March 15, 2016March 15, 2016

Fuzzing workflows; a fuzz job from start to finish

By @BrandonPrry (https://twitter.com/BrandonPrry)

Many people have garnered an interest in fuzzing in the recent years, with easy-to-use frameworks like American Fuzzy Lop showing incredible promise and (relatively) low barrier to entry. Many websites on the internet give brief introductions to specific features of AFL, how to *start* fuzzing a given piece of software, but never what to do when you decide to *stop* fuzzing (or how you decide in the first place?).

In this post, we'd like to go over a fuzz job from start to finish. What does this mean exactly? First, even finding a good piece of software to fuzz might seem daunting, but there is certain criteria that you can follow that can help you decide what would be useful and easy to get started with on fuzzing. Once we have the software, what's the *best* way to fuzz it? What about which testcases we should use to seed with? How do we know how well we are doing or what code paths we might be missing in the target software?

We hope to cover all of this to give a fully-colored, 360 view of how to effectively and efficiently go through a full fuzz job process from start to finish. For ease of use, we will focus on the AFL framework.

What should I fuzz? Finding the right software

AFL works best on C or C++ applications, so immediately this is a piece of criteria we should be looking for in software we would like to fuzz. There are a few questions we can ask ourselves when looking for software to fuzz.

1. *Is there example code readily available?*
   - Chances are, any utilities shipped with the project are too heavy-weight and can be trimmed down for fuzzing purposes. If a project has bare-bones example code, this makes our lives as fuzzers much easier.
2. *Can I compile it myself? (Is the build system sane?)*
   - AFL works best when you are able to build the software from source. It does support instrumenting black-box binaries on the fly with QEMU, but this is out of scope and tends to have poor performance. In my ideal scenario, I can easily build the software with afl-clang-fast or afl-clang-fast++.
3. *Are there easily available and unique testcases available?*
   - We are probably going to be fuzzing a file format (although with some tuning, we can fuzz networked applications), and having some testcases to seed with that are unique and interesting will give us a good start. If the project has unit tests with test cases of files (or keeps files with previously known bugs for regression testing), this is a huge win as well.

These basic questions will help save a lot of time and headaches later if you are just starting out.

The yaml-cpp project

Ok, but how do you *find* the software to ask these questions about? One favorite place is Github, as you can easily search for projects that have been recently updated and are written in C or C++. For instance, searching Github for all C++ projects with more than 200 stars led us to a project that shows a lot of promise: yaml-cpp (https://github.com/jbeder/yaml-cpp (https://github.com/jbeder/yaml-cpp)). Let's take a look at it with our three questions and see how easily we can get this fuzzing.

1. Can I compile it myself?
   - yaml-cpp uses cmake as its build system. This looks great as we can define which compilers we want to use, and there is a good chance afl-clang-fast++ will Just Work™. One interesting note in the README of yaml-cpp is that it builds a static library by default, which is perfect for us, as we want to give AFL a statically compiled and instrumented binary to fuzz.
2. Is there example code readily available?
   - In the util folder in the root of the project (https://github.com/jbeder/yaml-cpp/tree/master/util (https://github.com/jbeder/yaml-cpp/tree/master/util)), there are a few small cpp files, which are bare-bones utilities demonstrating certain features of the yaml-cpp library. Of particular interest is the parse.cpp file. This parse.cpp file is perfect as it is already written to accept data from stdin and we can easily adapt it to use AFL's persistent mode, which will give us a significant speed increase.
3. Are there easily available and unique/interesting testcases available?
   - In the test folder in the root of the project is a file called specexamples.h, which has a very good number of unique and interesting YAML testcases, each of which seems to be exercising a specific piece of code in the yaml-cpp library. Again, this is perfect for us as fuzzers to seed with.

This looks like it will be easy to get started with. Let's do it.

