Phalanx 2 Revealed: Using Volatility to Analyze an Advanced Linux Rootkit

Archived: 2026-04-02 11:36:58 UTC

Month of Volatility Plugins

In this blog post I will analyze the Phalanax2 rootkit using both Volatility as well as traditional malware analysis techniques.

Phalanx2

Phalanx2 (P2) is the latest version of a private rootkit, whose original source was leaked to PacketStorm back in late 2005. Since then there have been no public leaks of either the source code or the complete set of files for using the rootkit (backdoor, client, config instructions, etc). Instead, the only occurrences of it were from sysadmins and IR teams who found that their systems were infected with it. None of these teams have ever released the files public, so deep analysis of the rootkit is not available in any public forums.

With that said, we recently got our hands on a working sample (backdoor and config file only). Although we also are not allowed to release the sample, we can release our analysis of it.

The rootkit comes as a statically compiled userland ELF file with stripped symbols. The config file specifies the group ID of processes to hide, the prefix of filenames to hide, and a hash value that we did not determine the purpose of yet.

Analysis Setup & Approach

Analysis was done on a Debian 6.0.3 (Squeeze) 32 bit VMware virtual machine running the 2.6.32-5-686 #1 SMP kernel. I also compiled a custom kernel, explained later, that let me automate some of the analysis.

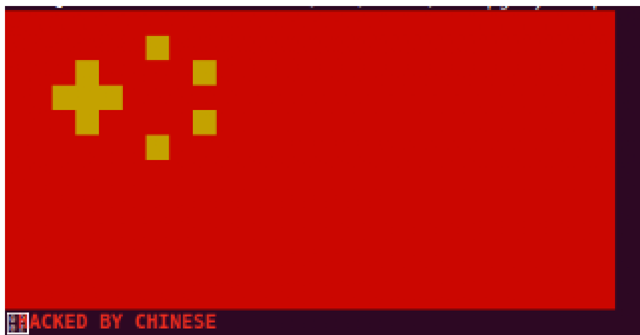
Our sample included three files, the userland binary that controls all functionality and a .config file that specified the group ID to hide, the hidden directory for files, and a .p2rc file used for when communicating with the backdoor. When running, P2 produced a file of recorded keystrokes into the hidden directory.

The approach during this analysis was a mix of both static and dynamic analysis using Volatility, IDA Pro, custom dynamic monitoring code, and the usual Linux analysis tools (gdb, strace, etc).

Analysis with Volatility

I will first walk through the analysis of P2 with Volatility. To determine the changes that the rootkit makes, I booted the VM, took a memory capture with LiME, installed P2, and then took another memory capture. Note that if you run the program without arguments you are greeted with a nice 'HACKED BY CHINESE' message that can be seen below:

 # insmod ./lime-2.6.32-5-686.ko "format=lime path=before-blog-post.lime"
 # ./phalanx2



 # ./phalanx2 i

(_- phalanx 2.5f -_)

; mmap failed..bypassing /dev/mem restrictions

; locating sys_call_table..

; sys_call_table_phys = 0x12742b0

; phys_base = 0x0

; sys_call_table = 0xc12742b0

; hooking.. [8=====================D]

; locating &tcp4_seq_show..................... found

>>injected

 # insmod ./lime-2.6.32-5-686.ko "format=lime path=after-blog-post.lime"

The output from P2, assuming that it can be trusted, seems like we will need to investigate /dev/mem, the system call table, and the */proc* handlers for the TCP protocol. If we tried to install P2 while the hooks are still active we get an error message:

 # ./phalanx2 i

(_- phalanx 2.5f -_)

fatal: already injected?

I then ran a number of plugins and diff'ed their output to determine the effects of the rootkit.

Hidden Processes

 # python vol.py --profile=Linuxthisx86 -f before-blog-post.lime linux_pslist > pslist-before

Volatile Systems Volatility Framework 2.2

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_pslist > pslist-after

Volatile Systems Volatility Framework 2.2

 # diff pslist-before pslist-after

64c64,65

< 0xf6698000 insmod   1292 0 0   Sun, 07 Oct 2012 03:47:59 +0000

> 0xf669e600 **Xnest**   1319 0 42779 Sun, 07 Oct 2012 03:52:58 +0000

> 0xf671aa80 insmod   1353 0 0   Sun, 07 Oct 2012 03:53:33 +0000

In this output we can see that insmod is different, as we should expect since we ran LiME twice and unloaded it in between runs. We also see a process named Xnest with a PID of 1319 and a GID of 42779.   This is the userland process spawned by P2, and 42779 is the GID to hide from userland. If we investigate this process with linux_psaux, which gathers arguments from userland, we see that the process name is different **-** it is disguised as a kernel thread (because the name is enclosed in brackets).

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_psaux -p 1319

Volatile Systems Volatility Framework 2.2

Pid   Uid   Gid   Arguments

1319   0   42779 [ata/0]

Although the Xnest process is hidden, it would look very strange on a normal system if it became uncovered, so P2 tries to blend the "Xnest" process name by disguising it as a normal kernel thread. An experienced investigator or system administrator would even find the new name suspicious though as the PID is very high for a kernel thread, which are all normally started soon after *init*. Similarly, the process will have memory maps even though the name is in brackets.

