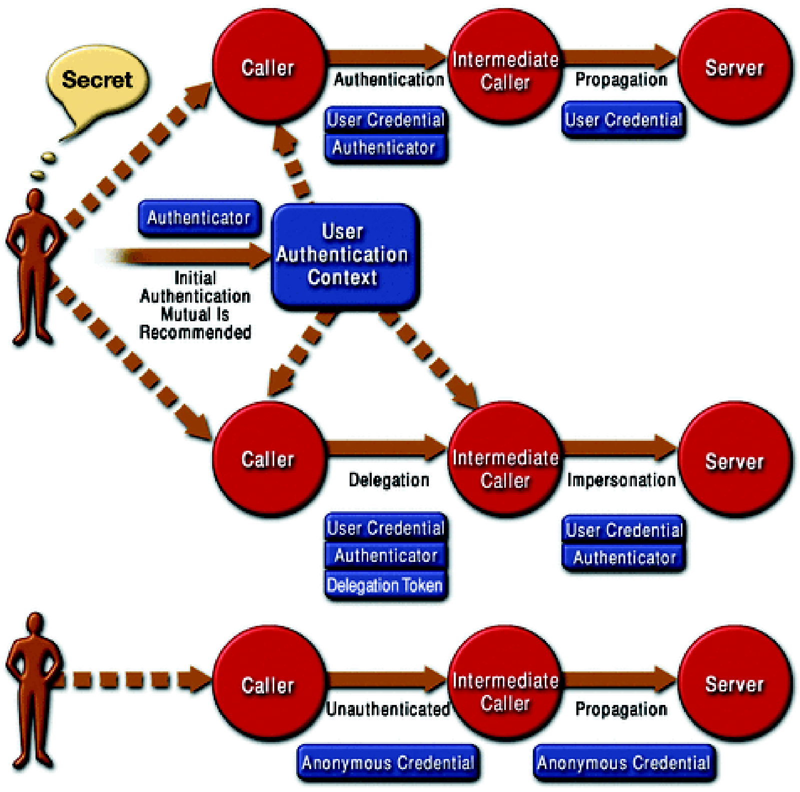


Figure 17: J2EE Authentication Scenarios [23]

As shown in the figure above, a calling component invoked by the user utilizes the user's authentication context to prove the user's identity to an intermediate component. If the identity is then propagated to the server and since no authentication tokens were provided, it will be accepted only if the two components reside in the same protection domain. On the other hand, if delegation rather than propagation occurs, access to authentication context is granted, allowing the intermediate caller to impersonate the user for all future calls. That means that propagation should only be used when all of the calls in the call chain originating from the user are part of the same protection domain. Delegation and impersonation should be used when authentication might be required over and over again. The figure also shows usage scenario for unauthenticated users utilizing anonymous credentials and propagation. A component that accepts anonymous credentials should not be part of any protection domain.

We already described earlier in this section some of the authentication mechanisms supported by J2EE Web containers. These include HTTP basic authentication, digest authentication, form-based authentication, or certificate authentication. The type of authentication to use can be specified declaratively via an XML deployment descriptor. Another type of authentication that makes use of certificate authentication is called mutual authentication. In mutual authentication, X.509 certificates are used by the client and the server to establish identity. SSL is used in mutual authentication to secure the communication channel. There is also support for hybrid authentication schemes involving HTTP basic authentication, form-based authentication, or HTTP digest authentication over SSL.

The type of authentication mechanism to use can be configured declaratively via the login-config element of the Web component deployment descriptor (Web.xml file). Below are a few examples of the ways that various authentication mechanisms can be specified in the Web.xml file.

HTTP Basic/Digest Authentication Configuration:

```
<web-app>
        <login-config>
                <auth-method>BASIC|DIGEST</auth-method>
                <realm-name>jpets</realm-name>
        </login-config>
</web-app>
```

Form-Based Authentication Configuration:

```
<web-app>
        <login-config>
                <auth-method>FORM</auth-method>
                <form-login-config>
                        <form-login-page>login.jsp</form-login-page>
                        <form-error-page>error.jsp</form-error-page>
                </form-login-config>
        </login-config>
</web-app>
```

Client Certificate Authentication Configuration:

```
<web-app>
        <login-config>
                <auth-method>CLIENT-CERT</auth-method>
        </login-config>
</web-app>
```

In order to protect the confidentiality of authentication tokens transmitted across insecure channels, authentication configurations above can be coupled with SSL-protected session. For instance HTTP basic authentication can be configured to happen over SSL as shown below. Same can be done for form-based authentication.

```
<web-app>
        <security-constraint>
                ...
                <user-data-constraint>
                        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
                </user-data-constraint>
        </security-constraint>
</web-app>
```

Setting the value of <transport-guarantee> tag to CONFIDENTIAL tells the J2EE Web container to use an SSL protected session for transmission of authentication tokens such as passwords.

Many deployments of enterprise web based applications built with J2EE do not provide separate authentication at the EJB tier components, and instead use protected domains that incorporate trusted Web components that are relied upon by the EJB components to have performed authentication. In this case, a protection domain spans J2EE containers, namely the Web and EJB containers. In that configuration a Web container is used to enforce protection domain boundaries for Web components and the enterprise beans called from those components. A diagram demonstrating this approach is shown below.

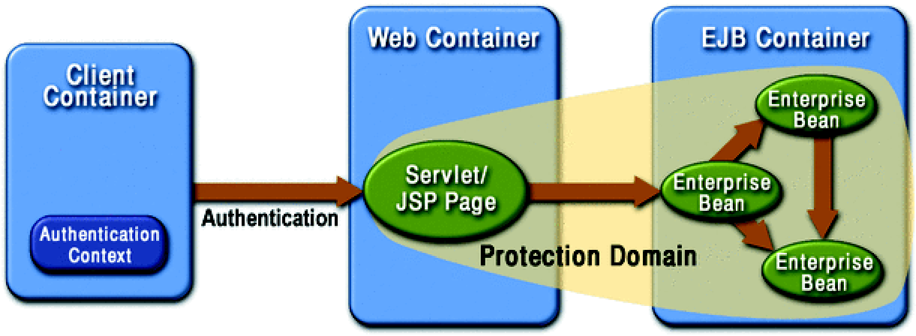


Figure 18: J2EE Protection Domains: Trusted Web Components [23]

The method for authentication in EJB components described above is less than ideal from a security standpoint. Having a protection domain that spans containers can be very tricky especially when some of the components in that protection domain are web components that may be externally accessible. Even if those components are trusted and coded properly (which is a big if), a compromise of one of the web components might make this trust misplaced if compartmentalization is not properly implemented. Additionally, web components will tend to be distributed across multiple machines, thus increasing a level of complexity. Additionally, the principle of defense in depth is violated in this scenario. In short, it would be far better if J2EE specification allowed for authentication to be performed at the EJB container level, because in general, data transferred from the middle tier to the business logic tier crosses a trust boundary and must thus be re-authenticated. At a minimum, unprotected Web components should never be able to call protected EJB components. Trusted Web components must be invoked first prior to calling protected EJB resources. Developers must make sure that protection mechanisms for access control cannot be circumvented when access to an EJB resource occurs in a Web component. Due to the J2EE specification, a lot of authentication in this case cannot be declarative and must be programmatic, introducing a greater risk factor.

As described earlier in this section, authentication at enterprise information systems layer is supported by J2EE via two primary mechanisms: container-managed resource manager sign-on and application-managed resource manager sign-on. In the former scheme, the calling container can be configured to manage the authentication to the resource for the calling component. This is the declarative approach. In the latter case, the components themselves manage caller identity and provide appropriate authenticators to the external resources at the EIS tier. This is the programmatic approach.

Container-managed resource manager sign-on uses a resource-ref element declared in the component's deployment descriptor that enumerates the resources used by the component. It is the responsibility of the component developer to provide this information in the deployment descriptor for all of the external resources being accessed. The sub-element res-auth specifies the type of sign on authentication. From inside the components, EJBContext.getCallerPrincipal and HttpServletRequest.getUserPrincipal methods can be invoked to obtain the identity of their caller. The component can then create a mapping between the caller identity to a new identity (or authentication secret) as required by a target resource being accessed at the EIS tier. The principal mapping is entirely performed by the container in container-managed resource manager sign-on scheme. In application-managed resource manager sign-on, the principal mapping is performed in the component itself, thus placing more responsibility on the programmer.

3.2.2 Authorization:

We already briefly discussed J2EE support for authorization services earlier in this section. Authorization, otherwise referred to as access control, is used to determine what resources should be accessible to authenticated users and what rights to should those users have while interacting with these resources. Only authentic caller identities are granted access to various J2EE components. Java 2 Standard Edition (J2SE) platform provides some mechanisms used to control access to code based on identity properties (e.g. the location and signer of the calling code). J2EE platform, which is built on top of J2SE, requires additional mechanisms for access control, where access to components must be limited based on who is using the calling code. As discussed in the authentication section, the caller may propagate the identity of its caller, select an arbitrary identity, or make an anonymous call.

A credential must be available to the called component that contains information describing the caller through its identity attributes. For anonymous calls, anonymous credentials are used. The access decision is made when comparing the caller's identity attributes with those required to access the called component. Like with authentication, a J2EE container serves as an authorization boundary between the components it contains and the callers to those components. An authorization boundary exists inside the authentication boundary of the container because it only makes sense to consider authorization after successful authentication. For all inbound calls, the security attributes in the caller's credentials must be compared with access control rules for the called component by the container. If the access control rules are satisfied, then the call is permitted. Otherwise, the call is rejected. Access control in J2EE is defined through permissions. For each resource, permissions enumerate who can perform various operations on the resource. Application deployer must configure a mapping between permission model of the application to the capabilities of the users in the operational environment. As with authentication, there are two types of authorization in J2EE: declarative and programmatic. We now describe both of these types.

The deployment descriptor of the application facilitates the mapping of application component capabilities to the capabilities of users in the corporate environment. This declarative approach to access control allows for container-enforced access control rules. The deployment descriptor defines security roles and associates them with components to define the privileges required to be granted permission to access components. "The deployment descriptor defines logical privileges called security roles and associates them with components to define the privileges required to be granted permission to access components. The deployer assigns these logical privileges to specific callers to establish the capabilities of users in the runtime environment. Callers are assigned logical privileges based on the values of their security attributes. For example, a deployer might map a security role to a security group in the operational environment such that any caller whose security attributes indicate that it is a member of the group would be assigned the privilege represented by the role. As another example, a deployer might map a security role to a list containing one or more principal identities in the operational environment such that a caller authenticated as one of these identities would be assigned the privilege represented by the role" [23].

At the EJB tier, access permission to a method is granted by the container only to callers that have at least one of the privileges associated with the method. At the Web tier, the Web container protects access to resources via security roles. For instance, a URL pattern and associated HTTP methods, such as GET, have associated permissions with them with which security roles must be compared. Web and EJB containers enforce authorization in similar ways. When a resource does not have an associated security role explicitly defined, access is granted to all who request the resource. In this declarative approach to authorization, the access control policy is specified at deployment time. The flexibility of this approach allows modifying the access control policy at any point. Privileges required to access components can be modified and refined. Additionally, the mapping between security attributes presenting by callers and container permissions can be defined. Those mappings are application specific and not container specific. That means that if a given container contains components from multiple applications, the privileges mapping may be different for each of the components, depending on what application it is associated with.

There may be situations when a finer grained access control may be required at the component level. For instance that may be the case if access control decision is based on the logic or state of the component. In this situation, access control decisions might be performed programmatically. A component can use two methods, EJBContext.isCallerInRole for entity java beans and HttpServletRequest.isUserInRole for web components allow for a finer-grained access control capability that may take into account factors such as parameters of the call, internal state of the component, time of call, etc. "The programmer of a component that calls one of these functions must declare the complete set of distinct roleName values used in all of its calls. These declarations appear in the deployment descriptor as security-role-ref elements. Each security-role-ref element links a privilege name embedded in the application as a roleName to a security role. It is ultimately the deployer who establishes the link between the privilege names embedded in the application and the security roles defined in the deployment descriptor. The link between privilege names and security roles may differ for components in the same application" [23].

Declarative access control policy provides more flexibility after the application has been written, is transparent and easy to comprehend. On the other hand, programmatic access control policy is not flexible after the application has been written, can only be completely understood by developers and requires changes to the code if it is to be modified or refined. A typical enterprise web based application built with J2EE technologies is likely to include a combination of declarative and programmatic access control policy. From the application security perspective, mistakes can be made in either one of the scenarios. In next section we describe some of the specifics of each technique.

It is important to make sure that access to resources is protected by the access control policy across all paths (component methods) that this access can take place. This goes back to the principle of securing the weakest link. Especially in cases with programmatic access control, it might make sense to have a few dedicated routines performing access to shared resources so that authorization code can be concentrated in as few places as possible. This is also referred to as encapsulation of access control with access components implementing an authorization barrier. The point is to be absolutely sure that authorization checks cannot be circumvented.

In order to control access declaratively to a Web resource, a security-constraint element with an auth-constraint sub-element needs to be specified in the Web deployment descriptor. The example below specifies that the URL with the pattern /control/placeorder can only be accessed by users acting in the role of customer. Thus the authorization constraint for web resource called placeorder is specified:

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>placeorder</web-resource-name>
                <url-pattern>/control/placeorder</url-pattern>
                <http-method>POST</http-method>
                <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
                <role-name>customer</role-name>
        </auth-constraint>
</security-constraint>
```

In order to control access declaratively to an EJB resource, a method-permission element can be specified in the deployment descriptor that would specify the methods of the remote and home interface that each security role is allowed to invoke. When the security roles required to access an enterprise bean are assigned only to authenticated users, then the bean in question is protected. An example of an enterprise bean authorization configuration is shown below:

```
<method-permission>
            <role-name>admin</role-name>
                        <method>
                                    <ejb-name>TheOrder</ejb-name>
                                    <method-name>*</method-name>
                        </method>
</method-permission>

<method-permission>
            <role-name>customer</role-name>
            <method>
                        <ejb-name>TheOrder</ejb-name>
                        <method-name>getDetails</method-name>
            </method>
            <method>
            ...
</method-permission>
```

Some resources may be unprotected and access to those resources may be permitted anonymously to users that have not been authenticated. Unprotected access to Web tier component can be enabled by leaving out an authentication rule. Unprotected access to an EJB component can be enabled by creating a mapping between at least one role allowed access to that resource and a universal set of users regardless of authentication.

An example below demonstrates how each application and each component within the application can specify individual authorization requirements. There are many reasons for which granular access control may be required and desirable. The example application below contains two enterprise beans, EJB 1 and EJB2, each with only one method. Each of the methods calls isCallerInRole where the role name is MANAGER. The deployment descriptor also specifies the security-role-ref element for the isCallerInRole in the enterprise beans. The security-role-ref for EJB1 links MANAGER to the role bad-managers and security-role-ref for EJB2 links MANAGER to the role bad-managers. The deployment descriptor also defines two method-permission elements that establish that the role employees can access all methods of EJB1 and that the role employees can also access all methods for EJB2. The deployment descriptor contains three security-role elements, namely employees, good-managers, and bad managers. User 1 is assigned to roles employees and good-managers and User 2 is assigned to roles employees and bad-managers. The second application only has one enterprise bean, EJB3, with only one method that also calls isCallerInRole with MANAGER as the role name. The deployment descriptor for the second application also contains security-role-ref elements to link MANAGER to the role good-managers. Method-permission element is defined to specify that the role employees can access all of the EJB3 methods. The deployment descriptor has two role elements, namely employees and good-managers. User 2 is assigned to roles employees and good-managers.

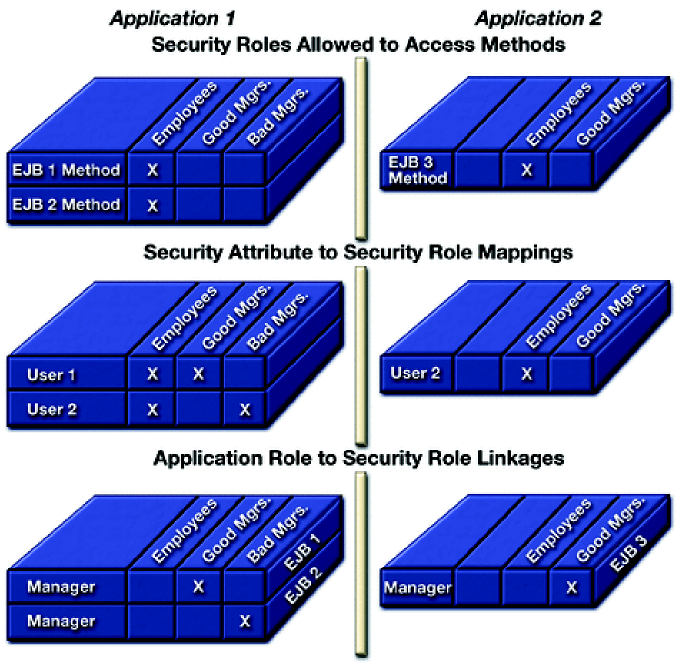

Application 1
Application 2
Security Roles Allowed to Access Methods
Employees
Good Mgrs.
Bad Mgrs.
EJB 1 Method
EJB 2 Method
EJB 3 Method
X
Security Attribute to Security Role Mappings
User 1
User 2
Application Role to Security Role Linkages
EJB 1
EJB 2
EJB 3
Manager

Figure 19: Sample J2EE Declarative Access Control Configuration [23]

Given the security mappings above, the table below demonstrates the access control decisions when various users (User 1, User 2 or User 3) invoke the various enterprise bean methods (EJB 1 method, EJB 2 method, EJB 3 method).

Authorization Decisions		
Call	Call Dispatched?	isCallerInRole?
User 1 - EJB 1	yes	true
User 1 - EJB 2	yes	false
User 1 - EJB 3	no	never called
User 2 - EJB 1	yes	false
User 2 - EJB 2	yes	true
User 2 - EJB 3	yes	true

Table 8: EJB Method Invocation Authorization Decisions [23]

3.2.3 Confidentiality:

As we previously mentioned, the value of <transport-guarantee> element to CONFIDENTIAL in the deployment descriptor tells the J2EE Web container to use an SSL protected session for transmission of authentication tokens such as passwords. Asymmetric (PKI with X.509 certificates) is used to distribute the shared symmetric key which is subsequently used for encryption. The J2EE specification recommends limiting cryptography usage to places where it is only absolutely necessary due to performance reasons. Most confidentiality issues are addressed at the deployment and not development time for the application when the deployer configures containers to apply confidentiality mechanisms to make sure that sensitive information is not disclosed to third parties. It is the responsibility of application assembler to provide the application deployer with the information on which method calls on which components feature parameters or return values that should be protected for confidentiality. The deployer than configures confidentiality in a way as to protect the method calls identified by the assembler will traverse open or insecure networks. In all instances where proper encryption is not used when appropriate, the calls should be rejected. From this we can gather that application of confidentiality mechanisms is very scarce and when required, very granular, in order to minimize the impact on performance. Nevertheless, precisely because the application of confidentiality is so conservative, it is more important than ever to be very careful. The application deployer needs to get complete and accurate information from the application assembler and then must configure the deployment descriptor appropriately.

Web resources have some special confidentiality requirements that must be addressed by application developers. For instance, it is important to be careful with properties of HTTP methods, especially with the consequences these properties have when a link is followed from one Web resource to another. When resources contain links to other resources, the nature of the links determines the ways in which protection context of the current resource determines the protection of requests made to the linked resources. With absolute links (beginning with https:// or http://) the protection context of the current resource is ignored, and the new resource will be accessed based on the protection context of the new URL. For URLs beginning with https://, a protected transport (usually via SSL) will be created with the server prior to sending the request. Conversely, for URLs beginning with http://, the request is sent over insecure transport. For relative links, the HTTP client container protects access to the linked resource based on the protection context of the current resource. An application developer should be aware of the link properties when linked requests must carry confidential data back to the server. In that case, a secure absolute link should be used (via https://). The downside of that approach is that it will have the effect of constraining the application to a very specific naming environment. Another approach would fall in the hands of application deployer who could configure the application so that in places where confidential interaction is required from one resource to another, both of those resources are deployed with confidential transport (by setting the <transport-guarantee> element to CONFIDENTIAL).

It is important to realize that many confidentiality issues will be addressed through network and not application security. In addition to performing the steps outlined above, an application deployer should also verify that all data is stored securely. All sensitive data should be stored encrypted. Things like passwords should be stored as hashes.

3.2.4 Integrity:

As we have already seen, a J2EE container serves as an authentication boundary between callers and the components it contains. The information flow to the component may be bi-directional (input or/and output). It is the responsibility of application deployer to configure containers via deployment descriptors in a way that would protect the interactions between components. Containers must be configured to implement integrity mechanisms when the call traverses open or insecure networks or in cases when calls are made between two components that do not trust each other (belong to different protection domains). Integrity checks must make it impossible for messages (method calls) to be used more than once. A container must compute and attach a message signature to the call request, and verify the association between the call response and the message signature attach to the call response. The called container verifies the association between the call request and the attached message signature, and then computes and attaches a message signature to the call response. This is all accomplished via hashing algorithms to create message digests. Timestamps can be attached to messages to make sure that they cannot be reused. All of the details are taken care of by the container. If the integrity check fails, the call should be abandoned and the caller notified of the failure.

3.2.5 Availability:

Availability disruption can occur as a side effect of some other problems within a web based enterprise application, such as poor exception management, buffer overflows, command injection, and other coding mistakes. There are no specific services to address the availability of the application. When an application is deployed however, components should be distributed to promote fault-tolerance and load balancing as that would help mitigate some of the security risks associated with availability (e.g. denial of service attacks).

3.2.6 Non-Repudiation:

As we mentioned in the discussion in integrity, the basic mechanism for ensuring non-repudiation is signing. Integrity and non-repudiation is addressed via cryptographic techniques. Those techniques mostly make use of various hashing algorithms. It is important to choose a hashing algorithm that is widely believed to be strong.

3.2.7 Accountability:

Keeping a record of security-related events that contain information on who has been granted access to what resources is an important tool for promoting accountability. These records can also be vital for recovery once a system has been breached and can also be used for intrusion detection purposes. They can also help track hackers down. Any good hacker will try to delete or modify all records of his or her activities and thus it is extremely important to keep those records secure and tamperproof. The deployer of the application should associate each of the container constraints for component interaction with a logging mechanism. The container can then audit one of the following events: all evaluations where the constraint was satisfied, all evaluations where the constraint was not satisfied, all evaluations regardless of the outcome and no evaluations. All changes to audit logs should also be audited. With J2EE responsibility for auditing and logging is shifted from developers to deployers of the application.

3.3 Programmatic Compensating Controls

As section 3.2 demonstrated, much of the J2EE security model and support for core security services is addressed through declarative security mechanisms many of which are configured by the application deployer and not application component developer. However, there are places where application developers of J2EE components will use programmatic security to support the core security services. In this section we discuss the various programmatic compensating controls. For coding examples that depend on the implementation, we focus the discussion on BEA Weblogic 8.1 implementation of the J2EE application server. BEA Weblogic 8.1 implements both the Web and EJB containers of J2EE. It also includes the implementation for all of the standard J2EE services. The discussion in this section focuses around the various J2EE security APIs, namely Java Authentication and Authorization Service (JAAS) API, SSL API and some other Java security APIs. We provide code samples for some of the programmatic security features utilizing the above mentioned security APIs. J2EE developers responsible for building security into enterprise web applications need to understand the nuts and bolts of how to use the various security APIs correctly.

3.3.1 Programmatic Authentication in Servlets

As we described earlier in the paper, there are instances where programmatic authentication may be appropriate. Weblogic application server provides a weblogic.servlet.security.ServletAuthentication API to provide programmatic authentication available to servlet applications. Recall that servlet components run in the J2EE Web container. With this API, servlet code can be written to authenticate users, log users, and associate users with the current session in order to register the user in the active security realm. Authentication can be performed using the SimpleCallbackHandler class or URLCallbackHandler class [19]. The two code examples included below show how to provide programmatic authentication:

With **weblogic.security.SimpleCallbackHandler:**

```
CallbackHandler handler = new SimpleCallbackHandler(username, password);
Subject mySubject = weblogic.security.services.Authentication.login(handler);
weblogic.servlet.security.ServletAuthentication.runAs(mySubject, request);
```

With **weblogic.security.URLCallbackHandler:**

```
CallbackHandler handler = new URLCallbackHandler(username, password);
Subject mySubject = weblogic.security.services.Authentication.login(handler);
weblogic.servlet.security.ServletAuthentication.runAs(mySubject, request);
```

In both cases request is an httpservletrequest object. As you may note, the code itself is very simple, with all of the authentication logic encapsulated from the programmer. All that a servlet programmer has to do is create a handler that contains the username and password information. Then weblogic.security.services.Authentication.login authentication service is invoked and the handler is passed to it. If authentication is successful, the authenticated user is associated with the current session using the weblogic.servlet.security.ServletAuthentication.runAs method [19].

3.3.2 JAAS API

J2EE client applications may use JAAS to provide authentication and authorization services. Some of the standard Java APIs to develop JAAS client applications include javax.naming, javax.security.auth, javax.security.auth.Callback, javax.security.auth.login, and javax.security.auth.SPI. Some JAAS security APIs specific to Weblogic 8.1 are weblogic.security, weblogic.security.auth, and weblogic.security.auth.Callback. The letter "x" after package name "java" (i.e. javax) stands for java extension, implying that JAAS API is an extension to the Java platform. If a J2EE application client needs to communicate with application servers that are not implemented via Weblogic, security APIs specific to Weblogic should not be used. In that case using those APIs would impact portability.

A client authenticated to the J2EE WebLogic application server with JAAS may be an application, applet, Enterprise Java Bean (EJB) or a servlet. JAAS is a standard extension to the Java Software Development Kit 1.4.1. JAAS allows enforcement of access control based on user identity. WebLogic application server uses only authentication capabilities of JAAS to support LoginContext and LoginModule functionalities. The WebLogic LoginModule weblogic.security.auth.login.UsernamePasswordLoginModule supports client user name and password authentication. For client certificate authentication, mutual SSL authentication should be used (provided by JNDI authentication).

JAAS can be used for external or internal authentication. Thus developers of custom authentication providers in J2EE applications, as well as developers for remote J2EE application clients may potentially need to understand JAAS. Users of Web browser clients or J2EE application component developers do not need to use JAAS. A typical JAAS authentication client application would include a Java client, LoginModule, Callbackhandler, configuration file, action file and a build script.

The key point to take away from this is that when weblogic.security.Security.runAs() method is executed, it associates the specified Subject with the permission of the current thread. After that the action is executed. If the Subject represents a non-privileged user, the default of the JVM will be used. Consequently, it is crucial to specify the correct Subject in the runAs() method. There are several options available to developers there. One is to implement wrapper code shown below.

Creating a wrapper for runAs() method:

```
import java.security.PrivilegedAction;
import javax.security.auth.Subject;
import weblogic.security.Security;
public class client
{
public static void main(String[] args)
{
Security.runAs(new Subject(),
new PrivilegedAction() {
public Object run() {
//
//If implementing in client code, main() goes here.
//
return null;
}
});
}
}
```

The discussion of various other methods for specifying the correct subject is omitted for the sake of brevity. There are eight steps to writing a client application using JAAS authentication in WebLogic J2EE application server. The first step is to implement LoginModule classes for each type of the desired authentication mechanisms. The second step is to implement the CallbackHandler class that will be used by the LoginModule to communicate with the user in order to obtain user name, password and URL. The third step is to write a configuration file that would specify which LoginModule classes would be used by the WebLogic Server for authentication and which should be invoked. The fourth step is to write code in the Java client to instantiate a LoginContext. The LoginContext uses the configuration file (sample_jaas.config) to load the default LoginModule configured for WebLogic Server. In step five, the login() method of the LoginContext instance is invoked. The login() method is used to invoke all of the LoginModules that have been loaded. Each LoginModule tries to authenticate the subject and the LoginContext throws a LoginException in the event that the login conditions specified in the configuration file are not met. In step six, Java client code is written to retrieve the authenticated Subject from the LoginContext instance and call the action as the Subject. When successful authentication of the Subject takes place, access controls can be placed upon that Subject by invoking the weblogic.security.Security.runAs() method, as was previously discussed. In step seven, code is written to execute an action if the Subject has the required privileges. Finally, step eight, a very important step, where the logout() method is invoked on the LoginContext instance. The logout() method closes the user's session and clears the Subject. It is very important for developers to follow all of these eight steps and do so properly in order for JAAS to be effective [19].

3.3.3 SSL API

Java Secure Sockets Extension (JSSE) provides support for SSL and TLS protocols by making them programmatically available. WebLogic implementation of JSSE also provides support for JCE Cryptographic Service Providers. HTTPS port is used for SSL protected sessions. SSL encrypts the data transmitted between the client and the server ensuring confidentiality of the username and password. SSL scheme uses certificates and thus requires certificate authentication not supported by JAAS. As the result, when SSL is used, an alternate authentication scheme must be used, one that supports certificates, namely Java Naming Directory Interface (JNDI) authentication. For client certificate authentication, a two-way SSL authentication scheme is used that is also referred to as mutual authentication. A common problem organizations have implementing SSL is failing to perform authentication properly. The result is secure communication with a remote host that has not been properly authenticated. It is thus critical to follow all of necessary steps to perform authentication correctly. A code example below demonstrates how one-way SSL authentication should be performed using JNDI authentication.

One-Way SSL Authentication:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
env.put(Context.PROVIDER_URL, “t3s://weblogic:7002”);
env.put(Context.SECURITY_PRINCIPAL, “javaclient”);
env.put(Context.SECURITY_CREDENTIALS, “javaclientpassword”);
ctx = new InitialContext(env);
```

SSL client application typically has several components. A java client initializes an SSLContextwith client identity, a HostnameVerifierJSSE, a TrustManagerJSSE, and a HandshakeCompletedListener. It then creates a keystore and retrieves the private key and certificate chain. An SSLSocketFactory is then used. Finally, HTTPS is used to connect to a JSP served by an instance of the application server. Another component is the HostnameVerifier that provides a callback mechanism so that developers can supply a policy for handling situations where the host is being connected to the server name from the certificate Subject Distinguished Name. A HandshakeCompletedListener defines how SSL client receives messages about the termination of an SSL handshake on a particular SSL connection. The number of times an SSL handshake can take place on a particular SSL connection is also defined. A TrustManager builds a certificate path to a trusted root and returns true if the certificate is valid. A build script compiles all of the files required and deploys them [19].

Two-way SSL authentication can be used if a mechanism is needed for the two parties to mutually authenticate each other. For instance, two servers may need to communicate to each other securely and may utilize two-way SSL authentication. This allows to have a dependable and secure communication. Typically mutual authentication is used in client-server environments, but it may also be used in server-server communication. An example below demonstrates establishment of a secure connection between two Weblogic server instances:

Two-Way SSL Authenticaiton Between Server Instances:

```
FileInputStream [] f = new FileInputStream[3];
f[0]= new FileInputStream("demokey.pem");
f[1]= new FileInputStream("democert.pem");
f[2]= new FileInputStream("ca.pem");
Environment e = new Environment ();
e.setProviderURL("t3s://server2.weblogic.com:443");
e.setSSLClientCertificate(f);
e.setSSLServerName("server2.weblogic.com");
e.setSSLRootCAFingerprints("ac45e2d1ce492252acc27ee5c345ef26");
e.setInitialContextFactory
("weblogic.jndi.WLInitialContextFactory");
Context ctx = new InitialContext(e.getProperties())
```

There are many steps to implementing SSL properly and developers should be aware of those. Misusing SSL could render it useless. There are too many detailed nuances to discuss all of them in this paper. Application component developers in charge of programmatic security need to have deep understanding of underlying technology and how to properly use it.

