Exploring AFL and Reproducing Bugs in the Perl Compiler

Jason Driver
Computer Science
University of California, Davis
Davis, USA
jydriver@ucdavis.edu

Abstract

Software testing and program verification are important tools in checking that code correctly executes in the way that it was intended. Software testing has become increasingly important as more software is being written and deployed inside embedded devices in addition to being important for mission critical applications on the Internet. Automating software bug detection through software testing tools has been effective at proving bugs exist within a program and can guide developers in software revisions. This paper will address brute force and guided software testing using American Fuzzy Lop. The importance of this is to discover previously unknown bugs without user interaction or manually writing unit tests. The primary software that will be test include the Perl compiler, the python compiler, and an image compression program called Guetzli.

1. Introduction

Software is eating the world, Marc Andreessen said in 2011. The coming of the Internet has drastically changed the way humans communicate, do business and work. Even older technologies such as radio and television, which were once considered the domain of electrical engineering analog has given way to digital radio and digital television channels. A new push into bringing intelligence to the edge of the network, in the embedded systems space, would drive in a new wave of computer science research. With seemingly every electronic device being controlled by microcontrollers, the importance of software will only grow. With more devices becoming intelligent, software will only grow in importance. When it comes to building large software systems, there also comes with it engineering mistakes, and finding these mistakes quickly so that the software can be revised is a crucial tool for the advance of the software industry.

There are many choices for software testing that could lead to bug discovery and subsequent bug fixes. A few include building numerous test cases for a test suite, deploying the software and having users report bugs, fuzzing, or combinations of these together. Vulnerabilities like the stack overflow vulnerability have been known for a long time, but still are one of the main causes behind software bugs because of the pervasive nature of C and C++ programing. With the advent of the embedded systems space, C and C++ will be very relevant for the foreseeable future, making these types of bugs hard to overlook. Historically, images and pdf files could be corrupted, and when a viewer opened them the underlying bug in the viewer might cause a vulnerability resulting in, for instance, a shell being opened and a host computer rooted. This could lead to financial damage for the computer owners, and such bugs should be actively looked for and reported.

Unit tests can be used in bug discovery before and after software is deployed. Automated testing used in jenkins, travis CI, and the cloud can be applied to programs to determine if bugs are present. One technique to increasing test coverage is through randomly generating input around a target test case and watching for the program to crash. This is generally referred to as software fuzzing, and one example of a fuzzer is the American Fuzzy Lop [4].

American Fuzzy Lop

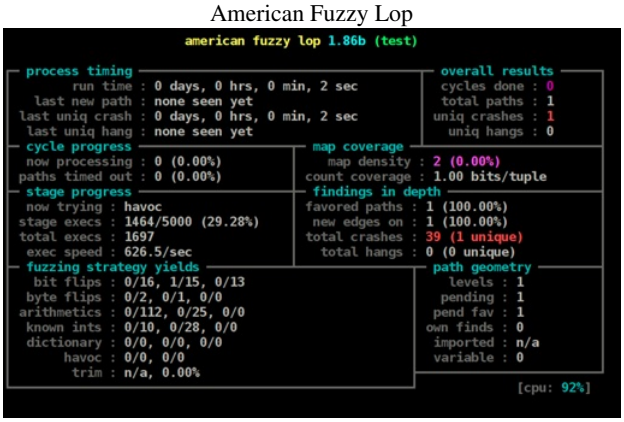

Figure 1. The picture depicts the UI for American Fuzzy Lop.

Fuzzing is a technique where many different, and exotic inputs are applied to a program's input in search of a program crash or other unintended behavior. The crash might then be traced to a bug, or possibly a vulnerability in the program. There are a few different types of fuzzing, some are brute force methods and others are more directed methods, starting with the test cases and mutating them from there. As with any software testing, there is a computational limit to the amount of random input a tester can feed a target program before it has to be deployed. Many programs have been fuzzed extensively, but this does not mean that all the bugs have been discovered through fuzzing. New programs are being released that have not been extensible fuzzed yet due to their newness. These programs, open source as well as commercial programs, always welcome the help of others to report bugs to help make their software better.