As final proof that our PID is not really a kernel thread, we can look at the output of the pstree plugin and see that Xnest is indeed not a child of the kernel thread daemon as all kernel threads should be:

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_pstree

<snip>

.Xnest   1319   0

[kthreadd]   2   0

.[migration/0]   3   0

.[ksoftirqd/0] 4 0

.[watchdog/0] 5 0

.[events/0] 6 0

.[cpuset] 7 0

.[khelper] 8 0

.[netns] 9 0

.[async/mgr] 10 0

.[pm] 11 0

<snip>

Memory Maps

Investigating the memory maps is pretty straightforward as the binary is statically compiled and does not load any libraries on its own. The memory maps between the binary and the stack may be of interest though, and the "rwx" mapping only adds to this interest. This is because memory mappings normally are either rw, ro, or rx, but not all three. The use of rwx pages is a common malware technique as it allows the malware to write to a memory buffer and then execute it.

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_proc_maps -p 1319

Volatile Systems Volatility Framework 2.2

9c27000 | 9c27000

0x8048000-0x805c000 r-x 0 8: 1 362195 /usr/share/XXXXXXXXXXXX.p2/.p-2.5f

0x805c000-0x805d000 rwx 81920 8: 1 362195 /usr/share/XXXXXXXXXXXX.p2/.p-2.5f

0x805d000-0x805f000 rwx 0 0: 0 0

0xaf891000-0xaf894000 rwx 0 0: 0 0

0xb7894000-0xb78ac000 rwx 0 0: 0 0

0xb78ac000-0xb78ad000 r-x 0 0: 0 0

0xbf874000-0xbf88a000 rwx 0 0: 0 0 [stack]

Open Files

If we investigate the open files we see that file descriptors 0, 1, and 2 are set to /dev/null, 3 is not present, and 4 and 5 are sockets. This is also fairly strange output as socket file descriptors are normally either dup'ed over the

inital 0, 1, and 2 file descriptors. In the binary analysis portion of this blog post we will see how these file descriptors are set.

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_lsof -p 1319

Volatile Systems Volatility Framework 2.2

Pid FD Path

-------- -------- ----

1319 0 /dev/null

1319 1 /dev/null

1319 2 /dev/null

1319 4 socket:[4441]

1319 5 socket:[4442]

Netstat

Next, we see that the open sockets actually connected over localhost to each other:

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_netstat

<snip>

TCP 127.0.0.1:40719 127.0.0.1:50271 ESTABLISHED Xnest/1319

TCP 127.0.0.1:50271 127.0.0.1:40719 ESTABLISHED Xnest/1319

<snip>

Again, this is quite abnormal…

Dmesg

 # python vol.py --profile=Linuxthisx86 -f before-blog-post.lime linux_dmesg > dmesg_before

Volatile Systems Volatility Framework 2.2

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_dmesg > dmesg_after

Volatile Systems Volatility Framework 2.2

 # diff dmesg_before dmesg_after

1150a1151,1159

> <6>[1573.826831] Program Xnest tried to access /dev/mem between 0->8000000.

> <4>[1576.063304] Xnest:1297 map pfn RAM range req write-back for 0-8000000, got uncached-minus

> <4>[1613.438913] [LiME] Parameters

> <4>[1613.438921] [LiME]   PATH: after-blog-post.lime

> <4>[1613.438925] [LiME]   DIO: 1

> <4>[1613.438928] [LiME]   FORMAT: lime

> <4>[1613.438931] [LiME] Initilizing Disk...

> <4>[1613.454205] [LiME] Direct IO may not be supported on this file system. Retrying.

> <4>[1613.454217] [LiME] Direct IO Disabled

In this output, we can see in the entries that the Xnest binary tried to access /dev/mem. The next line shows us the PID of Xnest, which the attentive reader will notice is different than the one we investigated with the previous plugins ;)

Loaded Modules

Investigating the loaded modules between memory captures produces interesting results. First, there is no difference between the two. This means that any active kernel module loaded by the rootkit would have to be hidden, but the check_modules plugin also reports no hidden modules. We will see in the binary analysis part why we cannot find any modules related to the rootkit.

 # python vol.py --profile=Linuxthisx86 -f before-blog-post.lime linux_lsmod > lsmod_before

Volatile Systems Volatility Framework 2.2

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_lsmod > lsmod_after

Volatile Systems Volatility Framework 2.2

 # diff lsmod_after lsmod_before

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_check_modules

Volatile Systems Volatility Framework 2.2

Module Name

Looking for Hooked Kernel Structures

As we have seen in [previous MoVP rootkits](), P2 hooks tcp4_seq_afino. This allows for trivially hiding network connections from userland.