4.0 .NET Security

4.1 Architecture Overview

We now introduce an architecture overview for the component technologies of Microsoft's .NET framework used in a typical enterprise web application. The discussion is focused on the technologies used across the various tiers, namely the client tier, presentation tier, business logic tier, and the database tier. The discussion in this section presents the architectural overview of some of the .NET technologies that will be instrumental for further discussion the support for core security services. The discussion of .NET will focus somewhat on ASP.NET since that technology in particular is most widely used for development of web based enterprise applications. .NET and J2EE are two major competing frameworks for development of enterprise applications. As we proceed through the discussion in this chapter we will compare and contrast the solutions offered by J2EE and .NET, specifically relating to support for the core security services.

4.1.1 .NET Framework Overview

Prior to diving into ASP.NET, a .NET technology used to build enterprise web based applications, it is helpful to first briefly introduce the .NET Framework that is the foundation for all the .NET technologies. The purpose of the .NET Framework is to provide support for building and running the next generation of applications and XML Web Services. Here are some of the objectives of the .NET Framework according to the .NET Framework Developer's Guide [28]:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

Note bullet three that sets as one of the objectives for the .NET Framework provision of a code-execution environment that facilitates safe execution of code even if the code is unknown and not trusted. This is Microsoft's recognition of the wide use of mobile code in distributed applications and an attempt to provide secure sandbox environments for execution of that code. We will come back to this point later on in the discussion.

The .NET Framework consists of two main components, namely the common language runtime (CLR) and the .NET Framework class library. CLR forms the foundation for the .NET Framework that promotes the principle of managed code. CLR manages code at execution time providing such services as memory management, thread management, remoting support, enforcement of strict type safety and support for code accuracy from a robustness and security standpoint. Code targeting the CLR is referred to as managed code, as opposed to unmanaged code, which does not target the CLR. Microsoft's .NET promotes the usage of managed code. The other main component of the .NET Framework is the class library, a "comprehensive, object-oriented collection of reusable types that you can use to develop applications ranging from traditional command-line or graphical user interface applications to applications based on the latest innovations provided by ASP.NET" [28]. The basic concept behind the common library is to promote reuse that would greatly facilitate and speed up development of reliable applications.

.NET Framework allows development of applications that use a combination of managed and unmanaged code. For instance, .NET Framework may be hosted by unmanaged components that can load the CLR into their processes and initiate the execution of managed code. This facilitates integration and allows usage of both managed and unmanaged code capabilities. ASP.NET uses CLR to provide scalable, server-side environment for managed code. CLR in ASP.NET is also used to enable XML Web services. Internet Explorer (IE) is an unmanaged application that hosts CLR as a MIME type extension. This enables IE to host embedded managed components such as HTML form controls. This also facilitates usage of managed mobile code that draws heavily from the least privilege principle to provide semi-trusted (sandboxed) execution and isolated file storage. The diagram below shows how CLR and the common language enable applications and systems. It also shows where managed code fits in.

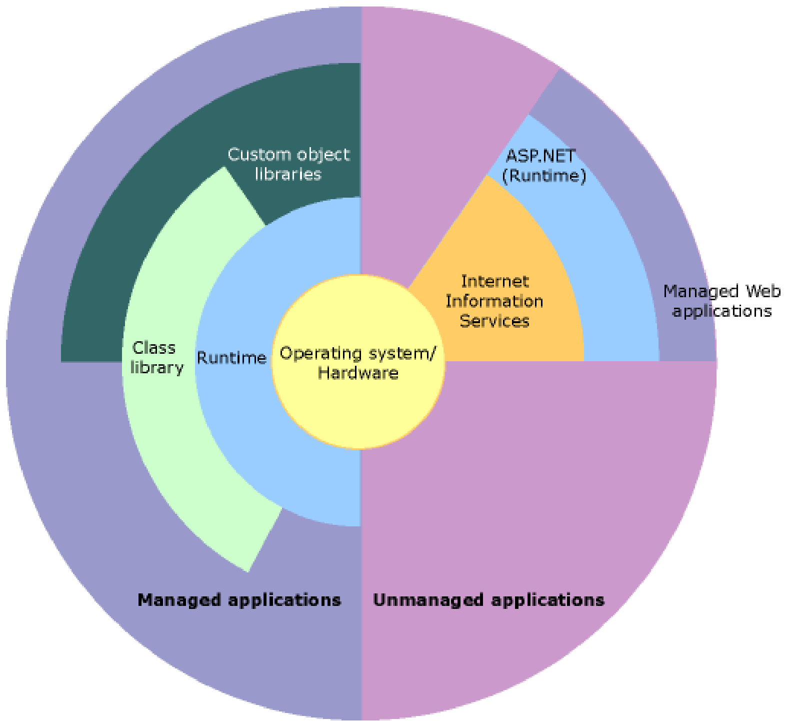


Figure 20: .NET Framework [28]

4.1.2 Common Library Runtime (CLR)

As we mentioned in the last section, the .NET common language runtime component manages memory, thread execution, code execution, code safety verification, compilation and various other system services for managed code running on it. The figure above reveals that ASP.NET uses Active Server Pages (ASP) technology that leverages CLR running on top of the Internet Information Services (ISS) Server. Factors such as code origin are used to assign a degree of trust to a managed component for the purposes of security. This means that security related decisions are made on per component basis, potentially restricting access to file-access operations, registry access-operations, etc. In the J2EE world that is analogous to declarative security that can be used to defined authorization on a component basis. Various components within the same application can have permissions that differ.

CLR enforces code access security. For instance, a managed component embedded in the web browser and providing GUI features can be guaranteed to not have access to the backend systems. Common language runtime also enforces code robustness through support for strict type-and-code-verification infrastructure known as the common type system (CTS). These robustness checks can also increase the overall security posture of the code. CTS guarantees that managed code is self describing and can take in the various managed types and instances that can then also be strictly enforced.

CLR addresses the problems associated with memory leaks and invalid memory references through handling of object layout, managing references to objects and freeing them when they cease being used. This is equivalent to the garbage collection services provided by the Java Virtual Machine. By taking the burden of cleanup away from the programmer, the chance of security issues related to memory management significantly decreases. CLR promotes interoperability and increase in productivity by allowing developers to write programs in any of the supported languages and still take advantage of CLR, the class library, and components written in other languages by other developers. Various compilers can target the CLR and make use of the various .NET Framework features to the existing code, thus migrating the migration process for existing applications. CLR also allows interoperability with existing technologies such as COM components and DLLs. CLR is geared towards performance and thus managed code is not interpreted. Just in time (JIT) compiling allows the managed code to run in the native machine language of the system on which it is executing. As this happens, memory manager performs memory defragmentation and attempts to increase spatial locality to further increase performance. Common language runtime can be hosted by Microsoft SQL Server and Internet Information Services Server. Thus the business logic and the database tiers can both make use of the common runtime library features. [28]

4.1.3 Class Library

The class library of the .NET Framework provides a collection of reusable types that work with the common language runtime. The class library is object oriented, thus types defined in the class library can be extended by managed code. Third-party components integrate seamlessly with the classes in the .NET framework. Classes collection in the .NET Framework includes a set of interfaces that can be implemented in the custom classes. This is similar to the interfaces provided by the various J2EE services that can be implemented by application component developers. .NET Framework types facilitate string management, data collection, database connectivity, and file access. The services supported by the .NET framework include console application, Windows GUI applications, ASP.NET applications, XML Web services and Windows services. For instances, .NET Framework Web Forms classes can be used in an ASP.NET enterprise web based application.

4.1.4 Web Based Application Architecture

Managed client-server applications are implemented using runtime hosts in .NET. Unmanaged application can host the CLR. This way all of the features of CLR and the class library can be harvested while providing performance, scalability and interoperability of the host server. A figure below shows a possible deployment model for a web based .NET application.

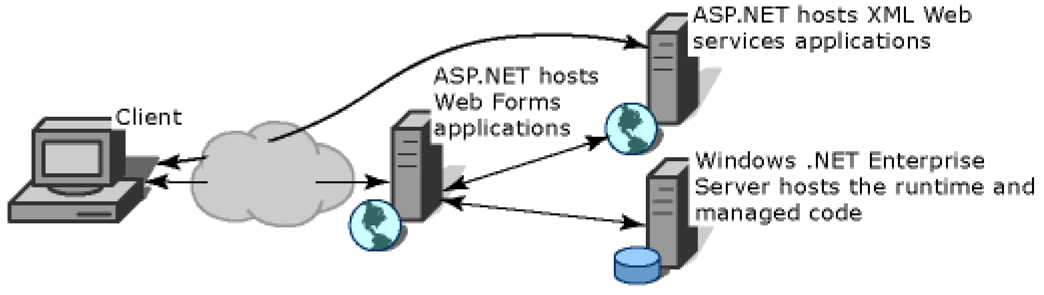

ASP.NET hosts XML Web services applications
ASP.NET hosts Web Forms applications
Client
Windows .NET Enterprise Server hosts the runtime and managed code

Figure 21: .NET Client-Server Application [28]

The diagram above shows that the .NET Enterprise Server hosts the runtime and the managed code. This is somewhat similar to the J2EE application server. Unlike J2EE however, the presentation logic and services are run on separate servers with ASP.NET Web Forms and ASP.NET XML Web Services respectively. ASP.NET is the underlying hosting environment that allows developers to use the .NET Framework to target Web-based applications. "ASP.NET is more than just a runtime host; it is a complete architecture for developing Web sites and Internet-distributed objects using managed code. Both Web Forms and XML Web services use IIS and ASP.NET as the publishing mechanism for applications, and both have a collection of supporting classes in the .NET Framework" [28].

ASP.NET builds on the ASP technology, but offers significant improvement with Web Forms, CLR and the class library. Web Forms pages can be developed in any language that supports the .NET Framework. Web Forms pages are the managed version of what used to be unmanaged ASP pages. However, Web Forms pages can take advantage of the runtime, while ASP pages are scripted and interpreted. ASP.NET pages are faster, have increased functionality and are easier to develop compared to unmanaged ASP pages because they make use of the common language runtime.

.NET framework also provides support for XML Web Services that support distributed Business to Business (B2B) communication. .NET Framework also provides a collection of classes and tools to facilitate implementation and usage of the XML Web Services applications, built on standards such as SOAP (a protocol for RPC), XML and WSDL (Web Services Description Language). Support for these standards provides interoperability for the .NET Framework.

A possible .NET usage scenario might involve querying XML web service, parsing its WSDL description, and producing C# or Visual Basic (VB) source code that an application can use to become a client of the XML Web service. Classes in the source code can be derived from classes in the class library to handle the communication using SOAP and XML parsing. If an XML Web service is developed and published, the .NET

Framework provides classes that follow the various communication standards like SOAP, WSDL and XML. This allows developers to focus on providing the logic in the service without having to worry about communications infrastructure. This is very much analogous to what is happening in J2EE. However, in terms of performance, .NET provides superior results. For instance, when publishing an XML service, the service will run with the speed of native machine language and make use of IIS scalable communication.

4.1.5 Building Web Applications with ASP.NET

Now that we have some understanding of the .NET Framework we can talk about the Microsoft technology that makes use of the .NET Framework capabilities and enables development of web based applications. That technology is ASP.NET. ASP.NET is the next version of Microsoft's Active Server Pages (ASP) technology. ASP.NET builds on ASP to provide additional services necessary to build enterprise web-based applications. To draw an analogy with the J2EE world, ASP may be loosely analogous to JSP, and ASP.NET in itself is analogous to the Web container of the J2EE application that implements both JSP and Servlets, as well as provides the various services, such as JMS, JTA, etc. Existing ASP applications can have ASP.NET functionality added into them in order to increase stability, scalability and functionality. ASP.NET code can be compiled and is based on the .NET Framework. Consequently, due to the common language runtime, code written in other languages based on the .NET Framework, such as Visual Basic .NET, C# .NET, and Jscript .NET is readily compatible with ASP.NET applications. Additionally, since ASP.NET is based on the .NET Framework, advantage can be taken of managed common language runtime environment, type safety, inheritance, and other features. One of the reasons that this is important from the security standpoint is that this means that enterprise web-based application built with ASP.NET can take advantage of all the security features of the .NET Framework.

Some of the important technologies available to programmers that are bundled under ASP.NET are Web Forms and XML Web Services. Web Forms enhance simple HTML based forms by facilitating construction of powerful forms-based Web pages. ASP.NET server controls can be used to create common UI elements that can be programmed for common tasks. These controls promote reusability by allowing a speedy construction of Web Form pages from reusable built-in or custom components. Reuse is very important from application security standpoint because it makes the introduction of bugs less likely. XML Web Services functionality allows a way to access server functionality in a remote manner. J2EE also provides support for web services. Support for remote access in J2EE is also provided by the Remote Method Invocation (RMI) technology. XML Web services allow exposure of programmatic interfaces to the enterprise web application data thus allowing the client to obtain and manipulate the data obtained from the enterprise web based application. XML Web services facilitate the exchange of data in client-server or server-server environments via XML messages and HTTP. Since Web Services are technology and calling convention independent, they can be provided and consumed by applications written in any language, using any component model, and running on any operating system that supports access to XML Web services.

ASP.NET adds object orientation and structure to the ASP technology. Thus, ASP.NET is not backwards compatible with ASP, and applications written with ASP will not work without changes under ASP.NET. If an existing ASP application makes use of embedded VB Script, changes will have to be made to the script due to extensive changes to Visual Basic .NET. ASP.NET provides database access through ADO.NET (analogous to JDBC in J2EE). ASP.NET developers can write logic that runs at the application level by writing code in the Global.asax text file or in a compiled class deployed as an assembly. Some of the logic that can be included is in the form of application-level events, but can be extended to meet the special need of the Web application. This is somewhat equivalent to declarative programming capabilities in J2EE that are supported by the container.

ASP.NET also provides application and session-state facilities that are also compatible with the .NET framework. Analogously, J2EE Web container provides session support. J2EE also provides session support through stateless session bean components running in the EJB container. To interact with request and response services of the IIS Web server IHttpHandler interface can be implemented. To include custom events that participate in every request made to the application server, IHttpModule can be implemented. HTTPRequest, HTTPRequest, and listeners are equivalent technologies in the J2EE world.

Since ASP.NET is built on top of the .NET Framework, it can also take some of the performance advantages of the .NET Framework and the common language runtime. ASP.NET applications are compiled, rather than interpreted (at run time), that allows early binding, strong typing, and just-in-time (JIT) compilation to native code. This has both security and performance advantages. ASP.NET is modular which allows developers to remove irrelevant modules to the developed application. ASP.NET also provides cashing services that can be either built-in or provided via caching APIs. ASP.NET also comes with performance counters that allow monitoring of the application to collect various metrics.

The .NET Framework and ASP.NET provide default authorization and authentication schemes for enterprise Web applications that can easily be removed, added to and replaced by other schemes. We will discuss the ways in which ASP.NET supports the core security services in the subsequent section. ASP.NET configurations are stored in XML files that are analogous to XML deployment descriptors in J2EE. Configurations can be customized to meet the needs of the application. Much of security in J2EE is declarative security, specified in deployment descriptors. We will see how this is done with ASP.NET. IIS 6.0 uses a new process model, referred to as the worker process isolation mode, which enhances security by promoting compartmentalization.

4.2 Support for Core Security Services

4.2.1 Role-Based vs. Code Access Security

Before we look at the various ways in which web-based enterprise applications built with ASP.NET with the .NET Framework foundation support the core security services across the various application tiers, we first introduce the .NET security model. .NET Framework provides two kinds of security: role-based security and code access security. As a synopsis, role-based security controls user access to application resources and operation, whereas code access security controls code access to various resources and privileged operations. These two kinds of security are complementary forms of security that are available to .NET Framework applications. User security focuses around the identity and the capabilities of the user, while code access security focuses around the source and author of the code and the associated permissions. Code security requires authorizing the rights of the code to access the file system, registry, network, directory services and directory services. The main distinction here is user capabilities vs. code permissions.

Role-based security in the .NET Framework enables a Web application to make security decisions that depend on the identity of the application user. These roles may be Windows groups, enterprise roles, etc. Principal objects passed to Web applications as part of the request would contain the identity of the authenticated user. The diagram below illustrates a common usage model for role-based security in a web application built with .NET.

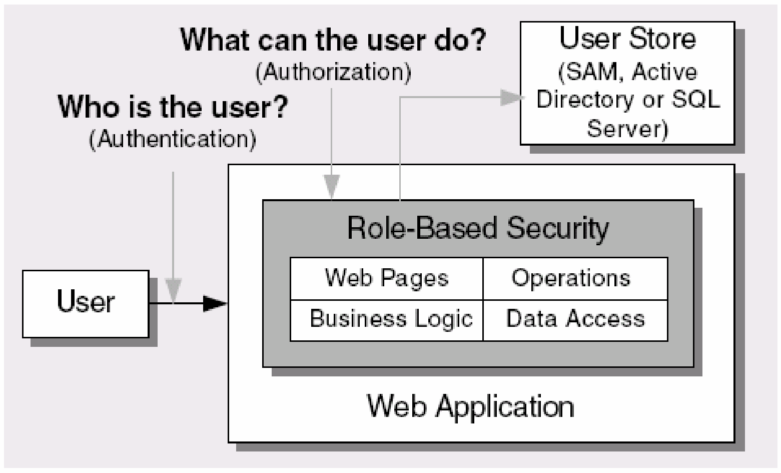


Figure 22: .NET Role-Based Security [10]

On the other hand, code access security performs authorization when code attempts to access the file system, registry, network or perform privileged operations, like calling unmanaged code or using reflection. Code access security provides an additional level of granularity by enabling to set permissions on a piece of code. Among other things, this approach can help enforce the principle of compartmentalization and help reduce the amount of damage if part of the system (i.e. some system process) is compromised. Analogously, we saw that with J2EE components permissions are specified via deployment descriptors and enforced by the container. There are some very important differences however between J2EE and .NET security model that will become apparent as we proceed with the discussion. The diagram below demonstrates the usage of code access security in .NET Web applications.

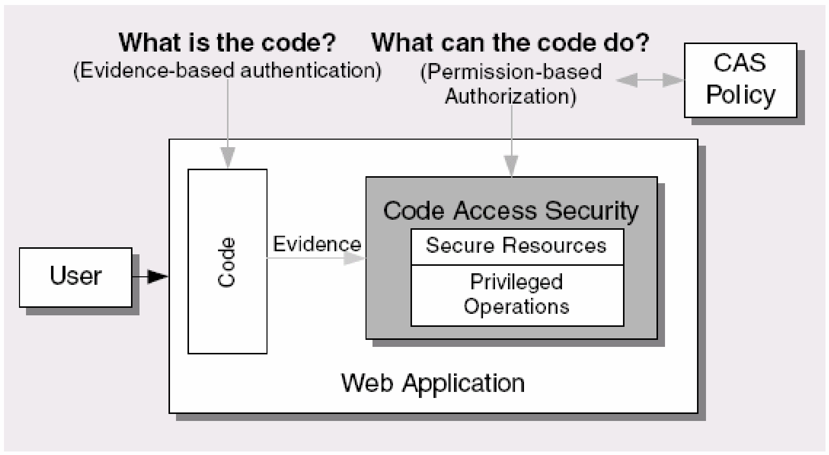


Figure 23: .NET Code Access Security [10]

In the context of code access security, authentication of code is based a variety of information about the code, such as its strong name, publisher, installation directory, etc. Authorization is based on the code access permissions that are specified by the security policy.

4.2.2 Security in Managed Code

In next section we will discuss some of the programmatic security issues that ASP.NET developers need to be concerned about, while at this point we focus on the .NET security model and the general support for core security services. Managed code is perhaps one of the single most important contributors to .NET security. Assemblies in .NET Framework (J2EE components and .NET assemblies mean roughly the same thing) are built with managed code. C#.NET and Visual Basic.NET (that are often used to implement much of the business logic in ASP.NET enterprise web application) are compiled to Microsoft intermediate language (MSIL) instructions, contained in Microsoft Windows .dll or .exe files. When an assembly is loaded and the requested method is called, the MSIL instructions of the method are compiled by the just-in-time (JIT) compiler into native machine instructions that are then executed. If a method is never called, it is never compiled. Intermediate language and run-time environment that are enabled by the common language runtime give assembly developers several security advantages. First, there is file format and metadata validation. The CLR ensures the validity of the MSIL file format and that addresses do not point outside the boundaries of the MSIL file. The CLR verifies the integrity of the metadata contained in the assembly. Second, type safety of MSIL code is verified at just-in-time compilation. This verification can prevent bad pointer manipulation, validate type conversions, check array bounds, etc. The result is practical elimination of buffer overflow problems in managed code, although developers still need to be careful with any unmanaged code that might be called. This is a great security feature since buffer overflows represent one of the most dangerous types of security vulnerabilities in the application. Furthermore, integrity checks take place in managed code for strong named assemblies to ensure that the code has not been altered in any way since it was built and signed. That is accomplished via digital signatures. Therefore attackers cannot alter MSIL instructions and as those alterations would be detected. Finally, managed code allows for code access security. CLR provides a virtual execution environment that enables performing additional security checks at runtime. Some of these checks involve run-time enforcement of code access permissions for the purposes of security decisions based on the identity of the calling code. [10]

4.2.3 Declarative vs. Imperative Security

Declarative security in .NET is a different concept from declarative security in J2EE. In the .NET world, through declarative security, developers can specify which users should have access to a class or a method. This is accomplished through adding a special attribute to the class or method definition called PrincipalPermissionAttribute. An attribute defined at a class level will apply to all the methods unless it is overridden in a method. On the other hand, imperative security can be used to provide a higher level of granularity when it becomes necessary. With imperative security, System.Security.Permissions.PrincipalPermission objects are used to create a PrincipalPermission object that has a blank user name along with a specified role name that then calls the Demand method. The CLR then interrogates the current Principal object associated with the current thread in order to check whether the related identity is a member of the specified role. If that is not the case, access is denied and SecurityException is thrown. Consequently, imperative security allows for finer grained access control than what can be achieved with declarative security.

Declarative and imperative security can be used with both role-based security and code access security in the .NET Framework. For the most part declarative security will be used, but imperative security has important applications, for instance when a security decision is based upon some variable value only available at runtime. Declarative security allows system administrator or assembly consumer to have a clear picture of what security permissions particular classes and methods have. Consequently, the right access security policy can be configured to accommodate those permissions. Declarative security also provides better performance because the checks are evaluated only once at load time, and not each time at run time (which is the case with imperative security). With declarative security, permissions on security attributes are checked before running any other code, thus eliminating any potential bugs that might have resulted from late enforcement of permissions. Finally, declarative checks can be applied to all class members, while imperative checks apply only to the routine being called. On the other hand, imperative security has the advantage of allowing dynamic changes to permissions based values only available at run time. Additionally, imperative security allows more granular authorization mechanisms by placing conditional logic in code. Declarative and imperative security in .NET loosely maps to declarative and programmatic security in J2EE. Developers are less likely to make mistakes with declarative security, since much detail is hidden from them, while bugs with imperative security are more likely. For instance, authorization checks in the code can be susceptible to race condition problems in multi-threaded environments which might enable a window of vulnerability for an attacker to circumvent the checks.

4.2.4 Security Namespaces

.NET Framework also supports the concept of security namespaces. Developers who program secure web applications in ASP.NET need to make use of these namespaces. Some of the .NET Framework security namespaces include System.Security, System.Web.Security, System.Security.Cryptography, System.Security.Principal, System.Security.Policy and System.Security.Permissions. A diagram illustrating each of the .NET security namespaces is provided below.

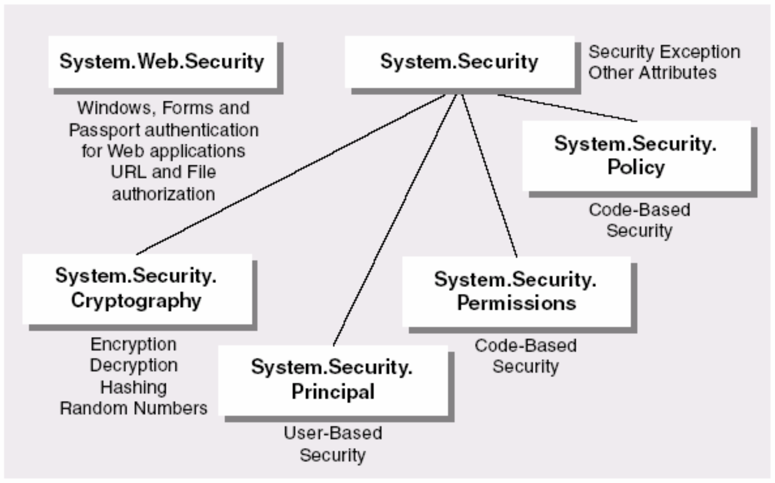

System.Web.Security
Windows, Forms and Passport authentication for Web applications URL and File authorization
System.Security
Security Exception Other Attributes
System.Security. Policy
Code-Based Security
System.Security. Cryptography
Encryption Decryption Hashing Random Numbers
System.Security. Permissions
Code-Based Security
System.Security. Principal
User-Based Security

Figure 24: .NET Security Namespaces [10]

.NET Framework namespaces are loosely analogous to the various security APIs in J2EE. In order to build secure web based enterprise applications with ASP.NET, developers need to be aware of the various services provided through the security namespaces and how to use them properly.

System.Security namespace contains the CodeAccessPermission base class that can be extended to derive all of the other code access permission types. This class obviously plays a critical role in .NET code-access security that we have previously discussed. Most often, specific permission types (derived from the base permission class) are used that represent the rights of code to have access to specific types of resources or perform various privileged operations. For instance, FileIOPermission may represent the rights of the code to perform file input and/or output. This particular security namespace also contains some classes that encapsulate permission sets, such as Permission Set and NamedPermissionSet. These permission sets encapsulate some common types widely used when building secure Web applications. SecurityException is an example of such a type and it is used to represent security errors. AllowPartiallyTrustedCallersAttribute is another commonly used type. This type is an assembly-level attribute used in strong named assemblies that provide support for partial trust callers. Without having this attribute, a strong named assembly could only be invoked by fully trusted users who have full and not restricted permissions. System.security .NET namespaces provides some key code-level authorization capabilities of which .NET web application developers should be aware.

System.Web.Security .NET namespace contains additional classes that can be used to build in Web application security programmatically, namely authentication and authorization services. Whereas System.Security namespace provides code-access authorization to system resources (e.g. files, registries), System.Web.Security provides authentication and authorization services to protect web application resources. Authentication and authorization is supported for windows, forms, URLs and files, etc. For instance, URL authorization is provided by the UrlAuthorizationModule and file authorization is provided through FileAuthorizationModule classes. There are several types that belong to this namespaces that are commonly used in Web applications. FormsAuthentication type provides static methods to assist with Forms authentication, as well as authentication ticket manipulation. FormsIdentity type encapsulates the identity of the user that has been authenticated by the Forms authentication. Since Microsoft Passport authentication service has been introduced, PassportIdentity can be used to encapsulate the user identity that has been authenticated through Passport authentication. System.Web.Security namespace provides extensive support for authentication services in .NET web applications.

System.Security.Cryptography namespace provides a .NET web application developer with types that can be used to perform encryption, decryption, hashing, and random number generation. Managed code provides a variety of encryption algorithms, while others are offered by types in this namespace that provide wrappers for cryptographic functionality of the Microsoft Win32-based CryptoAPI. System.Security.Cryptography namespace provides tools for the developer to implement confidentiality services.

System.Security.Principal security namespace provides types that support .NET role-based security. These types can be used to restrict which users can access classes and class members. IPrincipal and IIdentity interfaces are provided by this namespace. Some of the commonly used types from this namespace in Web applications are GenericPrincipal, GenericIdentity, WindowsPrincipal and WindowsIdentity. GenericPrincipal and GenericIdentity enable the developer to define custom roles and identities. They may be used with custom authentication mechanisms. WindowsPrincipal and WindowsIdentity types represent users authenticated with Windows authentication along with the associated Windows role list of the user.

System.Security.Policy namespace contains types that may be useful to implement the policy of the system as it pertains to code access security. This namespace contains types to represent code groups, membership conditions, policy levels, and evidence.

System.Security.Permissions .NET namespace contains most of the permission types that are used to encapsulate the rights of code to have access to various resources as well as perform operations that are privileged. The table below lists the permission types available in the System.Security.Permissions namespace.

Permission	Description
DirectoryServicesPermission	Required to access Active Directory.
DNSPermission	Required to access domain name system (DNS) servers on the network.
EndpointPermission	Defines an endpoint that is authorized by a SocketPermission object.
EnvironmentPermission	Controls read and write access to individual environment variables. It can also be used to restrict all access to environment variables.
EventLogPermission	Required to access the event log.
FileDialogPermission	Allows read-only access to files only if the file name is specified by the interactive user through a system-provided file dialog box. It is normally used when FileIOPermission is not granted.
FileIOPermission	Controls read, write, and append access to files and directory trees. It can also be used to restrict all access to the file system.
IsolatedStorageFilePermission	Controls the usage of an application's private virtual file system (provided by isolated storage). Isolated storage creates a unique and private storage area for the sole use by an application or component.
IsolatedStoragePermission	Required to access isolated storage.
MessageQueuePermission	Required to access Microsoft Message Queuing message queues.
OdbcPermission	Required to use the ADO.NET ODBC data provider. (Full trust is also required.)
OleDbPermission	Required to use the ADO.NET OLE DB data provider. (Full trust is also required.)
OraclePermission	Required to use the ADO.NET Oracle data provider. (Full trust is also required.)
PerformanceCounterPermission	Required to access system performance counters.
PrincipalPermission	Used to restrict access to classes and methods based on the identity and role membership of the user.
PrintingPermission	Required to access printers.
ReflectionPermission	Controls access to metadata. Code with the appropriate ReflectionPermission can obtain information about the public, protected, and private members of a type.
RegistryPermission	Controls read, write, and create access to registry keys (including subkeys). It can also be used to restrict all access to the registry.
SecurityPermission	This is a meta-permission that controls the use of the security infrastructure itself.
ServiceControllerPermission	Can be used to restrict access to the Windows Service Control Manager and the ability to start, stop, and pause services.
SocketPermission	Can be used to restrict the ability to make or accept a connection on a transport address.
SqlClientPermission	Can be used to restrict access to SQL Server data sources.
UIPermission	Can be used to restrict access to the clipboard and to restrict the use of windows to "safe" windows in an attempt to avoid attacks that mimic system dialog boxes that prompt for sensitive information such as passwords.
WebPermission	Can be used to control access to HTTP Internet resources.

Table 9: .NET Code Access Permission Types [10]

The SecurityPermission class from the table above specifies the rights of code to perform privileged operations, assert code access permissions, call unmanaged code, use reflection, control policy and evidence, etc. The Flags property of the SecurityPermission class determines the specific rights of the code. The Flags property can contain any of the values defined in the SecurtyPermissionFlags enumerated type. SecurityPermission class Flag property value of SecurityPermissionFlags.UnmanagedCode would indicate that given code has the right to invoke unmanaged code.

Developers of enterprise web based applications need to be very familiar with .NET security namespaces, as well as classes and types that they contain in order to be effective in building secure web application that provide support across all of the core security services. In this section we discussed role based security, code access security and managed code security concepts. We also discussed some ways to support the core security services in .NET applications through security namespaces and other mechanisms. We also saw that code access security provides a finer level of access granularity that can be used to support the compartmentalization principle. Next section builds on some of these principles to provide some examples of specific programming constructs that ASP.NET developers can make use of in order to build support for security into the enterprise web based applications [10].

4.3 Programmatic Compensating Controls

In this section we discuss some of the programmatic details of security in ASP.NET applications. We build on the information from the previous section and provide specific code examples of the ways developers can programmatically provide authentication, authorization, confidentiality and other core security services. The focus is on code access security and on building of secure .NET assemblies (analogous to J2EE components). One of the things to note about .NET is that it is very conservative with regards to the number of lines of code that the developer has to write to perform any task (including security related tasks), compared to J2EE for instance, and so less code decreases the probability of introducing bugs (security or others).

