Fuzzing with AFL

Peter Gutmann

University of Auckland

Fuzzing before AFL

Download a fuzzer

Stare at the extensive, half-page long manual for awhile

Figure out the arcane scripting notation used to generate input data

- Spend even longer trying to figure out whether you've got a good level of coverage

Run the fuzzer on your code

- Possibly under a coverage tool to tell you whether you're doing something useful or just wasting CPU cycles

Fuzzing after AFL

Download AFL

Get one or more test input files

- Example: Basic JPEG image for an image viewer
- (Actually you don’t even need that, see the next slide)

Run AFL on your code

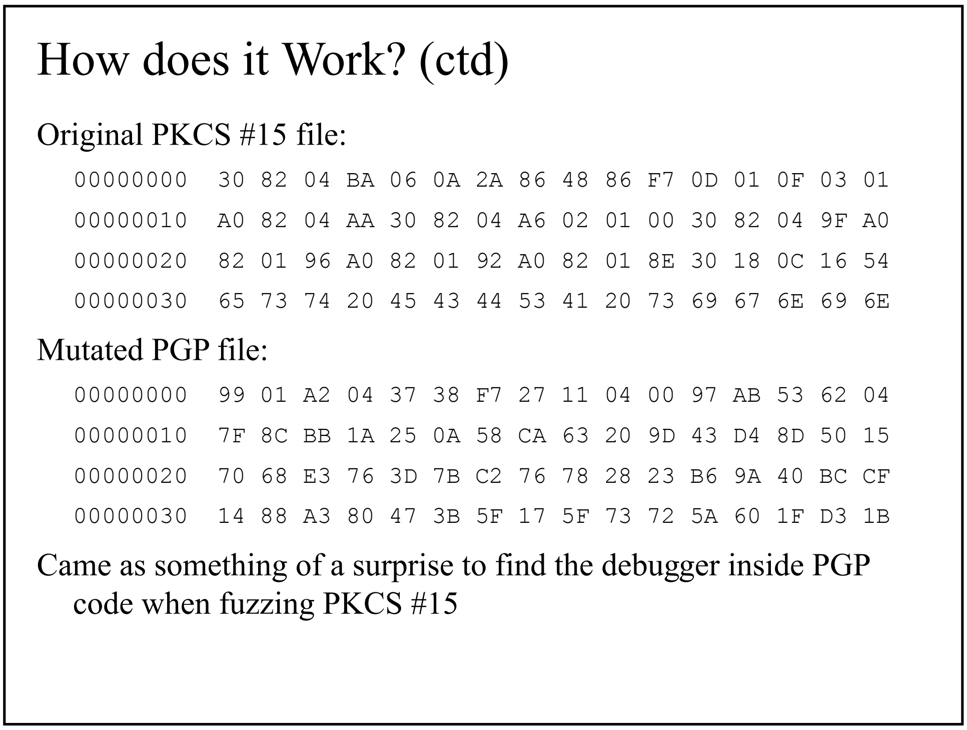
How does it Work?

It’s an instrumentation-guided fuzzer

- Use the compiler to instrument the code being fuzzed
- Mutate the input to explore all code paths

Gives it some pretty astounding capabilites

- Synthesised a series of valid JPEG files starting from the text string “hello”
  - NB: “Valid” doesn’t mean “decodes to produce a meaningful image” merely “is accepted by the parser as a JPEG file”
- Synthesised a PGP keyring starting from a PKCS #15 key file



How does it Work? (ctd)

