KIDS – Kernel Intrusion Detection System

Hack In The Box 2007 – DUBAI - UAE

Rodrigo Rubira Branco

<rodrigo@kernelhacking.com>

Domingo Montanaro

<conferences@montanaro.org>

Dubai, 05/04/2007

Disclaimer

We are just security guys who work for big companies.

This presentation is just about issues we have worked on in our own time, and is NOT related to the companies ideas, opinions or works.

Montanaro main research efforts are in Forensics and Anti-Forensics technics and backdoor detection/reversing

Rodrigo research efforts are in going inside the System Internals and trying to create new problems to be solved

Agenda

- Motivation – Actual Issues to be solved
- Tools that try to act on this issues and their vulnerabilities
- Differences between protection levels (software / hardware)
- The Forensics and Anti-Forensics challenge
- Our Proposal
- Comments on efforts of breaking our ideas
- Improvements on StMichael – Technical Stuff
- Questions and Astalavista baby :D

Motivation

- Linux is not secure by default (we know, many *secure* linux distributions exist...)
- Most of efferts till now on OS protection don’t really protect the kernel itself
- Many (a lot!) of public exploits were released for direct kernel exploitation
- Beyond of the fact above, it is possible to bypass the system’s protectors (such as SELinux)
- After a kernel compromise, life is not the same (never ever!)

Breaking into SELinux

Spender's (from grsecurity.net) released a public exploit with SELinux and LSM disable code...

“Bug in fs/splice.c was silently fixed in 2.6.17.7, even though the SuSE developer who fixed the bug knew it to be a "local DoS" Changelog stated only: "splice: fix problems with sys_tee()"

On LKML, the user reporting tee() problems said the oops was at ibuf->ops->get(ipipe, ibuf), where ibuf->ops was NULL

Exploitation is TRIVIAL, mmap buffer at address 0, 7th dword is used as a function pointer by the kernel (the get())”

Breaking into SELinux

```
pipebuf[6] = &own_the_kernel;

/* don't need PROT_EXEC since the kernel is executing it, bypasses SELinux's execmem restriction enabled by default in FC6 test1 */

ptr = mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_FIXED | MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);

memcpy(ptr, &pipebuf, sizeof(pipebuf));
```

We got own_the_kernel to be executed in kernel-mode

Breaking into SELinux

own_the_kernel

- get_current
- disable_selinux
- change gids/uids of the current
- chmod /bin/bash to be suid

Breaking into SELinux

disable_selinux

- find_selinux_ctxid_to_string()

/* find string, then find the reference to it, then work backwards to find a call to selinux_ctxid_to_string */

What string? "audit_rate_limit=%d old=%d by auid=%u subj=%s"

- /* look for cmp [addr], 0x0 */
then set selinux_enable to zero

- find_unregister_security();

What string? "<6>%s: trying to unregister a"
Than set the security_ops to dummy_sec_ops ;)

LSM Discussion

- Ok, because SeLinux uses the LSM framework, we will explain how the LSM framework works for the purpose of this presentation:

  * security_operations structure contains pointers to functions that will be called by the internal hooks
  * dummy implementation that does nothing and will call the loaded module hooks (stackable) -> First problem... the stackable module support depends entirely on the modules, it will inherit a lot of complexity into the code (kernel bugs)
  * all symbols are exported, so, anyone can use it in a backdoor (see references)

LSM Discussion – Dumb module

```
int myinode_rename(struct inode *old_dir, struct dentry *old_dentry,
                        struct inode *new_dir, struct dentry *new_dentry) {
        printk("\n dumb rename \n");

        return 0;
}

static struct security_operations my_security_ops = {
        .inode_rename           = myinode_rename,
};

register_security (&my_security_ops);
```

Kernel Backdoor Fragment

```
static int
execute(const char *string)
{

        ...

        if ((ret = call_usermodehelper(argv[0], argv, envp, 1)) != 0) {
                printk(KERN_ERR "Failed to run \"%s\": %i\n",
                        string, ret);
        }
        return ret;

}
```

OBS: call_usermodehelper replaces the exec_usermodehelper showed in the phrack article (see references)