4.3.1 Coding Secure Assemblies

Assemblies are the units of deployment, version control, reuse and code access security in the .NET Framework. Consequently, secure design and implementation of assemblies is absolutely critical for .NET security. The goal of this section is to consider ways in which secure assemblies can be constructed. Some vulnerabilities in .NET are eliminated through the use of managed code and the CLR. For instance, buffer overflows in managed code are eliminated through the use of type safe verification. However, if unmanaged code is invoked by managed code, then buffer overflow problems may still occur. In this section we examine how to perform file I/O, registry access, data access, and other tasks in a secure manner.

4.3.1.1 Top Threats

Some of the main threats facing .NET assemblies are unauthorized access/privilege elevation, code injection, information disclosure and tampering. A diagram below illustrates these main threats. As we have described in the threat modeling chapter, understanding what the main threats are helps focus attention during development towards provision of mitigations for the threats. Usually threats exist because an attacker can exploit existing vulnerabilities. The goal of the developer is to code securely in a way as to not introduce vulnerabilities into the application, thus mitigating the threats.

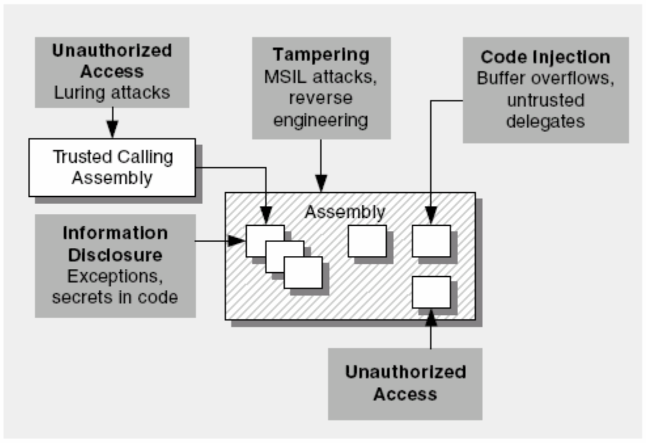


Figure 25: .NET Top Assembly Threats [10]

Unauthorized access, a condition that can often lead to privilege escalation, is manifested if an unauthorized user or unauthorized code calls an assembly and executes privileged operations that access restricted resources. Some vulnerabilities that can lead to unauthorized access and/or privilege elevation include weak or missing role-based authorization, exposure of internal types and type members, insecure implementation of code access security checks or non-sealed (non-final in Java talk) and unrestricted base classes that can be extended by untrusted code. Attackers may attempt to exploit these vulnerabilities through a luring attack where malicious code performs access to the assembly through an intermediary assembly that is trusted in order to bypass authorization mechanisms. An attacker can also use malicious code that bypasses access controls by calling classes that are not part of the assembly's public API. To counter the above mentioned threats, role-based authorization can be used to provide access control checks on all the public classes and class methods, type and member visibility can be restricted in order to minimize publicly accessible code, privileged code can be sandboxed to ascertain that calling code is authorized and meets required permission demands and make all classes that should not be extended final (sealed) or alternatively limit inheritance capabilities with code access security.

Code injections problems arise when an attacker is able to execute arbitrary code with the same permissions as those granted to the assembly process itself. This can be even a bigger risk if assembly code is called by unmanaged code and if the assembly process has root privileges. This is where least privilege can definitely help. At the assembly level, some common types of attacks using code injection are buffer overflows and running untrusted code. Buffer overflows are not really an issue in .NET unless unmanaged code is called. Some of the ways to counteract the code injection threats include validation of input parameters, validation of data passed to unmanaged code, not running untrusted code and using the least privilege principle.

Information disclosure can take place if an assembly leaks confidential data, like excessive exception details and unencrypted sensitive information to users (whether malicious or legitimate). Intellectual property threat also exists because MSIL code is easier to reverse engineer into source code than binary machine code. This also means that if the source code can be obtained through reverse engineering by an attacker, an attacker may be able to find more readily exploitable vulnerabilities in the application. Some vulnerabilities leading to information disclosure are bad exception handling and hard-coded secrets in the source code, such as passwords. Attackers may attempt to cause errors by passing bad input to the assembly thus causing possible availability problems and potentially information disclosure in the event of bad exception management. Information disclosure can then facilitate launch of additional attacks on the application. Some of the ways to avoid these vulnerabilities include good input validation, systematic exception handling that returns generic errors to the client, not storing secrets in code and obfuscation tools that confuse decompilers when those attempt to reverse engineer.

Tampering issues, also known as integrity problems, arise if an assembly is changed through alterations made to the MSIL instructions. Recall that MSIL is the intermediate representation of .NET Framework source. Mechanisms are available to protect against tampering, but those rely on strong name signatures. In the absence of a strong name signature, tampering may not be detected. Common ways that tampering may be used in an attack include either direct manipulation of MSIL instructions or reverse engineering of MSIL instructions. Tampering threat can be countered through usage of strong names to sign the assembly with a private key. When loading a signed assembly, the CLR will detect if the assembly has been altered in any way and will only proceed if no modifications have been made [10].

4.3.1.2 Privileged Code

When designing secure assemblies it is also important to identify privileged code since privileged code has an important impact on code access security. Privileged code is managed code that has access to various secured resources and can perform privileged operations such as calling unmanaged code. Privileged code must be granted elevated permissions by the code access security policy to fulfill its functions. Application designers need to pay special attention when defining interactions with privileged code. Privileged code provides access to privileged resources that require code access security permissions. These resources may include the file system, databases, registry, event logs, Web services, sockets, DNS, database, directory services, environment variables, among others. We will look in more detail later in this section at the various ways to provide secure access to the privileged resources. Privileged code also performs privileged operations that also require code access security permissions. These privileged operations may include calling unmanaged code, serialization, usage of reflection, creation and control over application domains, creation of Principal objects, and making changes to the security policy. Identifying privileged code is the first step to defining secure interactions between privileged code and other code so to not let an attacker perform any operations or access any resources in a manner that would constitute a privilege elevation. A threat modeling process should identify privileged code and the compensating controls will be found at the architecture, design and implementation stages.

4.3.1.3 Secure Class Design

Secure assembly design should take into consideration such factors as privileged code, trust level of target environment, public interfaces, etc. These issues should be ironed out at design time, although developers often become de facto designers of the application as well and should therefore be aware of the secure design principles. Another important thing to get right is the security in class design for which developers are responsible. In general, both with assembly and class design the aim should be to reduce the attack surface on the application. The attack surface would have been identified as part of the threat model. Classes that are built with security in mind would follow proper object-oriented design principles, prevent inheritance where it should not be allowed, restrict who and what code can call them. Microsoft provides the following recommendation for secure class design: restrict class and member visibility, seal non base classes, restrict which users can call the code, expose fields using properties. All of these recommendations make good sense from the object-oriented design perspective but are also very important for security. The same set of recommendations roughly holds true for Java classes used with J2EE as well.

The visibility of the class and the member methods that it contains should be restricted appropriately. For instance, class members need to be public only if they are part of assembly's public interface. Public classes are the only ones accessible from outside the assembly so it is desirable to have as few of them as possible to minimize the attack surface. Private class members are preferred for the most part and should be the default. Private classes pose the lowest security risk. Protected class members should be created only if the class member needs to be accessible to derived classes. Internal access modified should be used only if class members need to be accessible to other classes from the same assembly. If the class is not intended to be a base class, it should be sealed in order to prevent inheritance. An example below illustrates how this can be done.

```
public sealed class NobodyDerivesFromMe
{}
```

Furthermore, it is possible to restrict which users can call the code. Declarative principal permissions can be used to control which users have access to the class or its members. This can be specified at the class level or at the level of individual class methods. In the example below, only members of the Windows group can access the Orders class.

```
[PrincipalPermission(SecurityAction.Demand,
Role=@"DomainName\WindowsGroup")]
public sealed class Orders()
{
}
```

As a general principle in object oriented programming, fields should be exposed via properties. Typically getter and setter methods are provided for each of the private class data members. This is important from security standpoint because it allows the developer to add logic for input validation or permission checks inside those methods. An example of this is demonstrated in the code snippet below.

```
public sealed class MyClass
{
private string field; // field is private
// Only members of the specified group are able to
// access this public property
[PrincipalPermission(SecurityAction.Demand,
Role=@"DomainName\WindowsGroup")]
public string Field
{
get {
return field;
}
}
}
```

4.3.1.4 Strong Names

We have already alluded several times throughout this paper to strong names in .NET Framework and said that they can be used to ensure integrity. An assembly strong name is compromised of a text name, a version number, culture (optional), public key, and a digital signature. Machine.config can be used to reference the various constituents of the strong name. The example below demonstrates how System.Web assembly can be referenced in Machine.config.

```
<add assembly="System.Web, Version=1.0.5000.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" />
```

Ideally, all assemblies should be given strong names. Strong names make sure that partially trusted code cannot call the assembly, the assembly can be shared among multiple applications and that strong names are used as security evidence. Public key portion of the strong name provides cryptographically strong evidence for code access security. Cryptographically strong evidence can also contain Authenticode signature and an assembly’s cash. Authenticode signatures can be used if X.509 certificate was used to sign an assembly. Authenticode is not loaded by the ASP.NET host and thus not used with ASP.NET applications.

Application developers constructing assemblies should delay sign the code. This is a process of placing the public key (part of the strong name) in the assembly that can later be used as evidence to the code access security policy. A private key, that is securely stored in a central location, is used to sign the assembly and create its digital signature. Few trusted people have access to the private key. A single public key representing organization developing the software can be used by all of the developers. This is really important from the integrity standpoint hence all assemblies should be signed.

4.3.1.5 Exception Management

Poor exception management can lead to information disclosure which may help malicious hackers to compromise the application. Additionally, poor exception management can also result in availability issues by allowing an attacker to launch a denial or service attack. In short, all exceptions should be caught and handled properly. Microsoft provides several recommendations for proper exception management: use structured exception handling, do not log sensitive data, do not reveal system or sensitive application information, consider exception filter issues and consider an exception management framework.

Structured exception management in Visual C# and Visual Basic .NET does not differ from Java. In all of these languages, try / catch and finally constructs are used. It is important to remember to be very granular by catching specific exceptions rather than trying to lump them all into one by catching a generic exception. Structure exception handling guarantees that the system is always in a consistent state. The code snippet below demonstrates a structured approach to exception handling:

```
try
{
// Code that could throw an exception
}
catch (SomeExceptionType ex)
{
// Code to handle the exception and log details to aid
// problem diagnosis
}
finally
{
// This code is always run, regardless of whether or not
// an exception occurred. Place clean up code in finally
// blocks to ensure that resources are closed and/or released.
}
```

Exceptions tend to contain a lot of detailed information that could help an attacker compromise the system. Consequently, care should be taken when logging exception data and sensitive data should not be logged. The raw output of an exception should never propagate directly to the client (or go outside the application trust boundary for that matter) as it provides too much useful information. Some of these details may include operating system, .NET Framework version numbers, method names, computer names, SQL command statements, connection strings, along with other details. The information may be very useful to an attacker. For instance, knowing what version of the operating system is used, an attacker may exploit a known vulnerability for that operating system. Instead, generic messages about an exception should be returned to the client where appropriate. A lot of the time exception handling might make the exception transparent to the client, in which case nothing may be returned. In general, an exception management framework can ensure that all exceptions are properly detected, logged and processed in a way that avoids both information disclosure and availability issues [10].

4.3.1.6 File Input and Output

Developers should be aware of how to properly interact with the file system and avoid the common pitfalls. For instance, security decisions should not be based on input file names because of the many ways in which a single file name can be represented. This problem is also known as canonicalization. If the code needs to access a file by using a file name supplied by a user, developers must make sure that an assembly cannot be used by a hacker to get access to or overwrite sensitive data. Microsoft provides the following recommendations for performing input/output interactions with the file system securely: avoid untrusted input for file names, do not trust environment variables, validate input filenames and constraint file I/O within your application's context.

To avoid the possibility of granting an attacker access to arbitrary system files, code should not be written that accepts a file or path input from the caller. Instead, fixed file names and paths should be used when performing input and output. Wherever possible, absolute file paths should be used rather than constructing file paths through values of the environment variables, because it is not always possible to guarantee the value of the environment variable. If input file names absolutely have to be passed by the caller, extensive validation should be performed by the process to determine validity. For instance, the code should check to make sure that the file system names are valid and check that the location is valid, as defined by the application context. System.IO.Path.GetFullPath method should be used to validate the path and file names passed by the caller. An example of how this can be accomplished is provided below.

```
using System.IO;
public static string ReadFile(string filename)
{
// Obtain a canonicalized and valid filename
string name = Path.GetFullPath(filename);
// Now open the file
}
```

The GetFullPath method used in the code snippet above checks that the file name does not contain invalid characters (as defined in Path.InvalidPathChars), checks that the file name represents an actual file and not some other resource, checks for the length of the path to make sure that it is not too long, removes redundant characters and rejects file names that conform to the //?/ format. This takes some of the burden of writing custom validation logic away from the programmer, thus reducing the risk of canonicalization problems. Following validation of the file system file name, assembly code must validate that the location of the file is valid in the application context. For instance, it is necessary to verify that the location is within the directory hierarchy for the application and that code cannot gain access to arbitrary files on the file system [10].

4.3.1.7 Event Logs

Most enterprise web based application will use event logging in some shape or form and it is very important to ascertain that this is done securely. The fundamental threats to event logs involve tampering with the log, information disclosure of sensitive information stored in the log and log deletions in order to erase tracks. While some event log protection is provided by the security features in the Windows operating system, developers must ensure that event logging code cannot be used by an attacker to gain unauthorized access to the event log.

The first axiom of logging should be to not log sensitive data. We have already mentioned this in the secure exception management section. Additionally, if EventLog.WriteEvent is used, existing records cannot be read or deleted. This leads us back to the principle of least privilege. Do not give more privileges to the code updating the event logs than is absolutely necessary and grant those privileges for the shortest amount of time necessary. This can be accomplished by specifying EventLogPermission by way of code access security. A threat that should be addressed is preventing an attacker from invoking the code that does event logging so many times that would cause overwrite in previous log entries. An attacker can try and to this in an attempt to cover his tracks for example. A way to deal with that may be to use an alert mechanism that would signal the problem as soon as the event log approached a limit.

4.3.1.8 Registry Access

Developers often use the system registry as a place to provide secure storage to sensitive application configuration data. Configuration data can be stored under the local machine key (HKEY_LOCAL_MACHINE) or under the user key of the current user (HKEY_CURRENT_USER). For security, all data should be encrypted prior to being placed in the registry.

If configuration data is stored under HKEY_LOCAL_MACHINE then any process running on the local machine can possibly have access to the data. A restrictive access control list (ACL) should be used (applied to the specific registry key) to restrict access to the configuration data to the administrator and the specific process (or thread) that needs to be given access to the configuration data. If the configuration data is stored under HKEY_CURRENT_USER, an ACL does not need to be configured because the access to the configuration data is automatically restricted based on process identity. It is also important for developers to be aware of the ways to read securely from the registry. Code snippet below demonstrates how to read an encrypted database connection string from the HKEY_CURRENT_USER key using the Microsoft.Win32.Registry class.

```
using Microsoft.Win32;
public static string GetEncryptedConnectionString()
{
```

```
return (string)Registry.
CurrentUser.
OpenSubKey(@"SOFTWARE\YourApp").
GetValue("connectionString");
}
```

4.3.1.9 Data Access

When writing code that performs database access developers need to be aware of the ways to manage database connection strings securely and to validate SQL statements in order to prevent SQL injection attacks. Developers must also be aware of the ADO.NET permission requirements. We will come back to this topic later on in the paper.

4.3.1.10 Interacting with Unmanaged Code

It is possible that a .NET assembly developer will need to interact with code that predated the .NET Framework and therefore constitutes unmanaged code. Unmanaged code of course does not provide all of the safe guards of managed code, so special care should be taken to ensure secure interaction. When unmanaged code is called, it is important to have the managed code validate each parameter passed to the unmanaged API in order to protect against possible buffer overflows. It is also important to be careful when dealing with output parameters passed back from the unmanaged API. As a matter of good practice, calls made to unmanaged code within an assembly should be isolated in a separate wrapper in order to allow sandboxing of highly privileged code and isolate code access security permission requirements to a specific assembly. Microsoft provides the following recommendations for secure interaction with unmanaged API calls: validate input and output string parameters, validate array bounds, check file path lengths, compile unmanaged code with the /GS switch and inspect unmanaged code for insecure APIs.

In order to reduce the risk of buffer overflows, string parameters passed to unmanaged APIs should be thoroughly validated. The lengths of input strings passed as parameters to the unmanaged API from inside the assembly wrapper should be validated against expected lengths by the formal arguments in the unmanaged code APIs. In some cases (where a formal argument is a character pointer for instance), it may not be known what the safe length of the input string is without having access to the source code. This is demonstrated by the code snippet below.

```
void SomeFunction( char *pszInput )
{
char szBuffer[10];
// Look out, no length checks. Input is copied straight into the buffer
// Check length or use strncpy
strcpy(szBuffer, pszInput);
. . .
}
```

In the example above, an assembly developer would need to have access to the unmanaged source code to know that he should not pass a string longer than ten characters to SomeFunction method because otherwise a buffer overflow would occur in buffer szBuffer as the consequence of strcpy operation. This can also be tied in with the recommendation for inspection of unmanaged code for insecure APIs. For instance, in this example, the use of strcpy is insecure. strncpy should be used instead because it allows specifying the maximum number of bytes to copy. If the source code cannot be examined (if the developer's organization does not own it for instance), then an assembly developer should manually test the API by trying to pass inputs of varying length to the method and observing what happens to the stack in memory. If StringBuilder is used to receive strings from the unmanaged API, the code should make sure that the string can hold the longest possible value that can come from the unmanaged API.

All array bounds should be validated for index out of bounds conditions. If an input to unmanaged API is an array, an assembly coder should ensure that the array does not contain more information than the size with which it has been allocated. In the event that an unmanaged API accepts a file path, the wrapper in the assembly should verify that the path length does not exceed 260 characters (MAX_PATH constant). If the developer owns the unmanaged code, it should be compiled using the /GS switch in order to enable checks to detect buffer overflows. Unmanaged code should also be examined for insecure APIs, such as strcpy for instance. If unmanaged code is owned by the organization, all insecure APIs should be replaced by their safe alternatives (e.g. strncpy instead of strcpy).

4.3.1.11 Delegates

In .NET delegates are managed code equivalents of type safe function pointers and are used by the .NET Framework as a way to support events by maintaining a reference to a method that gets called when the delegate is invoked. Since events may have multiple methods registered as event handlers, all of the event handlers get called when the event happens. The key point to remember about delegates is not to accept them from untrusted sources. If a delegate or an event is publicly exposed by the assembly, any code can associate a method with the delegate. In this case, there is no way of knowing what that code does. The security axiom here is not to accept delegates from untrusted sources. If the assembly is strong named, only full trust callers can pass a delegate to the assembly (unless AllowPartiallyTrustedCallersAttribute is specified). In the event that partially trusted callers are allowed, a delegate could also get passed by malicious code.

4.3.1.12 Serialization:

The concept of serialization is used in both .NET and J2EE. Every class that needs to be marshaled by value between application domains, processes or computers needs to implement a serializable interface. In .NET, serialization support for a class is required if it needs to be marshaled by value across a .NET remoting boundary or if it needs to be persisted in the object state to create a flat data stream. In .NET, classes can be serialized only if they are marked with SerializableAttribute of if they derive from ISerializable.

The things that assembly developers who write serializable classes need to remember are to not serialize sensitive data and to validate serialized data streams.

As a general rule, classes that contain sensitive data should not be serialized. If they need to be serialized, however, it is important to not serialize the fields containing sensitive data. To accomplish this, ISerializable interface can be implemented to control serialization behavior or alternatively the fields containing sensitive data that should not be serialized can be tagged with the [NonSerialized] attribute. A code example of how this can be done is shown below.

```
[Serializable]
public class Employee {
// OK for name to be serialized
private string name;
// Prevent salary being serialized
[NonSerialized] private double annualSalary;
. . .
}
```

If sensitive data items must be serialized, it is possible to encrypt them first. The drawback of this approach is that the code de-serializing the object would then need to have the decryption key.

As object instances are created from serialized data streams (de-serialization), developers should make sure to check that the streams contain valid data in order to detect possibly malicious data being injected into the object. The code below shows how to validate each field as it is being de-serialized.

```
public void DeserializationMethod(SerializationInfo info, StreamingContext cntx)
{
string someData = info.GetString("someName");
// Use input validation techniques to validate this data
}
```

If partial trust callers are supported, then malicious code might pass a serialized data stream or malicious code may attempt to serialize some data on the assembly object. Additional checks are required there that we will cover later in this paper.

4.3.1.13 Issues with Threading

Multi-threaded applications are often subject to race conditions that can result in security vulnerabilities, application crashes, and other problems related to timing. Developers of multi-threaded assemblies should follow the following recommendations: not caching the results of security checks, considering impersonation tokens, synchronizing static class constructors and synchronizing dispose methods.

It is not a good idea for multi-threaded code to cache the results of security checks because the code may be vulnerable, as demonstrated by the example below.

```
public void AccessSecureResource()
{
_callerOK = PerformSecurityDemand();
OpenAndWorkWithResource();
_callerOK = false;
}
private void OpenAndWorkWithResource()
{
if (_callerOK)
PerformTrustedOperation();
else
{
PerformSecurityDemand();
PerformTrustedOperation();
}
}
```

If OpenAndWorkWithResource can be invoked by a separate thread on the same object, it is possible that the second thread would omit the security demand checks because it may see_callerOK=true set by the first thread. This is a classic Time of Check Time of Use (TOCTOU) problem where a race condition leaves a window of vulnerability which can lead in this case to circumvention of security checks. This situation may be resolved by not caching security checks or by providing locking mechanisms to enforce the critical section.

When a new thread is created, it takes on the security context defined by the process level token. If a parent thread is impersonating while it creates a new thread, the impersonation token is not passed to the new thread. If static class constructors are used, the code should ensure through synchronization that they are not susceptible to race conditions to avoid potential vulnerabilities. To avoid issues with freeing a resource more than once, Dispose implementations should be synchronized in multi-threaded environments. The example below is susceptible to this kind of problem.

```
void Dispose()
{
if (null != _theObject)
{
ReleaseResources(_theObject);
_theObject = null;
}
}
```

In the example above, the second thread may evaluate the conditional statement before the first thread has set _theObject to null, thus resulting in freeing a resource twice. This can create various security vulnerabilities, depending on how ReleaseResources is implemented. A solution to this issue would be to synchronize the Dispose method.

4.3.1.14 Reflection

Reflection is another is a concept applicable to both J2EE and .NET. Reflection allows dynamic loading of assemblies, discovery of information about types, and execution of code. De-serialization is typically supported through the use of reflection. Reflection is also critical in .NET remoting. Reflection also allows to obtain a reference to an object and invoke get/set methods on its private members. Developers should make sure that when using reflection to reflect on other types only trusted code can call the assembly. Code access security permission checks should be used to authorize the calling code. If assemblies are loaded dynamically, through use of System.Reflection.Assembly.Load, for instance, assembly or type names should not be used if they were passed from untrusted sources. Particularly when the caller has a lower trust level than the assembly generating the code, developers should ensure that if the assembly dynamically generates code to perform operations for the caller that the caller has no way to affect the code that is being generated. In the event that code generation relies on the input passed by the caller, assembly developers should make sure to validate input strings used as a string literal in the code being generated and escape quotation mark characters to ensure that the caller cannot break out of the literal in order to inject code. There should be no way for the caller to influence code generation or security vulnerabilities are likely.

4.3.1.15 Obfuscation

We already mentioned that it is fairly easy to use a decompiler tool on MSIL code of the assembly. This can result in loss of intellectual property since source code can easily be recovered. An obfuscation tool can be used to make this decompilation extremely difficult. In general, security solutions should not rely on obscurity, which is precisely what code obfuscation is. However, obfuscation is fairly successful against threats related to reverse engineering. Obfuscation tools tend to obscure code paths, change the names of internal member variables and encrypt strings. This has the affect of making it harder for an attacker attempting to reverse engineer MSIL to crack the security logic, understand the code, and search for specific strings in order to identify key sensitive logic. Obfuscation tools also introduce extra instructions into MSIL code, not part of the original assembly code, that does not do anything but makes it very confusing to perform reverse engineering.

4.3.1.16 Cryptography

Cryptography is vital to security for obvious reasons. Encryption can be used to protect data confidentiality, hashing can be used to protect integrity by making it possible to detect tampering, and digital signatures can be used for authentication. Cryptography is typically used to protect data in transit or in storage. The two biggest mistakes that developers can make related to cryptography are: using homegrown cryptographic solutions and not properly securing encryption keys. Developers need to pay special attention to the following issues in order for cryptography to be effective: using cryptographic services provided by the platform, secure key generation, secure key storage, secure key exchange, and secure key maintenance.

It is never a good idea for developers to use custom security solutions since they are almost guaranteed to be weaker than the industry standard. Instead, managed code developers should use algorithms provided by the System.Security.Cryptography namespace for encryption, decryption, hashing, random number generation, and digital signatures. We have already previously discussed the System.Security.Cryptography namespace. Many of the types in this namespace actually wrap around the CryptoAPI provided by the operated system.

During the key generation phase, developers must make sure that the keys generated are truly random, that PasswordDeriveBytes is used for password encryption and that the key sizes are sufficiently large. For programmatic generation of encryption keys, RNGCryptoServiceProvicer should be used for key creation and initialization vectors. Random class should never be used because it does not provide sufficient entropy and produces a reliably identical stream of random numbers for a given seed. Thus with the same seed, the random number stream is known when using the Random class. On the other hand, RNGCryptoServiceProvider creates cryptographically strong random numbers that are FIPS-140 compliant. The code below demonstrates how secure keys can be generated with RNGCryptoServiceProvider.

```
using System.Security.Cryptography;
. . .
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
byte[] key = new byte[keySize];
rng.GetBytes(key);
```

System.Security.Cryptography.DeriveByte namespace can be used to encrypt data that is based on a password that the user supplies. This can be accomplished via PasswordDeriveBytes methods. The reason that we need to have a separate way to encrypt data on user supplied input (like password) is because user input does not tend to be truly random and will not have the same level of randomness as a key generated with RNGCryptoServiceProvider. In order to perform decryption, the user would have to provide the same password as that which was used to encrypt. For password authentication, a password verifier may be stored as a hash value with a salt value. PasswordDeriveBytes takes in password, salt, encryption algorithm, hashing algorithm, key size in bits, and initialization vector data (if symmetric key algorithm is used) as arguments. After the key is used to encrypt the data, it should be cleared from memory, but salt and initialization vector should be stored securely, since they are needed for decryption.

Large keys are preferable to small keys since security is in the key. When generating encryption keys or key pairs, the largest possible key size should be used that a given algorithm would accommodate. Sometimes smaller key sizes may be necessary for different reasons (such as performance for instance), so the key size may be a judgment call made by the application designer or the developer. Larger keys do not enhance the security of the algorithm itself, but increase the amount of time it would take to perform a brute force attack on the key. The code snippet below demonstrates a way in which the largest supported key size for a given encryption algorithm can be found.

```
private int GetLargestSymKeySize(SymmetricAlgorithm symAlg)
{
KeySizes[] sizes = symAlg.LegalKeySizes;
return sizes[sizes.Length].MaxSize;
}
private int GetLargestAsymKeySize(AsymmetricAlgorithm asymAlg)
{
KeySizes[] sizes = asymAlg.LegalKeySizes;
return sizes[sizes.Length].MaxSize;
}
```

Ideally key management should be performed by a platform provided solution and not programmatically as part of the application. However, when encryption keys need to be stored, it is imperative to do so securely by storing then in a secure location. Microsoft recommends using DPAPI, a native encryption/decryption feature provided by Microsoft, Windows 2000 for key management. With DPAPI, the encryption key is managed by the operating system because it is created from the password that is associated with the process account calling the DPAPI functions. Encryption with DPAPI can be performed using a user key or a machine key. User key is the default, meaning that only a threat that runs under the security context of the user account that encrypted the data can decrypt the data. Alternatively, DPAPI can use the machine key. This can be accomplished by passing the CRYPOPROTECT_LOCAL_MACHINE flaw to the CryptProtectData API and then any user on the current machine can decrypt the data. The user key option requires a loaded user profile in the account used to perform the encryption. This somewhat limits portability, and so machine key should be used where portability is required. If machine key option is used, an ACL is required to secure the encrypted data. An optional entropy value can be passed to DPAPI if added security is desired. Of course then the entropy value has to also be managed. Alternatively, machine key can be used without an entropy value and then code access security can be used to validate users and code prior to calling the DPAPI code.

An axiom of key management states that keys should never be stored in code. Hard coded keys in the compiled assembly (MSIL) can be easily disassembled which would eliminate any benefits provided by cryptography. Additionally, access to stored keys should obviously be limited. Appropriate ACLs should be used to limit access to the key when keys are stored in a persistent storage to be used as the application is running. Access to the key should only be allowed to Administrators, SYSTEM, and the identity of the code at runtime (e.g. ASPNET identity). When backing up keys, they should be encrypted with DPAPI or a strong password and placed on removable media.

Key exchange is traditionally a hard problem to solve in any cryptosystem. The most widely used solution is usage of PKI to distribute symmetric keys. A symmetric key that needs to be exchanged is encrypted with the other party's public key that is obtained from a certificate that is valid. Valid certificates are not outdated, contain verifiable signatures along the certificate chain, are of correct type, verified up to a trusted certificate authority, and are no in the Certificate Revocation List (CLR) of the issuer. If an enterprise web application needs to engage in key exchange, assembly code must perform all of the above steps in order to perform secure key exchange. Sometimes it is hard for developers to remember to get all of the steps right, particularly since developers are usually not security professionals. A common problem is with the use of SSL, where proper authentication is not performed prior to communication. The net effect is secure communication with unauthenticated party.

Finally, keys must be securely maintained, which usually involves replacing the keys periodically and protecting exported private keys. Using the same key for a prolonged period of time is not a good strategy. It is also possible for keys to become compromised, either through theft, loss, or other methods. If the private key is compromised that is used for key exchange, users of the public key should be immediately notified that the key has been compromised. All documents digitally signed with the compromised private key need to be re-signed. If the private key that is used for certificate is compromised, the CA should be notified so that the certificate can be place on the CRL and key storage should be reevaluated. Exported private keys should be protected. PasswordDeriveBytes can be used to securely export RSA or DSA private keys. ToXmlString method in RSA and DSA classes can be used to export the public or private key (or both) from the key container, but it exports the keys in plain text. In order to export a private key securely, the key should be encrypted with PasswordDeriveBytes after exporting the key. The code snippet below shows how to use PasswordDeriveBytes to generate a symmetric key securely.

```
PasswordDeriveBytes deriver = new PasswordDeriveBytes(<strong password>, null);
byte[] ivZeros = new byte[8];//This is not actually used but is currently
required.
//Derive key from the password
byte[] pbeKey = deriver.CryptDeriveKey("TripleDES", "SHA1", 192, ivZeros);
```

4.3.2 Code Access Security

A few additional points of interest on .NET Framework code access security are offered in this section to add to what we have already discussed previously in this paper. As a summary, code access security in .NET is used to constraint code access to system resources and privileged operations. Code access security is solely concerned with code level authorization, independent of the user calling the code and the various authentication issues. Code access security is applied to restrict what the code can do, restrict who can call the code through use of the public key part of the assembly's strong name, and to identify code through use of strong names or hashes. The focus in this section is on specific things that developers should be aware of when dealing with .NET code access security. A diagram below is helpful for understanding how code access security works.

