CS4110 Software Testing and Reverse Engineering: Final Report
Fuzzing

Michael The Olaf Maas Willem Vaandrager

April 2016

1 Introduction

In this report, we will tackle the topic of fuzzing. We do this by analyzing and applying three tools in different areas: Wapiti for web applications, AFL for binaries, and Sulley, the fuzzer framework.

2 Wapiti

Wapiti is a free and Open Source black box fuzzing tool. Wapiti focus on finding vulnerabilities in web applications and offers an extensive toolbox for doing so. Wapiti has no user interface and comes in the form of a commandline tool. After the run it generates an html report with all found vulnerabilities and the payload used to find these vulnerabilities.

2.1 Cookies

Most modern web applications require some form of login to enter the application. It is possible to pass a cookie to wapiti which authenticates Wapiti. To generate this cookie the tool wapiti-getcookie can be used. This tool asks for the url of the login form and will ask the correct value of all fields in this form. It will login and store the cookies it got into a cookies.json file. This file can than be passed to the wapiti fuzzer.

2.2 Vulnerabilities

Wapiti consists of multiple modules, each module is able to detect a specific type of vulnerability. By default wapiti includes the following modules:

1. Backup: Module which looks for weakly secured backup scripts on the server which can be used to disclose data.

2. BlindSQL: This module will look for blind SQL injection vulnerabilities. Wapiti detects these vulnerabilities by using a time based SQL injection. The idea behind tbSQL is to use the vulnerabilities to pause the SQL database for a few seconds before returning the results. If this delay is present the SQL injection succeeded.

3. Exec: A module designed to find command execution vulnerabilities. This module will try to find certain patterns in the html page to detect whether a page is vulnerable. For example if PATH and PWD are in the response there is probably a vulnerability.

4. File: If arguments are directly passed to file reading functions a file disclosure vulnerabilities could be present. This module tries to change parameters to random file names and looks for patterns in the output.

5. HtAccess: This modules looks for parts protected by .htaccess files. If these files are misconfigured or can be bypassed this could lead to a vulnerability. Using evil requests this module will try to circumvent the protection.

6. PermanentXSS: This module looks for permanent XSS vulnerabilities. For example a comment system which are vulnerable to XSS injection. This module does not detect reflected XSS.

7. SQL: In contrary to the blindSQL scanner this module searches for SQL errors. So this module will try to break a query by changing the paremeters of a request. If it succeeds it will report an vulnerability.

8. XSS: This module searches for reflected XSS attacks within the application. It tries to inject javascript which will give an alert with the text XSS. If it succeeds it will report this.

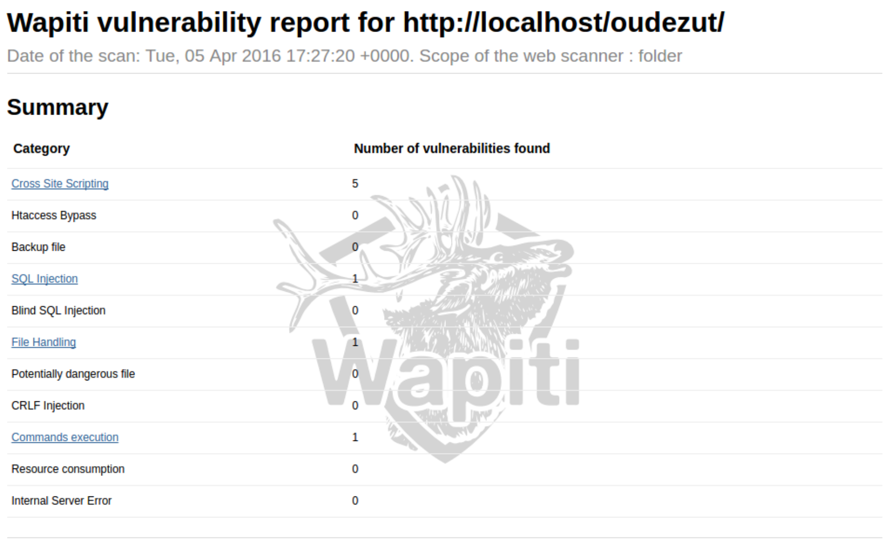
2.3 Crawling

As user the only thing you should pass to wapiti is a base url. Then wapiti will start crawling the application. In practice you do not want wapiti to crawl certain URL which can have unwanted side effects, for example the logout page. Luckily it is possible to exclude certain URLs.

Wapiti will look for get parameters in the url, forms and other user interactions for locations where it is able to fuzz. One limitation of Wapiti is that it only has a basic understanding of Javascript. It is not able to crawl sites which are heavily dependent on a front-end framework such as Angular or React.

2.4 Generated Report

Wapiti is able to generate reports in multiple formats such as HTML, XML, JSON and text. By default it generates an single page HTML report where per module it specifies the vulnerabilities found and the payload used to find the vulnerability. If possible it includes a link to the page. It will also report on HTTP 500 errors if wapiti find any due to the fact that an HTTP 500 erro can also be a sign of a vulnerability.



Wapiti vulnerability report for http://localhost/oudezut/

Date of the scan: Tue, 05 Apr 2016 17:27:20 +0000. Scope of the web scanner : folder

Summary

Category	Number of vulnerabilities found
Cross Site Scripting	5
Htaccess Bypass	0
Backup file	0
SQL Injection	1
Blind SQL Injection	0
File Handling	1
Potentially dangerous file	0
CRLF Injection	0
Commands execution	1
Resource consumption	0
Internal Server Error	0

Figure 1: Example of a Wapiti Report

2.5 Using Wapiti

We used wapiti on a number of web applications.