Kernel Backdoor Fragment

```
/* create a socket */
   if ( (err = sock_create(AF_INET, SOCK_DGRAM, IPPROTO_UDP, &kthread->sock))
        < 0)
         printk(KERN_INFO MODULE_NAME": Could not create a datagram socket,
                   error = %d\n", -ENXIO);
         goto out;
   }
   if ( (err = kthread->sock->ops->bind(kthread->sock, (struct sockaddr *)&kthread->
                   addr, sizeof(struct sockaddr))) < 0)
         printk(KERN_INFO MODULE_NAME": Could not bind or connect to socket,
                   error = %d\n", -err);
         goto close_and_out;
   }
   /* main loop */
   for (;;)
   {
         memset(&buf, 0, bufsize+1);
         size = ksocket_receive(kthread->sock, &kthread->addr, buf, bufsize);
```

OBS: See the references for a complete UDP Client/Server in kernel mode

Kernel Backdoor Fragment

```
static struct workqueue_struct *my_workqueue;

static struct work_struct Task;
static DECLARE_WORK(Task, intrpt_routine, NULL);

static void intrpt_routine(void *irrelevant)
{

        /* do the scheduled action here */

        if (!die)
                queue_delayed_work(my_workqueue, &Task, HZ);

}


my_workqueue = create_workqueue(MY_WORK_QUEUE_NAME);
queue_delayed_work(my_workqueue, &Task, 100);
```

OBS: StMichael uses this kind of schedule, it has been taken from the LKMPG Chapter 11 (see references)

Kernel Backdoor Fragment

- Presented by Richard Johnson at Toorcon 2004

```
int
_load_binary (struct linux_binprm *linux_binprm, struct pt_regs *regs)
{
      ...
}
```

The parameter regs isn't used...

Kernel Backdoor Fragment

```
int my_bprm_set_security (struct linux_binprm *bprm)
{

       if ( ! md5verify_sum(bprm->filename) )
       {
              printk("\n hey hey hey\n");
              return -1;
       }

       return 0;
}
```

Kernel Backdoor Fragment

- Putting all things together, so you have:

* UDP Client/Server -> You can use it to receive and respond to backdoor commands

* LSM registered functions (or hooks) -> Can intercept commands, hide things, and do interesting things (will be revised later)

* Execution from the kernel mode -> Can execute commands requested by the user

* Schedule tasks -> Permits scheduling the backdoor to run again (maybe to begin a new connection - connback), after a period of time

Yeah, only using public available sources!!

PaX Details

- **“The Guaranteed End of Arbitrary Code Execution”**
- **Address Space Layout Randomization (ASLR)**
- **Provides non-executable memory pages**

Seems good, but ....

- Various methods of by-passing some PAX resources were successfull demonstrated (H2HC 2005)
- Any kind of bug that leads to arbitrary kernel write/execute could be used to re-mark the page-protection mechanism (PaX KernSeal will try to prevent it)
- PAX needs complementary solutions (grsecurity)
- Most ppl think that PAX+SELinux is a perfect world, but it isn’t since SELinux doesn’t provide lsm modules that implements some protection that PAX needs

PaX Details

- KERNEXEC
* Introduces non-exec data into the kernel level
* Read-only kernel internal structures

- RANDKSTACK
* Introduce randomness into the kernel stack address of a task
* Not really useful when many tasks are involved nor when a task is ptraced (some tools use ptraced childs)

- UDEREF
* Protects agains usermode null pointer dereferences, mapping guard pages and putting different user DS

The PaX KERNEXEC improves the kernel security because it turns many parts of the kernel read-only. To get around of this an attacker need a bug that gives arbitrary write ability (to modify page entries directly).