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_check_afinfo

Volatile Systems Volatility Framework 2.2

Symbol Name Member Address

------------------- ----------- ----------

tcp4_seq_afinfo show 0xf7c7f000

We then see that P2 hooks a wide range of system calls:

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_check_syscall > p2syscalls

Volatile Systems Volatility Framework 2.2

 # grep HOOKED p2syscalls

32bit 0x3 0xf7c4f000 HOOKED

32bit 0x4 0xf7c53000 HOOKED

32bit 0x5 0xf7c40000 HOOKED

32bit 0xa 0xf7c57000 HOOKED

32bit 0xc 0xf7c3d000 HOOKED

32bit 0x25 0xf7c4c000 HOOKED

32bit 0x27 0xf7c6a000 HOOKED

32bit 0x53 0xf7c67000 HOOKED

32bit 0x60 0xf7c43000 HOOKED

32bit 0x66 0xf7c7c000 HOOKED

32bit 0x6a 0xf7c73000 HOOKED

32bit 0x6b 0xf7c70000 HOOKED

32bit 0x84 0xf7c5a000 HOOKED

32bit 0x8d 0xf7c3a000 HOOKED

32bit 0xc3 0xf7c79000 HOOKED

32bit 0xc4 0xf7c76000 HOOKED

32bit 0xdc 0xf7c37000 HOOKED

32bit 0x127 0xf7c64000 HOOKED

32bit 0x128 0xf7c6d000 HOOKED

32bit 0x12d 0xf7c61000 HOOKED

To determine exactly which system calls are hooked, I modified the plugin to print the decimal form of only the hooked system calls. I then used this numbers to grep on unistd:

 # grep -wf hooked-syscalls /usr/include/asm/unistd_32.h

#define __NR_read 3

#define __NR_write 4

#define __NR_open 5

#define __NR_unlink 10

#define __NR_chdir 12

#define __NR_kill 37

#define __NR_mkdir 39

#define __NR_symlink 83

#define __NR_getpriority 96

#define __NR_socketcall 102

#define __NR_stat 106

#define __NR_lstat 107

#define __NR_getpgid 132

#define __NR_getdents 141

#define __NR_stat64 195

#define __NR_lstat64 196

#define __NR_getdents64 220

#define __NR_openat 295

#define __NR_mkdirat 296

#define __NR_unlinkat 301

This tells us exactly which system calls are hooked and we can guess some of the rootkit's functionality based on this.

Dynamic & Binary Analysis

The Loaded Module

I started the analysis process by looking at the strings of the binary in hopes to find something interesting to target analysis on. The first thing I noticed was:

rm dummy.ko helper.ko p2.ko

From there I grepped for kernel modules and found:

 # grep \.ko strings

UN=`uname -r`;cp `find /lib/modules/$UN -name dummy.ko` .

could not find dummy.ko

helper.ko

ld -r helper.ko dummy.ko -o p2.ko

readelf -s p2.ko|grep ' init_module'|awk '{print $1}'

readelf -S p2.ko|grep symtab|awk '{print $5}'

insmod p2.ko 2>&1

rm dummy.ko helper.ko p2.ko

which showed quite a few interesting entries. From the output we can that P2 attempts to find the dummy network interface kernel module, and then links another module helper.ko with dummy.ko to produce p2.ko. This module (p2.ko) is eventually loaded and then the three modules are deleted.

At this point I wanted to get helper.ko and p2.ko so that I could reverse them to determine what in-kernel features P2 had. I was also intrigued as I did not see any calls to rmmod in the strings, although it could just have simply been obfuscated.

My first thought was to recover the deleted modules with the Sleuthkit, but unfortunately they were unrecoverable each time I tried. I then decided that I could simply get rid of the call to rm and they would stay on disk. For this, I made a copy of the rootkit and changed rm to aa (a non-existent command) so that the loading would error out. I then loaded the rootkit and was able to get the three kernel modules after the loading process aborted on attempting to execute the aa command.

I first examined helper.ko and saw that it only referenced two functions:

5: 00000000   48 FUNC   LOCAL DEFAULT   1 __memcpy

9: 00000000   142 FUNC   GLOBAL DEFAULT   4 module_helper

Investigation of module_helper inside of p2.ko (remember they are linked) showed that it was a fairly small function:

```
push    ebp             ; Alternative name is 'init_module'
mov     ebp, esp
sub     esp, 38h
mov     [ebp+hook_opcodes], 55h
mov     byte ptr [ebp-15h], 89h
mov     [ebp+var_14], 0E5h
mov     [ebp+var_13], 0B8h
mov     [ebp+var_12], 1
mov     [ebp+var_11], 0
mov     [ebp+var_10], 0
mov     [ebp+var_F], 0
mov     [ebp+var_E], 5Dh
mov     [ebp+var_D], 0C3h
mov     [ebp+page_size], 0
mov     eax, 0C101B165h ; devmem_is_allowed
mov     [ebp+devmem_is_allowed_addr], eax
mov     eax, 0C101CD6Eh ; set_memory_rw
mov     [ebp+set_memory_rw_addr], eax
mov     eax, 1000h
mov     [ebp+page_size], eax
cmp     [ebp+set_memory_rw_addr], 0
jz      short loc_8000181

mov     eax, [ebp+devmem_is_allowed_addr]
mov     edx, 0
div     [ebp+page_size]
mov     eax, [ebp-0Ch]
sub     eax, edx
mov     ecx, [ebp+set_memory_rw_addr]
mov     edx, 3
call    ecx

loc_8000181:
mov     edx, [ebp+devmem_is_allowed_addr]
mov     [esp+38h+var_30], 11h
lea     eax, [ebp+hook_opcodes]
mov     [esp+38h+var_34], eax
mov     [esp+38h+var_38], edx
call    __memcpy
mov     eax, 0FFFFFFFDh
leave
retn
module_helper endp
```

If you examine helper.ko's version of this function, the kernel address references 0xCxxxxxxx in p2.ko, are actually stored as 0xcacacaca, and are later fixed up with the addresses from the kernel that is going to be infected.

