# Project Triforce: AFL + QEMU + kernel = CVEs! (or) How to use AFL to fuzz arbitrary VMs

October 2016


# Intro

# Who?

- **Jesse Hertz**
  - Senior Security Consultant at NCC Group
  - Focus on low-level C/C++ auditing, enjoy escaping sandboxes
  - Been working on Linux container assessments recently, led to a focus on the kernel as the shared attacked surface.
- **Tim Newsham**
  - Distinguished Security Consultant at NCC Group
  - Operating Systems Junkie
  - Fuzzing Fan
- **Systems internals Enthusiasts** ☺
- **BTW**: All the info here (with even more details) will be in a whitepaper we have coming out soon.

# Why?

- **Frequent users of AFL on gigs**
  - It's a great fuzzer (@lcamtuf is awesome)
  - Its more than just a fuzzer though (more on that later)
- **Frequent users of VMs on gigs**
  - Client code may require specific environment
  - Team members may be using different host OS's
- **The challenge: Instrumentation**
  - We tend to spend a lot of time writing fuzzing harnesses for our targets
  - "Get a fuzzer running, then start auditing code"
  - Easy when a target is just a simple CLI program.
  - What about difficult to instrument code? Like a kernel…
- **Given arbitrary code that we have running in a VM, how can we fuzz it?**

# Let's Take a Step Back...

## How does AFL normally work?

- Instruments a binary at compile time.
- Uses a forkserver model, forking many copies of the binary to run test cases.
- With the binary instrumented, AFL can capture an "**edge trace**" that represents the paths through the control flow graph of the target binary.
- This gives it feedback, which is used in a **genetic algorithm** to create a new batch of test cases.
- This is often called "**feedback driven fuzzing**".

# Quick Primer on Genetic Algorithms

**We have a starting generation of test cases**

**We score them somehow**

**We keep the "interesting" ones, and make a new generation from them**

- **There are a variety of ways to do this**
  - Favor inputs with better scores, mutate them, and repeat
  - Can "thin the herd" by throwing out low scoring inputs
  - Can also do fun things like recombination, lots to learn from biology
- **Specific to fuzzing, AFL uses a score function that covers whether new edges were hit by a test case.**
  - This means… "it learns"
  - Famously, it can manufacture a valid (albeit boring) JPEG from a single byte.
  - By measuring not just coverage, but edges, AFL can discover lots of interesting test cases that trigger strange paths. Bugs!

# Using AFL on Hard Targets

- **At first, using AFL was only possible on programs you could build (specifically using afl-gcc or afl-clang), and focused on CLI applications.**
- **Then, qemu-mode was introduced:**
  - This uses QEMU's userland emulation to run a binary in a userland emulator
  - This lets you capture an edge-trace for a "random binary"
  - Still restricted to running linux binaries on linux hosts.
- **WinAFL also is an interesting fork to make AFL work on Windows.**
- **Other forks/shims emerged to fuzz network binaries, or other non-CLI binaries.**
- **TriforceAFL instead extends qemu_mode to QEMU's full-system emulation.**
  - This means the target can be an operating system kernel.
  - This means the target can be a binary running in any operating system that runs under QEMU.


# TriforceAFL

# What?

**A fever dream: “What If We Could Run AFL on the Linux Kernel?”**

- **Some issues:**
  - Instrumenting a kernel is not straightforward.
  - We’d need to move chunks of AFL into kernel-mode.
  - How do you fuzz system calls that AFL itself uses?
- **Prior Work:**
  - Trinity (the OG syscall fuzzer). We took a lot of inspiration from Trinity.
  - Oracle showed some interesting work compiling file system drivers with AFL
  - Google’s syzkaller is the closest to our approach (although we didn’t know about it when we started building TriforceAFL)
    - Uses coverage rather than edges

**So what’s our approach anyway?**

# What (specifically)?

**Extend AFL's (userland) QEMU support to fuzzing a VM running under QEMU's full-system emulation!**

- **What does this let us do?**
  - Fuzz anything that will run in QEMU's full-system emulation
    - Takes a performance hit, but not as bad as expected
  - Why just operating systems? How about just difficult to harness programs?
  - Get the full power of feedback driven fuzzing, targeted at whatever you want inside the VM.
  - We'll talk more about "uses" later, this is getting abstract
- **How does this work?**