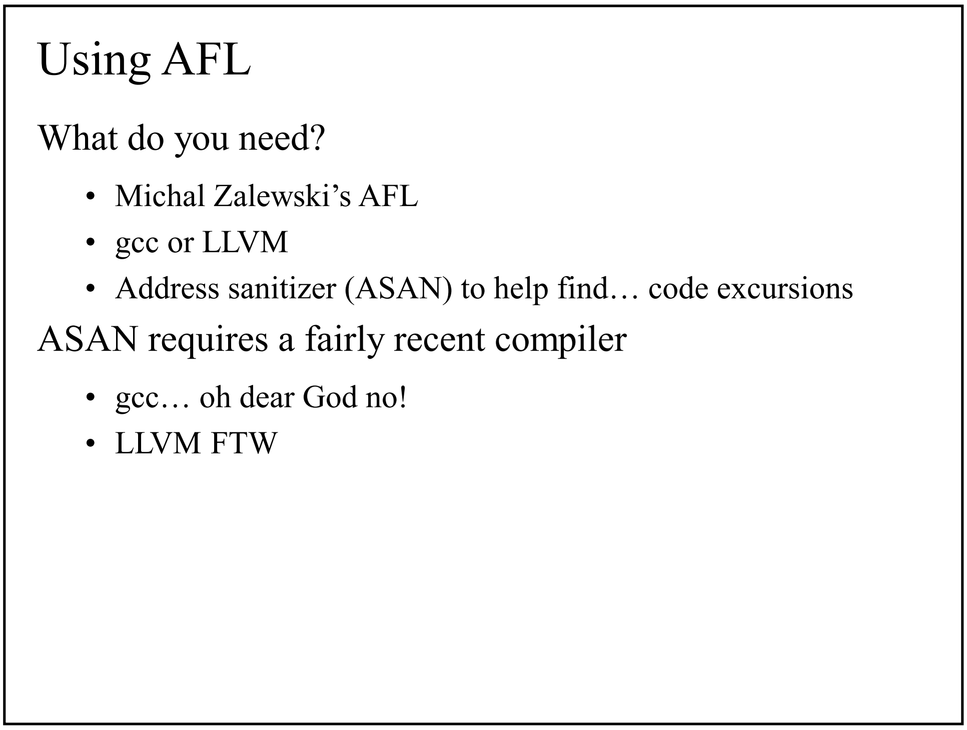
Original PKCS #15 file:

```
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
```

Mutated PGP file:

```
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
```

Came as something of a surprise to find the debugger inside PGP code when fuzzing PKCS #15



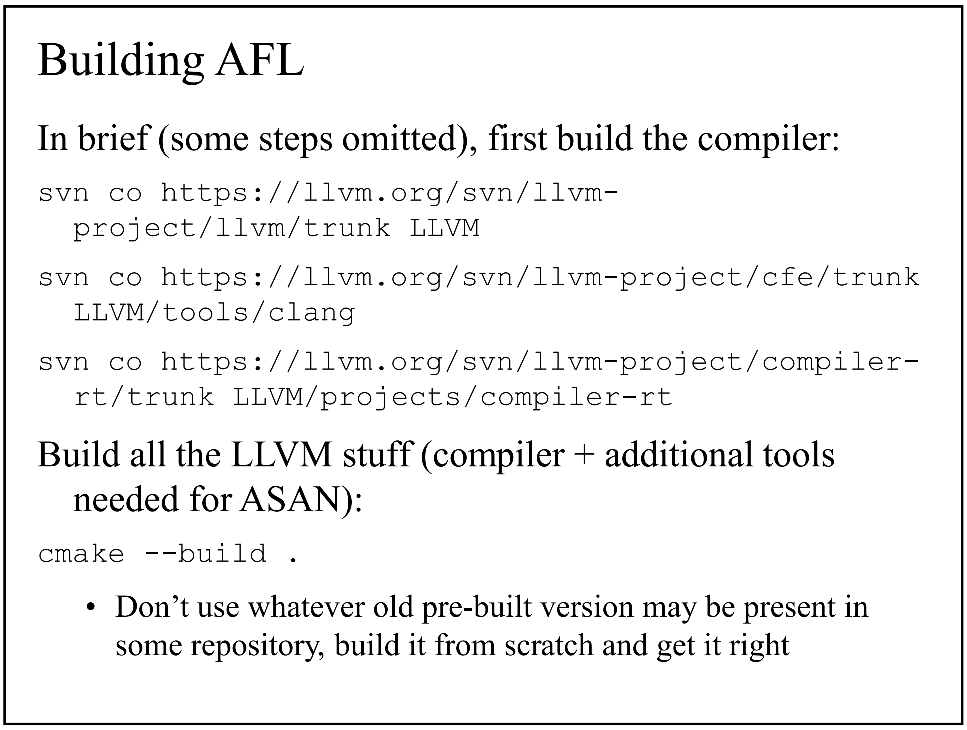
Using AFL

What do you need?

