pi3 blog

bug bugs bughunt exploit exploiting research vulnerabilities

"He who fights with monsters should be careful lest he thereby become a monster. And if thou gaze long into an abyss, the abyss will also gaze into thee."

~Friedrich Nietzsche

- Subscribe

Search

Press enter to search.

- ## Categories
    - Bughunt (26)
    - Exploiting (33)
    - Ideas (27)
    - LKRG (10)
    - Meeting (6)
    - O wszystkim i o niczym (31)
- ## Recent Posts
    - My talks @ BlackHat 2021 and DefCon29
    - LKRG 0.9.0 has been released!
    - Windows 7 TCP/IP hijacking
    - The short story of broken KRETPROBES and OPTIMIZER in Linux Kernel
    - CVE-2020-16898 – Exploiting "Bad Neighbor" vulnerability
- ## Archives
    - July 2021
    - April 2021
    - January 2021
    - December 2020
    - October 2020
    - September 2020
    - June 2020
    - May 2020
    - March 2020
    - October 2019
    - July 2019
    - May 2019
    - April 2019
    - February 2019
    - August 2018
    - July 2018
    - March 2018
    - February 2018
    - January 2018
    - April 2017
    - September 2016
    - July 2015
    - February 2015
    - March 2014
    - August 2013
    - July 2013
    - May 2013
    - April 2013
    - March 2013
    - February 2013
    - October 2012
    - May 2012
    - April 2012
    - December 2011
    - September 2011
    - August 2011
    - July 2011
    - March 2011
    - November 2010
    - October 2010
    - July 2010
    - May 2010
    - March 2010
    - February 2010
    - January 2010
    - December 2009
    - November 2009
    - October 2009
    - September 2009
- ## Friends
    - Bothunters – Borys Łącki
    - Gynvael Coldwind
    - Icewall
    - j00ru
    - Piotr Bania
- ## Pages
    - About

15
Dec

The short story of broken KRETPROBES and OPTIMIZER in Linux Kernel

by pi3

The short story of broken KRETPROBES and OPTIMIZER in Linux Kernel.

During the LKRG development process I've found that:

- KRETPROBES are broken since kernel 5.8 (fixed in upcoming kernel)
- OPTIMIZER was not doing sufficient job since kernel 5.5

First things first – KPROBES and FTRACE:

Linux kernel provides 2 amazing frameworks for hooking – K*ROBES and FTRACE. K*PROBES is older and a classic one – introduced in 2.6.9 (October 2004). However, FTRACE is a newer interface and might have smaller overhead comparing to K*PROBES. I'm using a word "K*PROBES" because various types of K*PROBES were availble in the kernel, including JPROBES, KRETPROBES or classic KPROBES. K*PROBES essentially enables the possibility to dynamically break into any kernel routine. What are the differences between various K*PROBES?

- KPROBES – can be placed on virtually any instruction in the kernel
- JPROBES – were implemented using KPROBES. The main idea behind JPROBES was to employ a simple mirroring principle to allow seamless access to the probed function's arguments. However, since 2017 JPROBEs were depreciated. More information can be found here:
  https://lwn.net/Articles/735667/
- KRETPROBES – sometimes they are called "return probes" and they also use KPROBES under-the-hood. KRETPROBES allows to easily execute user's own routine at the entry and return path to the hooked function.However, KRETPROBES can't be placed on arbitrary instructions.

When a KPROBE is registered, it makes a copy of the probed instruction and replaces the first byte(s) of the probed instruction with a breakpoint instruction (e.g., int3 on i386 and x86_64).

FTRACE are newer comparing to K*PROBES and were initially introduced in kernel 2.6.27, which was released on October 9, 2008. FTRACE works completely differently and the main idea is based on instrumenting every compiled function (injecting a "long-NOP" instruction – GCC's option "-pg"). When FTRACE is being registered on the specific function, such "long-NOP" is being replaced with JUMP instruction which points to the trampoline code. Later such trampoline can execute any pre-registered user-defined hook.

A few words about Linux Kernel Runtime Guard (LKRG)

In short, LKRG performs runtime integrity checking of the Linux kernel (similar to PatchGuard technology from Microsoft) and detection of the various exploits against the kernel. LKRG attempts to post-detect and promptly respond to unauthorized modifications to the running Linux kernel (system integrity) or to corruption of the task integrity such as credentials (user/group IDs), SECCOMP/sandbox rules, namespaces, and more.
To be able to implement such functionality, LKRG must place various hooks in the kernel. KRETPROBES are used to fulfill that requirement.

LKRG's KPROBE on FTRACE instrumented functions

A careful reader might ask an interesting question: what will happen if the function is instrumented by the FTRACE (injected "long-NOP") and someone registers K*PROBES on it? Does dynamically registered FTRACE "overwrite" K*PROBES installed on that function and vice versa?