# How?

**Extend AFL's (userland) QEMU support to fuzzing a VM running under QEMU's full-system emulation!**

- **Host OS runs QEMU and AFL.**
  - Currently, we only officially support Linux as the Host OS
- **Guest VM runs the target (is the target!). Notably, it also runs a small (userland) program called the driver. Userland driver. Userland!**
- **Highest level, do{**
  - The driver unpacks a serialized test case. It executes it.
  - QEMU records an edge trace through the control flow graph.
  - AFL uses the feedback to generate new test cases.
  - **} while(1);**

# How (specifically)?

- **Host OS runs QEMU and AFL.**
  - They communicate through Unix pipes. No need to be fancy here.
  - QEMU is augmented with dispatch for a fake x64 instruction: *aflCall (0x0F24*)
  - This lets the guest VM perform a “hypercall” and communicate with the host
- **Guest VM runs/is the target**. **Runs a small (userland) program: the driver.**
  - Drivers pins a buffer to a fixed (physical) address
  - Driver makes hypercall to tell AFL to put a serialized test case into this buffer
    - To AFL, this is just a buffer of bytes
  - Driver uses a hypercall to tell QEMU to start recording an edge trace.
  - Driver deserializes test case, and executes it.
  - The test case causes a panic, or the driver makes a hypercall to indicate the test case executed normally

# Fun with QEMU

- **Issues with "naïve fuzzing" using this approach:**
  - Booting a VM is slow
  - Test cases aren't idempotent (we need a consistent and reproducible environment).
- **Solution: We only boot the VM once, and then fork it per test-case**
  - This gives surprisingly good performance
    - We're not talking AFL on raw metal speeds, but its a lot better than we expected. More on this later.
  - Test cases are now fully isolated from each-other
  - We can adapt AFL's forkserver approach to run many test-cases in rapid sucession.
  - Requires all VM state to be in memory (no real disk images)

# The Full Run Loop

- TriforceAFL boots a QEMU VM, waits till VM makes the aflCall startForkserver (so the driver needs to be invoked on guest startup)
- The usual AFL loop of running a cycle of test cases, collecting feedback on the test case, and then creating a new generation of test cases at the end of the cycle still applies.
- The forkserver creates a fork of the VM for each test case. With a CoW view of memory, this lets each test case be isolated from each other! **From here on out, everything happens in a fork of the VM**.
    - The driver “wakes up” from its hypercall to startForkserver()
    - Makes a hypercall to getWork()
    - Deserializes the test case
    - Makes a hypercall to startWork() with the addresses to trace
    - Executes the test case
    - Either the test case crashes, or the driver makes a call to endWork() to signify the test case didn’t crash
        - When starting TriforceAFL, you provide an the address of a basic block that represents a crash


**In the most general case, TriforceAFL is a tool to allow AFL to find inputs that cause code executing in a VM to move to a basic block of interest.**

# aflCall details

- **Implemented as an invalid x64 instruction (*0x0F24)***
  - RAX determines which aflCall is used.
  - Arguments to the hypercalls are passed in subsequent registers.
  - A small C wrapper around these is provided for writing drivers
  - Helper functions were added to QEMU when encountering this instruction, and perform dispatch as needed
- **Four Hypercalls**
  - startForkserver
    - Starts AFL's forkserver. From here on out every operation in the VM runs in a forked copy of the VM that persists only until the end of a test case.
    - As a side effect, this call can either enable or disable the vCPU's timer.
  - getWork
    - Copies a test case into a buffer in the guest operating system.
  - startWork
    - This call enables tracing to AFL's edge map. Tracing is only performed for a range of virtual addresses specified.
  - endWork
    - Notify AFL that the test case has completed, optionally passing back a return code or exit status. AFL will then destroy this VM-fork.

# Making it Work

- **Getting startForkserver() to work turned out to be one of the hardest parts:**
  - QEMU uses 3 threads to do full system emulation (CPU, IO, RCU)
  - On Linux, only the thread calling fork has all its state preserved in the child process (TLS, mapped regions, stacks, registers, etc).
  - Fork also doesn't preserve important threading state and can leave locks, mutexes, semaphores, and other threading objects in an undefined state.

