Project Triforce: Run AFL on Everything!

tl;dr

This is a pretty long blogpost, so for those who want to jump right to the code:

TriforceAFL (https://github.com/nccgroup/TriforceAFL): A modified version of AFL that supports fuzzing using QEMU's full system emulation.
TriforceLinuxSyscallFuzzer (https://github.com/nccgroup/TriforceLinuxSyscallFuzzer): A Linux syscall fuzzer built on-top of TriforceAFL. It has already found several bugs:

- CVE-2016-4997 (https://github.com/nccgroup/TriforceLinuxSyscallFuzzer/tree/master/crash_reports/report_compatIpt) : Corrupted offset allows for arbitrary decrements in compat IPT_SO_SET_REPLACE setsockopt
- CVE-2016-4998 (https://github.com/nccgroup/TriforceLinuxSyscallFuzzer/tree/master/crash_reports/report_ipt) : Out of bounds reads when processing IPT_SO_SET_REPLACE setsockopt
- As well as some miscellaneous low-impact crashes we've analyzed: https://github.com/nccgroup/TriforceLinuxSyscallFuzzer/tree/master/crash_reports (https://github.com/nccgroup/TriforceLinuxSyscallFuzzer/tree/master/crash_reports)

We will be releasing a whitepaper shortly that covers all of TriforceAFL, the design and implementation of TriforceLinuxSyscallFuzzer and will also contain all the bug analyses. So keep a lookout for that. And now, onto the blogpost!

Inspiration, Motivation, and Fever Dreams

AFL (http://lcamtuf.coredump.cx/afl/) is an awesome tool. The power of an easy to use, feedback-driven fuzzer has produced an absolutely staggering number of bugs. Still, at first AFL required being able to build the executable, something sadly not available on a lot of targets. With the addition of AFL's qemu_mode, it became possible to fuzz binaries without source, exposing a whole new world of targets to AFL. I'd been on a number of Linux container engagements recently where we'd managed to escape through kernel exploits. I fell asleep one night to several AFL screens running, and I awoke suddenly with a crazy idea: "Run AFL on the Linux Kernel."

Well, this isn't exactly a totally new idea. In fact, Google has had a very successfully feedback-driven Linux syscall fuzzer, syzkaller (https://github.com/google/syzkaller). Trinity (https://github.com/kernelslacker/trinity) [1], perhaps the most successful Linux system call fuzzer, briefly considered adding feedback support (http://codemonkey.org.uk/2015/05/05/thoughts-feedback-loop-trinity/). Oracle recently showed some very intersting work on using AFL to fuzz Linux filesystem drivers (https://events.linuxfoundation.org/sites/events/files/slides/AFL%20filesystem%20fuzzing,%20Vault%202016_0.pdf).

TriforceAFL

After a lot of internal discussions about what it would even mean to "Run AFL on the Linux Kernel," Tim Newsham and myself set out to extend AFL's existing userland QEMU support to support fuzzing an OS running under QEMU's full system emulation mode.

This offers several advantages over previous techniques:

- Unlike Oracle's work, we won't need to recompile pieces of the kernel with AFL, or figure about how instrument core parts of the kernel.
- Unlike syzkaller, kernels don't need to be built with coverage support. Any kernel will do. And, since we're capturing edge info (rather than coverage), we get the full benefits of AFL's feedback engine.
- Because of the way we use AFL+QEMU, we fork right before decoding and executing the test case, offering surprisingly good performance, since we only have to incur the cost of booting the OS once.
- It's generic! This can be used to fuzz anything that can run under QEMU's x64 full system emulation mode (more on that later). This is the main reason I think this is so cool.

So, how about some details?

Important Note

The use of the word 'driver' from here on out refers to the userland programs that act as a 'decoder' of test cases, and run inside the VM. They are ***not*** *operating system drivers. Please excuse the poor decisions in terminology.*

The Design

Normally, when fuzzing with AFL, a target program is started for each test case and runs to completion, or until it crashes. When fuzzing in the context of a full system, this is not always desirable or possible. Our design allows the hosted operating system to boot up and load a 'driver' 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 test case.