PaX Details

Linux Kernel have some read-only portions since 2.2 with PaX kernexec, they are just putting more things in this protected section: .text, kernel page tables, IDT/GDT

You can do something like: 'readelf -e vmlinux'

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [12] .text             PROGBITS        00000000 301000 33f223 00  AX  0   0 4096
  [13] __ex_table        PROGBITS        c073f230 640230 000c00 00   A  0   0  8
  [14] .rodata.page_alig PROGBITS        c0740000 641000 005820 00   A  0   0 16
  [15] .rodata           PROGBITS        c0746000 647000 0ae53e 00   A  0   0 32
  [16] .eh_frame         PROGBITS        c07f4540 6f5540 0c67a0 00   A  0   0  4
  [17] .pci_fixup        PROGBITS        c08bace0 7bbce0 000408 00   A  0   0  4
  [18] __ksymtab         PROGBITS        c08bb0e8 7bc0e8 005a38 00   A  0   0  4
  [19] __ksymtab_gpl     PROGBITS        c08c0b20 7c1b20 001470 00   A  0   0  4
  [20] __ksymtab_gpl_fut PROGBITS        c08c1f90 7c2f90 000030 00   A  0   0  4
  [21] __ksymtab_strings PROGBITS        c08c1fc0 7c2fc0 00fb9d 00   A  0   0 32
  [22] __param           PROGBITS        c08d2000 7d3000 001018 00   A  0   0  4
  [23] .module.text      PROGBITS        c08d4000 7d5000 52c000 00  WA  0   0  1
  [24] .data             PROGBITS        c0e00000 d01000 10c61c 00  WA  0   0 32
```

The Virtual Address of .text is __KERNEL_TEXT_OFFSET, 0xc0400000, and all sections until .data are mapped read-only, something like 10 MB of memory in this case... to test you can just do:

```
dd if=/dev/zero of=/dev/mem bs=4096 seek=1024
```

This will (try to) write to physical address 4096*1024=4MB, that is, the beginning of .text and will end up in a harmless oops because of the read-only mapping. Don't try it w/o KERNEXEC enabled though as it'll trash your system!

Actual Problems

- Security normally runs on ring0, but usually on kernel bugs attacker has ring0 privilleges
- Almost impossible to prevent (Joanna said we need a new hardware-help, really?)
- Lots of kernel-based detection bypassing (more in the next slides)
- Detection on kernel-based backdoors or attacks rely on “mistakes” made by attackers

Trying to find the backdoor

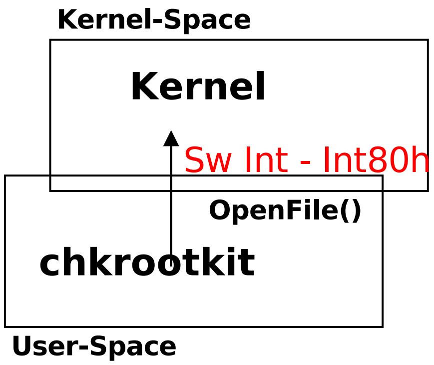
Different types of tools residing in the User-Space can easily be tricked

Linux

```
king:/mnt/sda1# chkrootkit
bla bla bla nothing found… -- Really?
```

Reliable method?

All methods covered by these tools can fail when someone is watching the int80h



- Adore
- Suckit
- Other Custom LKMs

Trying to find the backdoor

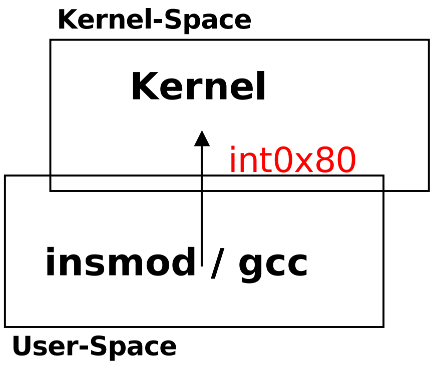
Ok, lets assume that our detection method is based on Kernel-Space tools