2. Related Work

Different types of fuzzing tools exist for random fuzzing, and guided fuzzing like AFL for linux, libFuzzer, and MiniFuzz for windows. Recently, Microsoft has released a cloud based fuzzing service called Project Springfield and Google has released their own cloud based fuzzing service called OSS-Fuzz. I found AFL most interesting for its out of the box ease of use, good documentation, and ability to perform more complex fuzzing techniques like instrumented fuzzing, use of a strong dictionary and guided fuzzing. There are a variety of different projects which have used AFL in pursuing exploitable bugs in software such as fuzzing Tar, wireshark, Putty, and Skype.

Additional work related to using AFL can be found on the AFL [4] website here at http://lcamtuf.coredump.cx/afl/ under the trophy case section. An example of a bug report found using AFL, applied to the Perl compiler, can be found here https://rt.perl.org/Public/Bug/Display.html?id=123539 [2]. This example is useful in guiding what information has to be discovered to make a useful bug report.

3. Approach

The approach taken is fairly straightforward. Setup linux as the test environment. Download the version of software where the bug was found in the trophy case. Compile the program using the afl compiler, afl-gcc and afl-g++. This will create an instrumented version of the compiled code which adds instructions to the Perl source code itself to detect code paths. This is useful for getting full coverage of all of the paths through the Perl compiler. Next, setup the AFL to work with the target program, and create test cases to reproduce the bug. The Perl bug chosen is caused by parsing of regular expression leading to a stack overflow. The initial test cases to attempt should then be just different regular expressions. It should also be noted that same bug can be found through a non-regular expression input, but this would then depend on the power of the computer used as well as the time the tester has. Parallel fuzzing are used to fully utilize system resources and to share interesting generated test cases across fuzzing threads.

The second approach used was to look at the top trending C++ projects on github. The top projects the author selected were the Python Programming Language [2] and Project Guetzli [3] by Google. Python, and specifically the cPython implementation of python, was targeted as AFL works best on C and C++ libraries. CPython is also the core development branch for python. Additional reasons to choose python include that it has a large programming following, an extensive code base, and bug bounty prizes. In addition, many websites are now running on python and many universities have switched to python to teach introductory computer science courses. Project Guetzli is a photo compression algorithm written by Google that has a better compression ratio than jpeg, and is based on the human sensory system. This program was chosen because of its relative newness, less than 4 months old since its release on github at the time of writing, as well as continued development of the project, and it is a C implementation.

4. Implementation and Results

4.1. Environment

An *unix environment for all experiments, Ubuntu 16.04, is used. Installed programs include AFL v1.96b, Perl 5.20.0, Perl 5.24.1 at the time of this writing is the newest release of the Perl compiler, as well as all related dependencies. In addition, cPython 3.6 compiled from source and guetzli compiled from source were installed.

4.2. Perl 5.20.0 Regex Bug

Navigate to the Perl Folder in Terminal. Mine is ∼/tmp/perl-5.20.0. This first run is without instrumentation mode set.

```
make test
make install
```

Create an example test case named test.pl, and put into ∼/perlTest. Contents of test.pl shown below

Initial test case, test.pl

```
$~='`';$_=$:=$~|'%';$;=$^='/'|$~;$;++;$\=$~|'"";$;++;$:.=++$;;$/=++$;;+$\++;$_.=
'#'|$~;$,=++$/;$_.="$\$^$\"";++$,;$_.='@'|'*'&~'!';$_.="$,$;$/$\"";$_.+='!.'|$~.
$~;$_.="$^$/$\$:$\"";$_='@'|':'&~'*';$_.=$:;$_.=$^&'|';$_.=$".$\;$_.=+"$~$~$~"|
'!#+';++$.;$.++;`$_$:;>&$.`;
```

Figure 2. Initial test case for Perl 5.20.0 Fuzzing.

The following is how to initiate the fuzzing process. Point the afl-fuzz section of the command to the compiled binary for afl-fuzz, or add it to your path like this example.

The flag `-n` indicates that non-instrumented dumb mode will be used. `-t50` indicates that the operation will time out after 50ms. `-m none` indicates that there will be no memory limit for each process. Also note that this is executed from within the ∼/tmp/perl-5.20.0 folder containing my compiled Perl program.

```
afl-fuzz -n -t50+ -m none -i ∼/perlTest
-o ∼/perlTest/perlOutput/ ./perl
```

After 2 hours, AFL finds 5 unique crashes. Enter command afl-plot to generate the following plots.