A typical TriforceAFL fuzzer would perform the following steps:

- Boot an operating system.
- The operating system would invoke a fuzz driver as part of its startup process.
- The driver process would 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:
  - Get a single test case.
  - Enable tracing of the parser.
  - Parse the test case.
  - Enable tracing of the kernel or some portion of the kernel.
  - Invoke a kernel feature based on the parsed input.
  - Notify that the test case completed successfully (if the test case wasn't terminated early by a panic).

Note that since the fuzzer runs in a forked copy of the virtual machine, the entire in-memory state of the kernel for each test case is isolated. If the operating system uses any other resources besides memory, these resources will not be isolated between test cases. For this reason, its usually desirable to boot the operating system using an in-memory filesystem, such as a Linux ramdisk image.

The Implementation

The driver communicates with the virtual machine through a special instruction that was added to the QEMU x64 CPU that we call `aflCall` (`0f 24`). It supports several operations:

- `startForkserver` - This call causes the virtual machine to start up the AFL fork server. Every operation in the virtual machine after this call will run in a forked copy of the virtual machine that persists only until the end of a test case. As a side effect, this call will either enable or disable the CPUs timer in each forked child based on an argument. Disabling the CPU timer can make the fuzz tests more deterministic, but may also interfere with the proper operation of some of the guest operating system's features.
- `getWork` - This call causes the virtual machine to read in the next input from a file in the host operating system and copy its contents 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 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 input file and then trace the kernel while performing a system call based on the input file. AFL's search algorithm will only be aware of the edges that are traced, and this call provides a means to adjust what parts of the system to trace.
- `endWork` - This call notifies the virtual machine that the test case has completed. It allows the driver to pass in an exit code. The forked copy of the virtual machine will exit with the specified exit code, which is communicated by the fork server back to AFL and used to determine the outcome of a test case.

Besides the driver calling `endWork`, the virtual machine can end a test case if a panic is detected. This is achieved by providing the QEMU virtual machine with an argument specifying the address of a panic function. If this function is ever called, the test case is terminated with a status of 32. Note that this argument can specify any basic block of interest, it need not represent the `panic` function of the operating system.

The virtual machine can also intercept a logging function by specifying an argument to QEMU with the address of Linux' `log_store` function. The virtual machine 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. 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 virtual machine can optionally bitwise OR in the value 64 to the exit code to indicate that logging occurred. However, we did not find this feature is particularly useful, so it is currently disabled in the source code.

QEMU and AFL Code

ProjectTriforce's version of QEMU borrows heavily from the AFL QEMU patches. These patches already included code to trace execution edges into AFL's edge map. However, we found that there was a subtle bug in the tracing due to QEMU's execution strategy. Sometimes QEMU starts executing a basic block and then gets 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 causes some extra edges to appear in the edge map, and introduces some non-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` to trace a basic block only after it has been executed to completion.

AFL's QEMU performs tracing in its CPU execution code. We experimented with performing tracing in the code generated for each basic block. This results in a performance gain since the hash function used to hash addresses is only computed once, at translation time. However, due to the issues with non-determinism when basic blocks are interrupted, and due to some other unresolved issues, we decided to continue using the existing tracing method.

AFL's QEMU has a feature 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 translate the block so that 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.[2] 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 fork server is started. This approach is an optimization that has boosted our performance a little but is not strictly necessary.

AFL is typically used with programs that are quite a bit smaller than a kernel. To cope with the larger program, we adjusted the edge map size from 2^16 to 2^21 to reduce edge collisions to an acceptable level, and we updated the hash function to a better hash recommended by Peter Gutmann. More information about the measurements that lead to this map size can be found in https://groups.google.com/forum/#!searchin/afl-users/hash/afl-users/iHCx2Z2WncI/Okyn1oXkIwAJ (https://groups.google.com/forum/#!searchin/afl-users/hash/afl-users/iHCx2Z2WncI/Okyn1oXkIwAJ).

To support panic and logging detection, we added new command line options 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. These 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: `helper_aflInterceptPanic` or `helper_aflInterceptLog`.

Communication from the driver in the guest host to the virtual machine is supported with an additional CPU instruction 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 straight-forward. The implementation of `startForkServer` is deceptively complicated.