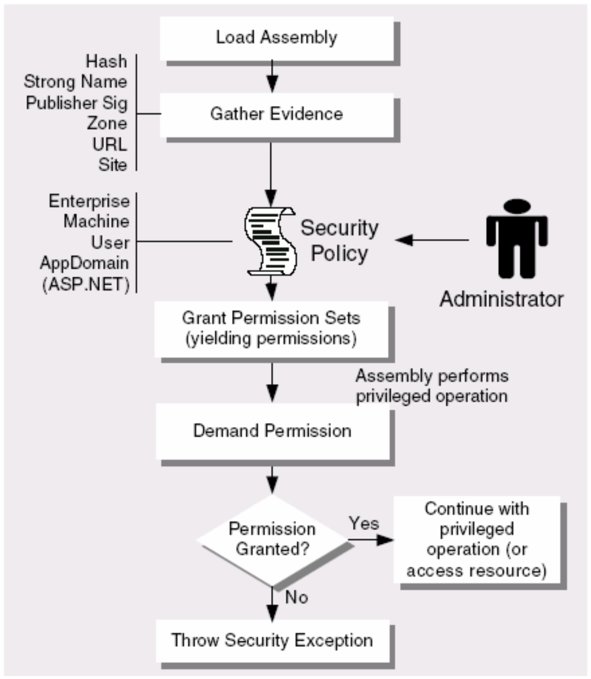


Figure 26: .NET Code Access Security Overview [10]

It is important to not that all of the .NET Framework classes that access resources or perform privileged operations need to contain appropriate permission demands. For instance, FileStream class demands the FileIOPermission and the Registry class demands the RegistryPermission. Web services and HTTP internet resources demand WebPermission.

Assemblies that have strong names cannot be called by partial trust assemblies by default because the default demand for calls is that of full trust. However, this default can be overwritten by specifying as follows: [assembly: AllowPartiallyTrustedCallersAttribute()]. This ensures that developers do not inadvertently allow partial trust and leave an assembly exposed to malicious code. As a general security principle, the use of partial trust code should be limited. Only use it when absolutely necessary and scrutinize the code that allows partial trust for any security problems. It is not uncommon for enterprise web applications to extend trust to partial trust users and it should be done very carefully.

The threat modeling of the application should identify a list of all resources accessed by the application and all of the privileged operations performed by the code. Since permission requirements are configured at deployment time, developers can provide assembly level declarative security attributes that specify minimum permission requirements necessary for their code. These would be place in Assemblyinfo.cs or Assemblyinfo.vb files. Requesting minimum permissions helps enforce least privilege principle. To accomplish this task, Security.Action.RequestMinimum method can be used with declarative permission attributes. A code snippet below demonstrates how to request minimum permissions for code that that needs to access the registry, but only needs to retrieve configuration data from a specific key and nothing else,

```
[assembly: RegistryPermissionAttribute(
SecurityAction.RequestMinimum,
Read=@"HKEY_LOCAL_MACHINE\SOFTWARE\YourApp")]
```

Even if an assembly known to run in a full trust environment, specification of minimum required security permissions is a good practice. If SecurityAction.RequestOptional method is used, then only the permissions requested through RequestMinimum and RequestOptional will be granted and no others, even if otherwise more permissions would have been granted to the assembly. SecurityAction.RequestRefuse can be used to explicitly disallow certain permissions to the assembly code. A code snippet below demonstrates how to explicitly disallow invocation of unmanaged code.

```
[assembly: SecurityPermissionAttribute(SecurityAction.RequestRefuse,
UnmanagedCode=true)]
```

Threat modeling process, architecture and design of the applications should provide developers with a complete understanding of permissions that their code requires. Developers should then request only the minimum permissions required and explicitly disallow all other permission. For instance, a way to do this would be to use the RequestMinimum attribute and an empty RequestOptional attribute.

For authorization purposes, it is important to restrict what code can call the assembly code. Explicit code access permission demands can be used to ensure that the code calling an assembly has the necessary permissions to access the resource or perform a privileged operation exposed by an assembly. As an alternative, identity permissions can be used to restrict the calling code based on identity evidence (e.g. public key derived from the strong name). This will only work for strong named assemblies.

Developers need to know how to restrict which code can call their code. Public methods can be called by any code and so restrictions are needed there. An example below demonstrates use of code access security identity permission demand to provide the restriction. A specified public key is required in order to gain access to the method.

```
public sealed class Utility
{
// Although SomeOperation() is a public method, the following
// permission demand means that it can only be called by assemblies
// with the specified public key.
[StrongNameIdentityPermission(SecurityAction.LinkDemand,
PublicKey="00240000048...97e85d098615")]
public static void SomeOperation() {}
}
```

To restrict unforeseen insecure extension of base classes, inheritance should be restricted unless it needs to be allowed. This can be done with an inheritance demand and a StrongNameIdentityPermission. An example of how this can be done is provided below.

This would prevent inheritance of the base class from any assembly that is not signed with the private key that corresponds to the public key in the demand.

```
// The following inheritance demand ensures that only code within the
// assembly with the specified public key (part of the assembly's strong
// name can sub class SomeRestrictedClass
[StrongNameIdentityPermission(SecurityAction.InheritanceDemand,
PublicKey="00240000048...97e85d098615")]
public class SomeRestrictedClass
{
}
```

Web applications often use data caching for performance reasons and cached data should be protected. If the data is cached, the same permission demands should be applied when accessing cached data as those that were used originally in order to ensure that the calling code is authorized to access the resource. For instance, if data was read from a file, and then cached, a FileIOPermission demand should be used when accessing the cached data. A code sample demonstrating how this can be done is shown below.

```
// The following demand assumes the cached data was originally retrieved from
// C:\SomeDir\SomeFile.dat
new FileIOPermission(FileIOPermissionAccess.Read,
@"C:\SomeDir\SomeFile.dat").Demand();
// Now access the cache and return the data to the caller
```

Custom resources should be protected with custom permissions. In other words, if a resource is exposed through say unmanaged code, the wrapper code that accesses unmanaged code should be sandboxed and custom permission demanded to authorize the calling code. Full trust users would get the permission by default if the permission type of the custom permission implements the IUnrestrictedPermission interface. Partial trust users would have to be explicitly granted the permission ensuring that untrusted code cannot call the assembly and access the exposed custom resources. UnmanagedCodePermission should never be used because that would allow all users to call the exposed unmanaged code resources.

Enterprise web application often use directory services (e.g. LDAP). By default, code using classes from System.DirectoryServices namespace to access directory services is granted full trust. However, DirectoryServicesPermission can be used to constraint the type of access and the particular directory services that the code can access. To constrain a directory service access, DirectoryServicesPermissionAttribute can be used along with SecurityAction.PermitOnly. This guarantees that the code can only connect to a specific LDAP path and can only perform an action of browsing the directory. A code sample to accomplish this is shown below.

```
[DirectoryServicesPermissionAttribute(SecurityAction.PermitOnly,
Path="LDAP://rootDSE",
PermissionAccess=DirectoryServicesPermissionAccess.Browse)]
public static string GetNamingContext(string ldapPath)
{
DirectorySearcher dsSearcher = new DirectorySearcher(ldapPath);
dsSearcher.PropertiesToLoad.Add("defaultNamingContext");
dsSearcher.Filter = "";
SearchResult result = dsSearcher.FindOne();
return (string)result.Properties["adsPath"][0];
}
```

In order to document the permission requirements for directory services access, DirectorySericesPermissionAttribute can be used with SecurityAction.RequestMinimum. This way, the assembly will not load unless sufficient permissions for directory services access are available. This can be done as shown below.

```
[assembly: DirectoryServicesPermissionAttribute(SecurityAction.RequestMinimum,
Path="LDAP://rootDSE",
PermissionAccess=DirectoryServicesPermissionAccess.Browse)]
```

Assembly code may also need to read or write environment variables using the System.Environment class and must be granted EnvrionmentPermission. The permission type is used to constrain access to environment variables with specific names.The code below demonstrates how to constrain access to environment variables.

```
[EnvironmentPermissionAttribute(SecurityAction.PermitOnly, Read="username")]
[EnvironmentPermissionAttribute(SecurityAction.PermitOnly, Read="userdomain")]
[EnvironmentPermissionAttribute(SecurityAction.PermitOnly, Read="temp")]
public static string GetVariable(string name)
{
return Environment.GetEnvironmentVariable(name);
}
```

Permissions required for environment variable access can be specified in the assembly's deployment descriptor as follows:

```
[assembly: EnvironmentPermissionAttribute(SecurityAction.RequestMinimum,
Read="username"),
EnvironmentPermissionAttribute(SecurityAction.RequestMinimum,
Read="userdomain"),
EnvironmentPermissionAttribute(SecurityAction.RequestMinimum,
Read="temp")]
```

Enterprise web applications would often use web services. Web services require the WebPermission from the code access security policy. WebPermission can be used to constraing access to any HTTP Internet based resources, something of great importance for a web application. The sample below demonstrates how to constrain web service connections. This code ensure that the PlaceOrder method and any method called by it can only invoke Web services on the http://somehost site.

```
[WebPermissionAttribute(SecurityAction.PermitOnly,
ConnectPattern=@"http://somehost/.*")]
[EnvironmentPermissionAttribute(SecurityAction.PermitOnly, Read="USERNAME")]
public static void PlaceOrder(XmlDocument order)
{
PurchaseService.Order svc = new PurchaseService.Order();
// Web service uses Windows authentication
svc.Credentials = System.Net.CredentialCache.DefaultCredentials;
svc.PlaceOrder(order);
}
```

ConnectPattern property of the WebPermissionAttribute takes a regular expression that matches the range of addresses to which a connection is allowed. An example below demonstrates how the Connect attribute can be used to explicitly restrict connections to a particular Web service (order.asmx in this case).

```
[WebPermissionAttribute(SecurityAction.PermitOnly,
Connect=@"http://somehost/order.asmx")]
```

ADO.NET data access to SQL Server data supports partial trust callers. Most other data providers, such as Oracle and ODBC providers require full trust callers. To connect to SQL Server data access requires the SqlclientPermission. This permission can be used to restrict the allowed range for name/value pairs in the connection string passed to the SqlConnection object. The code sample below also demonstrates how to perform a check to ensure that blank passwords cannot be used in a connection string. The code throws a SecurityException when a blank password is encountered.

```
[SqlClientPermissionAttribute(SecurityAction.PermitOnly,
AllowBlankPassword=false)]
public static int CheckProductStockLevel(string productCode)
{
// Retrieve the connection string from the registry
string connectionString = GetConnectionString();
. . .
}
```

It is important to control access to sockets in a web based enterprise application. Sockets in .NET may be used directly by the code through the System.Net.Sockets.Socket class and must be granted a SocketPermission. If the code uses DNS to map host names to IP address, a DnsPermission is also required. SocketPermission can be used to restrict access to certain ports on specified hosts. It is possible to specifywhether the socket is inbound or outbound. Transport protocol can also be specified (e.g. TCP, UDP). To constrain code so that it only uses sockets in a way that is determined by the application's security policy, a SocketPermissionAttribute can be used with the SecurityAction.PermitOnly. The code below shows how to connect only to a specific port on a specific host using the TCP protocol. Since Dns.Resolve is called by the code to resolve a host name, the DnsPermission is needed.

```
[SocketPermissionAttribute(SecurityAction.PermitOnly,
Access="Connect",
Host="hostname",
Port="80",
Transport="Tcp")]
[DnsPermissionAttribute(SecurityAction.PermitOnly, Unrestricted=true)]
public string MakeRequest(string hostname, string message)
{
```

```
Socket socket = null;
IPAddress serverAddress = null;
IPEndPoint serverEndPoint = null;
byte[] sendBytes = null, bytesReceived = null;
int bytesReceivedSize = -1, readSize = 4096;
serverAddress = Dns.Resolve(hostname).AddressList[0];
serverEndPoint = new IPEndPoint(serverAddress, 80);
socket = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
bytesReceived = new byte[readSize];
sendBytes = Encoding.ASCII.GetBytes(message);
socket.Connect(serverEndPoint);
socket.Send(sendBytes);
bytesReceivedSize = socket.Receive(bytesReceived, readSize, 0);
socket.Close();
if(-1 != bytesReceivedSize)
{
return Encoding.ASCII.GetString(bytesReceived, 0, bytesReceivedSize);
}
return "";
}
```

Requirements for socket permissions can be specified by using an assembly level SocketPermissionAttribute and a DnsPermissionAttribute with SecurityAction.RequestMinimum (to specify minimum required permissions). This is demonstrated by the code below.

```
[assembly: SocketPermissionAttribute(SecurityAction.RequestMinimum,
Access="Connect",
Host="hostname",
Port="80",
Transport="Tcp")
DnsPermissionAttribute(SecurityAction.PermitOnly, Unrestricted=true)]
```

In this section we have considered some of the nuts and bolts of .NET code access security and how it can be applied to constrain code access to various resources. We now shift gears and discuss some of the specific threats on enterprise web applications developed with ASP.NET, the vulnerabilities in the code that those threats exploit, and how they can be programmatically mitigated.

4.3.3 Building Secure ASP.NET Web Applications

Now that we have covered many of the fundamentals of the .NET Framework security we are ready to consider the various threats on web applications built with ASP.NET, the vulnerabilities those threats attempt to exploit, as well as programmatic compensating controls. An exhaustive list of threats and mitigations will not be provided here. Instead, we will focus on some of the more important threats and see how they can be mitigated. More complete descriptions of various web application threats and mitigations across the different application tiers can be found in the next chapter. The discussion in this section is geared more towards the threats and mitigations at the Web tier, since it is the front line for attackers.

If this section had to be summarized in three words, these words would be: perform input validation! Many attacks rely on malicious input being passed to the application as part of the HTTP request in hopes of forcing an application to perform unauthorized operations or disrupt availability. Consequently, careful input validation is a must and it helps mitigate many threats facing web applications, including cross site scripting, SQL injection, code injection, etc. Some of the top threats include code injection, session hijacking, identity spoofing, parameter manipulation, network eavesdropping and information disclosure. The diagram below illustrates these major threats on ASP.NET (as well as J2EE) web applications [10].

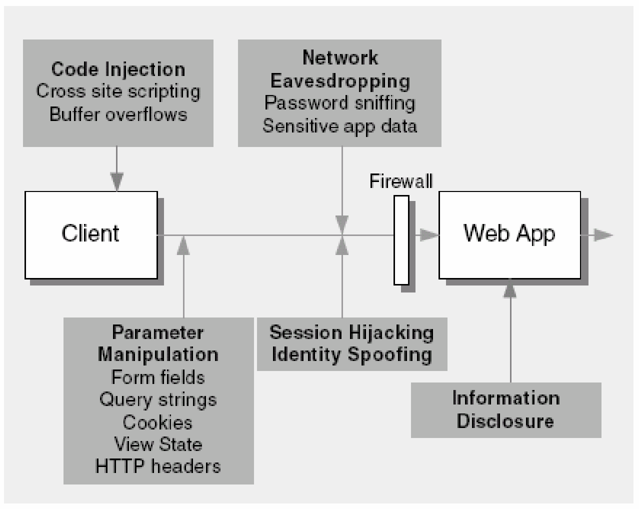
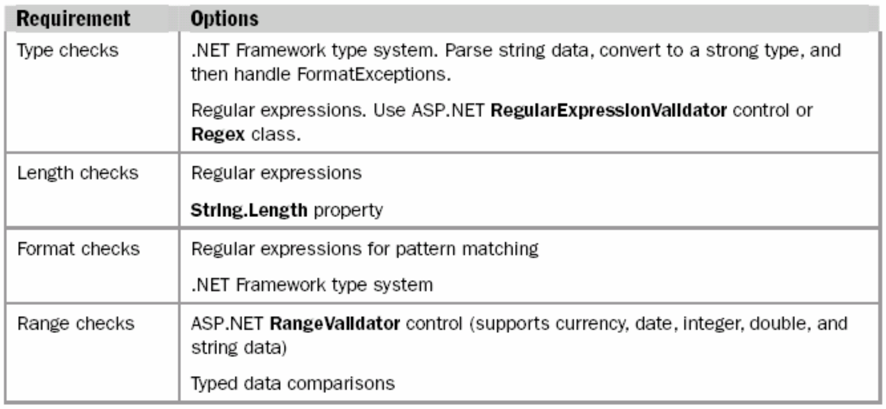


Figure 27: Major threats on an ASP.NET Web Application [10]

4.3.3.1 Input Validation

Since so many problems are the result of poor input validation, some ASP.NET programming techniques are provided here that would help developers to write secure input validation routines. The first step should be to validate input by performing type, length, format and range checks. Validation should be performed against a white list of allowed input, rather than a black list of prohibited input. The table below summarizes .NET classes that facilitate performing type, length, format and range checks.



Requirement	Options
Type checks	.NET Framework type system. Parse string data, convert to a strong type, and then handle FormatExceptions. Regular expressions. Use ASP.NET **RegularExpressionValidator** control or **Regex** class.
Length checks	Regular expressions **String.Length** property
Format checks	Regular expressions for pattern matching .NET Framework type system
Range checks	ASP.NET **RangeValidator** control (supports currency, date, integer, double, and string data) Typed data comparisons

Table 10: ASP.NET Facilities for Input Validation [10]

Regular expressions are an effective mechanism for restricting the range of valid characters, stripping unwanted characters and performing length and format checks. In order to restrict the input, regular expressions can be constructed that the input must match. For that purpose, RegularExpressionValidator control and the Regex class are available from the System.Text.RegularExpressions namespace. Web form input fields can be validated with the RegularExpressionValidator control. If HTML controls are used with no runat=”server” property, then the Regex class is used either on the page class or in a validation helper method [10].

Regular expressions can be used to validate string fields containing names, addresses, social security numbers, etc. In order to do that, first an acceptable range of input characters is defined. Then formatting rules are applied. Patterns can be based on phone numbers, ZIP codes, SSN. Then all the lengths are checked. A code sample below uses RegularExpressionValidator control to validate a name field.

```
<form id="WebForm" method="post" runat="server">
<asp:TextBox id="txtName" runat="server"></asp:TextBox>
<asp:RegularExpressionValidator id="nameRegex"runat="server"
ControlToValidate="txtName"
ValidationExpression="[a-zA-Z'.`-´\s]{1,40}"
ErrorMessage="Invalid name">
</asp:regularexpressionvalidator>
</form>
```

Another example illustrates how Web form fields accepting social security numbers can be validated.

```
<form id="WebForm" method="post" runat="server">
<asp:TextBox id="txtSSN" runat="server"></asp:TextBox>
<asp:RegularExpressionValidator id="ssnRegex" runat="server"
ErrorMessage="Invalid social security number"
ValidationExpression="\d{3}-\d{2}-\d{4}"
ControlToValidate="txtSSN">
</asp:RegularExpressionValidator>
</form>
```

Alternatively, System.Text.RegularExpression.Regex class can be used in the code if server controls are not used.

```
if (!Regex.IsMatch(txtSSN.Text, @"\d{3}-\d{2}-\d{4}"))
{
// Invalid Social Security Number
}
```

You will notice that in the core of these checks are regular expressions. Regular expressions are extremely important from the input validation perspective, and a developer writing input validation logic must be proficient with regular expressions.

Date fields that have equivalent .NET Framework types can make use of the .NET Framework type system. An input value for date can be converted to a variable of type System.DateTime and handle any format exceptions. A code sample to accomplish this is provided below.

```
try
{
DateTime dt = DateTime.Parse(txtDate.Text).Date;
}
// If the type conversion fails, a FormatException is thrown
catch( FormatException ex )
{
// Return invalid date message to caller
}
```

A sanity range check can also be performed on a data field:

```
// Exception handling is omitted for brevity
DateTime dt = DateTime.Parse(txtDate.Text).Date;
// The date must be today or earlier
if ( dt > DateTime.Now.Date )
throw new ArgumentException("Date must be in the past");
```

Numeric data can be validated by converting a string input to an integer form and usring Int32.Parse or Convert.ToIn32 and handling a FormatException that might occur with invalid numeric data types:

```
try
{
int i = Int32.Parse(txtAge.Text);
. . .
}
catch( FormatException)
{
. . .
}
```

In the above example, if txtAge.Text is not a numeric entry and cannot be parsed as an integer, a FormatException will be thrown.

It may be necessary for code to perform validation to ensure that input data falls within a certain concrete range. The code snippet below shows how Regex class can be used to perform the range check.

```
try
{
// The conversion will raise an exception if not valid.
int i = Convert.ToInt32(sInput);
if ((0 <= i && i <= 255) == true)
{
// data is valid, use the number
}
}
catch( FormatException )
{
. . .
}
```

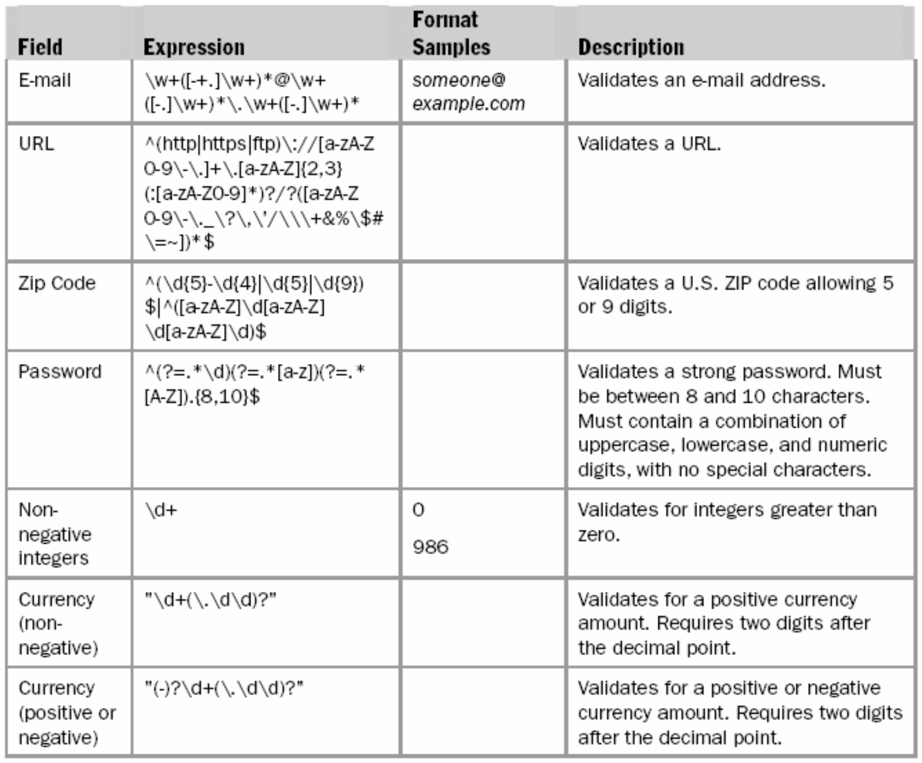
Input needs to be sanitized sometimes. This ensures that even if input was originally malicious, it will be made safe. What this essentially entails is replacing all of the "dangerous" characters. This enforces the defense in depth principle in the event that a regular expression validation is not sufficient. The sanitizer code below strips out possibly dangerous characters that include <, >, \, ", ', ; , ( ), &.

```
private string SanitizeInput(string input)
{
Regex badCharReplace = new Regex(@"([<>""'%;()&])");
string goodChars = badCharReplace.Replace(input, "");
return goodChars;
}
```

It is also critical to validate HTML controls. ASP.NET validator controls cannot be used if server controls are not used. In the case of HTML controls, the content must be validated with regular expression in the Page_Load event handler (when the page first loads). This is demonstrated by the code below.

```
using System.Text.RegularExpressions;
. . .
private void Page_Load(object sender, System.EventArgs e)
{
// Note that IsPostBack applies only for
// server forms (with runat="server")
if ( Request.RequestType == "POST" ) // non-server forms
{
// Validate the supplied email address
if( !Regex.Match(Request.Form["email"],
@"\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*",
RegexOptions.None).Success)
{
// Invalid email address
}
// Validate the supplied name
if ( !RegEx.Match(Request.Form["name"],
@"[A-Za-z'\- ]",
RegexOptions.None).Success)
{
// Invalid name
}
}
}
```

All input used for data access should be thoroughly validated, primarily in order to avoid SQL injection attacks which can happen if dynamic queries are generated based on user input without first thoroughly validating the user input. An attacker can then possibly inject malicious SQL commands that will be executed by the database. To validate input used for dynamic query construction regular expressions should be used to restrict input. For defense in depth the input should also be sanitized. Additionally, whenever possible, it is a good idea to use stored procedure for data access in order to make sure that type and length checks are performed on the data prior to it being used in SQL queries. The table below lists a common list of useful regular expressions that developers should use for input validation.



Field	Expression	Format Samples	Description
E-mail	\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*	*someone@example.com*	Validates an e-mail address.
URL	^(http|https|ftp)\://[a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,3}(:[a-zA-Z0-9]*)?/?([a-zA-Z0-9\-\._\?\,\'/\\\+&%\$#\=~])*$		Validates a URL.
Zip Code	^(\d{5}-\d{4}|\d{5}|\d{9})$\|^([a-zA-Z]\d[a-zA-Z]\d[a-zA-Z]\d)$		Validates a U.S. ZIP code allowing 5 or 9 digits.
Password	^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,10}$		Validates a strong password. Must be between 8 and 10 characters. Must contain a combination of uppercase, lowercase, and numeric digits, with no special characters.
Non-negative integers	\d+	0 986	Validates for integers greater than zero.
Currency (non-negative)	"\d+(\.\d\d)?"		Validates for a positive currency amount. Requires two digits after the decimal point.
Currency (positive or negative)	"(-)?\d+(\.\d\d)?"		Validates for a positive or negative currency amount. Requires two digits after the decimal point.

Table 11: Common Regular Expressions for Input Validation [10]

Developers armed with knowledge of the various ways to perform input validation and skills working with regular expressions are in position to mitigate (and virtually eliminate) application vulnerabilities that may cause buffer overflows, cross site scripting, SQL injection and code injection problems.

4.3.3.2 Cross Site Scripting

Cross site scripting (XSS) attacks are able to exploit vulnerabilities due to lack of proper input validation that allow injection of client-side script code. The code is sent to the application users and since it is downloaded from a trusted site it gets executed by the user's browser. Consequently, security zone restrictions in the browser provide no defense against cross site scripting. A severe case of XSS allows an attacker to steal the user's credentials which subsequently allows an attacker to spoof an identity of a legitimate user. This can be possible when an attacker creates a script that retrieves an authentication cookie used for authentication to a trusted site and then posts the cookie to some external Web site known to the attacker. An attacker then sends the malicious script to the server, the script eventually gets sent to a client, the script is executed by the client browser, an authentication cookie gets stolen and posted to the attacker's web site, and thus allowing an attacker to spoof the legitimate user's identity and gain illegal access to a trusted site. The common defenses against cross site scripting attacks are input validation and output encoding.

The first defense against cross site scripting is input validation as we have already discussed. All input entering the application from outside the application's trust boundary should be validated for type, length, format (via regular expressions) and range. A simple defense against cross site scripting is to encode all server output to a Web site that contains text to make sure that it does not contain any HTML special characters (i.e. <, > and &). This can be accomplished with HttpUtility.HtmlEncode method that replaces < with &lt, > with &gt, and & with &quot. Analogously, URL strings can be encoded with HttpUtility.UrlEncode. This will ensure that no special characters are passed to the client that prompt the browser to execute a script. An example of how to use HtmlEncode method is shown below:

```
Response.Write(HttpUtility.HtmlEncode(Request.Form[“name”]));
```

In ASP.NET data-bound Web controls do not encode output (with the exception of TextBox control when the TextMode property is set to MultiLine). Consequently, any control bound to data that contains malicious XSS code will result in the malicious script execution on the client. As the result, when in doubt, encoding should always be performed before passing the data to the Web controls. Additionally, free format input should always be sanitized. If a Web page contains a free-format text box that allows a comments fields designed to allow safe HTML elements like <b> and <i>, it can be handled properly with first using HtmlEncode and then selectively removing the encoding in places where special characters should be permitted. An example of this is shown below.

```
StringBuilder sb = new StringBuilder( HttpUtility.HtmlEncode(userInput) ) ;
sb.Replace("&lt;b&gt;", "<b>");
sb.Replace("&lt;/b&gt;", "</b>");
sb.Replace("&lt;i&gt;", "<i>");
sb.Replace("&lt;/i&gt;", "</i>");
Response.Write(sb.ToString());
```

There are also a few additional ways to defend against XSS attacks in ASP.NET that are worth mentioning. One option is to set correct character encoding. Constraining the ways in which input data is represented is critical towards restricting what data is valid on the Web pages of the application. In ASP.NET, it is possible to specify the allowed character set either at the page level or at the application level. The default encoding is ISO-8859-1 character encoding. In order to set the character encoding at the page level, two mechanisms can be applied. Both are shown below.

1) <meta http-equiv=”Content Type” content=”text/html; charset=ISO-8859-1” />
2) < @ Page ResponseEncoding=”ISO-8859-1” %>

Alternatively, character encoding can also be specified in the Web.config file, affecting the whole application. This is shown below.

```
<configuration>
        <system.web>
                        <globalization requestEncoding=”ISO-8859-1”
                        responseEncoding=”ISO-8859-1” />
        </system.web>
</configuration>
```

The latter approach has the benefit of consistently guaranteeing proper encoding for all text sent via request and response communications between the application server and the client, thus reducing the chance of a cross site scripting attack.

Another way to defend against XSS attacks is to use regular expressions to validate Unicode characters. This can be done with the code shown below.

```
using System.Text.RegularExpressions;
. . .
private void Page_Load(object sender, System.EventArgs e)
{
// Name must contain between 1 and 40 alphanumeric characters
// together with (optionally) special characters '`´ for names such
// as D'Angelo
if (!Regex.IsMatch(Request.Form["name"], @"^[\p{L}\p{Zs}\p{Lu}\p{Ll}]{1,40}$"))
throw new ArgumentException("Invalid name parameter");
// Use individual regular expressions to validate other parameters
. . .
}
```

Additionally, ASP.NET validateRequest option may be used (available in .NET Framework 1.1) which is set on the <page> element in the Machine.config file. This has the effect of telling ASP.NET to examine all data received from the browser to detect input that may be malicious, such as input containing <script> elements, various special characters (<, >, &), etc. This allows ASP.NET to examine all input received from HTML form fields, cookies, and query strings.

4.3.3.3 SQL Injection

We have already touched on SQL injection issues in this paper and we provide a little more detail on the topic in this section. SQL injection attacks can takes place when input to the application is used to construct dynamic SQL statements that access the database. SQL injection attacks can also occur if code uses stored procedures that are passed strings containing unfiltered input from the user. SQL injections can have the effect of allowing attackers to execute commands in the database and can become even a bigger issue if a process performing database access has excessive privileges. SQL injections can be mitigated by constraining input and using type safe SQL parameters.

Input should be constrained for type, length, format (via regular expressions) and range. If the input comes from a trusted source that has performed input validation, it may not be necessary to validate the input again (although defense in depth principle would recommend that you do so anyway). On the other hand, if the input is from an untrusted source, proper validation on the input should be performed prior to using that input to in SQL statement construction.

The Parameters collection in SQL can be used for input validation where input is treated as a literal value and SQL does not treat it as executable code. Parameters collection in SQL can be used to perform type checking and length validation which also helps enforce type and length checks. Values outside the valid ranges will trigger an exception. Stored procedure should be used where possible and they should be called with Parameters collection. The code below demonstrates how to use the Parameters collection with stored procedures. By themselves, stored procedures may be susceptible to SQL injections if they are passed unfiltered input, but coupled with usage of Parameters collection, the problem goes away.

```
SqlDataAdapter myCommand = new SqlDataAdapter("AuthorLogin", conn);
myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;
SqlParameter parm = myCommand.SelectCommand.Parameters.Add(
"@au_id", SqlDbType.VarChar, 11);
parm.Value = Login.Text;
```

In the case dynamic SQL, where stored procedures cannot be used, Parameters collection should still be used. An example of that is shown below.

```
SqlDataAdapter myCommand = new SqlDataAdapter(
"SELECT au_lname, au_fname FROM Authors WHERE au_id = @au_id", conn);
SqlParameter parm = myCommand.SelectCommand.Parameters.Add("@au_id",
SqlDbType.VarChar, 11);
parm.Value = Login.Text;
```

It is important to note that Parameters collection can also be used when concatenating several SQL statements to send a batch of statements to the server at one time. In this case, the names of the parameters should not be repeated.