Starting the fuzz job

We are not going to cover installing or setting up AFL, as we will assume that has already been done. We are also assuming that afl-clang-fast and afl-clang-fast++ have been built and installed as well. While afl-g++ should work without issues (though you won't get to use the awesome persistent mode), afl-clang-fast++ is certainly preferred. Let's grab the yaml-cpp codebase and build it with AFL.

```
# git clone https://github.com/jbeder/yaml-cpp.git
 (https://github.com/jbeder/yaml-cpp.git)# cd yaml-cpp
# mkdir build
# cd build
# cmake -DCMAKE_CXX_COMPILER=afl-clang-fast++ ..
# make
```

Once we know that everything builds successfully, we can make a few changes to some of the source code so that AFL can get a bit more speed. From the root of the project, in /util/parse.cpp, we can update the main() function using an AFL trick for persistent mode.

```
int main(int argc, char** argv) {
  Params p = ParseArgs(argc, argv);

  if (argc > 1) {
    std::ifstream fin;
    fin.open(argv[1]);
    parse(fin);
  } else {
    parse(std::cin);
  }

  return 0;
}
```

With this simple main() method, we can update the else clause of the if statement to include a while loop and a special AFL function called __AFL_LOOP(), which allows AFL to basically perform the fuzzing of the binary *in process* through some memory wizardry, as opposed to starting up a new process for every new testcase we want to test. Let's see what that would look like.

```
if (argc > 1) {
  std::ifstream fin;
  fin.open(argv[1]);
  parse(fin);
} else {
  while (__AFL_LOOP(1000)) {
    parse(std::cin);
  }
}
```

Note the new while loop in the else clause, where we pass 1000 to the __AFL_LOOP() function. This tells AFL to fuzz up to 1000 testcases in process before spinning up a new process to do the same. By specifying a larger or smaller number, you may increase the number of executions at the expense of memory usage (or being at the mercy of memory leaks), and this can be highly tunable based on the application you are fuzzing. Adding this type of code to enable persistent mode also is not always this easy. Some applications may not have an architecture that supports easily adding a while loop due to resources spawned during start up or other factors.

Let's recompile now. Change back to the build directory in the yaml-cpp root, and type 'make' to re-build parse.cpp.

Testing the binary

With the binary compiled, we can test it using a tool shipped with AFL called afl-showmap. The afl-showmap tool will run a given instrumented binary (passing any input received via stdin to the instrumented binary via stdin) and print a report of the feedback it sees during program execution.

```
# afl-showmap -o /dev/null -- ~/parse < <(echo hi)
afl-showmap 2.03b by <lcamtuf@google.com>
[*] Executing '~/parse'...

-- Program output begins --
hi
-- Program output ends --
[+] Captured 1787 tuples in '/dev/null'.
#
```

By changing the input to something that should exercise new code paths, you should see the number of tuples reported at the end of the report grow or shrink.

```
# afl-showmap -o /dev/null -- ~/parse < <(echo hi: blah)
afl-showmap 2.03b by <lcamtuf@google.com>
[*] Executing '~/parse'...

-- Program output begins --
hi: blah
-- Program output ends --
[+] Captured 2268 tuples in '/dev/null'.
#
```

As you can see, sending a simple YAML key (hi) expressed only 1787 tuples of feedback, but a YAML key with a value (hi: blah) expressed 2268 tuples of feedback. We should be good to go with the instrumented binary, now we need the testcases to seed our fuzzing with.

Seeding with high quality test cases

The testcases you initially seed your fuzzers with is one of, if not *the*, most significant aspect of whether you will see a fuzz run come up with some good crashes or not. As stated previously, the specexamples.h file in the yaml-cpp test directory has excellent test cases for us to start with, but they can be even better. For this job, I manually copied and pasted the examples from the header file into testcases to use, so to save the reader time, linked here (https://github.com/bperryntt/yaml-fuzz/tree/master/raw_testcases) are the original seed files I used, for reproduction purposes.

AFL ships with two tools we can used to ensure that:

- The files in the test corpus are as efficiently unique as possible
- Each test file expresses its unique code paths as efficiently as possible

The two tools, afl-cmin and afl-tmin, perform what is called *minimizing*. Without being too technical (this is a technical blog, right?), afl-cmin takes a given folder of potential test cases, then runs each one and compares the feedback it receives to all rest of the testcases to find the best testcases which most efficiently express the most unique code paths. The best testcases are saved to a new directory.