One of the biggest issues we faced when trying to support full-system emulation was getting the fork server 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 is preserved in the child process. Fork also does not 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 approach to starting the fork server. 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. The IO thread receives its notification and performs a fork. At this point there are only two threads -- the CPU thread and an internal RCU thread. The RCU thread is already designed to handle forks properly and needs not be stopped. In the child, the CPU is restarted using the previously recorded information and can continue execution where it left off.

AFL Utilities and Modifications

We did not need to make very many changes to the AFL utilities. The most ubiquitous change was to add a new `-QQ` option to many of the tools. The old `-Q` option enables QEMU user-mode emulation. The new `-QQ` mode enables full-system emulation in QEMU. Unlike the user-mode version, the system-mode feature does not attempt to be clever in setting up the command line, and expects the user to pass in the path to the QEMU emulator and its flags.

We also made some tuning changes to the AFL configuration. Since our operating system might take several minutes to start up before invoking the fork server, we increased the amount of time that AFL will wait for the fork server. We also increased the default memory limit.

Because our virtual machine indicates panics and other undesired behaviors with an exit code, `afl-fuzz` was altered to treat all non-zero exit statuses as a crash.

Several standard AFL utilities do not support using the fork server feature. This is usually acceptable when a test program can be executed in a fraction of a second. However, our test cases can only be started after a lengthy operating system boot process and the test case itself is only a portion of the entire execution. To run properly with our test cases, we require that the utilities use the fork server. We patched `afl-cmin` and `afl-showmap.c` to add fork server support. In the process we had to make some changes to how these programs work. `afl-showmap.c` now supports a batch mode where the maps for many inputs are written to separate files in an output directory. `afl-cmin` makes use of this batch mode and no longer supports the stdin feature, which doesn't make sense in this context. Although `afl-tmin.c` and `afl-analyze.c` have been patched to add support for the `-QQ` option, we have not yet added fork server support to these programs and they are likely too slow to be usable at the moment.

The Future and More

Using TriforceAFL, we built a Linux syscall fuzzer (TriforceLinuxSyscallFuzzer). We'll have a whitepaper coming out soon detailing how we built it, how we generated test cases, how it works, and analyzing the bugs it found.

Most importantly, TriforceAFL can be used to fuzz anything that can run inside QEMU's x64 emulator! Using this, it should be possible to fuzz a variety of operating systems that were previously inaccessible to feedback driven fuzzers. This isn't limited to fuzzing system calls either. This tool can be used to fuzz drivers, userland programs, etc. All you need is:

- A guest OS that runs entirely off a ramdisk and runs inside QEMU's x64 emulator. [3]
- A 'driver' program that runs inside the guest OS (and starts on boot), which decodes and executes test cases.
- Some initial test cases (again, see our upcoming whitepaper which will describe how we created a corpus of test cases for Linux system calls).

We're releasing this tool in the hope that it can be used to fuzz targets which were previously inaccessible to AFL. Feel free to send us a pull-request!

Code Diffs in Git

We tried to organize things 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 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.

---

1. The fuzzer our project alludes to in name. At an earlier stage, we planned to combine Trinity, AFL, and QEMU into a new thing. Being nerds, thought Triforce might be a fun name, and even had plans to name components Link and Navi and such. In the end, we just extended AFL+QEMU, and didn't use anything from Trinity. So the name Triforce is now essentially an orphaned initialism. ↩

2. A QEMU hacker who knows far more than me suggests that this issue will be solved if we merged operations from the RCU and IO threads into the main thread so threads don't need to be recreated on fork. Again, feel free to send us a PR! We may also fix this ourselves at some point. ↩

3. This is a known limitation (as discussed in the design section), and a number of solutions are possible for systems which require a real backing store. The solution we've been tossing around is to have the host maintain a base FS that is forked with the VM, and provides CoW semantics. Feel free to send us a pull request! We may also get around to writing this ourselves. ↩

Published date: 27 June 2016

Written by: Jesse Hertz and helped by Tim Newsham

(https://twitter.com/share)