**Problem: How can we make the forkserver work if we can't fork a VM?**

**Solution: Trampoline the VM**

- Set a flag to tell the vCPU to stop.
- When the vCPU sees this flag set:
  - It exits out of the CPU loop
  - Sends notification to the IO thread
  - Records some CPU state for later
  - Finally, 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.
  - The child thread resumes cleanly from startForkserver()

# Making it Work

- **Things we're doing to get deterministic edge-traces:**
  - Disabling Interrupts and Signals
  - Time warping to make sure all test cases see the same virtual clock.
  - Fixed a bug in AFL QEMU mode that made spurious self-edges when the JIT was interrupted (and so QEMU would re-JIT the block)
    - We did this by disabling QEMU's "chaining" feature
  - Performing tracing in the CPU execution code (instead of in the translated code):
    - If we performed tracing in the code generated for each basic block, we could potentially get a speed gain.
    - 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.
- **AFL is typically used with programs that are considerably smaller than a modern OS kernel.**
  - AFL uses a hash function to map edges to a hashmap.
  - We've had to make adjustments to the map size to accommodate the larger number of edges: the hashmap size has changed from $2^{16}$ to $2^{21}$
    - We were still encountering an unacceptable number of collisions at this hashmap size, but going bigger destroyed performance
    - Updated the hash function to a better hash recommended by Peter Gutmann.
    - Now collisions are <1% at this hashmap size.

# Making it Better

## Logging

- We pass TriforceAFL the address of the log_store function as well
- When QEMU hits this address, it triggers a function that prints the system logs to a file
- Also can flag the test case as having produced syslog output (currently disabled)
- This is Linux only right now, but can easily be translated to other OS's

## Heating

- The original AFL qemu_mode patches 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.
- Allows the parent process to cache the translated block, so future children don't have to repeat the work
- 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 currently disable this feature.
- Instead, we've taken an approach where we run a "heater" program before we run our test driver.
- We 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.


nccgroup
Fuzzing Linux with TriforceAFL

# The Userland Driver

- One of our goals from the beginning was to allow us to leverage the already existing strength of AFL's mutation engine.
- We are also strong believers in the principle of iteration, and always having a fuzzer running.
- We wrote a series of progressively more complicated drivers, each time converting our previous work queue to our new format.
- We started with a driver that could only fuzz system calls with purely numerical arguments.
- This let us observe that AFL was successfully reading edge traces and finding new paths (albeit not particularly interesting ones).
- This lead us to our first (very trivial crash): On certain older kernels, running *umount()* with the MNT_FORCE flag on “/” would cause a kernel panic. Root-only, and just a sanity check.
    - Especially impressive is that it managed to find a null-terminated pointer to “/”
- Our second driver allowed system calls with a single buffer:
    - The buffer could be written to a file (and have the file name passed to the system call)
    - Or be passed directly to the system call as a pointer

# AFL Writes A Shell Script

**So you think you're a shell script expert? What does the following do?**

```
#!/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
```

**We tried analyzing it for a bit, before finally just using strace**

- 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 error messages about malformed elements in the shell script.
- These ended up being written to the special "/proc/sysrq-trigger" file.
- One of these error messages was formatted in such a way to trigger sysrq to cause a kernel panic.

To quote one of my colleague's response: "AFL is skynet".

# AFL Learns To Make a Hypercall

- 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
- 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 an assert we had put in QEMU to 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).
- We took out the assert in the startForkserver() hypercall, and instead detect duplicate calls and exit the forked VM
- From the AFL readme:
  - “Occasionally, sentient machines rise against their creators. If this happens to you, please consult http://lcamtuf.coredump.cx/prep/.”

# Multibuf: Our Release Driver

- **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.**

**It features:**

- Support for multiple system calls in one test case.
- We use the Trinity approach of having a big table of interesting “files” to be opened (sockets, perf_event, files in /proc and /sys, etc)
  - When a test-case uses a FD from the table, the driver opens up the FD.
  - AFL can mutate the table index to change which file is used
- Support for multiple buffers
- And **types**!
  - In fairness, onebuf also had types :P

# Multibuf and Types

- **Multibuf supports a number of "types"**
  - Int
  - Buffers
  - BufferLengths
  - FileContents
  - FileNames
  - FileTableNumber
  - ProcessIDs referencing the fuzzer process, its parent, or a child.
  - Vectors of arguments of any of the supported types.

