Fuzzing Code with AFL

PETER GUTMANN

Peter Gutmann is a Researcher in the Department of Computer Science at the University of Auckland working on design and analysis of cryptographic security architectures and security usability. He helped write the popular PGP encryption package, has authored a number of papers and RFCs on security and encryption, and is the author of the open source cryptlib security toolkit *Cryptographic Security Architecture: Design and Verification* (Springer, 2003) and an upcoming book on security engineering. In his spare time he pokes holes in whatever security systems and mechanisms catch his attention and grumbles about the lack of consideration of human factors in designing security systems. pgut001@cs.auckland.ac.nz

Most programs are only ever used in fairly stereotyped ways on stereotyped input and will often crash in the presence of unexpected input. Test suites designed by humans, assuming there even is a test suite, are only as good as the people creating them and often only exercise the common code paths. This problem is where fuzzing comes in, the creation of input that exercises as many different code paths as possible in order to show up problems in the code. Until recently fuzzing has been a complex and tedious process, but with the appearance of instrumentation-guided fuzzers like AFL the task has become much easier. This article looks at how you can apply AFL to your code.

Fuzzing Software with AFL

Most software is pretty buggy. The reason why it works a lot of the time is that we use it in ways that don't trigger the bugs, either because the bugs are in obscure parts of the code that never get exercised or because they're in commonly used parts of the code but we know about them and avoid triggering them. Since most programs are used in stereotyped ways that exercise only a tiny portion of the total number of code paths, removing obvious problems from these areas will be enough to keep the majority of users happy. This was shown up in one study of software faults which found that one-third of all faults resulted in a mean time to failure (MTTF) of more than 5,000 years, with somewhat less than another third having a MTTF of more than 1,500 years [1].

On the other hand, when you feed unexpected input to these programs, meaning you exercise all the code paths that are normally left alone, you reduce the MTTF to zero. A study [2] that looked at the reliability of UNIX utilities in the presence of unexpected input, and later became famous for creating the field of fuzz-testing or fuzzing, found that one-quarter to one-third of all utilities on every UNIX system that the evaluators could get their hands on would crash in the presence of random input.

Unfortunately, when the study was repeated five years later [3] the same general level of faults was still evident.

Windows was no better. A study that looked at 30 different Windows applications [4]—including Acrobat Reader, Calculator, Ghostscript, Internet Explorer, MS Office, Netscape, Notepad, Paintshop, Solitaire, Visual Studio, and Wordpad, coming from a mix of commercial and non-commercial vendors—found that 21% of programs crashed and 24% hung when sent random mouse and keyboard input, and every single application crashed or hung when sent random Windows event messages.

Before the Apple fans get too smug about these results, OS X applications, including Acrobat Reader, Apple Mail, Firefox, iChat, iTunes, MS Office, Opera, and Xcode, were even worse than the Windows ones [5].

So what can we do about this?

Fuzz Testing

The answer to the question posed in the previous section is to test your app with random input through fuzz testing before someone else, possibly with less than good intentions, does it for you. Until now this has been quite a pain to deal with since the tools were under-documented and required a large amount of manual intervention to do their job. The process would typically involve downloading a fuzzer, staring at the extensive half-page-long manual for a while, and then settling down to trying to figure out whatever arcane scripting language the fuzzer used to get it to generate input for your app.

Even if you got that far, it was often a case of trial and error with a code profiler to determine whether you were getting any useful code coverage from the fuzzing or just wasting CPU cycles.

Eventually, with a large amount of effort and more than a little luck, you could start fuzzing your code.

All of this changed a few years ago with the introduction of instrumentation-guided fuzzers. These compile the code being fuzzed with a custom build tool that instruments the code being compiled and tries to ensure that the fuzzer generates input that exercises all of the different code paths. As a result, the fuzzer doesn't spend forever randomly generating test cases that exercise the same paths over and over, but generates cases that exercise as many different paths as possible. In addition it can prune the test cases to eliminate ones that are covered by other cases, minimizing the amount of effort expended in trying to find problems.

This strategy produces some truly impressive results. The fuzzer I'll be talking about here, American Fuzzy Lop (named after a breed of rabbit), or AFL, managed to produce valid JPEG files recognized by djpeg starting from a text file containing the string "hello" [6]. The files didn't necessarily decode to produce a photo of the Eiffel Tower but did produce valid if rather abstract-looking JPEG images.

When I ran it on my code, I was somewhat surprised to find myself stepping through PGP keyring code when I'd started with a PKCS #15 key file, which has a completely different format. The input file sample for fuzzing that AFL had started with was:

```
00000000  30 82 04 BA 06 0A 2A 86 48 86 F7 0D 01 0F 03 01
00000010  A0 82 04 AA 30 82 04 A6 02 01 00 30 82 04 9F A0
00000020  82 01 96 A0 82 01 92 A0 82 01 8E 30 18 0C 16 54
00000030  65 73 74 20 45 43 44 53 41 20 73 69 67 6E 69 6E
```

What AFL mutated this into over time was:

```
00000000  99 01 A2 04 37 38 F7 27 11 04 00 97 AB 53 62 04
00000010  7F 8C BB 1A 25 0A 58 CA 63 20 9D 43 D4 8D 50 15
00000020  70 68 E3 76 3D 7B C2 76 78 28 23 B6 9A 40 BC CF
00000030  14 88 A3 80 47 3B 5F 17 5F 73 72 5A 60 1F D3 1B
```

Like the JPEGs that started as the text string "hello," it was a syntactically valid PGP keyring, although semantically meaningless.