Well, this is a very common situation from LKRG's perspective, since it is placing KRETPROBES on many syscalls. Linux kernel uses a special type of K*PROBES in such case and it is called "FTRACE-based KPROBES". Essentially, such special KPROBE is using FTRACE infrastructure and has very little to do with KPROBES itself. That's interesting because it is also subject to FTRACE rules e.g. if you disable FTRACE infrastructure, such special KPROBE won't work either.

OPTIMIZER

Linux kernel developers went one step forward and they aggressively "optimize" all K*PROBES to use FTRACE instead. The main reason behind that is performance – FTRACE has smaller overhead. If for any reason such KPROBE can't be optimized, then classic old-school KPROBES infrastructure is used.

When you analyze all KRETPROBES placed by LKRG, you will realize that on modern kernels all of them are being converted to some type of FTRACE 🙂

LKRG reports False Positives

After such a long introduction finally, we can move on to the topic of this article. Vitaly Chikunov from ALT Linux reported that when he runs FTRACE stress tester, LKRG reports corruption of .text section:

https://github.com/openwall/lkrg/issues/12

I spent a few weeks (month+) on making LKRG detect and accept authorized third-party modifications to the kernel's code placed via FTRACE. When I finally finished that work, I realized that additionally, I need to protect the global FTRACE knob (*sysctl kernel.ftrace_enabled*), which allows root to completely disable FTRACE on a running system. Otherwise, LKRG's hooks might be unknowingly disabled, which not only disables its protections (kind of OK under a threat model where we trust host root), but may also lead to false positives (as without the hooks LKRG wouldn't know which modifications are legitimate). I've added that functionality, and everything was working fine…
… until kernel 5.9. This completely surprised me. I've not seen any significant changes between 5.8.x and 5.9.x in FTRACE logic. I spent some time on that and finally I realized that my protection of global FTRACE knob stopped working on latest kernels (since 5.9). However, this code was not changed between kernel 5.8.x and 5.9.x. What's the mystery?

First problem – KRETPROBES are broken.

Starting from kernel 5.8 all non-optimized KRETPROBES don't work. Until 5.8, when #DB exception was raised, entry to the NMI was not fully performed. Among others, the following logic was executed:
https://elixir.bootlin.com/linux/v5.7.19/source/arch/x86/kernel/traps.c#L589

```
if (!user_mode(regs)) {
    rcu_nmi_enter();
    preempt_disable();
}
```

In some older kernels function *ist_enter()* was called instead. Inside this function we can see the following logic:
https://elixir.bootlin.com/linux/v5.7.19/source/arch/x86/kernel/traps.c#L91

```
if (user_mode(regs)) {
    RCU_LOCKDEP_WARN(!rcu_is_watching(), "entry code didn't wake RCU");
} else {
    /*
     * We might have interrupted pretty much anything.  In
     * fact, if we're a machine check, we can even interrupt
     * NMI processing.  We don't want in_nmi() to return true,
     * but we need to notify RCU.
     */
    rcu_nmi_enter();
}

preempt_disable();
```

As the comment says "*We don't want in_nmi() to return true, but we need to notify RCU.*". However, since kernel 5.8 the logic of how interrupts are handled was modified and currently we have this (function "*exc_int3*"):
https://elixir.bootlin.com/linux/v5.8/source/arch/x86/kernel/traps.c#L630

```
/*
 * idtentry_enter_user() uses static_branch_{,un}likely() and therefore
 * can trigger INT3, hence poke_int3_handler() must be done
 * before. If the entry came from kernel mode, then use nmi_enter()
 * because the INT3 could have been hit in any context including
 * NMI.
 */
if (user_mode(regs)) {
    idtentry_enter_user(regs);
    instrumentation_begin();
    do_int3_user(regs);
    instrumentation_end();
    idtentry_exit_user(regs);
} else {
    nmi_enter();
    instrumentation_begin();
    trace_hardirqs_off_finish();
    if (!do_int3(regs))
        die("int3", regs, 0);
    if (regs->flags & X86_EFLAGS_IF)
        trace_hardirqs_on_prepare();
    instrumentation_end();
    nmi_exit();
}
```

The root of unlucky change comes from this commit:

https://github.com/torvalds/linux/commit/0d00449c7a28a1514595630735df383dec606812#diff-51ce909c2f65ed9cc668bc36cc3c18528541d8a10e84287874cd37a5918abae5

which was later modified by this commit:

https://github.com/torvalds/linux/commit/8edd7e37aed8b9df938a63f0b0259c70569ce3d2

and this is what we currently have in all kernels since 5.8. Essentially, KRETPROBES are not working since these commits. We have the following logic:

```
asm_exc_int3() -> exc_int3():
                    |
    ----------------|
    |
    v
...
nmi_enter();
...
if (!do_int3(regs))
       |
  -----|
  |
  v
do_int3() -> kprobe_int3_handler():
                    |
    ----------------|
    |
    v
...
if (!p->pre_handler || !p->pre_handler(p, regs))
                             |
    -------------------------|
    |
    v
...
pre_handler_kretprobe():
...
    if (unlikely(in_nmi())) {
        rp->nmissed++;
        return 0;
    }
```

Essentially, *exc_int3()* calls *nmi_enter()*, and *pre_handler_kretprobe()* before invoking any registered KPROBE verifies if it is not in NMI via *in_nmi()* call.