```
Linux
king:/mnt/sda1# gcc -c scprint.c -I/usr/src/linux/include/
king:/mnt/sda1# insmod sprint.o
-- outputs of syscalls addrs to syslog
```

Reliable method?

Can gcc, insmod, libs of include, etc, be tricked?

Of course YES ☺

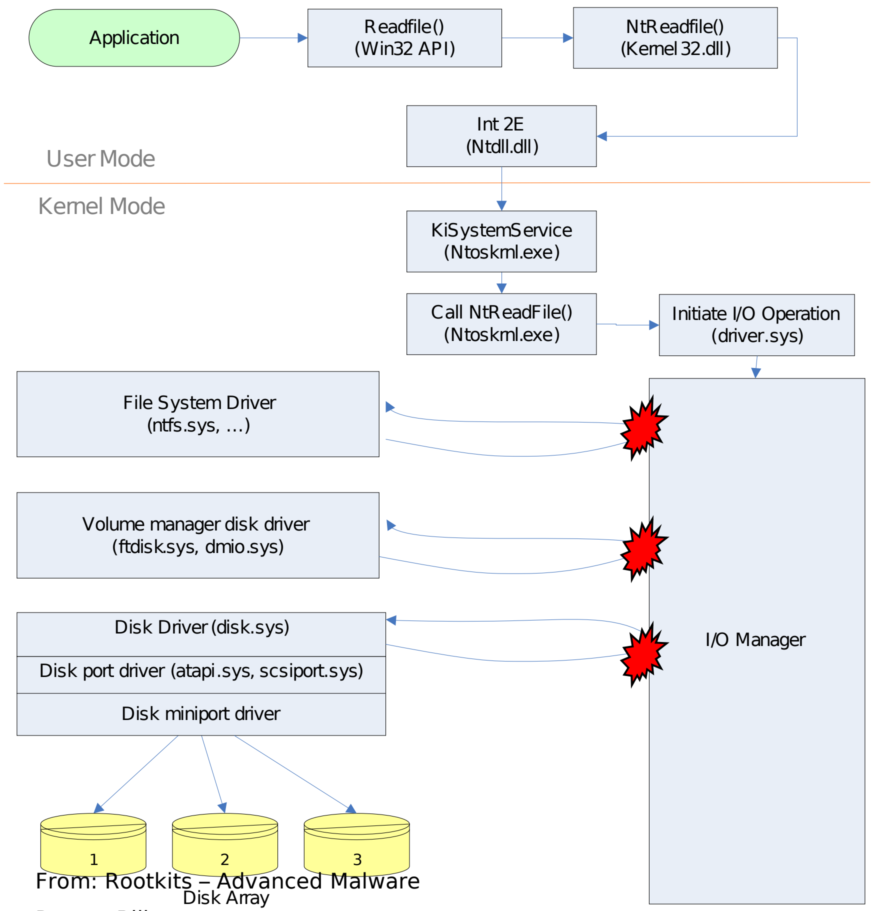



```
sys_read – 3
sys_open - 5
sys_getdents - 141
sys_query_module - 167
sys_execve - 11
sys_chdir - 12
```

Trying to find the backdoor

So if we want to inspect a file, its time to get the blocks directly from the HDD

Reliable method?



From: Rootkits – Advanced Malware
Darren Bilby

IRP (I/O Request Packet) Hooking

Linux
Implementation on the VFS or on fs_driver
Windows
He4Hook

RAM Forensics

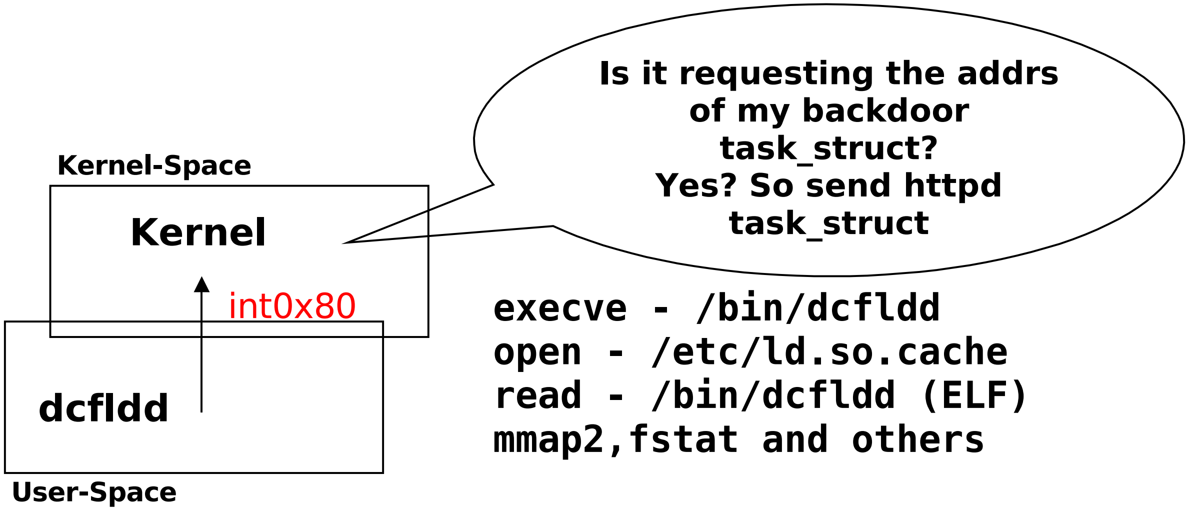
An alternative of inspecting directly the system for its process is to make a physical memory dump and post analysis to find the malware

Linux

