Exploiting the Kernel

Quora: What are some of the best resources for Kernel exploitation on Linux? (ans by Kaiwan NB).

From "A Guide to Kernel Exploitation – Attacking the Core", Perla & Oldani

If you dissect a "generic" exploit, you can see that it has three main components:

- Preparatory phase : Information about the target is gathered and a favorable environment is set up.
- Shellcode : This is a sequence of machine-level instructions that, when executed, usually lead to an elevation of privileges and/or execution of a command (e.g., a new instance of the shell). As you can see in the code snippet on the next page, the sequence of machine instructions is encoded in its hex representation to be easily manipulated by the exploit code and stored in the targeted machine's memory.
- Triggering phase : The shellcode is placed inside the memory of the target process (e.g., via input feeding) and the vulnerability is triggered, redirecting the target program's execution flow onto the shellcode.

(Below) Ref: *SMEP: What is It, and How to Beat It on Linux*, *Dan Rosenberg*
SMEP = Supervisor Mode Execution Protection

Brief Points:

- Linux kernel code can always access userspace code/data regions
- An attacker can carefully setup user memory with an attack payload, then
- (Re)Search the kernel for an exploitable bug(s)
  - Have the kernel run some buggy code*, that in turn, causes it to incorrectly access user regions
  - Remap the pointer(s) to point to the (userspace) attack code (the 'shellcode'), which will run in kernel mode
- Voila! Privilege escalation becomes easy

* Exploit buggy kernel code?
Yes, find a kernel bug: often useful, a null pointer dereference (or stack overflow). So, we craft stuff: mmap() the null address in our usermode process space, memcpy() our attack code (like, `commit_creds(prepare_kernel_creds(0));` into that memory region. Trigger the kernel bug; when the kernel dereferences the null pointer, it leads to (the IP/PC is set to) our exploit shellcode, which then runs, possibly giving us a root shell!
[Note- Hardening countermeasure: modern kernels run with *mmap_min_addr* set, so you can't typically mmap the null page].
[See this PDF, Sept 2012]

Ref:
"The Stack is Back", Jon Oberheide

Real-world stack overflow based exploits:

CVE-2010-3848
CVE-2010-3850

econet-sendmsg.
- VLA (var len array) on the stack with attacker controlled length!

[Demo:
● Achieve adjacent kstacks ● Process #2 goes to sleep ● Stack overflow in process #1 ● Overwrite return address on process #2 kernel stack ● Process #2 wakes up ● Process #2 returns to attacker control address ● Privilege escalation payload executed!
https://jon.oberheide.org/files/half-nelson.c
]

Mitigation: the new STACKLEAK feature!
"Trying to get STACKLEAK into the kernel", LWN, Sept 2018

STACKLEAK merged in 4.20 (only in 5.0?) Aug 2018 [commit].

Another approach: once you have a way to write into the kernel *(not unusual, see this link!)*, trap into one of several function pointers in kernel structures and use a kernel bug to remap them to the shady userspace (shell)code..

Below from: SMEP: What is It, and How to Beat It on Linux, Dan Rosenberg

On May 16, 2011, Fenghua Yu submitted a series of patches to the upstream Linux kernel implementing support for a new Intel CPU feature: Supervisor Mode Execution Protection (SMEP). This feature is enabled by toggling a bit in the cr4 register, and the result is the CPU will generate a fault whenever ring0 attempts to execute code from a page marked with the user bit.
...

RWX Kernel Pages

The first problem is the kernel's page permissions aren't yet in a completely sane state. By compiling a kernel with CONFIG_X86_PTDUMP (or using Kees Cook's modularized version of this feature), we can take a look at the permissions of kernel pages via the `/sys/kernel/debug/kernel_page_tables` debugfs file. In particular, we're interested in pages that are both writable and executable:

```
# grep RW /sys/kernel/debug/kernel_page_tables | grep -v NX
0xc009b000-0xc009f000           16K     RW              GLB x   pte
0xc00a0000-0xc0100000           384K    RW              GLB x   pte
0xc1400000-0xc1580000           1536K   RW              GLB x   pte
```

The first two regions are especially useful, since they will appear at static addresses on many modern 32-bit kernels. The first region is reserved for the BIOS, and the second is the so-called “I/O hole” used for DMA. While it’s probably best to avoid scribbling all over the I/O hole, as it’s commonly used at runtime, there’s no reason that writing into the BIOS region would cause any stability issues after booting is complete.