The rootkit relies on determining the address of devmem_is_allowed and set_memory_rw for the current kernel.   After gathering these addresses, the rootkit then marks the page for devmem_is_allowed writeable, and uses memcpy to overwrite the function. This code that is used to overwrite is declared in the beginning of the function starting with mov [ebp+hook_opcodes], 55h. If you disassemble these opcodes you will see:

 # perl -e 'print "\x55\x89\xE5\xB8\x01\x00\x00\x00\x5d\xc3"' > bin2

ndisasm -b32 bin2

00000000 55   push ebp

00000001 89E5   mov ebp,esp

00000003 B801000000   mov eax,0x1

00000008 5D   pop ebp

00000009 C3   ret

This function simply returns 1, which allows all addresses to be accessed. The outcome of this hooking is that the protections of /dev/mem are disabled and the full range of physical memory can be written to and read. If you remember from the output of running P2, we saw this line:

“; mmap failed..bypassing /dev/mem restrictions”

And now we know what it is referring to. You may also remember that the Volatility Linux plugins could not find any loadable kernel modules even though we are clearly seeing p2.ko being loaded into the system. The reason for this is evidence in the last few instructions of init_module*:*

mov eax,0xfffffffd

leave

ret

Since eax is used as the return value, this is the equivalent of “return -3” in C code. Any negative return value from an init_module function signifies an error to the LKM loader and will force it to unload the beginning pieces of the module and stop processing it. This has the effect of letting P2 disable /dev/mem protections without ever fully loading a kernel module; hence why we cannot find traces of it.

Already Injected?

When testing what P2 would do if you tried to inject it on an already infected system, I got the output from above about it being already injected. I then wanted to understand how P2 knew the system was already infected.

For this, I grepped the strings output for the userland binary for “injected”:

 # grep -i injected strings

[1;40minjected

/dev/shm/%s.injected

already injected?

 # grep shm strings

/dev/shm/....

; rm /dev/shm/.... and try again

/dev/shm/%s.injected

The /dev/shm line stood out quite a bit in this output as /dev/shm is always a tmpfs filesystem and rootkits often write to this directory as it does not persist across reboots. To determine what this injected file was, I used the tmpfs plugin to recover /dev/shm.

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_tmpfs -L

Volatile Systems Volatility Framework 2.2

1 -> /dev/shm

2 -> /lib/init/rw

 # python vol.py --profile=Linuxthisx86 -f after-blog-post.lime linux_tmpfs -S 1 -D tmpfs

Volatile Systems Volatility Framework 2.2

 # ls -lR tmpfs

tmpfs:

total 0

-rw------- 1 root root 0 Oct 7 2012 XXXXXXXXXXX.injected

As can be seen in the output, there is a file named as <prefix of hidden files>.injected under /dev/shmem. In this output I changed the prefix of our sample config to all Xs. We know now how P2 checks for the existence of a previous infection. If you are interested in how Volatility recovers tmpfs, please see this blog post.

Looking with strace

By analyzing the output of strace on the binary as it installs, we uncover a number of interesting activities. The first involves the use of */proc/self/exe* to execute:

chdir("/usr/share/XXXXXXXXXXXX.p2") = 0

symlink("/proc/self/exe", "Xnest") = 0

execve("./Xnest", ["Xnest", "i"], [/* 0 vars */]) = 0

readlink("/proc/self/exe", "/usr/share/XXXXXXXXXXXX.p2/.p-2.5f", 255) = 34

chdir("/usr/share/XXXXXXXXXXXX.p2") = 0

unlink("./Xnest") = 0

open(".config", O_RDONLY)   = 3

In this case, P2 is creating a symlink between /proc/self/exe and Xnest, which was the name of our hidden process. Xnest, which is a symlink to the same binary that is already running, is then re-executed with the same command-line parameters ("i"), and this time the Xnest binary is unlinked, followed by an opening of the .config file.

This output explains why the error we saw in dmesg that did not match the PID we investigated with Volatility. This abuse of /proc/self/exe also thoroughly confused gdb as the Xnest process is hidden upon the second execution and gdb cannot properly follow the child process.

Next, we see activity related to /dev/shm:

open("/dev/shm/....", O_RDWR)   = -1 ENOENT (No such file or directory)

open("/dev/shm/....", O_RDWR|O_CREAT|O_TRUNC, 0600) = 3

close(3)   = 0

open("/dev/shm/XXXXXXXXXXXX.injected", O_RDWR) = -1 ENOENT (No such file or directory)

In this case we see it create /dev/shm/…., and check for the non-existent injected file — remember we are tracing the install process.