```
king:/mnt/sda1# ./dcfldd if=/dev/mem of=memory.img bs=512 conv=noerror
```

Reliable method?

To further analysis, tools like Fatkit are used (Static memory dump file analysis)




```
execve - /bin/dcfldd
open - /etc/ld.so.cache
read - /bin/dcfldd (ELF)
mmap2,fstat and others
```

RAM Forensics

```
ssize_t h_read(int fd, void *buf, size_t count){
    unsigned int i;
    ssize_t ret;
    char *tmp;
    pid_t pid;

    If the fd (file descriptor) contains something
    that we are looking for (kmem or mem)

    return_address();
    At this point we could check the offset being
    required. If is our backdoor addr, send
    another task_struct
    ret=o_read(fd,buf,count);
    change_address();
    return ret;
    }
```

```
int return_address()
{
return our hacks to the
original state
}
```

```
int change_address()
{
put our hacks into
the kernel
}
```

Securely(?) Grabbing the RAM contents

Some hardwares attempt to get the RAM contents

These type of solutions rely on the DMA method of accessing the RAM and then acting on it (CoPolit) or dumping it (Tribble)

- Tribble – Takes a snapshot (dump) of the RAM

http://www.digital-evidence.org

- CoPilot – Audits the system integrity by looking at the RAM Contents

www.komoku.com/pubs/USENIX-copilot.pdf

- Other Firewire (IEEE 1394) Methods

Reliable method?

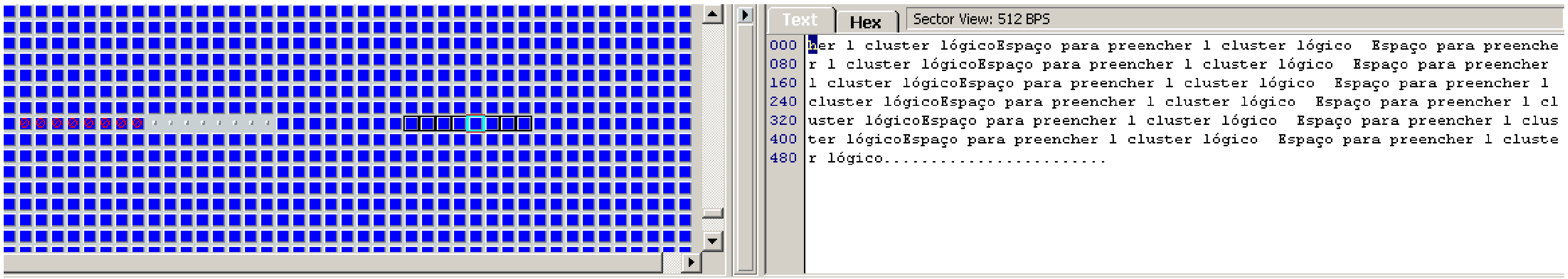
Joanna Rutkowska showed on BlackHat DC 2007 a technic using MMIO that could lead the attacker to block and trick a DMA access from a PCI card.

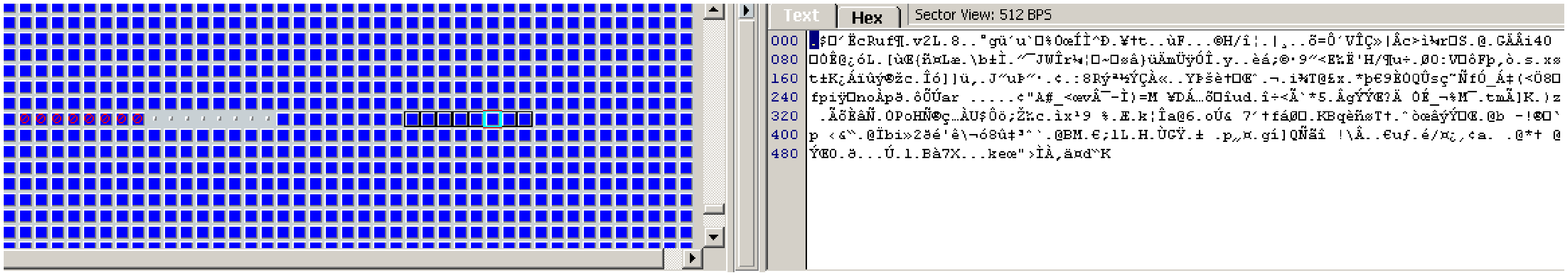
Slack Space

Non-addressable space in the MFT than can be written by specfic tools (RAW)

- NTFS uses logical cluster of 4kb
- Files less than 4kb use 4kb (outside MFT)
- Tools can build a own MFT and address directly on the disk its own blocks to use as a container for the backdoor (and can mark it as bad block to the filesystem, so it would not be overwritten)
- Combining this to crypto/steganographic technics should make the forensics job much harder (and most of times when it's well done, efforts will be lost)

Slack Space


Text
Hex
Sector View: 512 BPS
000 her 1 cluster lógicoEspaço para preencher 1 cluster lógico Espaço para preenche
080 r 1 cluster lógicoEspaço para preencher 1 cluster lógico Espaço para preencher
160 1 cluster lógicoEspaço para preencher 1 cluster lógico Espaço para preencher 1
240 cluster lógicoEspaço para preencher 1 cluster lógico Espaço para preencher 1 cl
320 uster lógicoEspaço para preencher 1 cluster lógico Espaço para preencher 1 clus
400 ter lógicoEspaço para preencher 1 cluster lógico Espaço para preencher 1 cluste
480 r lógico........................


Text
Hex
Sector View: 512 BPS
000
080
160
240
320
400
480

Slack Space