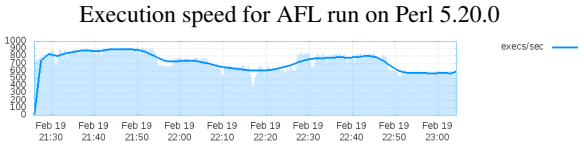

Figure 3. 2 hours of fuzzing.

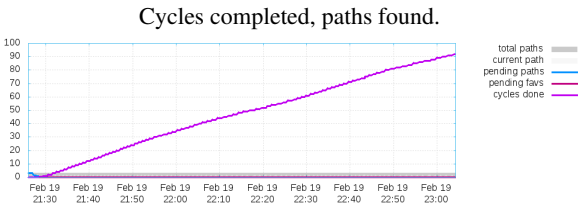

Figure 4. Shows metrics related to AFL coverage of execution paths through the Perl 5.20.0 compiler.

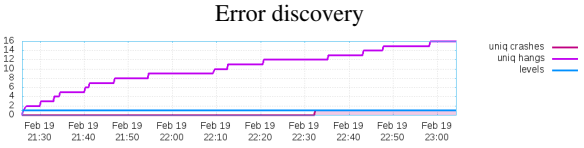

Figure 5. Shows metrics related to AFL errors discovered in thee Perl 5.20.0 compiler.

The following commands were issued to confirm the Perl compiler indeed had a segmentation fault when the target test case was input. Put the generated test case directly into the Perl compiler.

```
jydriver@jydriver-ThinkPad-T430:∼/tmp/perl
-5.20.0$ ./perl ∼/perlTest/perlOutput
crashes/id_000000,sig_11,src_000001,op
_havoc,rep_32
Segmentation fault
```

Now lets use valgrind to try to find the error.

```
jydriver@jydriver-ThinkPad-T430:∼/tmp/perl
-5.20.0$ valgrind --leak-check=full --show
-leak-kinds=all --track-origins=yes ./perl
∼/perlTest/perlOutput/crashes/id_000000
,sig_11,src_000001,op_havoc,rep_32
```

Full output long, important sections were:

```
==18892==    at 0x445D61: Perl_lex_read
_space (in /home/jydriver/tmp/perl-5.20.
0/perl)
==18892==    by 0x445EC6: S_skipspace_fl
ags (in /home/jydriver/tmp/perl-5.20.0/
perl)
==18892==    by 0x449DFD: S_scan_pat (in
/home/jydriver/tmp/perl-5.20.0/perl)
==18892==    by 0x45C425: Perl_yylex (in
/home/jydriver/tmp/perl-5.20.0/perl)
==18892==    by 0x45DE94: Perl_yyparse (
in /home/jydriver/tmp/perl-5.20.0/perl)
==18892==    by 0x43B027: perl_parse (in
/home/jydriver/tmp/perl-5.20.0/perl)
==18892==    by 0x41DD0F: main (in /home
/jydriver/tmp/perl-5.20.0/perl)
```

This resulted in a segmentation fault causing the Perl compiler to crash. Using valgrind, the author was able to trace the segmentation fault from the main Perl compile file, to `Perl_yyparse`, to a function called `Perl_lex_read_space`. This is consistent with the bug report assessment of the bug. The root cause, is when the Perl compiler allocates memory for a regular expression there is a three pass system used, where the first pass allocates memory and subsequent passes perform the regular expression function. There was a problem with the way this system was implemented, resulting in a stack overflow of the allocated memory on subsequent passes after the memory was allocated.

4.3. Perl 5.24.1 Fuzzing

Navigate to the Perl v5.24.1 folder in the terminal. Mine is `∼/Desktop/CS240/PerlCurrent`. Then specify the path to the AFL C and C++ compilers and save the the configuration file.

```
CC=/usr/bin/afl/afl-gcc
CXX=/usr/bin/afl/afl-g++ ./Configure
make test
make install
```

During this process, you should see output like this

```
Afl-as 196b by lcamtuf@google.com
[+] Instrumented 416 locations
```

The initiate the fuzzing process. Point the afl-fuzz section to the compiled binary for afl-fuzz, or add it to your path.

```
afl-fuzz -t50+ -m none -i /home/
jydriver/perlFuzz -o /home/
jydriver/perlFuzz/perlOutput
```

```
./perl @@
```

24 hour AFL run on Perl 5.24.1

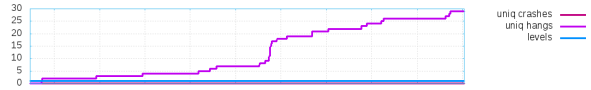

Figure 6. No unique crashes found in Perl 5.24.1 after fuzzing for 24 hours.