- Because AFL has no prior knowledge of the semantics of our serialized format, it will mutate type information:
  - e.g. turning (*buf, buf_len) → (*buf, int)
  - Take that type safety!
- By making an extra call to startWork() when we start deserializing the test case, AFL is encouraged to mutate test cases in a way that will cause different deserializations
  - AFL wants to find new edges in the parser
  - It also found a (stupid) crash in our driver through this tracing


nccgroup
Finding Linux Kernel Bugs

# Parallelization and Pollination

- **AFL natively supports distributed fuzzing.**
- **We can do some fun stuff with that.**
  - To start off with, we can run a number of instances of AFL in master/slave configuration
    - We built “min” and “fat” versions of the current release 2.X, 3.X, and 4.X kernels. Also got some fun stuff like KASAN builds in there (very slow!).
    - All these TriforceAFL instances used the same driver, so they can share a work queue (because they all parse the same serialized system call format).
    - By using a variety of different speed kernels, we can have the fast ones help out the slow ones (and vice-versa)
  - These different kernels cross-pollinate each other, which (we think) can help the fuzzer focus on differences between kernels (which often includes new functionality, and compatibility code, frequent areas of vulnerabilities).

# Performance

**These are just us trying to give some approximations of performance. YMMV HEAVILY, especially depending on things like size of L2 cache**

- We ran our ‘testAFL’ program on different kernels on a specific quad-core Linux machine, with no other major processes running on it (using only a single core).
- We used a fixed suite of test files gathered from earlier fuzzing, noted how many executions per second ‘testAFL’ 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).

# Doing Some Real Fuzzin'

- **NCC Group Senior Consultant Joel St. John was nice enough to give us (almost) free reign on his multi-core Linux server.**
  - 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.
  - 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.
  - Keep in mind: these numbers are imprecise, as we often trashed our work queues during driver iteration.
  - 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).
- **To the cloud!**
  - Recently, our friends are Digital Ocean were nice enough to give us some free credit to fuzz OSS, so we setup a cloud fuzzing cluster.
  - For an (imprecise) performance measure, each (single core, lowest specs possible) Linux droplet is currently averaging 58.86 exec/s.
  - This scales fairly nicely, although required us to solve some silly isues:
    - e.g. We ran out of inodes! (ext4 luckily lets us create a FS with many inodes)

# Corpus Generation

**Big thing we left out so far: how did we get a corpus of test cases?**

We started by doing static analysis of the syscall definitions to identify common "shapes" of syscall arguments (i.e. (fd,ptr,int) for a *write(2)* or *read(2)* syscall)

- This produced 116 common shapes
- Now, here's another either great solution or awful kludge depending on your POV
  - There are a lot of syscalls (roughly 400)
  - What if we took every syscall shape we had, made an example, and used it for every syscall number…
  - This produces a pretty large corpus, with a lot of syscalls that are completely invalid, or are redundant
- This sounds like a job for afl-cmin!
  - This is AFL's corpus minimization tool, that uses the feedback results from running test cases in order to build a minimized corpus of test cases
  - Very useful at the start of a fuzzing run, as feeding AFL with too many (or too large) test cases can lead to a lot of wasted cycles

# Forkserver Fixes All Problems

- **Issue: Stock afl-cmin doesn't use the forkserver**
  - Not a big deal on most binaries, that have relatively short startup times
  - Makes it very impractical to run in our setup (full VM boot) for several thousand test cases
- **Solution: Adapt afl-cmin to use the forkserver**
  - We also did this for afl-showmap

**afl-tmin (test minimization) and afl-analyze (used for crash analysis) have been patched to add support for full system emulation**

- We haven't yet added forkserver support to these programs
- They are likely too slow to be usable at the moment.
- Send us a pull request if you'd like to help out, on that, as those aren't the highest on our priority list ☺