```
19B751940  75 73 74 65 72 20 6C F3  67 69 63 6F 45 73 70 61  uster lógicoEspa
19B751950  E7 6F 20 70 61 72 61 20  70 72 65 65 6E 63 68 65  ço para preenche
19B751960  72 20 31 20 63 6C 75 73  74 65 72 20 6C F3 67 69  r 1 cluster lógi
19B751970  63 6F 0D 0A 45 73 70 61  E7 6F 20 70 61 72 61 20  co..Espaço para 
19B751980  70 72 65 65 6E 63 68 65  72 20 31 20 63 6C 75 73  preencher 1 clus
19B751990  74 65 72 20 6C F3 67 69  63 6F 45 73 70 61 E7 6F  ter lógicoEspaço
19B7519A0  20 70 61 72 61 20 70 72  65 65 6E 63 68 65 72 20   para preencher 
19B7519B0  31 20 63 6C 75 73 74 65  72 20 6C F3 67 69 63 6F  1 cluster lógico
19B7519C0  0D 0A 45 73 70 61 E7 6F  20 70 61 72 61 20 70 72  ..Espaço para pr
19B7519D0  65 65 6E 63 68 65 72 20  31 20 63 6C 75 73 74 65  eencher 1 cluste
19B7519E0  72 20 6C F3 67 69 63 6F  00 00 00 00 00 00 00 00  r lógico........
19B7519F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
19B751A00  10 24 8F 92 CB 63 52 75  66 B6 11 76 32 4C 01 38  .$▌´ËcRuf¶.v2L.8
19B751A10  18 01 B0 67 FC B4 75 60  8D 25 D2 9C CD CC 5E D0  ..°gü´u`▌%Ò▌ÍÌ^Ð
19B751A20  06 A5 86 74 06 12 F9 46  1B 02 17 A9 48 2F EE A6  .¥▌t..ùF...©H/î¦
19B751A30  10 7C B8 05 18 F5 3D D4  B4 56 CE C7 BB 2A 2A 2A  .|¸..õ=Ô´VÎÇ»***
19B751A40  2A 54 45 58 54 4F 20 45  53 43 4F 4E 44 49 44 4F  *TEXTO ESCONDIDO
19B751A50  2A 2A 2A 40 BF F3 4C 03  5B F9 8C 7B F1 A4 4C E6  ***@¿óL.[ù▌{ñ¤Læ
19B751A60  08 5C 62 B1 CC 07 94 AF  4A 57 CE 72 BC A6 9D 7E  .\b±Ì.▌¯JWÎr¼¦▌~
19B751A70  7F F8 E2 7D FC C4 6D DC  FF D3 CE 14 79 1E 18 E8  ▌øâ}üÄmÜÿÓÎ.y..è
19B751A80  E1 3B A9 B7 39 94 3C 45  89 CB 27 48 2F B6 75 F7  á;©·9▌<E▌Ë'H/¶u÷
19B751A90  02 D8 4F 3A 56 8D F4 46  FE 2C F2 0C 73 1A 78 F8  .ØO:V▌ôFþ,ò.s.xø
19B751AA0  74 B1 4B BF C1 EF FB FD  AE 9E 63 07 CE F3 5D 5D  t±K¿Áïûý®▌c.Îó]]
19B751AB0  FC 2C 0E 4A 94 75 DE 94  B7 0B A2 12 3A 38 52 FD  ü,.J▌uÞ▌·.¢.:8Rý
19B751AC0  AA BD DD C7 C0 AB 1D 07  59 DE 9A E8 86 81 8C 88  ª½ÝÇÀ«..YÞ▌è▌▌▌▌
19B751AD0  1C AC 04 69 BE 54 40 A3  78 0B 2A FE 80 39 C8 D2  .¬.i¾T@£x.*þ▌9ÈÒ
```

Sector 13482636 of 14329917 | Offset: 19B751981 | = 114

Introducing StMichael

- Generates and checks MD5 and, optionally, SHA1 checksum of several kernel data structures, such as the system call table, and filesystem call out structures;
- Checksums (MD5 only) the base kernel, and detect modifications to the kernel text such as would occur during a silvo-type attack;
- Backups a copy of the kernel, storing it in on an encrypted form, for restoring later if a catastrophic kernel compromise is detected;
- Detects the presence of simplistic kernel rootkits upon loading;
- Modifies the Linux kernel to protect immutable files from having their immutable attribute removed;
- Disables write-access to kernel memory through the /dev/{k}mem device;
- Conceals StMichael module and its symbols;
- Monitors kernel modules being loaded and unloaded to detect attempts to conceal the module and its symbols and attempt to "reveal" the hidden module.

Introducing StMichael

continuing..