The afl-tmin tool, on the other hand, works on only a specified file. When we are fuzzing, we don't want to waste CPU cycles fiddling with bits and bytes that are useless relative to the code paths the testcase might express. In order to minimize each testcase to the bare minimum required to express the same code paths as the original testcase, afl-tmin iterates over the actual bytes in the testcases, removing progressively smaller and smaller chunks of data until it has a removed any bytes that don't affect the code paths taken. It's a bit much, but these are very important steps to efficiently fuzzing and they are important concepts to understand. Let's see an example.

In the git repo I created with the raw testcases (https://github.com/bperryntt/yaml-fuzz/tree/master/raw_testcases) from the specexamples.h file, we can start with the 2 file (https://github.com/bperryntt/yaml-fuzz/blob/master/raw_testcases/2).

```
# afl-tmin -i 2 -o 2.min -- ~/parse
afl-tmin 2.03b by <lcamtuf@google.com>

[+] Read 80 bytes from '2'.
[*] Performing dry run (mem limit = 50 MB, timeout = 1000 ms)...
[+] Program terminates normally, minimizing in instrumented mode.
[*] Stage #0: One-time block normalization...
[+] Block normalization complete, 36 bytes replaced.
[*] --- Pass #1 ---
[*] Stage #1: Removing blocks of data...
Block length = 8, remaining size = 80
Block length = 4, remaining size = 80
Block length = 2, remaining size = 76
Block length = 1, remaining size = 76
[+] Block removal complete, 6 bytes deleted.
[*] Stage #2: Minimizing symbols (22 code points)...
[+] Symbol minimization finished, 17 symbols (21 bytes) replaced.
[*] Stage #3: Character minimization...
[+] Character minimization done, 2 bytes replaced.
[*] --- Pass #2 ---
[*] Stage #1: Removing blocks of data...
Block length = 4, remaining size = 74
Block length = 2, remaining size = 74
Block length = 1, remaining size = 74
[+] Block removal complete, 0 bytes deleted.

File size reduced by : 7.50% (to 74 bytes)
Characters simplified : 79.73%
Number of execs done : 221
Fruitless execs : path=189 crash=0 hang=0

[*] Writing output to '2.min'...
[+] We're done here. Have a nice day!

# cat 2
hr: 65 # Home runs
avg: 0.278 # Batting average
rbi: 147 # Runs Batted In
# cat 2.min
00: 00 #00000
000: 00000 #0000000000000000
000: 000 #000000000000000
#
```

This is a great example of how powerful AFL is. AFL has no idea what YAML is or what its syntax looks like, but it effectively was able to zero out all the characters that weren't special YAML characters used to denote key value pairs. It was able to do this by determining that changing those specific

characters would alter the feedback from the instrumented binary dramatically, and they should be left alone. It also removed four bytes from the original file that didn't affect the code paths taken, so that is four less bytes we will be wasting CPU cycles on.

In order to quickly minimize a starting test corpus, I usually use a quick for loop to minimize each one to a new file with a special file extension of .min.

```
# for i in *; do afl-tmin -i $i -o $i.min -- ~/parse; done;
# mkdir ~/testcases && cp *.min ~/testcases
```

This for loop will iterate over each file in the current directory, and minimize it with afl-tmin to a new file with the same name as the first, just with a .min appended to it. This way, I can just cp *.min to the folder I will use to seed AFL with.

Starting the fuzzers

This is the section where most of the fuzzing walkthroughs end, but I assure you, this is only the beginning! Now that we have a high quality set of testcases to seed AFL with, we can get started. Optionally, we could also take advantage of the dictionary token functionality to seed AFL with the YAML special characters to add a bit more potency, but I will leave that as an exercise to the reader.

AFL has two types of fuzzing strategies, one that is deterministic and one that is random and chaotic. When starting afl-fuzz instances, you can specify which type of strategy you would like that fuzz instance to follow. Generally speaking, you only need one deterministic (or master) fuzzer, but you can have as many random (or slave) fuzzers as your box can handle. If you have used AFL in the past and don't know what this is talking about, you may have only run a single instance of afl-fuzz before. If no fuzzing strategy is specified, then the afl-fuzz instance will switch back and forth between each strategy.

```
# screen afl-fuzz -i testcases/ -o syncdir/ -M fuzzer1 -- ./parse
# screen afl-fuzz -i testcases/ -o syncdir/ -S fuzzer2 -- ./parse
```

First, notice how we start each instance in a screen session. This allows us to connect and disconnect to a screen session running the fuzzer, so we don't accidentally close the terminal running the afl-fuzz instance! Also note the arguments -M and -S used in each respective command. By passing -M fuzzer1 to afl-fuzz, I am telling it to be a Master fuzzer (use the deterministic strategy) and the name of the fuzz instance is fuzzer1. On the other hand, the -S fuzzer2 passed to the second command says

to run the instance with a random, chaotic strategy and with a name of fuzzer2. Both of these fuzzers will work with each other, passing new test cases back and forth to each other as new code paths are found.