Filter By Service

- ☐ Software Escrow & Verification
- ☐ Security Consulting
- ☐ Risk Management & Governance
- ☐ Digital Transformation
- ☐ Strategy Consulting
- ☐ Website Performance
- ☐ Software Testing
- ☐ Corporate
- ☐ Business Insights
- ☐ Careers

Filter By Date

2017 (/uk/about-us/newsroom-and-events/blogs/?Year=2017)

September (17) (/uk/about-us/newsroom-and-events/blogs/?Year=2017&Month=9)

August (21) (/uk/about-us/newsroom-and-events/blogs/?Year=2017&Month=8)

July (12) (/uk/about-us/newsroom-and-events/blogs/?Year=2017&Month=7)

June (16) (/uk/about-us/newsroom-and-events/blogs/?Year=2017&Month=6)

May (16) (/uk/about-us/newsroom-and-events/blogs/?Year=2017&Month=5)

April (17) (/uk/about-us/newsroom-and-events/blogs/?Year=2017&Month=4)

March (18) (/uk/about-us/newsroom-and-events/blogs/?Year=2017&Month=3)

February (15) (/uk/about-us/newsroom-and-events/blogs/?Year=2017&Month=2)

January (10) (/uk/about-us/newsroom-and-events/blogs/?Year=2017&Month=1)

2016 (/uk/about-us/newsroom-and-events/blogs/?Year=2016)

2015 (/uk/about-us/newsroom-and-events/blogs/?Year=2015)

2014 (/uk/about-us/newsroom-and-events/blogs/?Year=2014)

2013 (/uk/about-us/newsroom-and-events/blogs/?Year=2013)

2012 (/uk/about-us/newsroom-and-events/blogs/?Year=2012)

2011 (/uk/about-us/newsroom-and-events/blogs/?Year=2011)

2010 (/uk/about-us/newsroom-and-events/blogs/?Year=2010)

Call us on:
+44(0)161 660 6993 (tel:+441616606993)

Newsroom & Events

In the media (/uk/about-us/newsroom-and-events/in-the-media/)
News (/uk/about-us/newsroom-and-events/news/)
Press Releases (/uk/about-us/newsroom-and-events/press-releases/)
Events (/uk/about-us/newsroom-and-events/events/)
Blogs (/uk/about-us/newsroom-and-events/blogs/)

About Us

History (/uk/about-us/what-we-do/history/)
Board & Senior Management (/uk/about-us/what-we-do/board-and-senior-management/)
Careers (/uk/about-us/careers/)
Resources (/uk/about-us/resources/)
Office Locations (/uk/about-us/what-we-do/office-locations/)

Investor Relations

Share Price (/uk/about-us/investor-relations/share-price/)
Results & Presentations (/uk/about-us/investor-relations/results-and-presentations/)
Stock Exchange Announcements (/uk/about-us/investor-relations/stock-exchange-announcements/)

Legal

Terms & Conditions (/uk/about-us/terms-and-conditions/)
Privacy Policy (/uk/about-us/privacy-policy/)
Cookie Policy (/uk/about-us/privacy-policy/cookie-policy/)
Accessibility (/uk/about-us/accessibility/)
Sitemap (/uk/sitemap/)

Latest from @NCCGroupplc (https://twitter.com/NCCGroupplc)

New NCC Group blog post exploring the latest news from @Splunkconf (https://twitter.com/Splunkconf)'s annual conference. Take a look here: https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2017/september/splunk-.conf2017-splunk-7-released-and-other-news/ … (https://t.co/Aw8HLcTfe1)

Reply (https://twitter.com/intent/tweet?in_reply_to=913829556968214528) Retweet (https://twitter.com/intent/retweet?tweet_id=913829556968214528) Favorite (https://twitter.com/intent/favorite?tweet_id=913829556968214528)

Get ready: Chrome will be flagging a lot more pages as insecure. Read our blog post to learn more https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2017/september/get-ready-chrome-is-about-to-flag-a-lot-more-pages-as-insecure/ … (https://t.co/pXkxptXUgz)

Reply (https://twitter.com/intent/tweet?in_reply_to=913811104052203520) Retweet (https://twitter.com/intent/retweet?tweet_id=913811104052203520) Favorite (https://twitter.com/intent/favorite?tweet_id=913811104052203520)