# Linux Kernel Bugs!

# The Trivial Linux Bugs

- **Even with our "stupid" drivers (and poor inputs to them) we still found some crashes**
  - The aforementioned (very silly) lack of **MNT_LOCKED** in certain 2.X and 3.X kernels.
    - This was just a kernel BUG_ON, root only DoS: overall very low impact.
- **Our smarter drivers started finding some more interesting crashes**
  - TIOCSSERIAL *ioctl(2)* DoS:
    - Found a number of issues in an *ioctl(2)* on the root serial device.
    - These were root only, so low impact.
    - Still, our first "real" bug, different versions triggered either a null-pointer-dereference, a WARN_ON call, or a divide-by-zero.
    - Interestingly, given one of the above, TriforceAFL will find the other two.
  - Linux 2.X Process Group 0 Crashes:
    - These were found by cross-pollination, test-cases that ran fine on Linux 3/4 kernels crashed Linux 2 kernels
    - These were all null pointer deferences only triggerable from a process in process group 0 (recall our driver was started as part of init).
    - Some were unprivileged! Some were root-only.

# Netfilter Crashes

- Both of these issues are in the complex (and buggy) Linux netfilter code
- 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.
  - Containers anyone ;) ?
- 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 good attack surface when we started fuzzing, TriforceAFL figured that out on its own).
- When we reported our two “high severity" issues, we found out there were already patches upstream that fixed them.
- It seems after ProjectZero reported their bug, the kernel maintainers had been overhauling the netfilter code, although these fixes had not yet been backported to any stable or distro release
- Since all the netfilters (arp, ip, ip6, eb) all shared the netfilter code, these issues (probably) effected all of these.
- 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, our bugs (and ProjectZero's bug) seem to have been the final straw causing multiple distros to change their defaults to disallow unprivileged users from creating user namespaces.

# CVE-2016-4998

## Heap Overread in setsockopt IPT_SO_SET_REPLACE

When installing an IP filter with the setsockopt(2) system call using the IPT_SO_SET_REPLACE command, the input record (a *struct ipt_replace*) and its payload (a *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.
- This issue 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.

# **CVE-2016-4998**: The Gory Details

- This triggers a call to *translate_table()*, 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.
    - Can also lead to log messages which may allow users to infer information about adjacent heap data.
- After returning, *translate_table()* accesses the target's *u.user.name* field using *target_offset*. This access can be out of bounds and can result in a kernel BUG.

# A little more on **CVE-2016-4998**

- 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.
  - An attacker may be able to consume large amounts of kernel memory with multiple simultaneous calls.
- We were happy our project had produced a (non-root, non-trivial) DoS.
  - A “legitimate” bug
- But we were quickly impressed/astounded by what AFL managed to mutate this into

# CVE-2016-4997

## Arbitrary Decrements in compat_setsockopt IPT_SO_SET_REPLACE

- Similar to the earlier bug, but in the 32-bit compatibility version of the syscall
- Due to incomplete validation of *target_offset* values in *check_compat_entry_size_and_hooks()*, a critical offset can be corrupted.
- Several important structures are referenced from **unvalidated** memory during error cleanup.
- Result: a malicious user can decrement arbitrary kernel integers when they are positive.

# **CVE-2016-4997**: Details

- In *check_compat_entry_size_and_hooks()* the entry is validated.
  - This function checks that *target_offset* is not too big, but does not check if it is too small!
  - This leads to initialization on an *'ematch'* being skipped
- A small value of *target_offset* lets the *ematch's* target pointer point into itself!
  - This allows further corruption of the *'ematch'* struct
- Later, when iterating over the same object in *compat_release_entry()*, the kernel iterates over matches that didn't exist earlier (when *target_offset* was too small to contain any)
  - These matches were never properly initialized!
- During cleanup of corrupted structs, the kernel performs an decrement/increment on an attacker supplied address, as part of calling *module_put()*.
  - Leads to code execution in kernel

**What happens is actually kernel version dependent:**

- 4: Arbitrary call to *atomic_dec_if_positive*
- 3: Arbitrary increment, but only to memory referenced through the %gs segment
- 2: Arbitrary decrement

# Analysis

- These issues are 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.
- 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.