When to stop and prune

Once the fuzzers have run for a relatively extended period of time (I like to wait until the Master fuzzer has completed it's first cycle at the very least, the Slave instances have usually completed many cycles by then), we shouldn't just stop the job and start looking at the crashes. During fuzzing, AFL has hopefully created a huge corpus of new testcases that could still have bugs lurking in them. Instead of stopping and calling it a day, we should minimize this new corpus as much as possible, then reseed our fuzzers and let them run *even more*. This is the process that no walkthroughs talk about because it is boring, tedious, and can take a long time, but it is crucial to highly-effective fuzzing. Patience and hard work are virtues.

Once the Master fuzzer for the yaml-cpp parse binary has completed it's first cycle (it took about 10 hours for me, it might take 24 for an average workstation), we can go ahead and stop our afl-fuzz instances. We need to consolidate and minimize each instance's queues and restart the fuzzing again. While running with multiple fuzzing instances, AFL will maintain a separate sync directory for each fuzzer inside of the root syncdir your specify as the argument to afl-fuzz. Each individual fuzzer syncdir contains a queue directory with all of the test cases that AFL was able to generate that lead to new code paths worth checking out.

We need to consolidate each fuzz instance's queue directory, as there will be a lot of overlap, then minimize this new body of test data.

```
# cd ~/syncdir
# ls
fuzzer1 fuzzer2
# mkdir queue_all
# cp fuzzer*/queue/* queue_all/
# afl-cmin -i queue_all/ -o queue_cmin -- ~/parse
corpus minimization tool for afl-fuzz by <lcamtuf@google.com>

[*] Testing the target binary...
[+] OK, 884 tuples recorded.
[*] Obtaining traces for input files in 'queue_all/'...
Processing file 1159/1159...
[*] Sorting trace sets (this may take a while)...
[+] Found 34373 unique tuples across 1159 files.
[*] Finding best candidates for each tuple...
Processing file 1159/1159...
[*] Sorting candidate list (be patient)...
[*] Processing candidates and writing output files...
Processing tuple 34373/34373...
[+] Narrowed down to 859 files, saved in 'queue_cmin'.
```

Once we have run the generated queues through afl-cmin, we need to minimize each resulting file so that we don't waste CPU cycles on bytes we don't need. However, we have quite a few more files now than when we were just minimizing our starting testcases. A simple for loop for minimizing thousands of files could potentially take days and ain't no one got time for that. Over time, I wrote a small bash script called afl-ptmin which parallelizes afl-tmin into a set number of processes and has proven to be a significant speed boost in minimizing.

```
#!/bin/bash

cores=$1
inputdir=$2
outputdir=$3
pids=""
total=`ls $inputdir | wc -l`

for k in `seq 1 $cores $total`
do
  for i in `seq 0 $(expr $cores - 1)`
  do
    file=`ls -Sr $inputdir | sed $(expr $i + $k)"q;d"`
    echo $file
    afl-tmin -i $inputdir/$file -o $outputdir/$file -- ~/parse &
  done

  wait
done
```

As with the afl-fuzz instances, I recommend still running this in a screen session so that no network hiccups or closed terminals cause you pain and anguish. It's usage is simple, taking only three arguments, the number processes to start, the directory with the testcases to minimize, and the output directory to write the minimized test cases to.

```
# screen ~/afl-ptmin 8 ./queue_cmin/ ./queue/
```

Even with parallelization, this process can still take a while (24 hours+). For our corpus generated with yaml-cpp, it should be able to finish in an hour or so. Once done, we should remove the previous queue directories from the individual fuzzer syncdirs, then copy the queue/ folder to replace the old queue folder.

```
# rm -rf fuzzer1/queue
# rm -rf fuzzer2/queue
# cp -r queue/ fuzzer1/queue
# cp -r queue/ fuzzer2/queue
```