Filter routines can be added to protect against SQL injection by replace all “unsafe” characters in the input that have a special meaning to SQL (e.g. a single apostrophe character). A way to do this is shown below.

```
private string SafeSqlLiteral(string inputSQL)
{
return inputSQL.Replace("'", "''");
}
```

However, a skillful attacker could use ASCII hexadecimal characters to bypass these checks, so the use of these filtering routines alone is not sufficient.

4.3.3.4 ASP.NET Authentication

One type of authentication is Form authentication, but it can often be susceptible to compromise through session hijacking and cookie replay attacks. It is also necessary to protect against SQL injection attacks when constructing dynamic database queries using the credentials supplied by the user. We will discuss SQL injection in more detail later on in this paper. It is also critical to enforce strong passwords and follow the principles of secure storage in order to ensure that authentication service is not compromised. The code below demonstrates how to configure Forms authentication properly in the Web.config.

```
<forms loginUrl="Restricted\login.aspx" Login page in an SSL protected folder
protection="All" Privacy and integrity
requireSSL="true" Prevents cookie being sent over http
timeout="10" Limited session lifetime
name="AppNameCookie" Unique per-application name
path="/FormsAuth" and path
slidingExpiration="true" > Sliding session lifetime
</forms>
```

In order to achieve secure Forms authentication with ASP.NET, Microsoft recommends the following solutions: partitioning the Web site, security restricted pages with SSL, using URL authorization, securing authentication cookies, using absolute URLs for navigation and using secure credential management.

Web site partitioning is a simple principle that ties in with the principle of compartmentalization. Basically it means that secure pages that require authenticated access should be placed in a different subdirectory from pages that may be accessed anonymously.

SSL should be used to protect the logon credentials sent from the login form to the server and to send the authentication cookie on subsequent requests to the server. This can be done by configuring the secure folders in the Internet Information Server to require SSL by setting AccessSSL=true attribute for the folder in the IIS metabase. HTTPS must be used the request URL to request pages in the secured folders. Additionally, server certificate must be installed on the Web server in order to use SSL.

URL authorization should also be used in order to ensure secure Forms authentication. To allow anonymous access to public pages, the following code can be used (in Web.config file).

```
<system.web>
<!-- The virtual directory root folder contains general pages.
Unauthenticated users can view them and they do not need
to be secured with SSL. -->
<authorization>
<allow users="*" />
</authorization>
</system.web>
```

To deny access to unauthenticated users and mandate a redirect to the login page , the following code can be added to Web.config file.

```
<!-- The restricted folder is for authenticated and SSL access only. -->
<location path="Secure" >
<system.web>
<authorization>
<deny users="?" />
</authorization>
</system.web>
</location>
```

It is very critical to be able to protect the authentication cookie because this is the token granted to the user after successful authentication and must be protected from being used in session hijacking or cookie replay attacks. Authentication cookies should only be passed over SSL connections using the HTTPS protocol. The cookie should also be encrypted before sending it to the client and the period for which the cookie is valid should be limited. Microsoft provides the following recommendations for protecting authentication cookies: restrict the authentication cookie to HTTPS connections, encrypt the cookie, limit cookie lifetime, use a fixed expiration period, do not persist authentication cookies, keep authentication and personalization cookies separate, and use distinct cookie names and paths.

The secure property of the cookie should be set to ensure that the cookie is sent by the browser only to a secure server page requested via HTTPS URL. This can be accomplished by setting requireSSL="true" on the <forms> element as shown below.

```
<forms loginURL="Secure\Login.aspx" requireSSL="true" . . . />
```

Alternatively, the secure property can also be set manually in the Application_EndRequest event handler in Global.asax file which would enforce this property across the entire web application.

```
protected void Application_EndRequest(Object sender, EventArgs e)
{
string authCookie = FormsAuthentication.FormsCookieName;
foreach (string sCookie in Response.Cookies)
{
if (sCookie.Equals(authCookie))
{
// Set the cookie to be secure. Browsers will send the cookie
// only to pages requested with https
Response.Cookies[sCookie].Secure = true;
}
}
}
```

Even though SSL is used, for additional protection it is a good idea to encrypt the cookie. In order to private privacy and integrity for the cookie, the protection attribute on the <forms> element should be set to “All”, as shown below.

```
<forms protection=”All” … />
```

In the event that cookie becomes stolen, it is a good idea to have a limit on the cookie lifetime to minimize the window of vulnerability. This goes back to the principle of least privilege, where minimum privilege should be granted for the minimum period of time. For instance, the code below shows how to set the cookie timeout for 10 minutes.

```
<forms timeout=”10” … />
```

As an alternative, it is also possible to use fixed expiration periods by setting slidingExpiration=”false” attribute on the <forms> element page to fix the cookie expiration instead of having to reset the timeout period after each Web request.

Cookies should not be persisted since they are stored in the user’s profile and can get stolen if an attacker finds a way to compromise the user’s machine. A non-persistent cookie can be specified when creating the FormsAuthenticationTicket as shown below.

```
FormsAuthenticationTicket ticket =
new FormsAuthenticationTicket(
1, // version
Context.User.Identity.Name, // user name
DateTime.Now, // issue time
DateTime.Now.AddMinutes(15), // expires every 15 mins
false, // do not persist the cookie
roleStr ); // user roles
```

Authentication and personalization cookies should be kept separately because they require different levels of protection. A stolen personalization cookie will reveal to the attacker user preferences and non sensitive information, whereas stolen authentication cookie will allow an attacker to impersonate a legitimate user and access the application.

Distinct names and paths for authentication cookies should be used and specified on the <forms> element. This helps address issues associated with hosting multiple applications on the same server where a user is authenticated in one application and could make request to another application and access the requested page directly, without going through authentication on the logon page.

Absolute URLs should be used for navigation because relative paths are open to an issue where navigations between public and restricted areas of the application occur since redirects use the protocol (whether HTTPS or HTTP) of the current page, and not the target page. This problem is eliminated when using absolute paths.   Code below shows how to redirect to the secure login page using absolute paths.

```
private void btnLogon_Click( object sender, System.EventArgs e )
{
// Form an absolute path using the server name and v-dir name
string serverName =
HttpUtility.UrlEncode(Request.ServerVariables["SERVER_NAME"]);
string vdirName = Request.ApplicationPath;
Response.Redirect("https://" + serverName + vdirName +
"/Restricted/Login.aspx");
}
```

Credential management should always be applied securely which generally means using one-way hashes for passwords, using strong passwords and preventing SQL injection. Secure credential management helps reduce the risk of brute force password attacks, dictionary attacks, and SQL injection attacks.

Instead of storing the actual passwords in the database, or even storing encrypted passwords, a better way is to store one-way password hashes with an added random salt value. Adding the salt value helps reduce the risk of brute force attacks (e.g. dictionary attacks). Using one-way hashes has several important properties. For once, it means that even if an attacker where to get access to the database storing password one-way hashes, he or she would not be able to login to the system using the user password since it would be extremely difficult to calculate the original password that hashed to the stored value (all good one-way hash functions have this property). When validation needs to happen, user's password is obtained, combined with the salt (that needs to be stored), the hash value gets recalculated and gets compared with the value stored in the database.

It is also imperative to use strong passwords that are resistant to dictionary attacks. Passwords should be of appropriate length and have a variety of numbers and characters in them. Ideally they should be generated with a good cryptographic pseudo random number generator. A weak password could compromise the effectiveness of the whole authentication system and thus the application. Regular expressions can be used to ensure that passwords comply with the strong password requirements. An example of how this can be done is provided below.

```
private bool IsStrongPassword( string password )
{
return Regex.IsMatch(password, @"^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,10}$");
}
```

Since user supplied credentials in Forms authentication is used in queries to the database there is a possibility of a SQL injection attack. In order to reduce that risk, all user input can be thoroughly validated with regular expression to ensure that they do not include any SQL characters, parameterized stored procedures can be used to access the database, and login to the database should run in a restricted and least privileged process to limit the amount of damage that can be done in the event of a successful SQL injection attack.

4.3.3.5 ASP.NET Authorization

We already discussed extensively authorization with code access security. Now we make a few points about authorization in the context of ASP.NET enterprise web applications. Authorization can be used as an access control mechanism to directories, individual Web pages, page classes, and methods. Authorization logic can be included in code of the invoked method. Microsoft provides several recommendation for authorization in ASP.NET: use URL authorization for page and directory access control, use file authorization with Windows authentication, use principal demands on classes and methods, and use explicit role checks for fine-grained authorization.

For page and directory level access control, URL authorization can be used. URL authorization can be configured by the <authorization> element in the Web.config file. Access to specific files or directories can be restricted via <location> element nested inside the <authorization> element.

File authorization should be used with Windows authentication since ASP.NET is configured for Windows authentication via FileAuthorizationModule that checks all requests for ASP.NET file types including page files (.aspx), user controls (.ascx), etc. FileAuthorizationModule can be configured via setting appropriate Windows ACLs on the files associated with ASP.NET.

Principal demands should be used on classes and methods that allow making an authorization decision based on the identity and role membership of the caller (part of role-based security in the .NET Framework). The identity of the caller and the caller's role membership are maintained by the principal object that is associated with the current Web request. Access controls on classes and methods can be provided as shown below.

```
// Declarative syntax
[PrincipalPermission(SecurityAction.Demand,
Role=@"DomainName\WindowsGroup")]
public void SomeRestrictedMethod()
{
}
```

Explicit role checks should be used for fine grained authorization. If additional checks inside a method are required to make authorization decisions, either imperative principal permission demands or explicit role checks using IPrincipal.IsInRole should be used. This makes use of runtime variables to allow a more granular authorization decision. An example below shows how an imperative principal permission demand can be used to allow a fine grained authorization.

```
// Imperative syntax
public void SomeRestrictedMethod()
```

```
{
// Only callers that are members of the specified Windows group
// are allowed access
PrincipalPermission permCheck = new PrincipalPermission(
null, @"DomainName\WindowsGroup");
permCheck.Demand();
// Some restricted operations (omitted)
}
```

We now move on to categories of the various threats and mitigations that occur at each of the tiers of an enterprise web based application. Armed with understanding of J2EE and .NET security we can draw from both for examples on how the threats are addressed.

5.0 Threats and Mitigations

5.1 Discussion of Threat Taxonomies

There are a lot of threats facing enterprise web-based applications today that have a potential to disrupt the core security services, namely confidentiality, authentication, authorization, integrity, availability, accountability and non-repudiation. In order to establish compensating controls to counteract these threats, a structured approach is needed for thinking about these threats. In chapter two we have discussed the threat modeling process that attempts to enumerate the various threats facing an application and to consider how these threats can be mitigated. This chapter is designed to aid a threat modeler of an enterprise web-based application to systematically evaluate these threats. This chapter also helps architects, designers and developers of enterprise web-based applications to understand what compensating controls are appropriate at each stage across each of the tiers of the application.

An enterprise web-based application is a multi-tiered application. Since trust boundaries are often drawn between the various tiers, we find it useful to consider the various threats across each of the tiers. This has the effect of enabling a more systematic and structured approach to threat modeling and also makes it clearer where the required compensating controls must be put in place in order to counteract the threats. In this chapter we look at the threats across the web, application and persistence tiers. The focus is on the OWASP top ten web application vulnerabilities, but we also consider additional categories of vulnerabilities [24]. For specific examples on possible mitigations we draw from the discussions in chapter three and four, J2EE and .NET security, respectively.

When we talk about each threat, at each of the tiers, we structure our discussion as follows.

- Overview
- Consequences
- Discussion
- Severity
- Likelihood of Exploit
- Mitigation

Threats on an enterprise web-based applications attempt to exploit a vulnerability or multiple vulnerabilities that may exist in the architecture, design or implementation of the application. At times, a problem may exist at many levels. In this chapter we look at the categories of application vulnerabilities that when exploited, lead to manifestation of the threats that in turn lead to a compromise of one or more of the core security services [11].

It should be noted that it may not always be a clear cut decision as to where a particular threat should be mitigated. Consequently, as we separate the threats across the various tiers, a reader should keep in mind that it might be possible to provide compensating controls for these threats at a different tier. Where the compensating controls are also depends on the particular design paradigm that is chosen for the application. We generally consider the presentation logic of the application to be at the web tier, the core functionality of the application to be at the business logic tier, and the databases for the application to be at the persistence tier. Outlining a complete list of all known application threats is beyond the scope of this chapter. We focus on the major categories of threats for enterprise web-based applications that also map to the OWASP top ten.

5.2 Web Tier Threats and Mitigations

The following threats are discussed for the Web tier of an enterprise web-based application:

- Poor Session Management
- Parameter Manipulation
- Path Traversal and Path Disclosure
- Canonicalization
- URL Encoding

5.2.1 Poor Session Management

Overview:

If proper steps are not taken to protect session tokens and account credentials, such as passwords, keys and session cookies, attackers can use those to defeat authentication checks and assume identities of other users. Authentication mechanisms can be circumvented if authentication credentials and tokens are not properly handled by credential management functions such as those to change password, retrieve forgotten password, account update, etc. Session tokens need to be properly protected against hijacking so that attackers cannot assume identities of authenticated users simply by hijacking the session after the authentication has taken place. Session tokens created should be strong and should be properly protected throughout the lifecycle of the session. Secure Sockets Layer (SSL) technology can go a long way towards creation of a secure session; however SSL is not properly implemented in many instances. Additionally, attacks like cross site scripting can allow an attacker obtain the session tokens even if SSL is used [10, 18, 24].

Consequences:

Poor session management results in the authentication service being compromised which allows an attacker to defeat authentication checks and assume an identity of a legitimate user. This invariably leads to violations of confidentiality, integrity, authorization, accountability and non-repudiation, essentially all the other security services.

Discussion:

Since HTTP is a stateless protocol, it is necessary to manage session tokens that are granted to a user who successfully completed proper authentication. These tokens may come in the form of cookies, static URL's, dynamic URLs. However, the most common session token is a cookie. There are four types of cookies, Persistent/Secure, Persistent/Non-Secure, Non-Persistent/Secure and Non-Persistent/Non-Secure. Persistent cookies are stored on the hardware of the client and are valid until the expiration date. Non-Persistent cookies are stored in RAM on the client and are valid until the browser is closed or until it is explicitly expired by a log-off mechanism. Secure cookies can only be transmitted over the HTTPS protocol used for SSL, while non-secure cookies can be transmitted over HTTPS or HTTP. Secure cookies (encrypted) are only secure during transmission, but not at the end user machine. Whenever SSL is used, secure cookies are required. Breakdown in authentication can occur if session cookies are stolen, if weak passwords are brute forced or if SSL is not properly implemented because some critical authentication steps are skipped. Also, if the client machine becomes compromised by an attacker, then the attacker would gain access to the application [10, 18, 24].

Severity:

High

Likelihood of Exploit:

High

Mitigation:

SSL should be used and be properly implemented. All session cookies should be encrypted. Passwords should be cryptographically strong. Session tokens should periodically expire. Re-authentication should be required for all significant system actions, at which point authentication tokens can be reissued upon successful authentication.

5.2.2 Parameter Manipulation

Overview:

Parameter manipulation attacks involve modification of data sent by the client to the server in a way that compromises one or more security services. Common problems involve attackers being able to modify session tokens, values stored in cookies, form fields, URL Query Strings and HTTP headers. Cryptography does not solve the parameter manipulation problem because it protects data while in transit, whereas parameter manipulation changes the data at the end points.

Consequences:

Parameter manipulation can have serious effects on the web application in terms of compromising authorization, authentication, confidentiality and integrity.

Discussion:

It is often easy to change session cookies. HTTP headers can be manipulated by changing the Referer field which may be used by the server to make authentication and authorization based decisions. HTML forms using hidden fields to hide authentication or authorization tokens will make it very easy for an attacker to compromise the web application. URL manipulation involves changing the URL address in an attempt to gain access to areas of the application for which the user is not authorized.

Severity:

High

Likelihood of Exploit:

High

Mitigation:

Never use hidden form fields for authentication or authorization, applies to both POST and GETS submission. Cookies should be encrypted and hashes maintained in order to prevent tampering. Never use URLs as a basis for granting access.

5.2.3 Path Traversal and Path Disclosure

Overview:

File system of the web server is often used in the presentation tier in order to provide temporary or possibly even permanent storage to information like image files, HTML files, CGI scripts, etc. The usual virtual root directory on the web server is WWW-ROOT and it can be accessed by the clients. In case where an application does not check and deal with meta characters that are used to specify paths, it is possible that an attacker can construct a malicious request that would disclose information about the physical file locations (e.g. /etc/password). This attack is often used in conjunction with other attacks. such as direct SQL or command injections by disclosing information about physical location [11].

Consequences:

This is primarily an information disclosure problem that provides very important information that can greatly facilitate other types of attacks that may compromise various core security services. For instance, if a command injection attack is facilitated by the path disclosure attack, failures in authorization and integrity are likely.

Discussion:

This is primarily an input validation issue and can be resolved by using white lists, regular expression filters and safe replacement functions. We rate this issue with medium severity because it does not lead to a compromise of a core security service by itself. The information disclosed through this attack may facilitate other attack, like command or SQL injection, but another relevant vulnerability must be present in order for that to occur.

Severity:

Medium

Likelihood of Exploit:

Medium

Mitigation:

Path normalization functions provided by the implementation language can be used. Always compare input against a white list. Remove strings of the form "../" as well as unicode representations of these strings. This is fundamentally an input validation issue. When we discussed .NET, we offered an extensive guidance on the various ways in which input validation can be performed.

5.2.4 Canonicalization

Overview:

Canonicalization problems occur because of the way that data is converted between forms. The term canonical means the simplest or standard form of something, and thus canonicalization is the process of converting data in one representation to its simplest form. In Web applications, many canonicalization problems are with URL encoding to IP address translation. A potentially exploitable condition may arise when security decisions are based on canonical forms of data.

Consequences:

Canonicalization issues may result in authentication, authorization and integrity services being compromised. It is a very serious problem that must be taken into account during the implementation.

Discussion:

Attackers will often try to defeat the input validation filters by encoding their input containing illegal characters with Unicode. Since there are a variety of encoding mechanisms, it may be difficult for application developers to check for them all. That is why it is important to perform input validation against a white list of allowable characters and not a black list [18].

Severity:

High

Likelihood of Exploit:

High

Mitigation:

A canonical form should be selected to which all user input should be canonicalized prior to making any authorization decisions based on the input. Security checks should be performed after UTF-8 deciding is completed and the program should check that the UTF-8 encoding is a valid canonical encoding for the represented symbol.

5.2.5 URL Encoding

Overview:

Similar to canonicalization, URL encoding attack involves placing malicious input that has been encoded in the URL that is then parsed by the server. Generally, the range of allowed characters for URLs and URIs is restricted to a subset of the US-ASCII character set. However, the data used by a web application is usually not restricted and can include any existing character set or binary data. Earlier versions of HTML 4.0 specification expanded to permit any character in the Unicode character set. URL-encoding a character can be performed by taking the character's 8-bit hexadecimal code and putting a "%" in front of it. An attacker may choose to encode malicious characters that would have been otherwise caught by an input validation filter. Since URL-encoding allows practically any data to be passed to the server via URLs, the server side code of a web application needs to make sure that no malicious input is allowed.

Consequences:

Accepting malicious input encoded in URLs can lead to serious problems including violations in authorization and integrity.

Discussion:

URL encoding may be used to facilitate both cross site scripting and SQL injection attacks. An example of this is shown below.

Cross Site Scripting Example:

Malicious script (in PHP) that an attacker intends to send to the server:

```
echo $HTTP_GET_VARS["mydata"];
```

This is how an attacker can encode the above script in the URL:

```
http://www.myserver.c0m/script.php?mydata=%3cscript%20src=%22http%3a%2f%2fwww.yourserver.
```

Here is the HTML that will be generated by the server:

```
<script src="http://www.yourserver.com/badscript.js"></script>
```

SQL Injection Example:

Here is an original database SQL query:

```
sql = "SELECT lname, fname, phone FROM usertable WHERE lname='" & Request.
QueryString("lname") & "';"
```

Here is the HTTP request that has been URL encoded by an attacker:

```
http://www.myserver.c0m/search.asp?lname=smith%27%3bupdate%20usertable%20set%20passwd%-%00
```

Here is the SQL query that actually gets executed after the parameters of the query are taken from the URL:

```
SELECT lname, fname, phone FROM usertable WHERE lname='smith';update usertable
set
```

Severity:

High

Likelihood of Exploit:

High

Mitigation:

Proper input validation is necessary and white lists should be used. Since this is partially a canonicalization problem, all user input should be canonicalized into the chosen form before performing any authorization decisions all using the input in any way.

5.3 Business Logic Tier Threats and Mitigations

The following threats are discussed for the Business Logic tier of an enterprise web-based application [10,18, 24]:

- Broken Access Control
- Input Validation Problems
  - Buffer Overflow
  - Cross Site Scripting (XSS)
  - SQL Injection
  - Command Injection
- Insecure Exception Management
- Insecure Use of Cryptography
- Insecure Default Configuration
- Insecure Auditing/Logging

5.3.1 Broken Access Control

Overview:

Broken access control (authorization) problems result when restrictions on what authenticated users are allowed to do are not properly enforced. Application vulnerabilities that fall in this category could allow attackers to access accounts of other users, view confidential information or use unauthorized functionality. There is a variety of attacks that fall into this category that could allow attackers to escalate privileges. For instance, reliance on hidden fields to establish identity for the purpose of access to web based administrative interfaces will allow an attacker unauthorized access because hidden fields can be easily manipulated. Exploitation of some other vulnerability in the application can cause violation in access control. For instance, crafting an attack that exploits a buffer overflow to modify some flag variable used for an authorization check could result in broken access control. Some key access controls issues include insecure ids, forced browsing past access control checks (URL tampering), path traversal, file permissions and client side caching.

Consequences:

Application vulnerabilities that fall into this category could compromise authentication, authorization and confidentiality core security services.

Discussion:

Most problems that result in the failure of an authentication service will also lead to broken access control. For instance, reliance on hidden fields to establish identity may lead to unauthorized access because they are easily manipulated. Vulnerabilities that allow escalation of privilege also belong to this category. These may be buffer overflow, command injection, and SQL injection vulnerabilities since all of them give an attacker elevated level of access to the application and the host machine. We have seen in J2EE and .NET that there are fundamentally two types of access control: role-based access control (RBAC) and code-based access control. With role-based access control, an authorization decision is based on the identity of the caller, whereas with code-based access control it is based on the calling code. There are also two fundamental ways to enforce access control: declarative and programmatic. Declarative access control is managed by the container (J2EE) or the CLR (.NET), whereas programmatic access control is implemented as a series of checks in the application code. Programmers need to be careful when writing code to ascertain that they are not introducing vulnerabilities that could lead to elevation of privilege (e.g. SQL injection, buffer overflow, command injection, format string, etc.). Additionally, the least privilege and compartmentalization principles should always be followed. Privileges should be released when appropriate and no covert channels should be allowed.

Severity:

High

Likelihood of Exploit:

High

Mitigation:

It is a good idea to try and reduce the amount of custom authorization logic that has to be written and instead rely on the authorization services provided by environment. Since most code written for enterprise web-based applications is managed code, it is a good idea to make use of the declarative authorization features that can be specified in a descriptor file and enforced by the managed environment itself. We have already discussed how both .NET and J2EE provide declarative authorization capabilities that can be enforced by the container. In general, if programmers have to write less custom code, there is less of an opportunity for them to introduce security bugs. That is true for authorization code in charge of access control and also true of other code.

5.3.2 Input Validation Problems

Overview:

Unvalidated input is a fairly broad vulnerability category that has very serious consequences. All web based applications need to handle input coming from a variety of untrusted sources (most notably the user of the application). If the input is not validated, attackers can use that opportunity to attack the backend components of the applications. In general, validation needs to be performed each time the data crosses a trust boundary. Validation may be performed on the client site, but for performance purposes early. Client site validation should never be relied upon for security. Validation also needs to happen at the web, application and database tiers. A variety of application level attacks could be avoided if input validation is performed properly; those include SQL injection, Cross Site Scripting (XSS), buffer overflows, format string, cookie poisoning, hidden field manipulation, command injections, etc. Unvalidated input could lead to compromise of authorization, integrity, authentication and availability security services. That is discussed in more detail later in the paper. All user input in HTTP requests should always be aggressively validated against white lists (list of allowable input) as opposed to black lists (list of input that is not allowed) [10,18,24].

Consequences:

Poor input validation can result in a compromise most security services, namely authorization, integrity and availability.

Discussion:

Buffer overflows are a classic example of poor input validation. The reason for this vulnerability is that some programming languages allow the input placed in a buffer to overflow past the allocated range for the buffer into adjacent areas of memory. For example, if a buffer overflow occurs on a stack and the overflow can reach a function return pointer, an attacker who carefully crafts a buffer overflow may overwrite the return pointer with his own value that will make the program to jump to the area where an attacker has injected some attack code and start executing it. There are two main kinds of buffer overflows, stack based and heap based. Both kinds are extremely dangerous because they cause availability problems (in the best case) and cause authorization failures (in the worst case) which can then lead to other problems. Languages like C are particularly prone to buffer overflows because it contains many inherently insecure functions. Luckily, buffer overflows are not a tremendous problem for enterprise web-based application since both J2EE and .NET provide automatic type checking that make buffer overflows very unlikely. However, application developers of enterprise web based applications still need to be very careful when dealing when providing integrations with legacy code.

Cross site scripting attacks exploits vulnerabilities that fall in the category of poor input validation. Essentially an attacker submits executable scripts as part of the input to the web application and those scripts are then executed on the browsers of other clients. Those attacks often lead to information disclosure of the end user's session tokens, attack the end user's machine or spoof content to fool the end user. Disclosure of session tokens can lead to session hijacking and allow an attacker to assume a valid user's identity (compromise authentication). Spoofing content can also lead to information disclosure if for instance a valid user input his/her login and password information into a form sent to an attacker. XSS attacks can occur at the web tier or at the application tier and aggressive white list input validation should be present in the application to thwart these attacks. There are two types of XSS attacks: stored and reflected. In stored XSS attacks, the malicious script injected by an attacker is permanently stored by the web application for later retrieval by the end user who requests the affected data. Since the malicious script at that point arrived from a trusted server, the client executes the script. In reflected attacks, the malicious script is transferred to the server and then is echoed back to the user either in an error message, search result, or some other response to the end user which includes some of the data fields into which a malicious script has been inserted as part of the request.

All input used for data access should be thoroughly validated, primarily in order to avoid SQL injection attacks which can happen if dynamic queries are generated based on user input without first thoroughly validating the user input. An attacker can then possibly inject malicious SQL commands that will be executed by the database. To validate input used for dynamic query construction regular expressions should be used to restrict input. For defense in depth the input should also be sanitized. Additionally, whenever possible, it is a good idea to use stored procedure for data access in order to make sure that type and length checks are performed on the data prior to it being used in SQL queries.

Enterprise web applications pass parameters when they access external systems, applications, or use local OS resources. Whenever possible, those parameters should not come directly from the user and be defined as constants. Otherwise, they should be rigorously validated prior to usage. If an attacker can embed malicious commands into these parameters, they may be executed by the host system when the access routines are invoked by the application. SQL injection is a particularly common and serious type of injection, where SQL statements are passed to the web application and then without validation are passed to the routine that accesses the database with that SQL statement. Command injections can be used to disclose information, corrupt data and pass malicious code to an external system application via the web application [10,18,24].

Severity:

High

Likelihood of Exploit:

High

Mitigation:

The answer to all of these problems is to perform input validation properly. This can be accomplished by validating against white list of allowable input, as opposed to a black list of invalid input. Correct regular expressions should be used to validate input and character replacement functions to make the input safe should be applied for defense in depth. Additionally, encoded characters should be given special attention because attackers will try to encode input in order to fool the validation checks. This will also help prevent canonicalization problems that we have previously discussed.

5.3.3 Insecure Exception Management

Overview:

Errors and exceptions occurring during the operation of the enterprise web application should be handled properly. Error information that is echoed to the user in its raw form can cause information disclosure. For instance, letting an attacker know the OS that the host machine is running, the version of the database and the database driver can allow the attacker to exploit existing vulnerabilities for those technologies. Improperly managed exceptions can result in disruption of availability or cause security mechanisms to fail.

Consequences:

Improper error handling and exception management can lead to information disclosure problems that can facilitate other types of attacks. Availability service might also be compromised.

Discussion:

Poor exception management can lead to information disclosure which may help malicious hackers to compromise the application. Additionally, poor exception management can also result in availability issues by allowing an attacker to launch a denial or service attack. In short, all exceptions should be caught and handled properly. Microsoft provides several recommendations for proper exception management: use structured exception handling, do not log sensitive data, do not reveal system or sensitive application information, consider exception filter issues and consider an exception management framework.

Structured exception management in Visual C# and Visual Basic .NET does not differ from Java. In all of these languages, try / catch and finally constructs are used. It is important to remember to be very granular by catching specific exceptions rather than trying to lump them all into one by catching a generic exception. Structure exception handling guarantees that the system is always in a consistent state. The code snippet below demonstrates a structured approach to exception handling:

```
try
{
// Code that could throw an exception
}
catch (SomeExceptionType ex)
{
// Code to handle the exception and log details to aid
// problem diagnosis
}
finally
```

```
{
// This code is always run, regardless of whether or not
// an exception occurred. Place clean up code in finally
// blocks to ensure that resources are closed and/or released.
}
```

Exceptions tend to contain a lot of detailed information that could help an attacker compromise the system. Consequently, care should be taken when logging exception data and sensitive data should not be logged. The raw output of an exception should never propagate directly to the client (or go outside the application trust boundary for that matter) as it provides too much useful information. Some of these details may include operating system, .NET Framework version numbers, method names, computer names, SQL command statements, connection strings, along with other details. The information may be very useful to an attacker. For instance, knowing what version of the operating system is used, an attacker may exploit a known vulnerability for that operating system. Instead, generic messages about an exception should be returned to the client where appropriate. A lot of the time exception handling might make the exception transparent to the client, in which case nothing may be returned. In general, an exception management framework can ensure that all exceptions are properly detected, logged and processed in a way that avoids both information disclosure and availability issues.

Severity:

Medium

Likelihood of Exploit:

Medium

Mitigation:

Programmers should make sure that all potential problems are caught and properly handled. As we have discussed, there are very good facilities available in both J2EE and .NET platforms for exception management.

5.3.4 Insecure Use of Cryptography

Overview:

Cryptography is vital to security for obvious reasons. Encryption can be used to protect data confidentiality, hashing can be used to protect integrity by making it possible to detect tampering, and digital signatures can be used for authentication. Cryptography is typically used to protect data in transit or in storage. The two biggest mistakes that developers can make related to cryptography are: using homegrown cryptographic solutions and not properly securing encryption keys. Developers need to pay special attention to the following issues in order for cryptography to be effective: using cryptographic services provided by the platform, secure key generation, secure key storage, secure key exchange, and secure key maintenance.

Consequences:

Since cryptography often supports confidentiality, integrity and authentication, insecure use of cryptography could compromise these services.