P2 then mmap's /dev/mem in order to start its reading and hooking of kernel memory:

13185 open("/dev/mem", O_RDWR)   = 3

13185 old_mmap(NULL, 134217728, PROT_READ|PROT_WRITE, MAP_SHARED, 3, 0xbf8533e0097c0020) = 0xaf866000

13185 write(1, "; locating sys_call_table..", 27) = 27

After the injection and hooking is completed, we see the activity we already noted from the userland process:

setresuid(0, 0, 0)   = 0

setresgid(42779, 42779, 42779)   = 0

getpid()   = 13188

setpriority(PRIO_PROCESS, 13188, 7) = 0

chdir("/")   = 0

setsid()   = 13188

open("/dev/null", O_RDWR)   = 3

dup2(3, 0)   = 0

dup2(3, 1) = 1

dup2(3, 2) = 2

close(3) = 0

In this case it is setting its privileges to UID 0 and its GID to 42779. We also see the call to setsid followed by the setting of file descriptors 0, 1, and 2 to /dev/null, which is then closed as file descriptor 3. This matches our output from linux_lsof.

The signal handlers for the hooked process are then set to avoid detection by sending of signals to random PIDs in order to find hidden processes. The set handlers will simply ignore any signals.

rt_sigaction(SIGHUP, {SIG_IGN, [HUP], SA_RESTORER|SA_NODEFER, 0x8059830}, {SIG_DFL, [], 0}, 8) = 0

rt_sigaction(SIGINT, {SIG_IGN, [INT], SA_RESTORER|SA_NODEFER, 0x8059830}, {SIG_DFL, [], SA_RESTORER, 0x8059830}, 8) = 0

rt_sigaction(SIGQUIT, {SIG_IGN, [QUIT], SA_RESTORER|SA_NODEFER, 0x8059830}, {SIG_DFL, [], SA_RESTORER, 0x8059830}, 8) = 0

rt_sigaction(SIGILL, {SIG_IGN, [ILL], SA_RESTORER|SA_NODEFER, 0x8059830}, {SIG_DFL, [], 0}, 8) = 0

rt_sigaction(SIGTRAP, {SIG_IGN, [TRAP], SA_RESTORER|SA_NODEFER, 0x8059830}, {SIG_DFL, [], 0}, 8) = 0

<snip>

rt_sigaction(SIGRT_28, {SIG_IGN, [RT_28], SA_RESTORER|SA_NODEFER, 0x8059830}, {SIG_DFL, [], 0}, 8) = 0

rt_sigaction(SIGRT_29, {SIG_IGN, [RT_29], SA_RESTORER|SA_NODEFER, 0x8059830}, {SIG_DFL, [], 0}, 8) = 0

rt_sigaction(SIGRT_30, {SIG_IGN, [RT_30], SA_RESTORER|SA_NODEFER, 0x8059830}, {SIG_DFL, [], 0}, 8) = 0

rt_sigaction(SIGRT_31, {SIG_IGN, [RT_31], SA_RESTORER|SA_NODEFER, 0x8059830}, {SIG_DFL, [], 0}, 8) = 0

Setting up of the network connections we saw in linux_netstat also appears, as well as a call to symlink with 0xdeadbeef as the first parameter and our newly accepted file descriptor 11/0xb being sent to sys_symlink. We will look into this shortly.

socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 9

bind(9, {sa_family=AF_INET, sin_port=htons(4161), sin_addr=inet_addr("127.0.0.1")}, 16) = 0

listen(9, 1)   = 0

socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 10

bind(10, {sa_family=AF_INET, sin_port=htons(0), sin_addr=inet_addr("0.0.0.0")}, 16) = 0

connect(10, {sa_family=AF_INET, sin_port=htons(4161), sin_addr=inet_addr("127.0.0.1")}, 16) = 0

accept(9, 0, NULL)   = 11

close(9)   = 0

symlink(0xdeadbeef, 0xb)   = -1 EFAULT (Bad address)

At the end of the installation process we see:

open("/dev/shm/XXXXXXXXXXXX.injected", O_RDWR|O_CREAT|O_TRUNC, 0600) = 3

close(3)   = 0

unlink("/dev/shm/....")   = 0

Which is P2 deleting its temporary marker and creating the real injected file, which will be hidden due to its prefix.

Monitoring Interactions with /dev/mem

Since I knew that P2 operated by mmap'ing /dev/mem I thought that I could simply record all of interactions by hooking the open, mmap, and lseek handlers for the device file. After I implemented this, I installed P2 and got this output from the kernel:

[40.552968] open_port Xnest 906 2014

[40.596772] Program Xnest tried to access /dev/mem between 0->8000000.

[41.020102] open_port Xnest 906 2014

[41.082741] mmap_mem Xnest 906 offset 0 count 134217728

[41.108800] Xnest:906 map pfn RAM range req write-back for 0-8000000, got uncached-minus

From the highlighted lines (output from my hooks), we can see that the Xnest process tried to access /dev/mem and was denied, and then it later was able to access it (after the protections were turned off). It then calls mmap from 0 to 134217728/0x8000000. Unfortunately, this is all the output we get and the lseek and mmap hooks are never triggered again.

At this point, I decided I would have to revisit this approach since I was obviously missing something related to how character devices are mmap'ed. I guess having a nice log of all reads and writes to kernel memory from the rootkit would have been too easy anyway!

Finding the mmap Code

My next approach was to add to my existing kernel hooks to print out the saved instruction and stack pointers from the userland process that triggers them (Xnest). These are the saved registers that the kernel uses to return control flow to userland after servicing some type of request (system call, ioctl, etc). Gathering of these registers would have the effect of telling me exactly where in the userland binary the hooks were being placed from.