With the new minimized queues in place, we can begin fuzzing back where we left off.

```
# cd ~
# screen afl-fuzz -i- -o syncdir/ -S fuzzer2 -- ./parse
# screen afl-fuzz -i- -o syncdir/ -M fuzzer1 -- ./parse
```

If you notice, instead of passing the -i argument a directory to read testcases from each time we call afl-fuzz, we simply pass a hyphen. This tells AFL to just use the queue/ directory in the syncdir for that fuzzer as the seed directory and start back up from there.

This entire process starting the fuzz jobs, then stopping to minimize the queues and restarting the jobs can be done as many times as you feel it necessary (usually until you get bored or just stop finding new code paths). It should also be done often because otherwise you are wasting your electricity bill on bytes that aren't going to pay you anything back later.

Triaging your crashes

Another traditionally tedious part of the fuzzing lifecycle has been triaging your findings. Luckily, some great tools have been written to help us with this.

A great tool is crashwalk (https://github.com/bnagy/crashwalk), by @rantyben (https://twitter.com/rantyben) (props!). It automates gdb and a special gdb plugin to quickly determine which crashes may lead to exploitable conditions or not. This isn't fool proof by any means, but does give you a bit of a head start in which crashes to focus on first. Installing it is relatively straight-forward, but we need a few dependencies first.

```
# apt-get install gdb golang
# mkdir src
# cd src
# git clone https://github.com/jfoote/exploitable.git
 (https://github.com/jfoote/exploitable.git)# cd && mkdir go
# export GOPATH=~/go
# go get -u github.com/bnagy/crashwalk/cmd/…
```

With crashwalk installed in ~/go/bin/, we can automatically analyze the files and see if they might lead to exploitable bugs.

```
# ~/go/bin/cwtriage -root syncdir/fuzzer1/crashes/ -match id -- ~/parse
@@
```

Determining your effectiveness and code coverage

Finding crashes is great fun and all, but without being able to quantify how well you are exercising the available code paths in the binary, you are doing nothing but taking shots in the dark and hoping for a good result. By determining which parts of the code base you aren't reaching, you can better tune your testcase seeds to hit the code you haven't been able to yet.

An excellent tool (developed by @michaelrash (https://twitter.com/michaelrash)) called afl-cov (https://github.com/mrash/afl-cov) can help you solve this exact problem by watching your fuzz directories as you find new paths and immediately running the testcase to find any new coverage of the codebase you may have hit. It accomplishes this using lcov, so we must actually recompile our parse binary with some special options before continuing.

```
# cd ~/yaml-cpp/build/
# rm -rf ./*
# cmake -DCMAKE_CXX_FLAGS="-O0 -fprofile-arcs -ftest-coverage" \
-DCMAKE_EXE_LINKER_FLAGS="-fprofile-arcs -ftest-coverage" ..
# make
# cp util/parse ~/parse_cov
```

With this new parse binary, afl-cov can link what code paths are taken in the binary with a given input with the code base on the file system.

```
# screen afl-cov/afl-cov -d ~/syncdir/ --live --coverage-cmd
"~/parse_cov AFL_FILE" --code-dir ~/yaml-cpp/
```

Once finished, afl-cov generates report information in the root syncdir in a directory called cov. This includes HTML files that are easily viewed in a web browser detailing which functions and lines of code were hit, as well as with functions and lines of code were not hit.

In the end

In the three days it took to flesh this out, I found no potentially exploitable bugs in yaml-cpp. Does that mean that no bugs exist and it's not worth fuzzing? Of course not. In our industry, I don't believe we publish enough about our failures in bug hunting. Many people may not want to admit that they put a considerable amount of effort and time into something that came up as what others might consider fruitless. In the spirit of openness, linked below are all of the generated corpus (fully minimized), seeds, and code coverage results (~70% code coverage) so that someone else can take them and determine whether or not it's worthy to pursue the fuzzing.

https://github.com/bperryntt/yaml-fuzz (https://github.com/bperryntt/yaml-fuzz)