I've reported this issue to the maintainers and it was addressed and correctly fixed. These patches are going to be backported to the stable tree (and hopefully to LTS kernels as well):

https://lists.openwall.net/linux-kernel/2020/12/09/1313

However, coming back to the original problem with LKRG… I didn't see any issues with kernel 5.8.x but with 5.9.x. It's interesting because KRETPROBES were broken in 5.8.x as well. So what's going on?

As I mentioned at the beginning of the article, K*PROBES are aggressively optimized and converted to FTRACE. In kernel 5.8.x LKRG's hook was correctly optimized and didn't use KRETPROBES at all. That's why I didn't see any problems with this version. However, for some reasons, such optimization was not possible in kernel 5.9.x. This results in placing classic non-optimized KRETPROBES which we know is broken.

Second problem – OPTIMIZER isn't doing sufficient job anymore.

I didn't see any changes in the sources regarding the OPTIMIZER, neither in the hooked function itself. However, when I looked at the generated vmlinux binary, I saw that GCC generated a padding at the end of the hooked function using INT3 opcode:

```
...
ffffffff8130528b:       41 bd f0 ff ff ff       mov    $0xfffffff0,%r13d
ffffffff81305291:       e9 fe fe ff ff          jmpq   ffffffff81305194
ffffffff81305296:       cc                      int3
ffffffff81305297:       cc                      int3
ffffffff81305298:       cc                      int3
ffffffff81305299:       cc                      int3
ffffffff8130529a:       cc                      int3
ffffffff8130529b:       cc                      int3
ffffffff8130529c:       cc                      int3
ffffffff8130529d:       cc                      int3
ffffffff8130529e:       cc                      int3
ffffffff8130529f:       cc                      int3
```

Such padding didn't exist in this function in generated images for older kernels. Nevertheless, such padding is pretty common.

OPTIMIZER logic fails here:

```
try_to_optimize_kprobe() -> alloc_aggr_kprobe() -> __prepare_optimized_kprobe()
-> arch_prepare_optimized_kprobe() -> can_optimize():

/* Decode instructions */
addr = paddr - offset;
while (addr < paddr - offset + size) { /* Decode until function end */
    unsigned long recovered_insn;
    if (search_exception_tables(addr))
        /*
         * Since some fixup code will jumps into this function,
         * we can't optimize kprobe in this function.
         */
        return 0;
    recovered_insn = recover_probed_instruction(buf, addr);
    if (!recovered_insn)
        return 0;
    kernel_insn_init(&insn, (void *)recovered_insn, MAX_INSN_SIZE);
    insn_get_length(&insn);
    /* Another subsystem puts a breakpoint */
    if (insn.opcode.bytes[0] == INT3_INSN_OPCODE)
        return 0;
    /* Recover address */
    insn.kaddr = (void *)addr;
    insn.next_byte = (void *)(addr + insn.length);
    /* Check any instructions don't jump into target */
    if (insn_is_indirect_jump(&insn) ||
        insn_jump_into_range(&insn, paddr + INT3_INSN_SIZE,
                 DISP32_SIZE))
        return 0;
    addr += insn.length;
}
```

One of the checks tries to protect from the situation when another subsystem puts a breakpoint there as well:

```
    /* Another subsystem puts a breakpoint */
    if (insn.opcode.bytes[0] == INT3_INSN_OPCODE)
        return 0;
```

However, that's not the case here. INT3_INSN_OPCODE is placed at the end of the function as padding.
I wanted to find out why INT3 padding is more common in the new kernels while it's not the case for older ones even though I'm using exactly the same compiler and linker. I've started browsing commits and I've found this one:
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7705dc8557973d8ad8f10840f61d8ec805695e9e

```
diff --git a/arch/x86/kernel/vmlinux.lds.S b/arch/x86/kernel/vmlinux.lds.S
index b06d6e1188deb..3a1a819da1376 100644
--- a/arch/x86/kernel/vmlinux.lds.S
+++ b/arch/x86/kernel/vmlinux.lds.S
@@ -144,7 +144,7 @@ SECTIONS
                *(.text.__x86.indirect_thunk)
                __indirect_thunk_end = .;
 #endif
-       } :text = 0x9090
+       } :text =0xcccc

        /* End of text section, which should occupy whole number of pages */
        _etext = .;
```

It looks like INT3 is now a default padding used by the linker.

I've brought up that problem with the Linux kernel developers (KPROBES owners), and Masami Hiramatsu prepared appropriate patch which fixes the problem:

https://lists.openwall.net/linux-kernel/2020/12/11/265

I've verified it and now it works well. Thanks to LKRG development work we helped identify and fix two interesting problems in Linux kernel 🙂

Thanks,
Adam

Filed Under Bughunt, Exploiting, Ideas, LKRG

Comments

Leave a Reply

Name (Required)

Email Address (Required)

Website

Submit Comment

CAPTCHA *

+ nine = 17