Discussion:

Attackers will try to compromise the weakest link of the software system and will use all possible techniques, including social engineering attacks. For instance, the weakest link of the system could be a member of technical support falling prey to social engineering. An attacker will not waste time trying to brute force a strong password to get into a system but look for other ways to get in instead. In general, cryptography (if used properly) is rarely the weakest link in the software system, so the attacker is much more likely to attack the endpoints. A good example of this is the notorious design flaw from a security standpoint in the gateway server used by wireless application protocol (WAP) to convert data encrypted with WTLS to SSL. An attacker does not have to try and attack the data while it is encrypted, but just wait for the gateway server to decrypt the data. Consequently, any exploitable flaws in the gateway server software (e.g. buffer overflows) could be used as a way to compromise data confidentiality. Consequently, encryption should be used as part of end-to-end solution and both ends should be properly secured.

SSL is often an example of insecure use of cryptography, where they are used as a drop in for standard sockets where some critical authentication steps are skipped. A server might want to use SSL, but a client may not support it. In this case a client downloads proper socket implementation from the server at runtime. This can be a potentially serious security problem because the server has not yet authenticated itself to the client when the download occurs and thus a client could really be downloading malicious code from a malicious host. In this situation trust is extended where it should not be. The failure occurs when the client fails to establish a secure connection using default libraries, but instead establishes a connection using whatever software it downloads from an unauthenticated and thus untrusted remote host (server). If secure failure principle was followed, the client would first authenticate the server using the default libraries prior agreeing to the download of additional SSL libraries at runtime [10, 18, 24].

Severity:

High

Likelihood of Exploit:

High

Mitigation:

The various available cryptographic APIs to support the core security services were already discussed for both J2EE and .NET. Developers need to be aware of the available APIs, when to use them and how to use them properly. Most importantly, it is important to know the limitations of each cryptographic scheme -- never use custom or insecure cryptographic algorithms and use strong keys.

5.3.5 Insecure Default Configuration

Overview:

Most default configurations on many commercial hardware and software products do not provide a sufficient level of security and must be modified. True to the least privilege principle, configuration should always allow for the minimum privilege necessary. Secure configuration is required on the web, application and database server. In order to securely deploy an enterprise web based application, all configurations must be properly performed. Some configuration problems can be due to unpatched security flaws present in the server software, improper file and directory permissions, error messages providing too much information, improper configuration of SSL certificates and encryption settings, use of default certificates, improper server configurations enabling directory listing and directory traversal attacks, among others.

Consequences:

Insecure default configuration often has an impact on authentication, authorization and confidentiality. However, other security services can also be compromised.

Discussion:

Since default configurations are often not changed, insecure default configurations will increase the likelihood of vulnerabilities. Application developers need to be aware of what the defaults are and configure them properly to match the desired level of security.

Severity:

Medium

Likelihood of Exploit:

High

Mitigation:

Software and platform vendors should do their part by providing more secure default configurations and making it simple to configure different settings. Application developers need to be aware of the default configuration and change it appropriately.

5.3.6 Insecure Auditing/Logging

Overview:

Most enterprise web based application will use event logging in some shape or form and it is very important to ascertain that this is done securely. The fundamental threats to event logs involve tampering with the log, information disclosure of sensitive information stored in the log and log deletions in order to erase tracks. While some event log protection is provided by the security features in the Windows operating system, developers must ensure that event logging code cannot be used by an attacker to gain unauthorized access to the event log.

Consequences:

Compromised logs will compromise non-repudiation and accountability services.

Discussion:

The first axiom of logging should be to not log sensitive data. We have already mentioned this in the secure exception management section. For example, with .NET, if EventLog.WriteEvent is used, existing records cannot be read or deleted. This leads us back to the principle of least privilege. Do not give more privileges to the code updating the event logs than is absolutely necessary and grant those privileges for the shortest amount of time necessary. This can be accomplished by specifying EventLogPermission by way of code access security. A threat that should be addressed is preventing an attacker from invoking the code that does event logging so many times that would cause overwrite in previous log entries. An attacker can try and to this in an attempt to cover his tracks for example. A way to deal with that may be to use an alert mechanism that would signal the problem as soon as the event log approached a limit.

Keeping a record of security-related events that contain information on who has been granted access to what resources is an important tool for promoting accountability. These records can also be vital for recovery once a system has been breached and can also be used for intrusion detection purposes. They can also help track hackers down. Any good hacker will try to delete or modify all records of his or her activities and thus it is extremely important to keep those records secure and tamperproof. The deployer of the application should associate each of the container constraints for component interaction with a logging mechanism. The container can then audit one of the following events: all evaluations where the constraint was satisfied, all evaluations where the constraint was not satisfied, all evaluations regardless of the outcome and no evaluations. All changes to audit logs should also be audited. With J2EE responsibility for auditing and logging is shifted from developers to deployers of the application [10, 18, 24].

Severity:

Medium

Likelihood of Exploit:

Medium

Mitigation:

Encryption should be used to protect system logs. Mechanisms should be provided to detect when logging reaches allocated limit in order to mitigate the risk of logs being overwritten. Additionally, sensitive information should not be logged.

5.4 Persistence Tier Threats and Mitigations .

The following threats are discussed for the Persistence tier of an enterprise web-based application:

- Insecure Storage

5.4.1 Insecure Storage

Overview:

Data spends far more time in storage than it does in transit and must therefore be stored in a secure manner. Encryption of data is not a bad idea to promote confidentiality and protect application data, passwords, keys, etc. Even when encryption is used, it is not often used properly. Some of the common types of mistakes make that fall into this category include failure to encrypt critical data, insecure storage of keys, certificates and passwords, improper storage of secrets in memory, poor sources of randomness, poor choice of cryptographic algorithms, and homegrown encryption algorithms. While encryption can help protect confidentiality of stored data, hashing can be used to ascertain integrity. A central point here is that as little as possible of sensitive information should be stored by the enterprise web applications. For instance, it might make sense to ask the users to reenter their credit card number each time instead of persisting it.

Consequences:

Insecure storage may result in compromise of the confidentiality security service.

Discussion:

It is important to remember that data typically spends far more time in storage than it does in transit, and consequently protecting data during storage from the prying eyes of attackers is essential. In some cases, only hashes of data should stored, like with passwords for instance. Additionally, some data should not be stored at all. An intelligent strategy for data storage is essential.

Severity:

High

Likelihood of Exploit:

Medium

Mitigation:

Data should be encrypted using strong cryptographic algorithms and strong keys. Passwords should be hashed. If data does not have to be stored, it should not be stored.

6.0 Analysis

The primary goal of this paper has been to identify secure coding practices and constructs that remediate application level vulnerabilities which could, if went unmitigated, result in realization of a variety of threats on enterprise web-based applications. As a way of conducting that discussion, we specifically talked about J2EE and .NET platforms for enterprise web application development. While some issues discussed where specific to these frameworks, most of the application security concepts which we have covered actually have a broad application that would transcend beyond these particular technologies. For instance, principles of compartmentalization and least privilege are not tied to any particular implementation.

We have seen the various compensating controls in J2EE and .NET that support the core security services and ensure that they are not compromised by the various threats. We have also discussed how developers need to use various security features that are provided by J2EE and .NET appropriately and looked at certain secure coding guidelines required in order to avoid potentially costly pitfalls. Throughout chapters three and four we pointed out some of the differences and similarities between J2EE and .NET security models and APIs that are available to programmers. We summarize some of this discussion in this section, highlight the key similarities and differences between these two frameworks, and illuminate on some key strengths and weaknesses of each. The intention here is not to suggest to the reader that one framework may be more secure than the other, but rather point out the unique security features of each that are important to understand when building secure enterprise web-based applications.

6.1 J2EE vs. .NET Security

We have discussed the various security features of both J2EE and .NET security models and how they help protect the core security services. Additionally, we covered some of the specific secure programming practices that developers of enterprise web-based applications should follow in order to promote security. For the most part, the programming guidelines have been similar, but the details were different. For instance, we discussed that good input validation against a white list of allowed input is the key to success. However, the actual mechanisms by which programmers can provide input validation in J2EE and .NET differ slightly (although similar). We now compare and contrast some of the security features of J2EE and .NET that support core security services. Some of the points we discuss are: Java Virtual Machine (JVM) vs. CLR (Common Language Runtime) security, J2EE class loader security vs. .NET code access security, J2EE vs. .NET security policy, cryptography. We also discuss role-based security in both J2EE and .NET. Special attention is also given to programmatic and declarative security.

6.1.1 JVM vs. CLR

JVM is responsible for executing an intermediate representation of the Java program classes (i.e. bytecode) on the machine on which it is resides. JVM also plays an essential security role in Java and consequently J2EE security. For instance, "the JVM provides a secure runtime environment by managing memory, providing isolation between executing components in different namespaces, array bounds checking, etc." [29]. JVM memory management and array bounds checking protects Java against the single gravest vulnerability category that plagues C and C++ applications, that is buffer overflows. This greatly supports authorization, confidentiality, integrity, and other security services. Since JVM allocates the various memory areas dynamically, it is very difficult for an attacker to figure out where to insert malicious code in a way that would change the flow of program execution. Beyond that, bounds checking on arrays, by preventing unreferenced memory accesses, leave an attacker looking for other, far more complex and rare, ways to inject malicious code. Additionally, memory management (such as garbage collection facilities) improve functional system stability and also protect J2EE applications against a wide variety of problems related to poor memory management, including availability issues that may otherwise lead to denial of service attacks. JVM also provides isolation between executing component, enforcing the principle of compartmentalization.

Additionally, JVM contains a class file verifier that examines classes for basic class file structure when loading. This protection mechanism is necessary to ensure that any malicious bytecode inserted by an attacker is detected. Several passes are used by JVM to inspect the bytecode, first for physical attributes (size, some magic number and length of attributes), then for attribute correctness in field and method references, and then for valid instruction parsing for each method.

The CLR in .NET serves a very similar security purpose to JVM in J2EE. CLR provides a secure execution environment through managed code and code access security. Analogous to JVM, the CLR is used to run the intermediate representation (e.g. intermediate language (IL)) of the .NET program code. CLR provides type checking facilities through Common Type Specification (CTS) that essentially make it impossible for buffer overflows to occur in managed code. Consequently, both J2EE and .NET runtime environments provide defense in depth against buffer overflow problems. Code access security determines the level of trust assigned to a particular piece of code and it allows for fine grained access control. There is similar support for this in J2EE through the class loader architecture [29].

Both J2EE and .NET may get into serious trouble when it comes to security when outside code needs to be called. For instance, in .NET, if a piece of code needs to be called that is not managed code (e.g. code in a unsupported programming language) then this code will not be able to take advantage of any of the security features of the CLR and thus it will be more difficult to keep it secure from things like buffer overflows and other problems. Analogously, if J2EE needs to work with code written in a language like C or C++, it will be open to similar problems since non of the JVM security features will be useful. Additionally, it will not be possible to apply finer grained access control to outside code, and it will always have to be treated as untrusted.

6.1.2 J2EE Class Loader vs. .NET Code Access Security

J2EE class loader architecture helps provide some of the fine grained authorization capabilities that allow execution of partially trusted code. There are two types of class loaders in J2EE: primordial class loader and object class loader. Primordial class loader is part of the JVM and it bootstraps the JVM by loading the base classes. On the other hand, object class loader is used to load the classes requested by the application. Without going into the specifics here, it suffices to say that primordial class loader in the JVM ensures that all classes loaded by the object class loader are properly authorized and that untrusted code cannot be made to replace the trusted code (base classes). "The loading of class loaders describes a tree structure with primordial class loader at the root and classes as the leaf nodes" [29]. Class loaders prevent class spoofing by passing requests back up the tree, through the parent class loader, until the class loader that loaded the requested class is reached. This is similar to tracing a certificate used in PKI to a trusted CA. Class loaders support additional security through managing the security namespaces, where a particular class loader may only reference other classes that are part of the same namespace (i.e. loaded by the same class loader or its parents).

Code access security is enabled in .NET through usage of evidence-based security and permissions. While we have already discussed both of these in far more detail in chapter four of the paper, we briefly touch on them here for the sake of discussion. Analogously to JVM, the CLR inflicts certain checks on .NET assemblies to ascertain to what degree they can be trusted and what code they can access. Evidence-based security uses two main criteria in the assembly querying: where did the code originate and who created the assembly. Digital certificate signed by the assembly publisher or a strong assembly name comprised of the unique identifier, text name, digital signature and a public key, can serve as evidence-based security tokens. For the purpose of determining who created the assembly, assembly's metadata is used that includes information on types, relationships with other assemblies, security permissions requested and assembly description. Metadata is examined by the CLR at various stages. The verifier module in the CLR ensures that IL types are correct before compiling IL to the native code and that assembly metadata is valid. The CLR further examines the metadata to establish the identity of the assembly and the permissions granted to the assembly based on the security policy [29].

Permissions inside the CLR define the rights of a piece of code to access a particular resource or perform a particular operation. An assembly will not be given greater permission than what is allowed by the security policy, but may be given lesser permission (in line with the least privilege principle). The assembly requests certain permissions at runtime, which are either granted or denied by the CLR. The CLR performs a "stack walk" to determine whether the given assembly has the necessary permissions to be granted access to the requested resource.

6.1.3 J2EE vs. .NET Security Policy

The basis for code access permissions, in both J2EE and .NET is the underlying security policy. In J2EE, the access controller makes the permissions decisions based on the security policy. The CLR resolves permission decisions by querying the access controller about the security policy as it relates to the particular permission of the assembly being evaluated. As we previously discussed, according to J2EE security policy, permission sets are grouped in J2EE into protection domains associated with the various code sources. Groups of permissions are associated with groups of classes and classes are grouped by origin [29]. Signed Java code is assigned permissions in accordance with the system policies as applied to the protection domain from which the code originates. It should be noted that the default configuration is not necessarily secure, since by default, Java applications are not associated with any permission domain, and thus have full access to system resources.

Analogously, .NET security policy is managed by the Code Access Security Policy Tool. The CLR uses the information in the security policy to determine at run-time the permissions that the assembly has. This is done after the assembly is identified through evidence-based security techniques that we have described. Similarly to J2EE protection domains that are assigned certain permissions, the security policy in .NET groups the categories based on the assembly evidence information, such as the zone from which the code is loaded. There are certain permissions associated with each of the zones, however, individual assemblies may have finer grained permissions. In this way, .NET security zones allow for more flexibility than J2EE protection domains, where all of the components in the protection domain have the same permissions.

6.1.4 Role-Based Security

In addition to code access security, both J2EE and .NET provide support for role-based security, where access control decisions to resources and operations are made based on the role of the individual on whose behalf the permission is requested. .NET defines various membership roles for the application. .NET uses role-based security for authenticating and authorizing individual users. When a user is identified, his authenticated identity and role membership (e.g. John Smith, CFO) becomes a principal.

Principals can be members of one or more roles (e.g. John Smith, CFO, Executive). The access control decision is similar to that for code access security, with the exception that the permission structure now depends on and is managed through the PrincipalPermission object [29]. J2EE also supports role-based security through a concept of a principal (an authenticated and authorized identity) that has certain permissions. In J2EE, the principals are often specified declaratively (via container's deployment descriptors) and may be accessed programmatically in order to check for permissions.

6.1.5 Programmatic and Declarative Security

Both J2EE and .NET support programmatic security, where developers can program security checks in the code. This makes it critical for developers to be aware of the proper ways to make use of the various security APIs and features provided by these platforms. Developers be may make use of code access security, role-based security, cryptography functionality, and other security related functionality in a programmatic manner. J2EE also makes extensive use of declarative security that is specified via XML deployment descriptors and is handled by the container. The deployment descriptors may be specified at deployment time. Additionally, declarative security places fewer burdens on the programmer, thus reducing the risk of insecure programming. For instances, security roles and permissions can be specified through deployment descriptors. .NET does not provide an equivalent mechanism for declarative security.

6.1.6 Cryptography

Both J2EE and .NET provide adequate cryptographic functionality, that when used properly, will support the core security services, including confidentiality, integrity, etc. .NET base classes provide support for encryption, key generation and HMAC through a variety of accepted cryptographic algorithms. .NET framework also provides tools for certificate management and manipulation. For instance, X.509 certificates can be created and managed. We have discussed some of the cryptographic APIs available in J2EE including JCE and JAAS. J2EE also contains APIs that contain SSL and TLS functionality that developers may use.

7.0 Conclusions

From the analysis above it is apparent that J2EE and .NET platforms both have fairly comparable security models that provide extensive support for all of the core security services. The decision to use one over the other is probably not going to be security driven. The discussion of how to choose one over the other for a particular development project is outside the scope of this paper. The important thing to take away from this is that both J2EE and .NET require a developer to make use of various security features via programmatic methods. Failure to use those feature properly could undermine all built in security functionality and offer no more than a false sense of security. The focus of this paper has been to demonstrate how to program enterprise web-applications securely using J2EE and .NET technologies. While network and host security were given some consideration in this paper, the main emphasis has been on application security, as it is the most immature and is in need of many improvements help at this point.

We took a broader approach to application security in this paper than just looking at specific examples of how the core security services are supported in J2EE and .NET. For instance, we first tried to understand what an enterprise web-based application really is what makes it different from other applications, and what are some of the unique security requirements for these applications. That led us to consider the various application vulnerabilities that constitute the major risk areas. That discussion was driven by the OWASP top ten list. We then talked about the key security principles, such as least privilege and compartmentalization that need to be followed in all stages of enterprise web-application development and deployment because of the critical role that they plan in security. Throughout this paper, as we discussed many of the specific compensating controls, we very often came back and tied our discussion to the key security principles. We cannot emphasize enough how critical it is for all members of the software project team to be familiar with and follow these principles because only solutions that comply with these principles will have any chance of providing true protection for the core security services.

We introduced threat modeling as a critical ingredient for development of secure enterprise web-based applications. It is impossible to defend against the threats without first understanding what the threats are. To this end we recommended identifying all resources that need protection (assets), documenting security assumptions, identifying attack surface as well as input and output attack vectors, combining these vectors into attack trees (scenarios) and ensuring that proper mitigations are put in place at the appropriate places. We also suggested that defense in depth is a useful strategy. We looked at a sample threat model for a J2EE application (JPetStore 4.0) and then considered the threat modeling technique employed by Microsoft that uses STRIDE and DREAD. The chapter on threat modeling is a critical chapter in this paper, because without a solid threat model, it is not possible to build secure systems.

The threats and mitigations for enterprise web-based applications were considered systematically across the various tiers of the application. This view also helps better understand where the compensating controls must be placed. For each of the vulnerability categories we provided a detailed discussion, mitigation strategies with examples, as well as likelihood of exploit and severity. The goal of that discussion was to give a reader a more in depth view of the main areas of concern in enterprise web-based applications, the insecure programming pitfalls that allow for exploitation, and the ways to correct these problems. A threat model for an enterprise web-based application will include much of the information provided in chapter five of this paper, but will have a more complete coverage and follow the strategies outlined in chapter two.

It is also important to remember that development and deployment of secure web-based applications is contingent upon a secure development process. While we did not provide a detailed discussion of secure development process in this paper due to time limitation, the baseline process that we mentioned is derived from CLASP. Additionally, C&A

should be made part of this process, in order to manage both local and enterprise risk, as outlined in NIST SP800-37. In addition to a secure development process that is aware of the various application security issues and makes it an explicit part of the process, an ESA is required to support development and deployment of secure enterprise web-based application. We discussed the ESA rings 1-5 that is part of the EA, such as the Zachman Framework. While most of the issues in this paper fall in rings 4 and 5, we showed how many of the CLASP activities actually map to rings 1-3. In general, an entire organization needs to get behind security with security awareness programs, monitoring of security metrics, setting of security policies, and other initiatives, in order to attain measurable and repeatable improvements in security of enterprise web-based applications. It is time to stop applying bandage treatments for security problem symptoms and instead focus on eliminating security problems at the source.

Appendix A – List of Acronyms

ACL – Access Control List

API – Application Programming Interface

DREAD - Damage, Reproducibility, Exploitability, Affected users, Discoverability

DSA – Digital Signature Algorithm

ebXML – Electronic Business Extensible Markup Language

EJB – Entity Java Bean

HMAC – Hashing and Message Authentication

JAAS – Java Authentication & Authorization API

PKI – Public Key Infrastructure

RBAC – Role Based Access Control

RNG – Random Number Generator

RSA – Rivest, Shamir and Adleman

SAML – Security Assertions Markup Language

SOAP – Simple Object Access Protocol

SQL – Structured Query Language

SSL – Secure Sockets Layer

STRIDE - Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege

TLS – Transport Layer Security

WSDL – Web Service Definition Language

WTLS – Wireless Transport Level Security

XSS – Cross Site Scripting

Appendix B – Annotated Glossary

Access Control List:

The access control list (ACL) is a concept in computer security, used to enforce privilege separation. It is a means of determining the appropriate access rights to a given object given certain aspects of the user process that is requesting them, principally the process's user identity. The list is a data structure, usually a table, containing entries that specify individual user or group rights to specific system objects, such as a program, a process, or a file.

Each accessible object contains an identifier to its ACL. The privileges or permissions determine specific access rights, such as whether a user can read from, write to or execute an object. In some implementations an Access Control Entry (ACE) can control whether or not a user, or group of users, may alter the ACL on an object.

ACL implementations can be quite complex. ACLs can apply to objects, directories and other containers, and for the objects and the containers created within this container. ACLs cannot implement all of the security measures that one might wish to have on all systems, and a fine-grained capability-based operating system may be a better approach, with the authority transferred from the objects being accessed to the objects seeking access — allowing for much finer-grained control.

In networking, the term Access Control List (ACL) refers to a list of the computing services available on a server, each with a list of hosts permitted to use the service. On a router an access list specifies which addresses are allowed to access services. Access lists are used to control both inbound and outbound traffic on a router.[1]

ACL can be used as a basis to implement code access security for enterprise web based applications. For the most part, the implementation of the ACL is supported by the operating system. There are a variety of possible implementations for access control lists.

Attack Vectors & Trees:

An attack vector is a path or means by which a hacker can gain access to a computer or network server in order to deliver a payload or malicious outcome. Attack vectors enable hackers to exploit system vulnerabilities, including the human element.
Attack vectors include viruses, e-mail attachments, Web pages, pop-up windows, instant messages, chat rooms, and deception. All of these methods involve programming (or, in a few cases, hardware), except deception, in which a human operator is fooled into removing or weakening system defenses.

[1] http://en.wikipedia.org/wiki/Access_control_list

To some extent, firewalls and anti-virus software can block attack vectors. But no protection method is totally attack-proof. A defense method that is effective today may not remain so for long, because hackers are constantly updating attack vectors, and seeking new ones, in their quest to gain unauthorized access to computers and servers. The most common malicious payloads are viruses (which can function as their own attack vectors), Trojan horses, worms, and spy ware. If an attack vector is thought of as a guided missile, its payload can be compared to the warhead in the tip of the missile. Identification of attack vectors through the various program entry points is an essential step of the threat modeling process. [2]

An attack tree is a way of collecting and documenting the potential attacks on your system in a structured and hierarchical manner. The tree structure gives a descriptive breakdown of various attacks that the attacker uses to compromise the system. Attack trees represent a reusable representation of security issues that helps focus efforts. Test teams can create test plans to validate security design. Developers can make tradeoffs during implementation and architects or developer leads can evaluate the security cost of alternative approaches. Attack patterns are a formalized approach to capturing attack information in your enterprise. These patterns can help identify common attack techniques. [3]

A good way to go about creating attack trees is to identify goals and sub-goals of an attack as well as other actions necessary for a successful attack. An attack tree can be represented as a hierarchical diagram or as an outline. The desired end result of the whole exercise is to have something that portrays an attack profile of the application. This will help in evaluation if likely security risks and guide the choice of mitigations. At this stage, flaws in architecture or design might become apparent and would need to be rectified. Attack trees can be started by first identifying the root nodes, representing the ultimate goals of an attacker, and working towards the leaf nodes, representing the techniques and methodologies used by an attacker to achieve the goals. Each of the leaf nodes might be a separate individual security threat that was previously identified.

Buffer Overflow:

Buffer overflow attacks are possible if no proper bounds checking is performed on the buffer to which user input is written. Carefully crafted input that writes data to the buffer past the allocated range can be used to overwrite the return pointer on the stack and point the program counter to a location where malicious shell code has been planted. The attack code is then executed resulting in severe authorization breach on the application (execution of arbitrary code). Arbitrary code can be executed on the host system with the same privileges as those that were granted to the web application. Following the principle of least privilege could help limit the amount of damage an attacker can cause following a successful exploitation of the buffer overflow vulnerability. Buffer overflows can be avoided by proper input validation. Additionally, the likelihood of introducing buffer overflows into the application can be significantly reduced if safe

[2] http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci1005812,00.html
[3] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/threatcounter.asp

string and memory manipulation function are used. While execution of arbitrary code and taking control of the application process are the more drastic possible consequences of a successful exploitation of the buffer overflow vulnerability, a more frequent impact could be on system availability since buffer overflows will often cause system crashes. Besides the custom application code, components that can be vulnerable to buffer overflows may include CGI components, libraries, drivers and web application server components. Since JAVA is generally considered a safe language in terms of bounds checking and thus fairly protected against buffer overflows, enterprise web applications built with J2EE are generally considered protected against buffer overflows. However, that is not entirely the case. Buffer overflow attacks, while more difficult and rare with JAVA based applications, can still take place. An example may be a buffer overflow in JVM itself. Format string attacks are a subset of buffer overflow attacks.

Buffer overflows are a classic example of poor input validation. The reason for this vulnerability is that some programming languages allow the input placed in a buffer to overflow past the allocated range for the buffer into adjacent areas of memory. For example, if a buffer overflow occurs on a stack and the overflow can reach a function return pointer, an attacker who carefully crafts a buffer overflow may overwrite the return pointer with his own value that will make the program to jump to the area where an attacker has injected some attack code and start executing it. There are two main kinds of buffer overflows, stack based and heap based. Both kinds are extremely dangerous because they cause availability problems (in the best case) and cause authorization failures (in the worst case) which can then lead to other problems. Languages like C are particularly prone to buffer overflows because it contains many inherently insecure functions. Luckily, buffer overflows are not a tremendous problem for enterprise web-based application since both J2EE and .NET provide automatic type checking that make buffer overflows very unlikely. However, application developers of enterprise web based applications still need to be very careful when dealing when providing integrations with legacy code.

Role Based Access Control:

Role Based Access Control (RBAC) is a system of controlling which users have access to resources based on the role of the user. Access rights are grouped by role name, and access to resources is restricted to users who have been authorized to assume the associated role. For example, if a RBAC system were used in a hospital, each person that is allowed access to the hospital's network has a predefined role (doctor, nurse, lab technician, administrator, etc.). If someone is defined as having the role of doctor, than that user can access only resources on the network that the role of doctor has been allowed access to. Each user is assigned one or more roles, and each role is assigned one or more privileges to users in that role. [4]

Both J2EE and .NET have mechanisms for enforcing RBAC in order to provide authorization services to users of enterprise web-based applications. In J2EE and .NET that there are fundamentally two types of access control: role-based access control

[4] http://www.webopedia.com/TERM/R/RBAC.html

(RBAC) and code-based access control. With role-based access control, an authorization decision is based on the identity of the caller, whereas with code-based access control it is based on the calling code. There are also two fundamental ways to enforce access control: declarative and programmatic. Declarative access control is managed by the container (J2EE) or the CLR (.NET), whereas programmatic access control is implemented as a series of checks in the application code.

Command Injection:

Enterprise web applications pass parameters when they access external systems, applications, or use local OS resources. Whenever possible, those parameters should not come directly from the user and be defined as constants. Otherwise, they should be rigorously validated prior to usage. If an attacker can embed malicious commands into these parameters, they may be executed by the host system when the access routines are invoked by the application. SQL injection is a particularly common and serious type of injection, where SQL statements are passed to the web application and then without validation are passed to the routine that accesses the database with that SQL statement. Command injections can be used to disclose information, corrupt data and pass malicious code to an external system application via the web application.

Untrusted code should never be invoked by trusted code and input that is not trusted and has not been validated should never be passed to the trusted code routines. Command injection attacks are made possible because the parameters passed to the trusted code routines come from input that has not be properly validated. The best strategy is to only use constants or trusted data for access to trusted code routines. As part of the threat model, all instances where trusted code is used should be documented along with the parameters passed to the trusted code and any code that might be invoked by trusted code.

Injection flaws allow attackers to relay malicious code through a web application to another system. These attacks include calls to the operating system via system calls, the use of external programs via shell commands, as well as calls to backend databases via SQL (i.e., SQL injection). Whole scripts written in perl, python, and other languages can be injected into poorly designed web applications and executed. Any time a web application uses an interpreter of any type there is a danger of an injection attack.
Many web applications use operating system features and external programs to perform their functions. Sendmail is probably the most frequently invoked external program, but many other programs are used as well. When a web application passes information from an HTTP request through as part of an external request, it must be carefully scrubbed. Otherwise, the attacker can inject special (meta) characters, malicious commands, or command modifiers into the information and the web application will blindly pass these on to the external system for execution. [5]

[5] http://www.owasp.org/documentation/topten/a6.html

DREAD:

DREAD is Mirosoft's approach to ratings of risks at the application level. A proper rating can help identify next steps for risk mitigation and also facilitate in prioritizing remediation. The final step in the threat modeling process is to rate all of the threats that were identified. That is done by evaluating potential impact of each of the threats on the system. The purpose of this exercise is to help prioritize the threats. It may be unrealistic to expect that under the pressures of a typical software development schedule all of the threats will be mitigated. This may be impossible due to time and money constraints. After all, functionality has to come first. However, having a good way to rate the threats based on the greatest security impact on the application as a whole will help make inform decisions as to what threats must be addressed first. The formula for calculating risk is: **RISK = PROBABILITY * DAMAGE POTENTIAL**. .

In order to prioritize the threats, high, medium and low ratings can be used. Threats rated as high pose a significant risk to the application and should be addressed as soon as possible. Medium threats need to be addressed, but are less urgent than high threats. Low threats should only be addressed if the schedule and cost of the project allows. Microsoft has also developed a more sophisticated rating system called DREAD that makes the impact of the security threat more explicit. Adding additional dimensions to consider makes it easier for a team performing threat modeling to agree on the rating. DREAD model is used to calculate risk at Microsoft instead of the simplistic formula above. The following questions must be asked when using DREAD to arrive at the risk for a particular threat:

- Damage potential: How great is the damage if the vulnerability is exploited?
- Reproducibility: How easy is it to reproduce the attack?
- Exploitability: How easy is it to launch an attack?
- Affected users: As a rough percentage, how many users are affected?
- Discoverability: How easy is it to find the vulnerability?

DREAD questions can be extended to meet the particular needs of the application. There might be other dimensions of great importance to a particular application being evaluated. A sample rating table is shown below that can be useful when prioritizing threats.

	Rating	**High (3)**	**Medium (2)**	**Low (1)**
D	Damage potential	The attacker can subvert the security system; get full trust authorization; run as administrator; upload content.	Leaking sensitive information	Leaking trivial information

R	Reproducibility	The attack can be reproduced every time and does not require a timing window.	The attack can be reproduced, but only with a timing window and a particular race situation.	The attack is very difficult to reproduce, even with knowledge of the security hole.
E	Exploitability	A novice programmer could make the attack in a short time.	A skilled programmer could make the attack, then repeat the steps.	The attack requires an extremely skilled person and in-depth knowledge every time to exploit.
A	Affected users	All users, default configuration, key customers	Some users, non-default configuration	Very small percentage of users, obscure feature; affects anonymous users
D	Discoverability	Published information explains the attack. The vulnerability is found in the most commonly used feature and is very noticeable.	The vulnerability is in a seldom-used part of the product, and only a few users should come across it. It would take some thinking to see malicious use.	The bug is obscure, and it is unlikely that users will work out damage potential.