- Ioctl() hooking
- Call flags test (and sets it again) -> and returns the original call
- Self protection : Removes itself from the module list
- Uses encrypted messages to avoid signature detection of its code
- Random keys
- MBR Protection
- Modules syscalls hooked (create_module,init_module,etc)

Efforts on bypassing StMichael

- Julio Auto at H2HC III proposed an IDT hooking to bypass StMichael
- Also, he has proposed a way to protect it hooking the init_module and checking the opcodes of the new-inserted module
- It has two main problems:
  - Can be easily defeated using polymorphic shellcodes
  - Just protect against module insertion not against arbitrary write (main purpose of stmichael)

Efforts on bypassing StMichael

- The best approach (and easy?) way to bypass StMichael is:
  - Read the list of VMA's in the system, detecting the ones with execution property enabled in the dynamic memory section
  - Doing so you can spot where is the StMichael code in the kernel memory, so, just need to attack it...

That's the motivation in the Joanna's comment about we need new hardware helping us... but...

Where do we wanna go?

- StMichael must be a SW independent of other set of programs that try to defend the system
- We will put another layer of protection between the system's auditors/protectors/verifiers and the hardware
- Are the researchers wrong about the impossibility of protecting the O.S. without a hw-based solution?

How? SMM!

SMM – System Management Mode

The Intel System Management Mode (SMM) is typically used to execute specific routines for power management. After entering SMM, various parts of a system can be shut down or disabled to minimize power consumption. SMM operates independently of other system software, and can be used for other purposes too.

From the Intel386tm Product Overview – intel.com

How does it work?

- Chip is programmed to grab and recognize many type of events and timeouts
- When this type of event happens, the chipset gets the SMI (System Management Interrupt)
- In the next instruction set, the processor saves it owns state and enters SMM
- When it receives the SMIACT, redirects all next memory cycles to a protected area of memory (specially reserved for SMM)
- Received SMI and Asserted the SMIAct output? -> save internal state to protected memory
- When contents of the processor state are fully in protected memory area, the SMI handler begins to execute (processor is in real-mode with 4gb segments limit)
- SMM Code executed? Go back to the previous enviroment using the RSM instruction

Going deeper

- Where will be our handler? In the memory, so someone can attack it?
- Protection of the memory pages (already supported by PaX)
- Possibility to add watchpoints in memory pages (detect read at VMAs? At our code? Or writes against our system?)
- DR7 Register!

The Debug Register 7 (DR7) has few unducumented bits that completely modifies the CPU behavior when entering SMM (earlier ICE – In-Circuit Emulation → previous of SMM)