- Michal Zalewski’s AFL
- gcc or LLVM
- Address sanitizer (ASAN) to help find… code excursions

ASAN requires a fairly recent compiler

- gcc… oh dear God no!
- LLVM FTW



Building AFL

In brief (some steps omitted), first build the compiler:

```
svn co https://llvm.org/svn/llvm-
  project/llvm/trunk LLVM
```

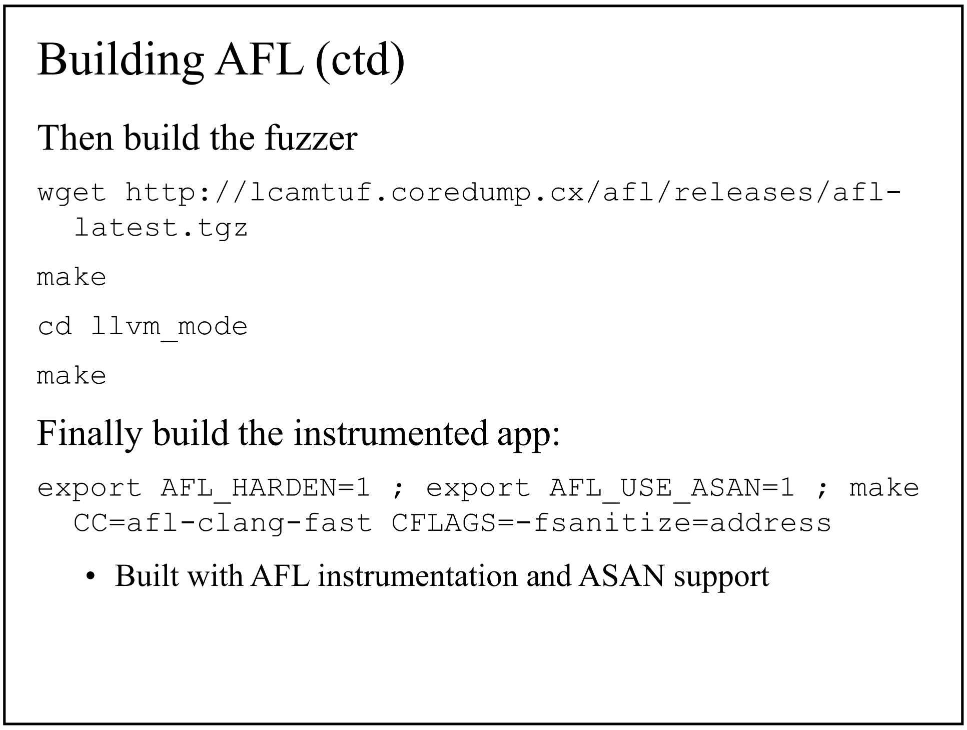
```
svn co https://llvm.org/svn/llvm-project/cfe/trunk
  LLVM/tools/clang
```

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

Build all the LLVM stuff (compiler + additional tools needed for ASAN):

```
cmake --build .
```

- Don’t use whatever old pre-built version may be present in some repository, build it from scratch and get it right



Building AFL (ctd)

Then build the fuzzer

```
wget http://lcamtuf.coredump.cx/afl/releases/afl-
  latest.tgz
```

```
make
```

```
cd llvm_mode
```

```
make
```

Finally build the instrumented app:

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

- Built with AFL instrumentation and ASAN support

Running AFL

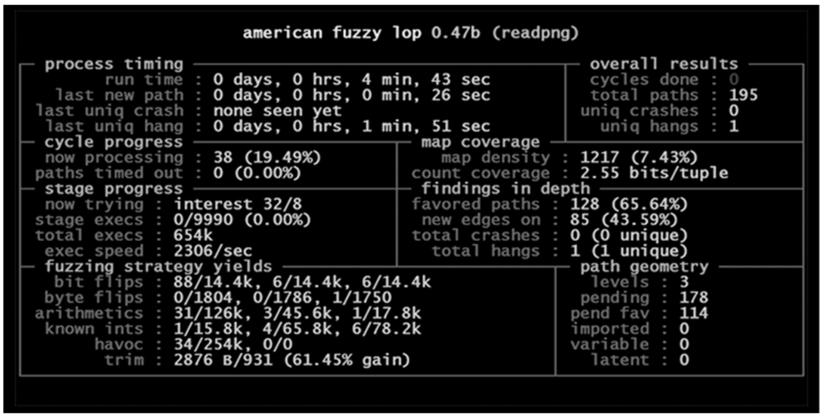
Run your code under afl-fuzz, telling it where to find input files and where to put output files

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

- ‘@@’ is a placeholder arg for AFL’s fuzzed file

Running AFL (ctd)

Results shown on annoying screen-hogging status page



```
                  american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec        │  total paths : 195    │
│ last uniq crash : none seen yet                       │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec        │   uniq hangs : 1      │
├─ cycle progress ──────────────────┬─ map coverage ────┴───────────────────────┤
│  now processing : 38 (19.49%)     │    map density : 1217 (7.43%)            │
│ paths timed out : 0 (0.00%)       │ count coverage : 2.55 bits/tuple         │
├─ stage progress ──────────────────┼─ findings in depth ──────────────────────┤
│  now trying : interest 32/8       │ favored paths : 128 (65.64%)             │
│ stage execs : 0/9990 (0.00%)      │  new edges on : 85 (43.59%)              │
│ total execs : 654k                │ total crashes : 0 (0 unique)             │
│  exec speed : 2306/sec            │   total hangs : 1 (1 unique)             │
├─ fuzzing strategy yields ─────────┴─────────────┬─ path geometry ────────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k        │    levels : 3              │
│  byte flips : 0/1804, 0/1786, 1/1750            │   pending : 178            │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k         │  pend fav : 114            │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k         │  imported : 0              │
│       havoc : 34/254k, 0/0                      │  variable : 0              │
│        trim : 2876 B/931 (61.45% gain)          │    latent : 0              │
└─────────────────────────────────────────────────┴────────────────────────────┘
```

- Prevents running more than one copy on multi-core systems
- Use nohup to deal with it

Running AFL (ctd)

The useful stats…

- Execs per second: How fast are you going?
- Total execs: How far have you got?
- Unique hangs and crashes: Self-explanatory
  - Hangs aren't terribly useful, mostly false positives due to timing glitches (page faults, I/O, etc)
- Cycles done: Number of full sets of mutations exercised

Data from cycle *n* is fed into cycle *n*+1

- Cycles can take from days to weeks to complete

Optimisations

Many apps have a high startup overhead

AFL uses a fork server to preload the app, but this still triggers the startup overhead

- Defer the forking until the startup has completed

In your code, manually start the fork server after the startup has completed

```
__afl_manual_init();
```

Tell AFL that you're using deferred init

```
export AFL_DEFER_FORKSRV=1
```

Notes on Use

You’re going to make yourself unpopular when you run this…

- 100% CPU per afl task
  - Maximise the loa^H^H^H^Hutilisation with one task per `getconf _NPROCESSORS_ONLN`
- 20 terabytes of VM for ASAN on x64
  - Yes, that’s 20,000,000 megabytes
  - Uses it as shadow memory to detect out-of-bounds accesses, see “AddressSanitizer: A Fast Address Sanity Checker”, Usenix Annual Technical Conference
  - Tested on someone else’s Linux box, it works fine even in a VM
- Weeks or months of runtime

Conclusion