So, if we have a kernel write primitive, all we have to do is write our payload into the BIOS region and divert execution there. If the target kernel leaks symbol locations via `/proc/kallsyms` or similar, then diverting execution is a simple matter of resolving the address of a suitable function pointer, overwriting it, and triggering it.
<< Led to the script leaking_addresses.pl (4.14, Nov 2017) : Tobin Harding (32-bit contribs from Kaiwan NB) >>

Otherwise, it’s trivial to issue a `sidt` instruction to retrieve the address of the IDT and set up a trap handler pointing into the payload. SMEP will have nothing to complain about, since we never cause the kernel to attempt to execute from user pages.

Stack Metadata

A second way to bypass this protection is to leverage the `addr_limit` variable, which resides in the `thread_info` structure at the base of each process’ kernel stack.

As described in Jon Oberheide’s and my presentation on Stackjacking, it’s possible to exploit the leakage of uninitialized stack data, a common bug, in order to infer the address of the base of a process’ kernel stack. I developed a library called *libkstack* to do so generically. Once this address is inferred, a kernel write vulnerability can simply write `ULONG_MAX` (`0xffffffff`) into the `addr_limit` variable, which is at a reliable offset from the kernel stack base.

At this point, arbitrary kernel memory can be read from and written to, since all kernel copy functions will accept kernel pointers as user arguments. For example, you can do a `write(pipefd, kernel_addr, len)` to read the data from `kernel_addr` into a pipe, to be retrieved later.

Once you have an arbitrary kernel read and write, the current process’ `cred` structure can be found and written into, escalating privileges to root. Again, this attack does not require executing any user code with kernel privileges, so SMEP cannot stop it.

Update: it’s worth noting that grsecurity protects against this type of attack by removing the `thread_info` structure from the kernel stack.
<< So does Linux now! From 4.10 for ARM64 ; the commit: “
This patch moves arm64's struct thread_info from the task stack into task_struct. This protects thread_info from corruption in the case of stack overflows, and makes its address harder to determine if stack addresses are leaked, making a number of attacks more difficult. ...” >>

Return-Oriented Programming

In the event that kernel symbols can be resolved on the target kernel (especially common on distro kernels) and the attacker has a stack overflow or another vulnerability that allows pivoting the stack pointer into an area of attacker controlled data, kernel ROP is possible. Fortunately, the `setup_smep` function, which has code to both enable and disable the SMEP bit in the cr4 register, is marked `__init`, so it’s likely to have been cleaned up by the kernel after initialization

and is not a good candidate for ROP. However, more complex ROP payloads are certainly possible, as I hope to demonstrate later this year.

What Needs to be Fixed

Some progress on removing useful sources of information leakage has been made with the kptr_restrict and dmesg_restrict sysctls. Continued work on plugging similar leaks should improve the usefulness of these features. However, it's still trivial to resolve the locations of kernel code and data on distribution kernels, since they are shipped as binaries that are identical across all machines with the same kernel version. This is demonstrated perfectly by Jon Oberheide's ksymhunter project.

The solution I'm currently working on is implementing randomization of the address at which the kernel is decompressed at boot << KASLR*! 2.6.12 (Jun 2005)* >>. This way, even if an attacker can download an identical kernel image as the target host, he won't know where kernel data and code resides in a running kernel, assuming an absence of information leakage.

In order to be effective, this solution requires relocating the IDT – otherwise, it will reside at the location pointed to by the `idt_table` symbol, and an `sidt` instruction would allow an attacker to calculate the offsets of every other kernel symbol relative to the address of the IDT. This has its own challenges, but I'm making progress and hope to submit a working version in the coming weeks. This will also have the useful side effect of marking the IDT read-only, which will prevent it from being a generic target for kernel write vulnerabilities.

Next, more work needs to be done on making sure page protections in the kernel are sane. Most importantly, RWX mappings should be removed and function pointer tables should be enforced read-only. Fortunately, efforts are underway in this area as well, with help from Kees Cook. Hopefully, with the combined efforts to remove information leakage via restricting leaks and kernel image randomization, stronger page protections in the kernel, and SMEP, the Linux kernel will have significantly raised the bar for exploitation.

Source: 'Writing Kernel Exploits'

The "full-nelson" : 2010, Dan Rosenberg:
Steps to exploit:

- Create a thread
- Set its clear_child_tid to an address in kernel memory
- Thread invokes splice on an Econet socket; crashes
- Kernel writes 0 to our chosen address
- We exploit that corruption somehow

full-nelson: exploiting a zero write

On i386, kernel uses addresses 0xC0000000 and up.
Use the bug to clear the top byte of a kernel function pointer.
Now it points to userspace; stick our payload there.
Same on x86_64, except we clear the top 3 bytes.

Then, the 'half-nelson' is described briefly.

Kernel Driver mmap Handler Exploitation, Fruba [2017.09.08, PDF]