Many of the bugs in the AFL trophy case were actually targeting this particular vulnerability in the regular expression parser. It looks like the new implementation of the parser fixed this problem, but only 24 hours of fuzzing directed at regular expressions was applied and further investigation is needed. Somewhere around the 12 hour mark resulted in more unique hangs being generated.

4.4. cPython 3.6 Fuzzing

CPython was downloaded from the corresponding github account [1] and compiled from binary sources. The test case for this run is the following:

```
class TestObject(object):
    def tester(self)
        for j in range(100):
            z += j +5
            yield z
TestO = TestObject()
```

24 hour AFL run on cPython.

Figure 7. 24 hours of fuzzing gives many unique crashes.

Only 1 million executions and hundreds of unique crashes were discovered. Many of the unique crashes are due to the same code area, and this has already been reported to the cPython development team. The marshal module for .pyc files is the problem, but the python team acknowledges this and gives a warning that the marshal module is not intended to be secure against erroneous or malicious constructed data. The author did not looked at all of the crash reports, there are a lot of ”unique crashes” to sift through so there is the chance that a unique bug does exist. Another strategy would be disabling the marshal module when building cPython to stop this type of crash from being reported.

4.5. Guetzli Fuzzing

Guetzli is downloaded from the corresponding github account [3] and compiled from binary sources. The test cases for this run are the .png and .jpg test examples from the AFL website [4]. These image files are known to cause bugs in previous image handling software and were used as the starting places for fuzzing. The following command was run:

```
afl-fuzz -n -t50+ -m none -i ~/Desktop/
CS240/guetzli/test/images -o ~/Desktop
/CS240/guetzli/test/Output ./guetzli @@
```

24 hour AFL run on Guetzli.

Figure 8. 24 hours of fuzzing gives no unique crashes.

It should be noted that the total paths found and the path geometry numbers quickly maxed out. This might suggest that the algorithm for image compression only has a small number of possible paths. After 50.5 million executions and no unique crashes, it might be possible to assume Guetzli has already been fuzzed using the known corrupt jpg and png images and already revised. This is a Google product, so that may not be surprising.

5. Conclusion

This project demonstrates the success of reproducing a regular expression parsing bug in the Perl 5.20.0 compiler. The cause of which is a stack over flow error in the implementation of the parser. More than a few of the Perl trophies actually came from this problem are in the code for regular expression parser. Fuzzing the newest Perl compiler v5.24.1 for 24 hours utilizing the same starting input, and a couple other regular expression test cases, did not produce any unique crashes. The bug fix seemed to have effectively patched the regular expression parsing issue. Further more, the author fuzzed the C implementation of Python 3.6 and discovered many hundreds of unique crashes. This is slightly misleading as this is a known problem with the

cPython marshal module. That does not mean that there is not a unique crash that was generated by AFL in the hundreds of unique crashes though. Using github metrics to find trending C and C++ programs to target for fuzzing is not a great way to find programs that have not been extensively fuzzed. Despite Guetzli being less than 4 months since being released on github, the known corrupt png and jpg files were not able to produce a unique crash. In conclusion, fuzzing is an indispensable tool for any software engineer developing code, as it provides a fast, easy, but computationally demanding work flow for software testing. This can be partly fixed through guiding the process by giving the fuzzer a good place to begin through a well guided initial test case. The next steps in this research area seems to be coupling different methods from programming languages research like symbol analysis to guide the fuzzing input generation around trouble areas. Machine learning methods could also be applied to further guide these fuzzing efforts.

References

[1] P. S. Foundation. cpython. *https://github.com/python/cpython*.

[2] T. P. Foundation. Perl compiler. *http://www.perl.org/*.

[3] Google. Guetzli perceptual jpeg encoder. *https://github.com/google/guetzli*.

[4] M. Zalewski. American fuzzy lop. *http://lcamtuf.coredump.cx/afl/*.