DSA:

Digital Signature Algorithm (DSA) is a public-key method based on the discrete log problem. The Digital Signature Algorithm is mandated by the Federal Information Processing Standard FIPS 186. This is a public key system, but unlike RSA it can only be used for making signatures. It is a public key algorithm that is used as part of the Digital Signature Standard (DSS). DSA was developed by the U.S. National Security Agency to generate a digital signature for the authentication of electronic documents. It cannot be used for encryption, only for digital signatures. The algorithm produces a pair of large numbers that enable the authentication of the signatory, and consequently, the integrity of the data attached. DSA is used both in generating and verifying digital signatures. [6]

The DSS standard specifies a Digital Signature Algorithm (DSA) appropriate for applications requiring a digital rather than written signature. The DSA digital signature is a pair of large numbers represented in a computer as strings of binary digits. The digital signature is computed using a set of rules (i.e., the DSA) and a set of parameters such that the identity of the signatory and integrity of the data can be verified. The DSA provides the capability to generate and verify signatures. Signature generation makes use of a private key to generate a digital signature. Signature verification makes use of a public key which corresponds to, but is not the same as, the private key. Each user possesses a private and public key pair. Public keys are assumed to be known to the public in general. Private keys are never shared. Anyone can verify the signature of a user by employing that user's public key. Signature generation can be performed only by the possessor of the user's private key.

A hash function is used in the signature generation process to obtain a condensed version of data, called a message digest. The message digest is then input to the DSA to generate the digital signature. The digital signature is sent to the intended verifier along with the signed data. The verifier of the message and signature verifies the signature by using the sender's public key. The same hash function must also be used in the verification process. The hash function is specified in a separate standard, the Secure Hash Standard (SHS),

[6] http://www.auditmypc.com/acronym/DSA.asp

FIPS 180. Similar procedures may be used to generate and verify signatures for stored as well as transmitted data. [7]

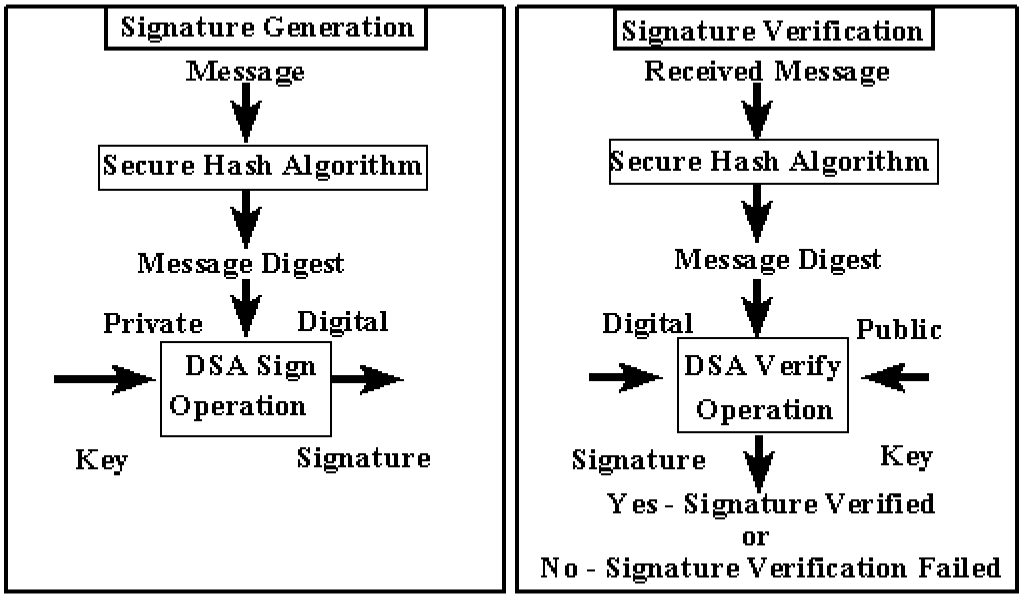


Figure: Using the SHA with the DSA [8]

Dynamic Security Analysis:

Dynamic analysis is any software analysis that involves actually running the software. Dynamic analysis tools provide a way for security professionals to test, before deployment, an application for vulnerabilities associated with the operation of the application, and to get a quick overview of the application's overall security posture. Dynamic analysis is frequently used to assess web-based applications for known vulnerabilities associated with a given operating environment or platform. The downside to dynamic analysis tools is that they inject analysis late in the development cycle, when the cost of fixing vulnerabilities is higher; and that they require a fully functioning and staged application, raising the tools' total cost of ownership. Another issue with dynamic analysis is that it essentially treats the system as a black box, making it difficult to confirm that the coverage was thorough and also difficult to point to the specific problem for the purposes of remediation. Additionally, the code has to run in order for it to be analyzed by the dynamic analysis tools. That means that if any "dormant" pieces of code are present in the application that only runs some of the time, that code might be missed by the dynamic analyzer.

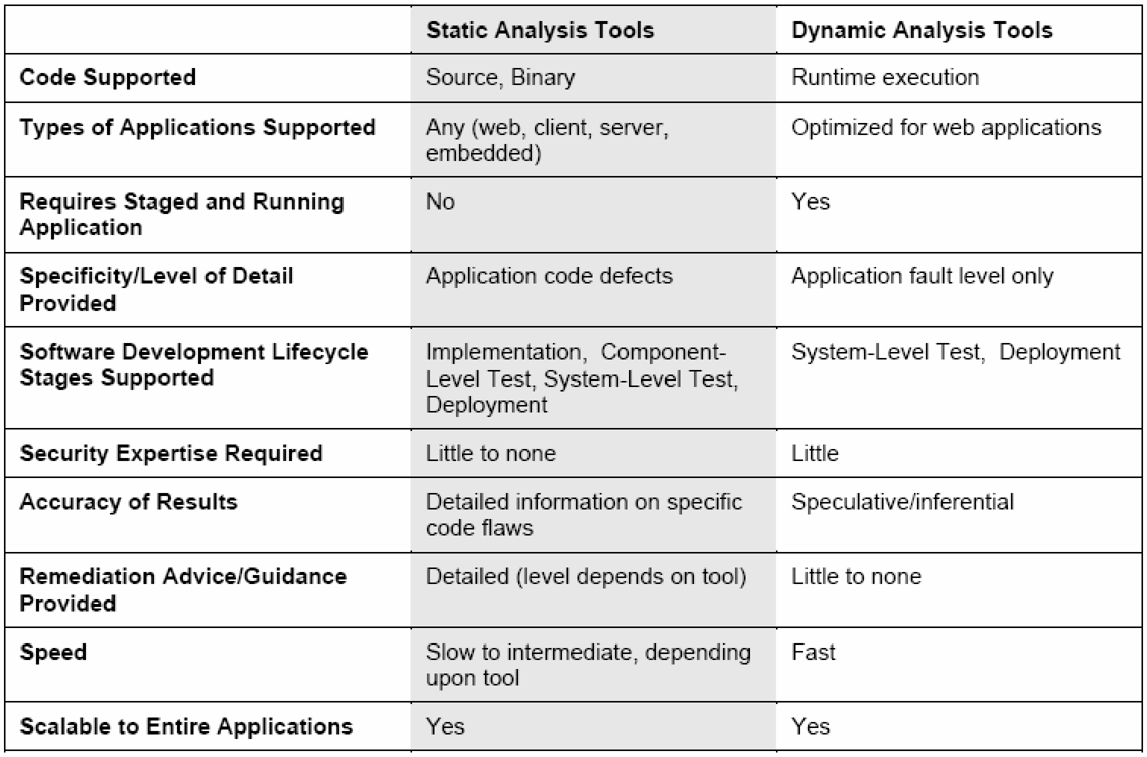
Dynamic analysis "exercises" an application for vulnerabilities likely to be caused by real-life operational use, including security problems introduced by users, configuration changes in databases and servers as applications move into production, and so on.

[7] http://www.itl.nist.gov/fipspubs/fip186.htm
[8] http://www.itl.nist.gov/fipspubs/fip186.htm

Dynamic analysis lets you test "what you know": your assumptions about how an application is likely to respond under known operating conditions. Dynamic analysis treats code like a black box. In a process known as red-teaming, QA and testing staff use simple metrics to see if an exploit is likely. Dynamic analysis generates a set of inputs to an application and tries to induce conditions not anticipated by the software developer. It then examines the responses, looking for a match that would suggest the presence of vulnerability. Today's dynamic testing tools automate this process. For example, a tester might feed a long string of characters into an application. If the application "crashes," the crash suggests a problem with buffer overflow (a common vulnerability).

Dynamic analysis is most useful for analyzing applications in production. Dynamic analysis can find new vulnerabilities introduced by the operational environment, such as vulnerabilities created by the misconfiguration of the production host; or related tools and libraries that may cause otherwise secure code to develop security problems.
For example, dynamic analysis could detect that an enterprise application was built using a database and that the developer had erroneously left the default database accounts and passwords intact. On the plus side, dynamic analysis with today's tools is fast and relatively inexpensive. It requires little skill; it can be used by QA professionals, testers and other non-developers. Its output is more readily understandable by non-developers and can provide easily understandable metrics for senior management. On the minus side, dynamic analysis lacks specificity and depth, limiting its usefulness to software development project teams. For example: although the application's reaction to a long string of characters does indicate that a buffer-overflow vulnerability is likely, this finding is not definitive – it could just as likely be an input validation problem. The developer must figure it out. Because dynamic analysis tests an application as a black box, it's nearly impossible to test all conditions, only the conditions testers can subject the code to; as a result, current automated dynamic analysis tools test only the most feasible conditions. Current dynamic analysis tools are also designed mostly for web applications, limiting their use across a broader application portfolio or code base. Dynamic analysis is relatively superficial – similar to an Easter Egg hunt, in that a developer who needs to remediate the vulnerability is given information about the vulnerability but not its location in the code base. The output from dynamic analysis thus has little context and is not easily actionable, requiring considerable interpretation to translate the results into what exactly needs to be fixed in the code. In real-world development environments, it may not even be clear how risky each vulnerability is or which developer "owns" which vulnerability. This lack of context can introduce or exacerbate organizational friction, because one group is analyzing another group's work in an over-the-wall fashion. Dynamic analysis is generally not cost-effective to integrate into the development cycle, primarily because of the cost of preparing for analysis. It requires the project team to build a running version of the full application for testing and then stage the application in its target operating environment (including configuring servers, networks, etc.). This makes the total cost of ownership (TCO) of dynamic analysis tools very high.

Dynamic analysis also has little residual value beyond basic analysis, because its results lack specificity and context. So it doesn't contribute to knowledge transfer or the improvement of coding practices. The table below compares the capabilities of dynamic analysis tools and static analysis tools for security (please see definition of static analysis below). [9]



	Static Analysis Tools	**Dynamic Analysis Tools**
Code Supported	Source, Binary	Runtime execution
Types of Applications Supported	Any (web, client, server, embedded)	Optimized for web applications
Requires Staged and Running Application	No	Yes
Specificity/Level of Detail Provided	Application code defects	Application fault level only
Software Development Lifecycle Stages Supported	Implementation, Component-Level Test, System-Level Test, Deployment	System-Level Test, Deployment
Security Expertise Required	Little to none	Little
Accuracy of Results	Detailed information on specific code flaws	Speculative/inferential
Remediation Advice/Guidance Provided	Detailed (level depends on tool)	Little to none
Speed	Slow to intermediate, depending upon tool	Fast
Scalable to Entire Applications	Yes	Yes

Figure: Dynamic vs. Static Security Analysis Tools [10]

FIPS-140:

Federal Information Processing Standard 140-2 (FIPS 140-2) is entitled "Security Requirements for Cryptographic Modules." It's a standard that describes government requirements that hardware and software products should meet for Sensitive but Unclassified (SBU) use. The standard was published by the National Institute of Standards and Technology (NIST), has been adopted by the Canadian government's Communications Security Establishment (CSE), and is being adopted by the financial community through the American National Standards Institute (ANSI).

The [FIPS 140-2] standard specifies the security requirements that are to be satisfied by a cryptographic module utilized within a security system protecting unclassified information within computer and telecommunication systems (including voice systems). The standard provides four increasing, qualitative levels of security: Level 1, Level 2,

[9] "Risk in the Balance: Static Analysis vs. Dynamic Analysis" Secure Software Inc., 2004. Available from: www.securesoftware.com
[10] "Risk in the Balance: Static Analysis vs. Dynamic Analysis" Secure Software Inc., 2004. Available from: www.securesoftware.com

Level 3, and Level 4. These levels are intended to cover the wide range of potential applications and environments in which cryptographic modules may be employed. The security requirements cover areas related to the secure design and implementation of a cryptographic module. These areas include basic design and documentation, module interfaces, authorized roles and services, physical security, software security, operating system security, key management, cryptographic algorithms, electromagnetic interference/ electromagnetic compatibility (EMI/EMC), and self-testing.

FIPS 140-2 is actually the third version of the FIPS 140 standard. NIST reviews the FIPS 140 standard every five years to determine if further updates are needed. At this time, NIST only accepts applications for FIPS 140-2 certificates. However, any product previously evaluated for FIPS 140-1 can still be purchased by the federal government today. It is important to note that the FIPS 140-1 or 140-2 certificate applies only to that version of the product that was submitted for validation, and all product updates are subject to re-evaluation against the current version of the standard. [11]

The security requirements for cryptographic key management encompass the entire lifecycle of cryptographic keys, cryptographic key components, and critical security primaries (CSPs) employed by the cryptographic module. Key management includes random number and key generation, key establishment, key distribution, key entry/output, key storage, and key zeroization. A cryptographic module may also employ the key management mechanisms of another cryptographic module. Encrypted cryptographic keys and CSPs refer to keys and CSPs that are encrypted using an Approved algorithm or Approved security function. Cryptographic keys and CSPs encrypted using a non-Approved algorithm or proprietary algorithm or method are considered in plaintext form, within the scope of this standard. Secret keys, private keys, and CSPs shall be protected within the cryptographic module from unauthorized disclosure, modification, and substitution. Public keys shall be protected within the cryptographic module against unauthorized modification and substitution. Documentation shall specify all cryptographic keys, cryptographic key components, and CSPs employed by a cryptographic module. [12]

JAAS:

The Java Authentication and Authorization Service (JAAS) is a set of APIs that can be used for two purposes: For authentication of users, to reliably and securely determine who is currently executing Java code, regardless of whether the code is running as an application, an applet, a bean, or a servlet; and For authorization of users, to determine reliably and securely who is currently executing Java code, regardless of whether the code is running as an application, an applet, a bean, or a servlet; and JAAS authentication is performed in a pluggable fashion. This permits Java applications to remain independent from underlying authentication technologies. New or updated technologies can be plugged in without requiring modifications to the application itself. JAAS authorization extends the existing Java security architecture that uses a security policy to

[11] http://www.corsec.com/fips_faq.php
[12] http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf

specify what access rights are granted to executing code. That architecture, as introduced in the Java 2 platform, was code-centric. That is, the permissions were granted based on code characteristics: where the code was coming from and whether it was digitally signed and if so by whom. With the integration of JAAS into the Java 2 SDK, the java.security.Policy API handles Principal-based queries, and the default policy implementation supports Principal-based grant entries. Thus, access control can now be based not just on what code is running, but also on who is running it. [13]

J2EE client applications may use JAAS to provide authentication and authorization services. Some of the standard Java APIs to develop JAAS client applications include javax.naming, javax.security.auth, javax.security.auth.Callback, javax.security.auth.login, and javax.security.auth.SPI. Some JAAS security APIs specific to Weblogic 8.1 are weblogic.security, weblogic.security.auth, and weblogic.security.auth.Callback. The letter "x" after package name "java" (i.e. javax) stands for java extension, implying that JAAS API is an extension to the Java platform. If a J2EE application client needs to communicate with application servers that are not implemented via Weblogic, security APIs specific to Weblogic should not be used. In that case using those APIs would impact portability.

A client authenticated to the J2EE WebLogic application server with JAAS may be an application, applet, Enterprise Java Bean (EJB) or a servlet. JAAS is a standard extension to the Java Software Development Kit 1.4.1. JAAS allows enforcement of access control based on user identity. WebLogic application server uses only authentication capabilities of JAAS to support LoginContext and LoginModule functionalities. The WebLogic LoginModule weblogic.security.auth.login.UsernamePasswordLoginModule supports client user name and password authentication. For client certificate authentication, mutual SSL authentication should be used (provided by JNDI authentication).

JAAS can be used for external or internal authentication. Thus developers of custom authentication providers in J2EE applications, as well as developers for remote J2EE application clients may potentially need to understand JAAS. Users of Web browser clients or J2EE application component developers do not need to use JAAS. A typical JAAS authentication client application would include a Java client, LoginModule, Callbackhandler, configuration file, action file and a build script.

The key point to take away from this is that when weblogic.security.Security.runAs() method is executed, it associates the specified Subject with the permission of the current thread. After that the action is executed. If the Subject represents a non-priveleged user, the default of the JVM will be used. Consequently, it is crucial to specify the correct Subject in the runAs() method. There are several options available to developers there. One is to implement wrapper code shown below.

Creating a wrapper for runAs() method:

[13] http://java.sun.com/products/jaas/overview.html

```
import java.security.PrivilegedAction;
import javax.security.auth.Subject;
import weblogic.security.Security;
public class client
{
public static void main(String[] args)
{
Security.runAs(new Subject(),
new PrivilegedAction() {
public Object run() {
//
//If implementing in client code, main() goes here.
//
return null;
}
});
}
}
```

The discussion of various other methods for specifying the correct subject is omitted for the sake of brevity. There are eight steps to writing a client application using JAAS authentication in WebLogic J2EE application server. The first step is to implement LoginModule classes for each type of the desired authentication mechanisms. The second step is to implement the CallbackHandler class that will be used by the LoginModule to communicate with the user in order to obtain user name, password and URL. The third step is to write a configuration file that would specify which LoginModule classes would be used by the WebLogic Server for authentication and which should be invoked. The fourth step is to write code in the Java client to instantiate a LoginContext. The LoginContext uses the configuration file (sample_jaas.config) to load the default LoginModule configured for WebLogic Server. In step five, the login() method of the LoginContext instance is invoked. The login() method is used to invoke all of the LoginModules that have been loaded. Each LoginModule tries to authenticate the subject and the LoginContext throws a LoginException in the event that the login conditions specified in the configuration file are not met. In step six, Java client code is written to retrieve the authenticated Subject from the LoginContext instance and call the action as the Subject. When successful authentication of the Subject takes place, access controls can be placed upon that Subject by invoking the eblogic.security.Security.runAs() method, as was previously discussed. In step seven, code is written to execute an action if the Subject has the required privileges. Finally, step eight, a very important step, where the logout() method is invoked on the LoginContext instance. The logout() method closes the user's session and clears the Subject. It is very important for developers to follow all of these eight steps and do so properly in order for JAAS to be effective.

Kerberos:

Kerberos is a secure method for authenticating a request for a service in a computer network. Kerberos was developed in the Athena Project at the Massachusetts Institute of Technology (MIT). Kerberos lets a user request an encrypted "ticket" from an authentication process that can then be used to request a particular service from a server. The user's password does not have to pass through the network. A version of Kerberos (client and server) can be downloaded from MIT or you can buy a commercial version. [1]

The core of a Kerberos architecture is the KDC (Key Distribution Server). The KDC stores authentication information and uses it to securely authenticate users and services. This authentication is called secure because it does not occur in plaintext, does not rely on authentication by the host operating system, does not base trust on IP addresses Does not require physical security of the network hosts, The KDC acts as a trusted third party in performing these authentication services. Due to the critical function of the KDC, multiple KDC's are normally utilized. Each KDC stores a database of users, servers, and secret keys. Kerberos clients are normal network applications which have been modified to use Kerberos for authentication. In Kerberos slang, they have been Kerberized. [2] The logical diagram for Kerberos is shown below:

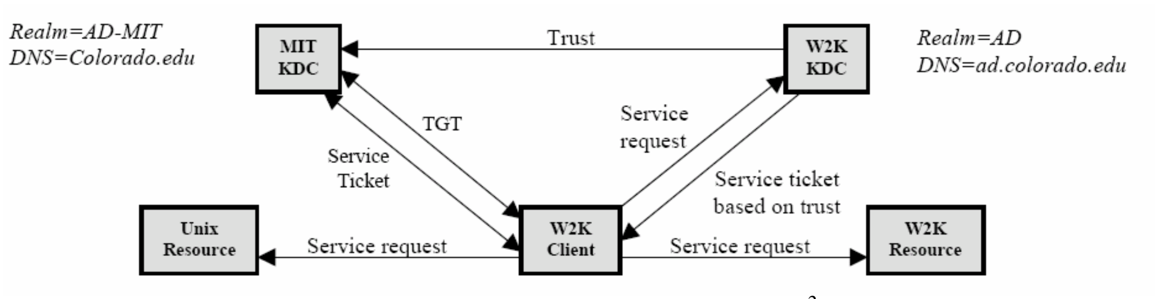


Figure: Kerberos Logical Diagram [3]

In the first step of Kerberos, a W2K Client asks KDC for a ticket granting ticket (TGT). The KDC then forwards a TGT back to the W2K client who can use it to obtain a service ticket via a service request from the W2K KDC. Now the W2K client can communicate with both UNIX resources and W2K resources.

Code Access Security:

Code access security allows code to be trusted to varying degrees depending on where the code originates and on other aspects of the code's identity. Code access security also enforces the varying levels of trust on code, which minimizes the amount of code that must be fully trusted in order to run. Using code access security can reduce the likelihood that the code can be misused by malicious or error-filled code. It can reduce liability

[1] http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci212437,00.html
[2] http://www.tech-faq.com/kerberos.shtml
[3] http://www.colorado.edu/its/windows2000/itsresources/kerbprop.pdf

because it is possible to specify the set of operations in the code that should be allowed to perform as well as the operations that the code should never be allowed to perform. Code access security can also help minimize the damage that can result from security vulnerabilities in the code.[1]

Code access security is a mechanism that helps limit the access code has to protected resources and operations. In the .NET Framework, code access security performs the following functions:

- Defines permissions and permission sets that represent the right to access various system resources.

- Enables administrators to configure security policy by associating sets of permissions with groups of code (code groups).

- Enables code to request the permissions it requires in order to run, as well as the permissions that would be useful to have, and specifies which permissions the code must never have.

- Grants permissions to each assembly that is loaded, based on the permissions requested by the code and on the operations permitted by security policy.

- Enables code to demand that its callers have specific permissions.

- Enables code to demand that its callers possess a digital signature, thus allowing only callers from a particular organization or site to call the protected code.

- Enforces restrictions on code at run time by comparing the granted permissions of every caller on the call stack to the permissions that callers must have.

To determine whether code is authorized to access a resource or perform an operation, the runtime's security system walks the call stack, comparing the granted permissions of each caller to the permission being demanded. If any caller in the call stack does not have the demanded permission, a security exception is thrown and access is refused. The stack walk is designed to help prevent luring attacks, in which less-trusted code calls highly trusted code and uses it to perform unauthorized actions. Demanding permissions of all callers at run time affects performance, but it is essential to help protect code from luring attacks by less-trusted code. To optimize performance, you can have your code perform fewer stack walks; however, you must be sure that you do not expose a security weakness whenever you do this.

The stack walk in the figure below results from method in assembly A4 requesting that its callers have permission P.

[1] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcodeaccesssecurity.asp

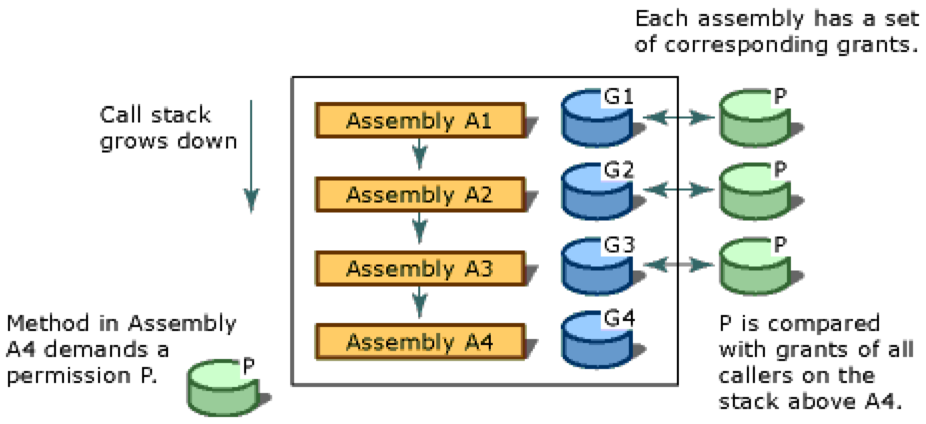

Each assembly has a set of corresponding grants.
Call stack grows down
Assembly A1
Assembly A2
Assembly A3
Assembly A4
G1
G2
G3
G4
P
P
P
Method in Assembly A4 demands a permission P.
P
P is compared with grants of all callers on the stack above A4.

Figure: Stack Walk (Method in A4 Demands Permission P from Callers) [1]

In one typical use of code access security, an application downloads a control from a local intranet host Web site directly to the client so that the user can enter data. The control is built using an installed class library. The following are some of the ways code access security might be used in this scenario:

- Before load time, an administrator can configure security policy to specify that code be given special authority (more permission than local internet code would usually receive) if it has a particular digital signature. By default, the predefined **LocalIntranet** named permission set is associated with all code that is downloaded from the local intranet.

- At load time, the runtime grants the control no more permissions than those associated with the **LocalIntranet** named permission set, unless the control has a trusted signature. In that case, it is granted the permissions associated with the **LocalIntranet** permission set and potentially some additional permissions because of its trusted signature.

- At run time, whenever a caller (in this case the hosted control) accesses a library that exposes protected resources or a library that calls unmanaged code, the library makes a security demand, which causes the permissions of its callers to be checked for the appropriate permission grants. These security checks help prevent the control from performing unauthorized actions on the client's computers. [2]

.NET Framework provides two kinds of security: role-based security and code access security. As a synopsis, role-based security controls user access to application resources and operation, whereas code access security controls code access to various resources and privileged operations. These two kinds of security are complementary forms of security

[1] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconintroductiontocodeaccesssecurity.asp

[2] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconintroductiontocodeaccesssecurity.asp

that are available to .NET Framework applications. User security focuses around the identity and the capabilities of the user, while code access security focuses around the source and author of the code and the associated permissions. Code security requires authorizing the rights of the code to access the file system, registry, network, directory services and directory services. The main distinction here is user capabilities vs. code permissions.

PKI:

Public-key infrastructure (PKI) is the combination of software, encryption technologies, and services that enables enterprises to protect the security of their communications and business transactions on the Internet. PKIs integrate digital certificates, public-key cryptography, and certificate authorities into a total, enterprise-wide network security architecture. A typical enterprise's PKI encompasses the issuance of digital certificates to individual users and servers; end-user enrollment software; integration with corporate certificate directories; tools for managing, renewing, and revoking certificates; and related services and support.[1]

A PKI enables users of a basically unsecured public network such as the Internet to securely and privately exchange data and money through the use of a public and a private cryptographic key pair that is obtained and shared through a trusted authority. The public key infrastructure provides for a digital certificate that can identify an individual or an organization and directory services that can store and, when necessary, revoke the certificates.

The public key infrastructure assumes the use of public key cryptography, which is the most common method on the Internet for authenticating a message sender or encrypting a message. Traditional cryptography has usually involved the creation and sharing of a secret key for the encryption and decryption of messages. This secret or private key system has the significant flaw that if the key is discovered or intercepted by someone else, messages can easily be decrypted. For this reason, public key cryptography and the public key infrastructure is the preferred approach on the Internet. (The private key system is sometimes known as symmetric cryptography and the public key system as asymmetric cryptography.)

A public key infrastructure consists of:

- A certificate authority (CA) that issues and verifies digital certificate. A certificate includes the public key or information about the public key
- A registration authority (RA) that acts as the verifier for the certificate authority before a digital certificate is issued to a requestor
- One or more directories where the certificates (with their public keys) are held
- A certificate management system

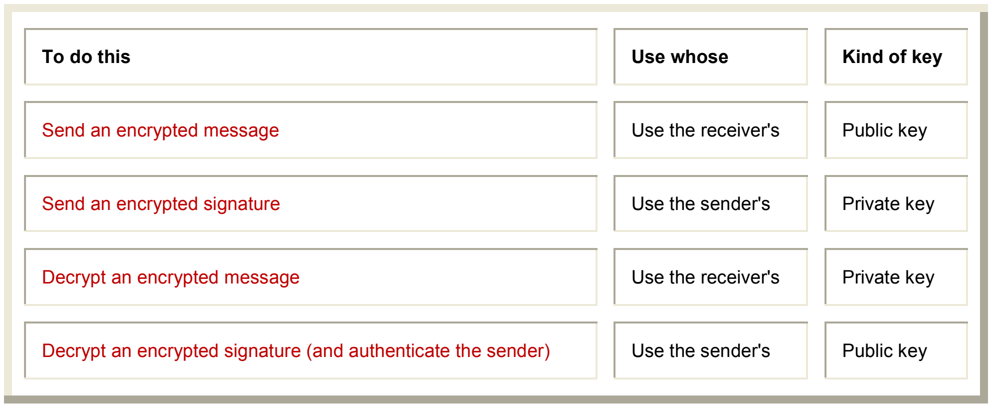
[1] http://verisign.netscape.com/security/pki/understanding.html

In public key cryptography, a public and private key are created simultaneously using the same algorithm (e.g. RSA) by a certificate authority (CA). The private key is given only to the requesting party and the public key is made publicly available (as part of a digital certificate) in a directory that all parties can access. The private key is never shared with anyone or sent across the Internet.[1]

RSA:

A public-key cryptosystem for both encryption and authentication, invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman. Its name comes from their initials. The RSA algorithm works as follows: take two large prime numbers, p and q, and find their product n = pq; n is called the modulus. Choose a number, e, less than n and relatively prime to (p-1)(q-1), and find its inverse, d, mod (p-1)(q-1), which means that ed = 1 mod (p-1)(q-1); e and d are called the public and private exponents, respectively. The public key is the pair (n,e); the private key is d. The factors p and q must be kept secret, or destroyed. It is difficult (presumably) to obtain the private key d from the public key (n,e). If one could factor n into p and q, however, then one could obtain the private key d. Thus the entire security of RSA depends on the difficulty of factoring; an easy method for factoring products of large prime numbers would break RSA. [2] RSA is the most popular algorithm in use to support the PKI.

The private key is used to decrypt text that has been encrypted with the public key. Thus, if Bob sends Alice a message, Bob can find out Alice's public key (but not Alice's private key) from Alice's certificate provided by the certificate authority (CA) and encrypt a message to Alice using her public key. When Alice receives it, she decrypts it with her private key. In addition to encrypting messages (which ensures privacy), Bob can authenticate himself to Alice (so Alice knows that it is really Bob who sent the message) by using his private key to sign the message. When Alice receives it, she can use Bob's public key to decrypt it.[3]



To do this	Use whose	Kind of key
Send an encrypted message	Use the receiver's	Public key
Send an encrypted signature	Use the sender's	Private key
Decrypt an encrypted message	Use the receiver's	Private key
Decrypt an encrypted signature (and authenticate the sender)	Use the sender's	Public key

[1] http://whatis.techtarget.com/definition/0,289893,sid9_gci214299,00.html
[2] http://dict.die.net/rsa%20encryption/
[3] http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci214273,00.html

Figure: Public/Private Key Usage [1]

Salt Value:

In order to make encrypted data (e.g. passwords) more resistant to attacks known as dictionary attacks that fall in the general category of "chosen plaintext" attacks, a salt value may often be "mixed" with the plaintext before it is encrypted. Some transformation function may use a salt value (random string of bits) to transform the plaintext into some intermediate form before it is encrypted. This makes it substantially harder for an attacker, who does not know the salt value, to launch a "chosen plaintext" or a dictionary attack.