Building AFL

Using AFL requires AFL itself [7], a compiler, and a compiler tool called Address Sanitizer [8], or ASAN, that's used to detect code excursions. ASAN requires a fairly recent compiler, quite possibly a more recent one than whatever crusty old version your OS ships with, so don't use the version you find in some repository but build it yourself so that you know it'll be done right. You can use either gcc or clang; if you value your sanity I'd recommend clang.

Start by getting the various pieces of the compiler suite that you'll need (clang is part of the LLVM toolset):

```
svn co https://llvm.org/svn/llvm-project/llvm/trunk LLVM
svn co https://llvm.org/svn/llvm-project/cfe/trunk LLVM/tools
    /clang
svn co https://llvm.org/svn/llvm-project/compiler-rt/trunk
    LLVM/projects/compiler-rt
```

Then build clang and the related tools:

```
cd LLVM
mkdir build
cd build
export MAKEFLAGS="-j`getconf _NPROCESSORS_ONLN`"
cmake -DCMAKE_BUILD_TYPE=RELEASE ~/LLVM
cmake --build .
```

If you don't have CMake installed, then either get it from your favorite repository or build it from source [9].

The clang build process will take an awfully long time even spread across multiple CPUs (which is what the `MAKEFLAGS` line does), so you can go away and find something else to do for a while. If you run out of virtual memory during the build process, decrease the -j argument, which spreads the load across fewer processors and uses less resources.

Eventually, the whole thing will be built and you'll have the necessary binaries present in the LLVM/build directory tree.

Now that you've got the tools that you need to build AFL, you can build AFL itself:

```
wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz
tar xvfz afl-latest.tgz
rm afl-latest.tgz
cd `find . -maxdepth 1 -type d -print | sort -r | head -1`
export PATH=~/LLVM/build/bin:~/LLVM/tools/clang/tools
    /scan-build:$PATH
make
cd llvm_mode
make
```

After all that, you've finally got the AFL tools ready to go.

Building Your App

Now you need to build your app. First, you need to modify it to take as input the data generated by AFL. If your app is something that takes a filename on the command line, as the djpeg example mentioned earlier does, this is pretty straightforward. If the app is a bit more complex than that, for example a GUI app, then you'll need to modify your code to allow the test data to be fed in. In my code I use a custom build that no-ops out a lot of the code that isn't relevant to the fuzzing and allows the test data to be injected directly into the data-processing code.

If you're fuzzing a network app then things get a bit more complicated. There's ongoing work to add support for fuzzing programs that take input over network sockets, one example being [10], but since it's work-in-progress it could well be superseded by the time you read this. A much easier option is to modify your code to take input from a file instead of a network socket, which also avoids the overhead of dealing with a pile of networking operations just to get the test data into your app.

Finally, if your app has a relatively high startup overhead, then AFL provides additional support for dealing with this, which I'll describe later in the section on optimizing AFL use.

Once you've got your code set up to take input from AFL, you can build it as you normally would, specifying the use of the AFL tools instead of the usual ones. For example, if you build your app using a makefile, you'd use:

```
export AFL_HARDEN=1 ; export AFL_USE_ASAN=1 ;
make CC=afl-clang-fast CFLAGS=-fsanitize=address
```

which builds the code with instrumentation and ASAN support. `afl-clang-fast` is the AFL-customized version of clang that adds the necessary instrumentation needed by the fuzzing. As the code is built, you'll see status reports about the instrumentation that's being applied.

One thing that you need to make sure of is that your app actually crashes on invalid input, either explicitly by calling `abort()` (typically via an assertion) or implicitly with an out-of-bounds memory access or something similar that ASAN can detect. ASAN inserts guard areas around variables and can detect out-of-bounds and other normally undetectable errors. But if your program simply continues on its way with invalid input, then AFL can't detect a problem. The easiest way to ensure an AFL-detectable exit is to sprinkle as many sanity-check assertions as possible throughout your code, which means that if any pre- or post-condition or invariant is violated by the input that AFL generates, it can be detected.

Fuzzing Your App

Now you're finally ready to fuzz your app. To do this, run the fuzzer as `afl-fuzz`, giving it an input directory to take sample files from and an output directory telling it where to store statistics and copies of files that produce crashes. If your app was built as `a.out`, you'd use:

```
afl-fuzz -i in -o out ./a.out @@
```

The `@@` is a placeholder that `afl-fuzz` replaces with the path to the fuzzed data files that it generates.

When the fuzzer is running, the results will be displayed on an annoying screen-hogging live status page that prevents you from running more than one copy on multicore systems. To deal with this, use `nohup` to get rid of the full-screen output. The AFL tools have built-in support for running across multiple cores or servers but the details are a bit too complex to go into here and have evolved over time; see the AFL Web pages for more information.