Unfortunately, while I later figured out my code was correct, I thought this approach was wrong as all the registers pointed into the kernel. This should never occur as userland code cannot execute within the kernel address space. I chalked this up to P2 tampering and decided to move on.

Symlink Hook

At this point, I was out of methods to fully automate the analysis and also seemingly could not locate where the userland binary was executing code. I then decided to start analyzing functions that I knew would be interesting. My first thought was to track down the handler for that strange symlink call that had 0xdeadbeef as the first parameter and the socket file descriptor number as the second. These were obviously not normal usages of the symlink system call.

I could have got the address of this function from the check_system_call output, but at this point I really did not trust anything in memory. Instead, I did a search with IDA for the immediate value of 0xdeadbeef. That brought me to this function, which is the symlink hook handler:

In the first basic block we see a compare of the first argument with 0xdeadbeef. IDA is also telling us the function is at 0x804CAEF. This is interesting as we know the handler has to be in the kernel, but is instead a regular function of our non-PIE userland binary - as opposed to some hook that gets copied into the kernel address space. Wanting to test this out, I uninstalled P2, loaded it into GDB, set a breakpoint on the symlink hook function (in userland), and then installed it.

Or I at least tried to install it… Instead I got a kernel OOPs backtrace inside GDB:

[13088.506133] int3: 0000 [#1] SMP

[13088.524203] last sysfs file: /sys/module/vt/parameters/default_utf8

[13088.544028] Modules linked in:

[13088.562724]

[13088.580913] Pid: 3469, comm: Xnest Not tainted (2.6.32 #19) VMware Virtual Platform

[13088.619038] EIP: 0060:[<f7d02001>] EFLAGS: 00200283 CPU: 0

[13088.638601] EIP is at 0xf7d02001

[13088.657447] EAX: 00000053 EBX: deadbeef ECX: 00000007 EDX: 000000e0

[13088.676627] ESI: bffffcf0 EDI: 00000000 EBP: f6758000 ESP: f6759fb0

[13088.696022] DS: 007b ES: 007b FS: 00d8 GS: 00e0 SS: 0068

[13088.715510] Process Xnest (pid: 3469, ti=f6758000 task=f6192e10 task.ti=f6758000)

[13088.754439] Stack:

[13088.773388] c100309b deadbeef 00000007 00000000 bffffcf0 00000000 bffffdb8 00000053

[13088.793871] <0> 0000007b 0000007b 00000000 00000033 00000053 b7fff424 00000073 00200213

[13088.835624] <0> bffffcd0 0000007b 00000000 00000000

[13088.878145] Call Trace:

[13088.899668] [<c100309b>] ? sysenter_do_call+0x12/0x28

[13088.921467] Code: <89> e5 56 81 ec 84 00 00 00 b8 00 30 c3 f7 89 45 b4 81 7d 08 ef be

[13088.967353] EIP: [<f7d02001>] 0xf7d02001 SS:ESP 0068:f6759fb0

[13089.017713] ---[end trace 12c622b2a9c13906]---

[13089.041329] note: Xnest[3469] exited with preempt_count 1

[13089.064735] BUG: scheduling while atomic: Xnest/3469/0x10000001

For those who have not seen kernel backtraces before, the first line:

[13088.506133] int3: 0000 [#1] SMP

and this line:

[13089.064735] BUG: scheduling while atomic: Xnest/3469/0x10000001

Are basically telling us that our userland breakpoint (int3 is the debug trap on Intel) triggered inside the kernel! We also see that EBX is 0xdeadbeef, which we know is the parameter sent by the rootkit during install.

At this point, I realized that P2 was doing some sort of page mirroring between userland and the kernel, and many things that I saw over two days of analysis and reversing started to make sense. First, all the references I saw into the kernel from userland were indeed valid. This includes the saved registers I was retrieving from my mmap hooks – meaning my code was correct.

It also clarified why, if I pulled a function from userland memory (e.g 0x804xxxx), its relocatable references like we saw in the module helper code, were still 0xcacacaca even though they would have to be fixed up for the code to operate. Pulling the functions from a function's kernel memory equivalent revealed the correct address for references though and all references were fixed up.

Note: Pulling arbitrary kernel and userland addresses was done using the linux_dump_address_range plugin that will be a part of the 2.3 Volatility release and in SVN shortly.

Now that I knew what to expect between the different addresses, I realized that I could analyze a function contained in the userland binary with IDA and look at its in-kernel memory equivalent to help resolve any relocatable addresses. Having reversed quite a few functions before this stage, I noticed a pattern of (remember this was before I could fix relocations):

```
mov     eax, 0CACACACAh
mov     [ebp+XXX], eax
mov reg, [ebp+XXX]
mov reg2, [reg+OFFSET]
call reg2
```

So I knew that the symbol that every function was referencing was some type of global data structure that at the very least held function pointers.

Now that I could resolve references, I could finally get the address of this data structure and analyze it. I then dumped it to disk with the dump_address_range plugin, and immediately saw a few things in the output.