```
3                  1 1 1 1 1 1         0
1                  5 4 3 2 1 0         0
+-----------------+-+-+-+-+-+-+---------+
|                 |T|T|G|I| | |         |
|                 |2|R|D|R| | |         |
+-----------------+-+-+-+-+-+-+---------+
                   | | | |
                   | | | +-- IceBp  1=INT01 causes emulator
                   | | |              to break emulation
                   | | |            0=CPU handles INT01
                   | | +---- General Detect = Yeah, we can spot CHANGES in the Registers
                   | +------ Trace1 1=Generate special address
                   |                  cycles after code dis-
                   |                  continuities.  On Pentium,
                   |                  these cycles are called
                   |                  Branch Trace Messages.
                   +-------- Trace2 1=Unknown.
```

collateral effects!

Compability Problems

- **Yeah, we have SMM just in the Intel platform... but:**
  - **Many platforms already supports something like firmware interrupts**
  - **Although any platform have some way to instrument it to debug agains hardware problems**

Another Difficulties

- **Do you ever know kdump/kexec?**
- **It's a kernel dump and recovering utility and is really interesting for debugging purposes and to keep the system availibility**
- **The problem:**
  - We have another kernel image
  - An attacker who could execute some code in kernel mode can just change this kernel image (this resides in an unprotected memory area) and then, get the system to cause a crash
  - We can solve this in two ways:
    - Signature analysis before run the new kernel
    - Memory protection in the 'guest' kernel

Future? Who wanna test?

- We are looking for a secure OS that wants to try our proposal
- Theo De Raadt is seeing this:

But we want to propose to test our ideas under a “secure” operating system such as OpenBSD. :-)

Acknowledges

Spender for help into many portions of the model

PaX Team for solving doubts about PaX and giving many help point directly to the pax implementation code

HackInTheBox crew – We'll surely steel some ideas for the H2HC

REFERENCES

Spender public exploit:
http://seclists.org/dailydave/2007/q1/0227.html

Pax Project:
http://pax.grsecurity.net

Joanna Rutkowska:
http://www.invisiblethings.org

Julio Auto @ H2HC – Hackers 2 Hackers Conference:
http://www.h2hc.org.br

Phalanx:
http://packetstormsecurity.org/UNIX/penetration/rootkits/phalanx-b6.tar.bz2

Kernel UDP Client/Server:
http://www.kernelnewbies.org/Simple_UDP_Server

Chapter 11 (Scheduling Tasks):
http://lkmpg.cvs.sourceforge.net/lkmpg/2.4/

Thanks!

Questions?

Thank you :D

Rodrigo Rubira Branco
<rodrigo@kernelhacking.com>
Domingo Montanaro
<conferences@montanaro.org>