Alongside the status screen, stats are written to a file `fuzzer_stats` in the fuzzer output directory. The important values are `execs_per_sec`, which indicate how fast you're going; `execs_done`, how far you've got; `unique_crashes` and `unique_hangs`, which are pretty self-explanatory (although the hangs aren't terribly useful unless you set the threshold fairly high; they'll be mostly false positives due to timing glitches like page faults and I/O); and finally `cycles_done`, the number of full sets of mutations exercised. Depending on the complexity of your input data, it can take days or even weeks to complete a cycle. There's a tool `afl-tmin` that tries to help you minimize the size of the test cases; again, see the AFL Web page for details.

For each crash or hang, AFL will write the input that caused it to the `crashes` or `hangs` subdirectories in the output directory. You can then take the files and feed them to your app running under your debugger of choice to see what's going on.

If you're running AFL on someone else's machine, you're going to make yourself somewhat unpopular with it. It uses 100% of the CPU per AFL task, and if you maximize the overall utilization with one task per `'getconf _NPROCESSORS_ONLN'` you're going to also have a load average of `'getconf _NPROCESSORS_ONLN'`.

In addition, ASAN uses quite a bit of virtual memory, around 20 terabytes on x64. Yes, that's 20,000,000 megabytes, which it uses as shadow memory to detect out-of-bounds accesses. While this may seem like a gratuitous stress test of your server's VM subsystem, when I ran it on someone else's Linux box it ran without any problems. Just be aware that you're going to really hammer anything that you run this on.

Optimizing the Fuzzing

Many applications have a high startup overhead. As part of its operation AFL uses a fork server in which it preloads the app once rather than reloading it on each run [11], but this still triggers the startup overhead. The way to avoid this is to defer the forking until the startup has completed and tell AFL to fork after that point. So if your app has a code flow that's a bit like:

```
init_app();
process_input();
```

then you'd insert a call to the function `__afl_manual_init()` between the two:

```
init_app();
__afl_manual_init();
process_input();
```

which defers the forking until that point. This means the startup code is run once and then the initialized in-memory image is cloned on each fuzzing run, which can greatly speed up the fuzzing process. If you use this optimization, make sure that you insert the call at the right place. If, for example, you do some of the input processing before the AFL fork-server call, then you'll be reusing a copy of the same input on each fuzzing run rather than reading new input each time.

Beyond that there are various other tweaks that you can apply, which you can also find on the AFL Web page.

So that's how you can test your code's behavior on unexpected input. Using fuzzing may seem like a lot of work to set up initially, but once it's done you can roll it into an automated test system that both identifies existing problems in your code and later checks that you haven't introduced new ones in any updates you make.

References

[1] Edward Adams, "Optimizing Preventive Service of Software Products," *IBM Journal of Research and Development,* vol. 28, no. 1 (January 1984), pp. 2–14.

[2] Barton Miller, Lars Fredriksen, and Bryan So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM,* vol. 33, no. 12 (December 1990), pp. 32–44: http://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf.

[3] Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan and Jeff Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services," *University of Wisconsin—Madison Computer Sciences Technical Report,* #1268, April 1995: ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-revisited.pdf.

[4] Justin Forrester and Barton Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," *Proceedings of the 4th USENIX Windows Systems Symposium (WinSys '00),* August 2000, p. 59: https://www.usenix.org/legacy/events/usenix-win2000/full_papers/forrester/forrester.pdf.

[5] Barton Miller, Gregory Cooksey, and Fredrick Moore, "An Empirical Study of the Robustness of MacOS Applications Using Random Testing," *SIGOPS Operating Systems Review,* vol. 41, no. 1 (January 2007), pp. 78–86: ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-nt.pdf.

[6] Michal Zalewski, "Pulling JPEGs Out of Thin Air," November 7, 2014: http://lcamtuf.blogspot.co.nz/2014/11/pulling-jpegs-out-of-thin-air.html.

[7] Michal Zalewski, "American Fuzzy Lop": http://lcamtuf.coredump.cx/afl/.

[8] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Proceedings of the 2012 USENIX Annual Technical Conference,* June 2012, p. 309: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf.

[9] "CMake": http://www.cmake.org/.

[10] Hanno Böck, "Network Fuzzing with American Fuzzy Lop," October 27, 2015: https://blog.fuzzing-project.org/27-Network-fuzzing-with-american-fuzzy-lop.html.

[11] Michal Zalewski, "Fuzzing Random Programs without execve()," October 14, 2014: http://lcamtuf.blogspot.co.nz/2014/10/fuzzing-binaries-without-execve.html.

ENIGMA

MORE TO DECIPHER

It's time for the security community to take a step back and get a fresh perspective on threat assessment and attacks. This is why in 2016 the USENIX Association launched Enigma, a new security conference geared towards those working in both industry and research. Enigma will return in 2017 to keep pushing the community forward.

Expect three full days of high-quality speakers, content, and engagement for which USENIX events are known.

The Call for Participation will be available soon.

enigma.usenix.org

JAN 30–FEB 1 2017

OAKLAND, CALIFORNIA, USA