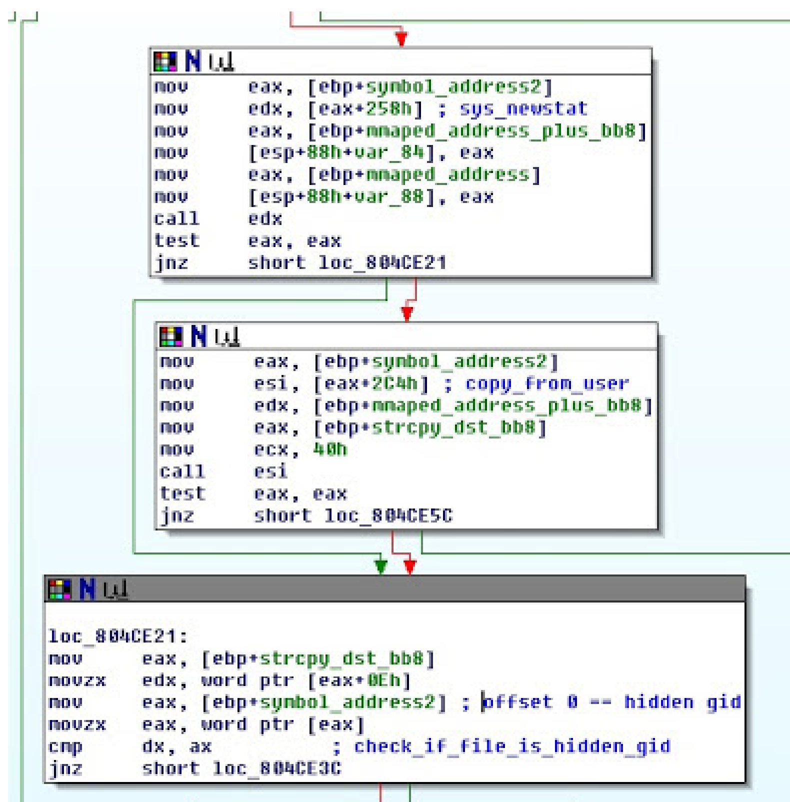
1) The short at offset 0 held the hidden GID
2) The ASCII char array after the GID stored the prefix used to hide files

I now knew I was on to something good…

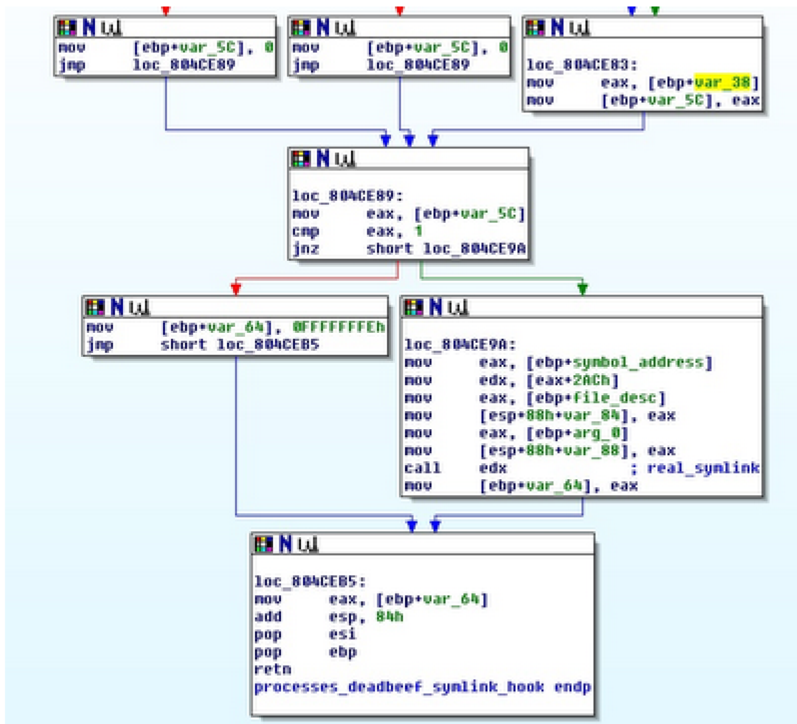
I then started reversing the actual symlink handler (previous picture) and noticed that the first thing it does is check for 0xdeadbeef and skips some code if it is not the first parameter. If it is the first parameter, then it checks a magic number, which in all my tests leads to the basic block on the bottom right. This basic block calls the kernel's fget function with the second file descriptor, which returns the file structure corresponding to the file descriptor. This is then stored at offset 0x10f8 within the global structure. sockfd_lookup is then called with the same file descriptor in order to retrieve its sock structure. This reference is stored at offset 0x10fc within the global structure.

Note: I was able to determine which functions are being called by finding the instruction that references them from the global structure (e.g. mov edx, [eax+260h] for fget), then looking at the offset (260h) in a hex editor view of the global structure I dumped. This gave me a function address in little endian that I could grep for in System.map to determine the actual function being called.