Salt values are often used to protect system passwords. When a user picks a system password, it is first mixed with the salt value, then hashed, and only then stored in the database. Salt value must also be stored securely. Subsequently, when a user attempts to log into the system, his supplied password is again mixed with the salt value, hashed, and the value compared with that stored in the database. If the two hashes match, then authentication is successful. If an attacker were to compromise that database storing all of the hashed password values, it would be more difficult for the attacker to get the plaintext password value that hashed to the stored hash value because the randomness of the password would have been increased by the salt value. In other words, salt value is just a way to increase the randomness of the plaintext. In the case with passwords, if the original password is generated in a random fashion, as opposed to being picked by a user, then the salt value is not necessary.

Salt values may also be often used with encryption. Since the data encrypted is not random most of the time, salt values can be used to increase the randomness of that data and make it more resistant to "chosen plaintext" attacks. Salt values may also be used as a collision control mechanisms for hashing.

SAML:

SAML (Security Assertion Markup Language) is an Extensible Markup Language (XML) standard that allows a user to log on once for affiliated but separate Web sites. SAML is designed for business-to-business (B2B) and business-to-consumer (B2C) transactions. SAML specifies three components: assertions, protocol, and binding. There are three assertions: authentication, attribute, and authorization. Authentication assertion validates the user's identity. Attribute assertion contains specific information about the user. And authorization assertion identifies what the user is authorized to do.
Protocol defines how SAML asks for and receives assertions. Binding defines how SAML message exchanges are mapped to Simple Object Access Protocol (SOAP) exchanges. SAML works with multiple protocols including Hypertext Transfer Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP) and also supports SOAP, BizTalk, and Electronic Business XML (ebXML). The Organization for

[1] http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci214273,00.html

the Advancement of Structured Information Standards (OASIS) is the standards group for SAML.[1] The diagram below illustrates how SAML works.

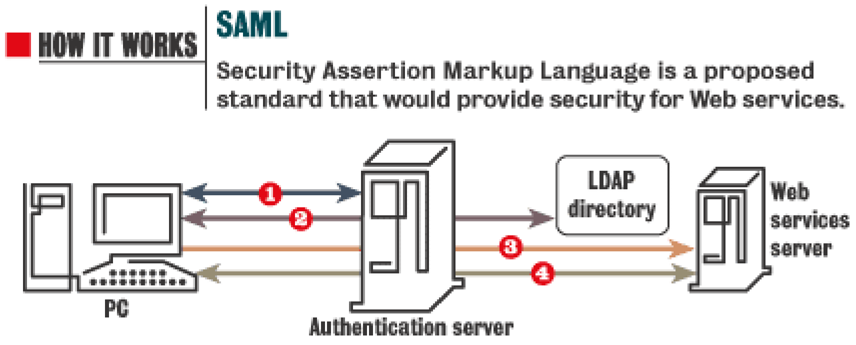

HOW IT WORKS
SAML
Security Assertion Markup Language is a proposed standard that would provide security for Web services.
1
2
3
4
LDAP directory
Web services server
PC
Authentication server

Figure: SAML – How it works[2]

The four steps in the diagram above are:[3]

1) End user's browser accesses authentication server, and authentication server asks for user ID and password.

2) End user enters ID and password. Authentication server checks with LDAP directory and then authenticates end user.

3) End user requests a resource from destination/Web services server. Authentication server opens a session with destination server.

4) Authentication server sends uniform resource identifier (URI) to end user. End user browser is redirected to URI, which connects end user to Web service.

SOAP:

SOAP (Simple Object Access Protocol) is a way for a program running in one kind of operating system (such as Windows 2000) to communicate with a program in the same or another kind of an operating system (such as Linux) by using the World Wide Web's Hypertext Transfer Protocol (HTTP) and its Extensible Markup Language (XML) as the mechanisms for information exchange. Since Web protocols are installed and available for use by all major operating system platforms, HTTP and XML provide an already at-hand solution to the problem of how programs running under different operating systems in a network can communicate with each other. SOAP specifies exactly how to encode an HTTP header and an XML file so that a program in one computer can call a program in another computer and pass it information. It also specifies how the called program can return a response.

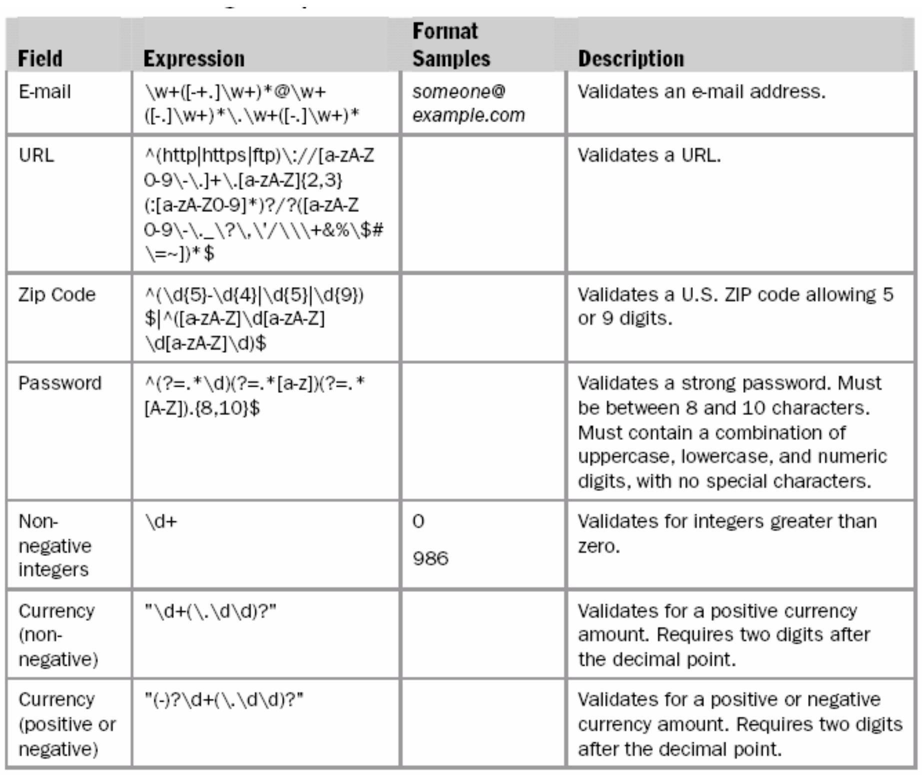
[1] http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci839675,00.html
[2] http://www.nwfusion.com/details/539.html
[3] http://www.nwfusion.com/details/539.html

SOAP was developed by Microsoft, DevelopMentor, and Userland Software and has been proposed as a standard interface to the Internet Engineering Task Force (IETF). It is somewhat similar to the Internet Inter-ORB Protocol (IIOP), a protocol that is part of the Common Object Request Broker Architecture (CORBA). Sun Microsystems' Remote Method Invocation (RMI) is a similar client/server interprogram protocol between programs written in Java. An advantage of SOAP is that program calls are much more likely to get through firewall servers that screen out requests other than those for known applications (through the designated port mechanism). Since HTTP requests are usually allowed through firewalls, programs using SOAP to communicate can be sure that they can communicate with programs anywhere.[1]

SQL Injection:

SQL injection is a particularly common and serious type of injection, where SQL statements are passed to the web application and then without validation are passed to the routine that accesses the database with that SQL statement. All input used for data access should be thoroughly validated, primarily in order to avoid SQL injection attacks which can happen if dynamic queries are generated based on user input without first thoroughly validating the user input. An attacker can then possibly inject malicious SQL commands that will be executed by the database. To validate input used for dynamic query construction regular expressions should be used to restrict input. For defense in depth the input should also be sanitized. Additionally, whenever possible, it is a good idea to use stored procedure for data access in order to make sure that type and length checks are performed on the data prior to it being used in SQL queries. The table below lists a common list of useful regular expressions that developers should use for input validation.

[1] http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci214295,00.html?Offer=SEcpwslg25



Field	Expression	Format Samples	Description
E-mail	\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*	*someone@example.com*	Validates an e-mail address.
URL	^(http|https|ftp)\://[a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,3}(:[a-zA-Z0-9]*)?/?([a-zA-Z0-9\-\._\?\,\'/\\\+&%\$#\=~])*$		Validates a URL.
Zip Code	^(\d{5}-\d{4}|\d{5}|\d{9})$\|^([a-zA-Z]\d[a-zA-Z]\d[a-zA-Z]\d)$		Validates a U.S. ZIP code allowing 5 or 9 digits.
Password	^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,10}$		Validates a strong password. Must be between 8 and 10 characters. Must contain a combination of uppercase, lowercase, and numeric digits, with no special characters.
Non-negative integers	\d+	0 986	Validates for integers greater than zero.
Currency (non-negative)	"\d+(\.\d\d)?"		Validates for a positive currency amount. Requires two digits after the decimal point.
Currency (positive or negative)	"(-)?\d+(\.\d\d)?"		Validates for a positive or negative currency amount. Requires two digits after the decimal point.

Table 11: Common Regular Expressions for Input Validation

Developers armed with knowledge of the various ways to perform input validation and skills working with regular expressions are in position to mitigate (and virtually eliminate) application vulnerabilities that may cause buffer overflows, cross site scripting, SQL injection and code injection problems.

Some code examples from ASP.NET are offered below for mitigating the SQL injection risks. SQL injection attacks can takes place when input to the application is used to construct dynamic SQL statements that access the database. SQL injection attacks can also occur if code uses stored procedures that are passed strings containing unfiltered input from the user. SQL injections can have the effect of allowing attackers to execute commands in the database and can become even a bigger issue if a process performing database access has excessive privileges. SQL injections can be mitigated by constraining input and using type safe SQL parameters.

Input should be constrained for type, length, format (via regular expressions) and range. If the input comes from a trusted source that has performed input validation, it may not be necessary to validate the input again (although defense in depth principle would recommend that you do so anyway). On the other hand, if the input is from an untrusted source, proper validation on the input should be performed prior to using that input to in SQL statement construction.

The Parameters collection in SQL can be used for input validation where input is treated as a literal value and SQL does not treat it as executable code. Parameters collection in SQL can be used to perform type checking and length validation which also helps enforce type and length checks. Values outside the valid ranges will trigger an exception. Stored procedure should be used where possible and they should be called with Parameters collection. The code below demonstrates how to use the Parameters collection with stored procedures. By themselves, stored procedures may be susceptible to SQL injections if they are passed unfiltered input, but coupled with usage of Parameters collection, the problem goes away.

```
SqlDataAdapter myCommand = new SqlDataAdapter("AuthorLogin", conn);
myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;
SqlParameter parm = myCommand.SelectCommand.Parameters.Add(
"@au_id", SqlDbType.VarChar, 11);
parm.Value = Login.Text;
```

In the case dynamic SQL, where stored procedures cannot be used, Parameters collection should still be used. An example of that is shown below.

```
SqlDataAdapter myCommand = new SqlDataAdapter(
"SELECT au_lname, au_fname FROM Authors WHERE au_id = @au_id", conn);
SqlParameter parm = myCommand.SelectCommand.Parameters.Add("@au_id",
SqlDbType.VarChar, 11);
parm.Value = Login.Text;
```

It is important to note that Parameters collection can also be used when concatenating several SQL statements to send a batch of statements to the server at one time. In this case, the names of the parameters should not be repeated.

Filter routines can be added to protect against SQL injection by replace all “unsafe” characters in the input that have a special meaning to SQL (e.g. a single apostrophe character). A way to do this is shown below.

```
private string SafeSqlLiteral(string inputSQL)
{
return inputSQL.Replace("'", "''");
}
```

However, a skillful attacker could use ASCII hexadecimal characters to bypass these checks, so the use of these filtering routines alone is not sufficient.

SSL:

The Secure Sockets Layer (SSL) is a commonly-used protocol for managing the security of a message transmission on the Internet. SSL has recently been succeeded by Transport Layer Security (TLS), which is based on SSL. SSL uses a program layer located between the Internet's Hypertext Transfer Protocol (HTTP) and Transport Control Protocol (TCP) layers. SSL is included as part of both the Microsoft and Netscape browsers and most Web server products. Developed by Netscape, SSL also gained the support of Microsoft and other Internet client/server developers as well and became the de facto standard until evolving into Transport Layer Security. The "sockets" part of the term refers to the sockets method of passing data back and forth between a client and a server program in a network or between program layers in the same computer. SSL uses the public-and-private key encryption system from RSA, which also includes the use of a digital certificate.

TLS and SSL are an integral part of most Web browsers (clients) and Web servers. If a Web site is on a server that supports SSL, SSL can be enabled and specific Web pages can be identified as requiring SSL access. Any Web server can be enabled by using Netscape's SSLRef program library which can be downloaded for noncommercial use or licensed for commercial use. TLS and SSL are not interoperable. However, a message sent with TLS can be handled by a client that handles SSL but not TLS.[1]

Many programming languages provide APIs that give programmers access to SSL functionality. Java Secure Sockets Extension (JSSE) provides support for SSL and TLS protocols by making them programmatically available. WebLogic implementation of JSSE also provides support for JCE Cryptographic Service Providers. HTTPS port is used for SSL protected sessions. SSL encrypts the data transmitted between the client and the server ensuring confidentiality of the username and password. SSL scheme uses certificates and thus requires certificate authentication not supported by JAAS. As the result, when SSL is used, an alternate authentication scheme must be used, one that supports certificates, namely Java Naming Directory Interface (JNDI) authentication. For client certificate authentication, a two-way SSL authentication scheme is used that is also referred to as mutual authentication. A common problem organizations have implementing SSL is failing to perform authentication properly. The result is secure communication with a remote host that has not been properly authenticated. It is thus critical to follow all of necessary steps to perform authentication correctly. A code example below demonstrates how one-way SSL authentication should be performed using JNDI authentication.

One-Way SSL Authentication:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
env.put(Context.PROVIDER_URL, “t3s://weblogic:7002”);
env.put(Context.SECURITY_PRINCIPAL, “javaclient”);
env.put(Context.SECURITY_CREDENTIALS, “javaclientpassword”);
ctx = new InitialContext(env);
```

SSL client application typically has several components. A java client initializes an SSLContextwith client identity, a HostnameVerifierJSSE, a TrustManagerJSSE, and a HandshakeCompletedListener. It then creates a key store and retrieves the private key and certificate chain. An SSLSocketFactory is then used. Finally, HTTPS is used to connect to a JSP served by an instance of the application server. Another component is

[1] http://whatis.techtarget.com/definition/0,289893,sid9_gci343029,00.html

the HostnameVerifier that provides a callback mechanism so that developers can supply a policy for handling situations where the host is being connected to the server name from the certificate Subject Distinguished Name. A HandshakeCompletedListener defines how SSL client receives messages about the termination of an SSL handshake on a particular SSL connection. The number of times an SSL handshake can take place on a particular SSL connection is also defined. A TrustManager builds a certificate path to a trusted root and returns true if the certificate is valid. A build script compiles all of the files required and deploys them.

Two-way SSL authentication can be used if a mechanism is needed for the two parties to mutually authenticate each other. For instance, two servers may need to communicate to each other securely and may utilize two-way SSL authentication. This allows to have a dependable and secure communication. Typically mutual authentication is used in client-server environments, but it may also be used in server-server communication. An example below demonstrates establishment of a secure connection between two Weblogic server instances:

Two-Way SSL Authentication Between Server Instances:

```
FileInputStream [] f = new FileInputStream[3];
f[0]= new FileInputStream("demokey.pem");
f[1]= new FileInputStream("democert.pem");
f[2]= new FileInputStream("ca.pem");
Environment e = new Environment ();
e.setProviderURL("t3s://server2.weblogic.com:443");
e.setSSLClientCertificate(f);
e.setSSLServerName("server2.weblogic.com");
e.setSSLRootCAFingerprints("ac45e2d1ce492252acc27ee5c345ef26");
e.setInitialContextFactory
("weblogic.jndi.WLInitialContextFactory");
Context ctx = new InitialContext(e.getProperties())
```

There are many steps to implementing SSL properly and developers should be aware of those. Misusing SSL could render it useless. There are too many detailed nuances to discuss all of them in this paper. Application component developers in charge of programmatic security need to have deep understanding of underlying technology and how to properly use it.

Static Security Analysis:

Static analysis is any software analysis that involves analyzing the software without executing it. Because static analysis entails analyzing source code, it enables application project teams to get a comprehensive assessment of an application's functionality earlier in the development cycle, when the cost of fixing vulnerabilities is much less. Only static analysis can provide the level of assurance required for business-critical proprietary applications. Vulnerabilities in these applications are typically unknown, unexamined by the global security community, and unprotected by operational vulnerability scanning methods. The downside to static analysis of security vulnerabilities has been the lack of enterprise-class tools, making the approach too costly and complex to use broadly.

Whereas dynamic analysis looks for circumstantial evidence, static analysis looks for facts. Static analysis tools enable deep structural analysis of code and its intended behavior, such as data flow, function calls, and loop-and-branch analysis. In effect, static analysis lets you test "what you don't know," or the possible root cause of misbehaving code, by comparing code against a database of known vulnerabilities. On the plus side, static analysis provides concrete results that identify vulnerabilities, their source in the code base, and their potential outcomes. Because static analysis works on source code and can test code without running a fully staged version of an application, it can be used more interactively by project teams as part of their daily workflow. Project teams can thus identify problems and fix them earlier in the software development cycle, when they are much less expensive to fix. The total cost of ownership of automated static analysis tools is low, because they don't require staging a functional version of the application. By identifying and prioritizing vulnerabilities during the development cycle, static analysis also reduces the cost of identifying and fixing security vulnerabilities. (The cost of handling a security vulnerability includes the labor and management effort required to identify, test, remediate, track and document a vulnerability throughout the development cycle.) Vulnerabilities can be corrected by a few keystrokes by a developer, instead of by a multi-step, multi-person process. Static analysis also strengthens "defense-in-depth" strategies by providing detailed information about an application's possible failure points. Project teams can then classify and prioritize vulnerabilities based on level of current threat. As hackers develop different exploit techniques that have a greater percentage of completion against an application, developers can respond more quickly because vulnerabilities have been classified and thoroughly documented. Static analysis also provides improved return on investment (ROI), by providing value beyond analysis. The in-depth information provided by the static analysis can be used to identify patterns indicative of skills discrepancies or bad coding habits, enabling these to be corrected through training. On the minus side, static analysis traditionally hasn't been able to test for vulnerabilities introduced or exacerbated by the operating environment. In addition, static analysis has required significant security expertise to perform it and interpret the results, making it too expensive to use extensively in the development cycle. Also, first-generation automated static-analysis tools typically have lacked the analytical rigor and context to analyze security problems reliably and accurately, and they haven't been able to scale beyond code sections to entire applications. However, a new generation of automated, enterprise-class solutions is overcoming many of the traditional disadvantages of static analysis – providing enhanced simplicity, usability, specificity, reliability and scalability for managing security vulnerabilities. [1]

STRIDE:

Microsoft has developed a STRIDE model for categorizing software threats that can be used to drive the threat modeling process. The summary of the model is offered in the table below:

[1] "Risk in the Balance: Static Analysis vs. Dynamic Analysis" Secure Software Inc., 2004. Available from: www.securesoftware.com

Term	Definition
Spoofing identity	Illegally obtaining access and use of another person's authentication information, such as a user name or password.
Tampering with data	The malicious modification of data.
Repudiation	Associated with users who deny performing an action, yet there is no way to prove otherwise. (Non-repudiation refers to the ability of a system to counter repudiation threats, and includes techniques such as signing for a received parcel so that the signed receipt can be used as evidence.)
Information disclosure	The exposure of information to individuals who are not supposed to have access to it, such as accessing files without having the appropriate rights.
Denial of service	An explicit attempt to prevent legitimate users from using a service or system.
Elevation of privilege	Where an unprivileged user gains privileged access. An example of privilege elevation would be an unprivileged user who contrives a way to be added to the Administrators group.

Figure: STRIDE Model[1]

Threat Modeling:

One of the big questions that architects and designers of enterprise web applications need to answer is: what are the threats on this application? This question is very relevant, because after all, it is impossible to make sure that an application is secure without understanding the types of attacks that adversaries may attempt to launch on the system. The technique for formal evaluation of threats on the application is commonly called threat modeling. Threat modeling process usually starts at the information gathering phase, proceeds to the analysis phase, and culminates with a report that can be used by application architects, designers, developers and security auditors to drive construction and verification.

A typical threat model would first document all of the protected resources of the application, such as data and execution resources. After all, attackers often try to get at those protected resources so it is imperative to know what these are. To get to those resources an attacker would have to use some input or output stream flowing to or from the application. The next step in threat modeling is to isolate the input and output vectors of the application, document the various threats as they apply to each of those input and output vectors and construct attack trees that combine the various input and output vector threats to abstract the various attack scenarios on the application being evaluated. While it is impractical and probably impossible to try and document every single possible attack scenario on the enterprise web application, threat modeling process provides a systematic way of considering the threats thus reducing the chance that some are overlooked and pointing out some of the potential weak or problem areas that need to be given special attention. Threat modeling helps comply with the principle of securing the weakest by

[1] http://msmvps.com/secure/archive/2004/06/22/8728.aspx

facilitating in understanding of what the weak links are. As part of threat modeling it is important to document all of the security related assumptions and decisions made as this information will help understand what affect architectural changes will have on the security posture of the application. After having evaluated the threats, it should be documented what threats are mitigated by the architecture, design and implementation and how that is accomplished. All threats that are not addressed should be clearly documented as well. The threats should also be rated based on severity and potential impact. Microsoft's approach to thereat modeling focuses on the STRIDE model.

TLS:

Transport Layer Security (TLS) is a protocol that ensures privacy between communicating applications and their users on the Internet. When a server and client communicate, TLS ensures that no third party may eavesdrop or tamper with any message. TLS is the successor to the Secure Sockets Layer (SSL). TLS is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. The TLS Record Protocol provides connection security with some encryption method such as the Data Encryption Standard (DES). The TLS Record Protocol can also be used without encryption. The TLS Handshake Protocol allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before data is exchanged. The TLS protocol is based on Netscape's SSL 3.0 protocol; however, TLS and SSL are not interoperable. The TLS protocol does contain a mechanism that allows TLS implementation to back down to SSL 3.0. The most recent browser versions support TLS. The TLS Working Group, established in 1996, continues to work on the TLS protocol and related applications.[1]

Web Services:

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.[2]

Web services (sometimes called application services) are services (usually including some combination of programming and data, but possibly including human resources as well) that are made available from a business's Web server for Web users or other Web-connected programs. Providers of Web services are generally known as application service providers. Web services range from such major services as storage management and customer relationship management (CRM) down to much more limited services such as the furnishing of a stock quote and the checking of bids for an auction item. The accelerating creation and availability of these services is a major Web trend.
Users can access some Web services through a peer-to-peer arrangement rather than by going to a central server. Some services can communicate with other services and this

[1] http://searchsecurity.techtarget.com/gDefinition/0,294236,sid14_gci557332,00.html
[2] http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/

exchange of procedures and data is generally enabled by a class of software known as middleware. Services previously possible only with the older standardized service known as Electronic Data Interchange (EDI) increasingly are likely to become Web services. Besides the standardization and wide availability to users and businesses of the Internet itself, Web services are also increasingly enabled by the use of the Extensible Markup Language (XML) as a means of standardizing data formats and exchanging data. XML is the foundation for the Web Services Description Language (WSDL).
As Web services proliferate, concerns include the overall demands on network bandwidth and, for any particular service, the effect on performance as demands for that service rise. A number of new products have emerged that enable software developers to create or modify existing applications that can be "published" (made known and potentially accessible) as Web services.[1]

WSDL:

The Web Services Description Language (WSDL) is an XML-based language used to describe the services a business offers and to provide a way for individuals and other businesses to access those services electronically. WSDL is the cornerstone of the Universal Description, Discovery, and Integration (UDDI) initiative spearheaded by Microsoft, IBM, and Ariba. UDDI is an XML-based registry for businesses worldwide, which enables businesses to list themselves and their services on the Internet. WSDL is the language used to do this. WSDL is derived from Microsoft's Simple Object Access Protocol (SOAP) and IBM's Network Accessible Service Specification Language (NASSL). WSDL replaces both NASSL and SOAP as the means of expressing business services in the UDDI registry.[2]

As communications protocols and message formats are standardized in the web community, it becomes increasingly possible and important to be able to describe the communications in some structured way. WSDL addresses this need by defining an XML grammar for describing network services as collections of communication endpoints capable of exchanging messages. WSDL service definitions provide documentation for distributed systems and serve as a recipe for automating the details involved in applications communication.

A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Hence, a WSDL document uses the following elements in the definition of network services:

[1] http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci750567,00.html
[2] http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci521683,00.html

Types– a container for data type definitions using some type system (such as XSD).

Message– an abstract, typed definition of the data being communicated.

Operation– an abstract description of an action supported by the service.

Port Type–an abstract set of operations supported by one or more endpoints.

Binding– a concrete protocol and data format specification for a particular port type.

Port– a single endpoint defined as a combination of a binding and a network address.

Service– a collection of related endpoints.

It is important to observe that WSDL does not introduce a new type definition language. WSDL recognizes the need for rich type systems for describing message formats, and supports the XML Schemas specification (XSD) as its canonical type system. However, since it is unreasonable to expect a single type system grammar to be used to describe all message formats present and future, WSDL allows using other type definition languages via extensibility.

In addition, WSDL defines a common binding mechanism. This is used to attach a specific protocol or data format or structure to an abstract message, operation, or endpoint. It allows the reuse of abstract definitions.
In addition to the core service definition framework, this specification introduces specific binding extensions for the following protocols and message formats:
SOAP 1.1, HTTP GET / POST and MIME.

Although defined within this document, the above language extensions are layered on top of the core service definition framework. Nothing precludes the use of other binding extensions with WSDL.[1]

X509 Certificate:

A public-key certificate binds a public-key value to a set of information that identifies the entity (such as person, organization, account, or site) associated with use of the corresponding private key (this entity is known as the "subject" of the certificate). A certificate is used by a "certificate user" or "relying party" that needs to use, and rely upon the accuracy of, the public key distributed via that certificate (a certificate user is typically an entity that is verifying a digital signature from the certificate's subject or an entity sending encrypted data to the subject). The degree to which a certificate user can trust the binding embodied in a certificate depends on several factors. These factors include the practices followed by the certification authority (CA) in authenticating the subject; the CA's operating policy, procedures, and security controls; the subject's obligations (for example, in protecting the private key); and the

[1] http://www.w3.org/TR/wsdl#_introduction

stated undertakings and legal obligations of the CA (for example, warranties and limitations on liability).

A X.509 certificate may contain a field declaring that one
or more specific certificate policies applies to that certificate. According to X.509, a certificate policy is "a named set of rules that indicates the applicability of a certificate to a particular community and/or class of application with common security requirements." A certificate policy may be used by a certificate user to help in deciding whether a certificate and the binding therein, are sufficiently trustworthy for a particular application. X.509 certificates are widely used as part of PKI.[1]

Cross Site Scripting (XSS):

Cross site scripting attacks exploits vulnerabilities that fall in the category of poor input validation. Essentially an attacker submits executable scripts as part of the input to the web application and those scripts are then executed on the browsers of other clients. Those attacks often lead to information disclosure of the end user's session tokens, attack the end user's machine or spoof content to fool the end user. Disclosure of session tokens can lead to session hijacking and allow an attacker to assume a valid user's identity (compromise authentication). Spoofing content can also lead to information disclosure if for instance a valid user input his/her login and password information into a form sent to an attacker. XSS attacks can occur at the web tier or at the application tier and aggressive white list input validation should be present in the application to thwart these attacks. There are two types of XSS attacks: stored and reflected. In stored XSS attacks, the malicious script injected by an attacker is permanently stored by the web application for later retrieval by the end user who requests the affected data. Since the malicious script at that point arrived from a trusted server, the client executes the script. In reflected attacks, the malicious script is transferred to the server and then is echoed back to the user either in an error message, search result, or some other response to the end user which includes some of the data fields into which a malicious script has been inserted as part of the request.

[1] http://www.faqs.org/rfcs/rfc2527.html

Appendix C – References

[1] Viega, J., McGraw, G. (2002). "Building Secure Software." Addison-Wesley, New York, NY.

[2] Coulouris, G., Dollimore, J., Kindberg, T. (2001). "Distributed Systems Concepts and Design." Addison-Wesley, 3rd edition, New York, NY.

[3] Pfleeger, C. (1997). "Security in Computing." Prentice Hall PTR, 2nd edition, Upper Saddle River, NJ.

[4] Asbury, S., Weinter, S. (1999). "Developing Java Enterprise Applications." Wiley Computer Publishing, New York, NY.

[5] Conklin, W., White, G., Cothren, C. (2004). "Principles of Computer Security." McGraw Hill Technology Education, Burr Ridge, IL.

[6] Viega, J., Messier, M. (2003). "Secure Programming Cookbook." O'Reilly, First Edition, Sebastopol, CA.

[7] Williams, Jeff. "How to Attack the J2EE platform," JavaOne Sun's Worldwide Java Developers Conference, 2004.

[8] Begin, Clinton. "Implementing the Microsoft .NET Pet Shop using Java," www.iBatis.com, 2002.

[9] The Middleware Company Case Study Team, "J2EE and .NET (Reloaded) Yet Another Performance Case Study," The Middleware Company Performance Case Study, 2003.

[10] Meier, J.D. (2003). "Improving Web Application Security: Threats and Countermeasures," Microsoft Corporation, Redmond, WA. (at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/threatcounter.asp)

[11] Viega, J. (2005). "CLASP Reference Guide." Secure Software Inc., McLean, VA.

[12] Lipner, S. "The Trustworthy Computing Security Development Lifecycle," Microsoft Corporation, Redmond, WA. (at: http://msdn.microsoft.com/security/default.aspx?pull=/library/en-us/dnsecure/html/sdl.asp)

[13] DeLooze, L. (2004). "Applying Security to an Enterprise using the Zachman Framework," (at: http://www.securitydocs.com/library/2129)

[14] Swanson, M. (2001). "Security Self-Assessment Guide for Information Technology Systems," NIST SP800-26.

[15] Ross, R. (2005). "Recommended Security Controls for Federal Information Systems," NIST SP800-53.

[16] Grance, T. (2004). "Security Considerations in the Information System Development Life Cycle," NIST SP800-64.

[17] "Business-Centric Methodology for Enterprise Agility and Interoperability," Oasis, 2003.

[18] Curphey, M. "A Guide to Building Secure Web Applications," OWASP. (at: http://www.cgisecurity.com/owasp/html/)

[19] "BEA Weblogic Server: Programming Weblogic Security," BEA Systems Inc., 2004. (at: http://e-docs.bea.com/wls/docs81/pdf/security.pdf)

[20] Ross, R. (2004). "Guide for the Security Certification and Accreditation of Federal Information Systems," NIST SP800-37.

[21] Lyons, B. "The Platform Wars: .NET vs. J2EE," Number Six Software, Inc. (at: http://www.numbersix.com/v4/csdi/resources.html)

[22] Watkins, D. (2002). "An Overview of Security in the .NET Framework," Microsoft Corporation, Redmond, WA.

[23] "J2EE BluePrints," (at: http://java.sun.com/blueprints/guidelines/designing_enterprise_applications/apmTOC.html)

[24] "OWASP Top Ten," OWASP. (at: http://www.owasp.org/documentation/topten.html)

[25] "Secure Coding Guidelines for the .NET Framework," Microsoft Corporation, Redmond, WA, 2002. (at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/seccodeguide.asp)

[26] BEA Product Documentation, "Securing Web Applications," BEA Systems. (at: http://e-docs.bea.com/wls/docs81/security/thin_client.html)

[27] MacLennan, B. (1987). "Principles of Programming Languages." Holt, Rinehart and Winston.

[28] .NET Framework Developer's Guide, "Overview of the .NET Framework," MSDN, Microsoft Corporation, Redmond, WA. (at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/threatcounter.asp)

[29] Mulcahy, G. "J2EE and .NET Security," CGI Security, 2002. (at: http://www.cgisecurity.com/lib/J2EEandDotNetsecurityByGerMulcahy.pdf)