# Analysis (II)

- Interestingly, after we published the advisories on oss-sec (in coordination with patches being ported to the –stable kernels), a twitter user came forward with a weaponized exploit for CVE-2016-4998 on Ubuntu 16 (priv-esc to root).
  - Interestingly, the PoC had a date in it the comments which (if true), places its authorship after the ProjectZero advisory, but before our bug disclosures.
- What can we conclude from this?
  - To some (unknown) degree, exploits for Linux 0-days are in the wild
  - Fuzzing can find these issues, and (arguably) improve the state of OSS security
  - Reviewing upstream branches, previous bug reports, and similar can often lead to finding 0-days (although not for us)


nccgroup
Fuzzing OpenBSD

# Target Switching to OpenBSD

- **With Linux, we had the advantage that the Linux kernel naturally supports booting off a ramFS from a CPIO archive**
  - This means that we got idempotence in test cases (and so could fork VMs freely).
  - Without idempotence, test cases would trample on a shared backing store.
- **OpenBSD (like most modern OS's) wants a real backing store to run off**
- **Brilliant Solution / Kludge #3:**
  - Configure OpenBSD to boot off a read-only emulated SCSI disk
  - Use the work of the FlashRD project, pivot the entire FS to a ram-based FS.
  - Now we get the idempotence needed, forks work properly.
- **Because our driver was written as a standard POSIX C program, it was trivial to port to OpenBSD.**
- **Because of our somewhat "interesting" approach to generating test cases (and the large similarities between the POSIXs), we could use the same trick with afl-cmin with all possible syscall numbers to get a corpus for OpenBSD**
- **And so with approximately one person-week (Tim-week, but still) we could fuzz OpenBSD 5.9**
  - For reference, building TriforceAFL and the Linux Syscall fuzzer had taken on the order of 10-person weeks.

# Some minor OpenBSD bugs

**These are minor issues, so we'll just give a brief description:**

- **CVE-2016-6242:** A DoS where providing an overly large value 'kqueue_register() will trigger a 'kassert' in 'mallocarray' . Triggerable by any user.
- **CVE-2016-6242:** Creation of a tmpfs mount with invalid combinations of flags can lead to a sanity check panic(). This is root only, unless kern.usermount is enabled.
- **CVE-2016-6247:** When using unmount() with the MNT_DOOMED flag, a sanity check causing a kernel panic is triggered. Similar to the above, it is root only unless kern.usermount is enabled.
  - OpenBSD disabled this flag by default in 5.9 ☺
- **CVE-2016-6350:** A null pointer deference when trying to use one of tmpfs's vfs_ops that was null. Triggerable by any user.
- **CVE-2016-6244:** Integer overflow in __thrsigdivert() leads to a negative timer value, triggering a sanity check that called panic().
  - Amost identical to the above is **CVE-2016-6243** in __thrsleep
- **CVE-2016-6245:** In the getdents() call on the UFS filesystem, an arbitrary large buffer can be requests, leading to a kernel panic.
  - Can be triggered by any user who can read a directory on a UFS filesystem.

# Memory Corruption in *mmap(2)*

**"What would be the funniest call to find a memory corruption issue in"**

**(CVE-2016-6239 and CVE-2016-6240)**

**This is so far my favourite issue we've found as part of ProjectTriforce.**

- It is also one of the least understandable. I can't remember how it works half the time, but let's give it a shot.

- When a user provides the __MAP_NOFAULT flag to *mmap(2)*, the kernel calls *amap_alloc()* which calls *malloc(2)* with a size derived from the user-passed size.
  - The *amap_alloc()* call is reachable whenever the UVM_FLAG_OVERLAY flag has been selected.
  - This happens when mapping a file with the __MAP_NOFAULT or when making a MAP_ANON mapping.
  - However, the MAP_ANON performs validation and prevents large allocations from happening in *amap_alloc()*.
- The call chain goes through *sys_mmap()*, *uvm_mmapfile()* and *uvm_map()* without validating the user-provided size.
- This can result in a panic in *malloc(2)*.