So now that I knew what the purpose of the strange call to symlink was, which is to store a reference to the file descriptor that P2 connects to itself with, I then wanted to see what the rest of the function did. The rest of the hook focuses on keeping users from setting symlinks to hidden files. To accomplish this, the hook walks each portion of the target path given (every directory and subdirectory as well as the final file), and calls stat on it. It then checks to see if the current portion of the path is owned by the hidden GID. It needs to check each part of the path so that a person cannot symlink deep into the hidden directory tree if they know of a certain file name. This process can be partially seen in the following picture, notice that I have commented the system calls and a few other instructions:



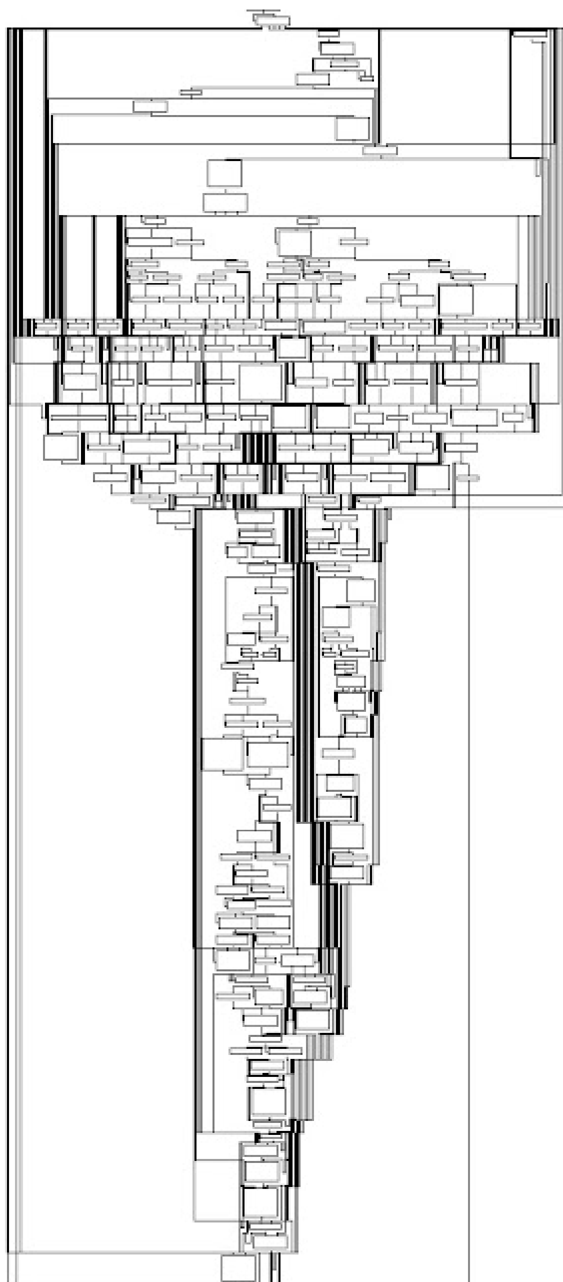
The last part of the hook is the rootkit either returning an error (-2) or allowing the real symlink system call to process the request. It is hard to tell from the limited picture, but the left and middle basic blocks at the top come from code paths that P2 allows to call the real symlink, so they set the check variable (var_5C) to 0, while the top right basic block will set var_5C to one. This is the path when P2 wants to stop the symlink operation. We then see in the last basic block that the return value comes from var_64 which is set by the two basic blocks above it. This value will either be the -2 error value or whatever value the actual symlink call produces:



In summary, the symlink hook serves both as a backdoor communication channel as well as a way for the rootkit to stop ordinary users from accessing its hidden files on a live machine.

Other Hooks

After spending two or three days analyzing P2 with Volatility, IDA, and dynamic analysis through standard Linux tools and my own kernel modules, I pretty much wanted a break… but I thought analyzing another system call would be interesting for the blog post. I then decided to analyze the sys_read hook and was presented with the star destroyer:



Analyzing this hook was “interesting” as it seems to bail on hooking if the file descriptor sent is <=3 or if the read request is larger than 0x3fff.   The rest of the function was pretty complicated and called into P2’s ioctl infrastructure which is something that may be its own blog post in the future...

Misc Thoughts

There were a number of interesting things about P2 that were discovered but did not really fit into their own category or were not understood enough to post.

1) If you look at the strings output from the binary, you see many interesting strings – system call names, error/debug messages, send_file(), get_file() – but none of them have any cross references. I believe this is because P2, like other rootkits, declare their hooks as a C char array that contains only the opcodes of the hook. Since hooks are never called directly from the inside the rookit and instead are only used to override function

pointers, RE tools like IDA will not detect/analyze them as code or detect any of the references. This makes finding the code that does the hooking more difficult.

2) After loading, P2 sits in a loop of:

   ioctl(<fd sent to symlink>, ….)

   getsockopt(<fd sent to symlink>, …)

   nanosleep()

I have not fully analyzed the ioctl or getsockopt handlers yet, but they are main priorities for future analysis. We also see many calls to ioctl with a special value from a number of the hooks, so it is definitely an important code path.

Conclusion

We have showed how to detect and analyze P2 in a number of ways. Between the userland anti-debugging techniques, the quirky way of loading a 30 instruction module, the statically declared hooks, and the page mirroring (which is not 100% understood yet), P2 has proven to be by far the most interesting piece of Linux malware I have seen. I will definitely be spending more time with it in the future as I know I have not yet analyzed large pieces of its functionality.

If you have any questions please either comment on the blog post or you can find me on Twitter (@attrc).

Source: https://volatility-labs.blogspot.com/2012/10/phalanx-2-revealed-using-volatility-to.html