# Fun in *amap_alloc()*

- **If we avoid causing the panic, the *amap_alloc()* code can also miscalculate the allocation size (through integer overflows)**
  - This causes an undersized allocation in *amap_alloc1()*, which can lead to memory corruption later.
- **There are two vulnerable paths**
  - *amap_alloc()* makes a truncation in converting the requested *size_t* size into an *int* variable *slots* (representing the number of slots needed)
    - If the *size* is larger than 0x1000.0000.0000, it will result in a truncated value of *slots*, resulting in an undersized amap.
  - The more complicated/interesting path also occurs in the interaction between *amap_alloc()* and utility function: *amap_alloc1()*.

# *amap_alloc()* and *amap_alloc1()*

- **In *amap_alloc1()***
  - The number of slots is rounded up so that the slot entries fill pages.
  - This rounding up happens in the int *totalslots* variable, and can overflow the original *slots* value.
  - For example, requesting an allocation of size 0xfff.ffff.0000:
    - In this case *amap_alloc()* computes that 0xffff.fff0 slots are needed
    - *amap_alloc1()* computes that zero *totalslots* are needed
      - allocates an amap of zero-bytes.
  - If the *amap->am_slots*, *amap->am_bckptr* or *amap->am_anonfields* are later accessed, it can lead to out-of-bounds reads and writes on the kernel allocation heap.
    - Many accesses through these pointers are guarded by *am_slots* (in the example given: 0xfffffff0) rather than *am_maxslots* (which, in the example, contains the flawed slot count of zero).
    - This might lead to kernel code execution and privilege escalation!
    - Weaponization left as exercise to readers ☺


nccgroup
Current Work, and the Future

# Current Work - privmem

- **One of the biggest issues holding us back is necessity on ram-disk only operating systems**
  - For OpenBSD we wrote a kludge (and used the work done by the FlashRD project) to get around that
  - We want a generic solution
    - Both to fuzz Operating Systems that may not support RAM only filesystems
    - And to make this more practical to use for fuzzing an arbitrary binary running in a VM (but perhaps not a Linux ramdisk VM)

- **Solution:** We wrote a QEMU backing store driver that emulates a normal IDE disk and supports CoW semantics.
  - Its called **privmem**
    - It doesn't quite work yet, but we will (probably) have it working soon
    - Code is available in an experimental branch on our GitHub if you want to help get it working ☺

# Current Work – arm32

**Someone said: Why do you guys only support x64?**

- Well, because we wanted to figure out how to use AFL on kernels, and it was the easiest and whatever
- But… valid point! Let's prove that this approach can be generalized
- We wrote an extension to ARM32
  - Surprisingly, not that hard, just inserting the relevant hooks into the functions in qemu/target-<arch>/translate.c (and having a bit of knowledge of the assembly for the architecture)
  - This opens up the fuzzing of Android devices and other things built on ARM*.
    - You just need an Android "distro" that runs on the QEMU "hardware"
    - Or, port our existing work to the Android QEMU emulator
  - To see if it worked, we fuzzed a Linux kernel built for ARM32 and compared it against a Linux kernel built for x64, on the same host
    - Ironically, they perform about the same

*this may be harder than expected

# Current Work: Corpus Generation

- **Building a library of syscalls**
  - At least one (manually verified) valid syscall for a given architecture. Ideally, we cover roughly the manpage for the syscall.
    - We're mostly done with Linux and OpenBSD on this.
- **Syscall-Recorder**
  - Capturing common patterns of syscalls from binaries programmatically. Think of this as "strace on steroids"
    - Issue with "just use the strace output": most of it is pointers to the process's address space
    - Need to rip these structs out and include them in the test case
  - Have most of this written, just missing the part that detects the lengths of the structs (several ideas we're throwing around on how best to do that)
- **Triforce-Invoker**
  - Take a utility with a manpage, lets say 'ls'
  - Parse all the flags to ls, feed those as a dictionary to AFL
  - Write small shim that passes its input as args to 'ls'
  - Use AFL for its "true purpose" of synthesizing a corpus of inputs that explore the different paths through 'ls'
  - Use that with above, "auto-record" a ton of useful/good/common syscall patterns. Free test cases!

# More Ideas

- **Better  Categorization of Results**
  - Current system just runs cases and collect dmesg output
    - We use regexs to bucket crashes (forgive us)
  - Automatically cross-running queues and crashes on different kernels, recording dmesg differences
- **Add some structural mutational types into to AFL's mutation engine.**
  - This would be architecture specific (probably Linux specific first)
  - Smart way to do this would be to add a new mutation type as a plugin to AFL, which would unpack, mutate, and repack test cases.
    - AFL has been talking about refactoring to become more modular, this would fit in nicely with that
    - This would mean we could keep being modular and write a plugins per target.
    - We can even use some of syzkaller's mutation code here ☺
- **Sharing JIT between forks, doesn't currently work due to threading issues.**
  - We'd need to make QEMU full system emulation single-threaded.
  - There would be other performance benefits out of this as well.
- **Driverless Triforce**
  - What if we didn't have to write a driver, and Triforce automatically snapshotted a VM while processes were executing a system call, then used that system call as the initial seed, and fuzzed from there?

**This is a really important idea: fuzzing real syscalls in real programs in their "natural" context**

# Improving Performance

- **Cache Optimization**
  - Because the targets we've been investigating at significantly larger than a usual binary, we increased the size of afl's edgeMap significantly.
    - This can cause a lot of cache pressure, especially considering we have QEMU constantly JITing code blocks.
  - This could possibly be fixed by building fuzzing servers with larger caches, or by clever trickery involving integrating the edgeMap into QEMU's code generator.

- **In general, we could do greater profiling of this toolset and look for large performance gains**
  - So far we've mainly stuck to the "pre-mature optimizatiion is the root of all evil" quote, and throwing resources at things is a great way to fix speed issues ☺

- **Switching to a KVM-based execution engine could offer significant speed benefits**
  - However, this is non-trivial, as it requires writing a tracing dynarec KVM-module, as well as moving certain other pieces of the toolset into the KVM layer.
  - This also would not allow us to use cloud resources, as (for very good security reasons) do not allow their users to run code in their hypervisors.

# Fuzz All the Things!

- **With privmem, and some of the other tooling we have (library of syscalls, adapted afl-cmin, ideally a working recorder/invoker pair), we can start targeting other POSIXs**
  - FreeBSD
  - NetBSD
  - HardenedBSD
  - PureDarwin
- **Its possible we can apply this approach to commercial operating systems too, such as OSX and Windows**
  - These of course come with caveats, OSX requires some interesting QEMU-KVM support to run, and Windows has a very different syscall architecture than POSIX (or so I've been told, I am not a Windows expert)

# A Whole New World of Targets

**Again, IFF they can run in QEMU or similar!**

- **Embedded device firmware**
- **Drivers**
- **Hypervisors (what is keeping your VMs on the cloud safe?!?)**
- **Userland processes that are difficult to instrument effectively**
- **Any relatively stateless part of a complex stack**
  - A “server” can be fuzzed using the receiving of messages as the input
  - I’m curious whether memory corruption in systems like HTTP stacks or network stacks that have traditionally been very difficult to fuzz can benefit from this sort of approach
- **More importantly, we don’t need to search for crashes. The ‘goal’ basic block can be anything we desire.**
  - Set the goal to be the call to *system(),* add some taint analysis to see whether input made it to the output, and use this as a CMDi fuzzer
  - Set the goal to be the ‘alert’ function of a Browser, add a dictionary of XSS payloads, and use this as an XSS fuzzer
    - Note: please don’t do this on someone else’s site! The VM should probably be running the server for the target as well for this model to work anyway.
  - AFL is a tool to create corpuses and explore paths, use it!

# Conclusion

- **AFL, QEMU, and automation are all awesome.**
- **We want to work with everyone to fuzz more things.**
- **All code is available on our GitHub.**
  - https://github.com/nccgroup/
  - Full bug reports and crash analysis are also available there
  - Send us a pull request!
- **Help us expand this to more targets, help us generate interesting test cases, donate/use your server-space to fuzz things, and lets run AFL on everything!**
- **Or don't, we're gonna keep at it anyway ☺**

**We love talking about this stuff, so feel free to shoot us emails:**

**jesse.hertz@nccgroup.trust**

**tim.newsham@nccgroup.trust**


nccgroup
Thanks for listening to us for so long!


nccgroup
Q+A / Lets talk about fuzzing stuff


nccgroup