2.5.1 DVWA

As an easy test to start working with Wapiti we decided to use it on the Damn Vulnerable Web application. This is a web application developed for educational purposes which is vulnerable to a number of commonly known vulnerabilities.

- Brute Forcable Login: Not found (Wapiti is not able to detect it)
- CSRF: Not found (Wapiti is not able to detect it)
- Command Injection: Found
- File Inclusion: Found
- File Upload: Not Found (Due to the fact that the server was configured too safe)
- SQL Injection: Found

- Blind SQL Injection: Not Found (Due to the fact that the database was configured too safe)
- Xss Reflective: Found
- Xss Stored: Found

2.5.2 Own site

We then decided to run Wapiti on a website which was build recently by one of our group members. This website is a basic Symfony website which uses an ORM and a standard form generator which handles most of the requests. We did not found any vulnerabilities in this site. There were however some HTTP 500 error.

Note that these results were obtained with some admin pages excluded. When these admin pages were enabled starting to find vulnerabilities took more than 3 hours. We then killed the process as we suspected that Wapiti was caught in a crawl loop. We found out that it is possible to define a crawl depth which could resolve this issue.

2.5.3 Wordpress With Vulnerable Plugins

In this test Wapiti was used on a wordpress website with a number of vulnerable plugins. Wapiti only found one XSS vulnerability in a job submission form. This was a severe vulnerability as the XSS was executed from within the administration panel.

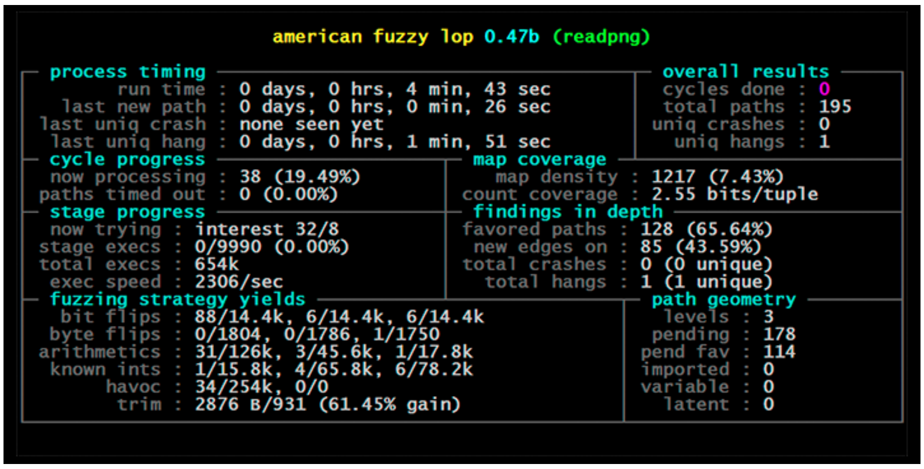
The other vulnerabilities were not found however. When testing the application manually it was easy to find these XSS and SQL vulnerabilities however. But when we turned off Javascript we were not able to exploit the vulnerabilities without looking at the javascript code. This is one of the main weaknesses of Wapiti, most web application communicate asynchronously with an external API. Being able to interpret Javascript and extracting parameters to fuzz the API is really useful when fuzzing modern web applications.

3 American Fuzzy Lop

American Fuzzy Lop (AFL) [1] is an open source fuzzer that uses genetic algorithms to discover interesting test cases that trigger new states in the target program. Its aim is to be a practical fuzzer and uses a collection of techniques that have been found to be effective. An example of the diagnostics screen can be seen in figure 2. In this section, we will describe how AFL works. We will also give the results of running AFL on two versions of ImageMagick.

3.1 The workings of AFL

Shortly said, AFL works as follows:



```
                 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         │
└──────────────────────────────────────────────────────┴───────────────────────┘
```

Figure 2: An example of AFL in action. [1]

1. The user must supply AFL with a set of intial test cases to be used as input.

2. Test cases are trimmed without altering the program's behaviour.

3. Test cases are repeatedly mutated using a number of strategies.

4. If any of the mutations result in a new state transition inside the targeted program, it adds the mutation to the queue of test cases.

This process is repeated indefinitely or until the user stops it. We will describe each of the steps in more detail.

3.1.1 Measuring coverage

Mutated test cases that result in new state transitions are seen as valuable test cases. The reason is obvious: if a new state transition is seen, a new section of the program is executed and increases our chance of finding bugs. In order to detect these state transitions, and when source code is available, the targeted program can be injected with instrumentation that detect these transitions. AFL itself ships with its own version of gcc, g++, and clang, that add the needed instrumentation during compilation. Third-party projects have also made it possible for AFL to target other languages, e.g., python [2], rust [3], OCaml [4].

There are also cases when source code is not available. Experimental support for using AFL with binaries only is accomplished by running QEMU [5], which emulates a hardware environment and allows callers to obtain the needed instrumentation. This comes at a performance cost, with QEMU mode running 2-5x slower.

The specific type of coverage captured by this instrumentation is branch coverage. Previous approaches would only consider block coverage [6], but branch coverage gives more insight into the execution path of the targeted program. For example, it can differentiate between the following execution paths:

$$A \to B \to C(tuples : AB, BC)$$

$$A \to C \to B(tuples : AC, CB)$$

In short, a new execution path is a path containing new edges, which are called tuples in AFL. New test cases that contain new execution paths are kept and put in the queue to be processed.

3.1.2 Trimming test cases

To kick the fuzzer off, AFL needs to be supplied with one or more files that are examples of input for the targeted program. AFL takes this input and mutates it to arrive at new state transitions, as explained in section 3.1.1. These new mutations are also added to the set of test cases. Two things are important here: 1. test cases should be kept small and 2. test cases should not be similar to each other.

The file size of a test case should be kept small as it has a major impact on fuzzing performance. Large files make the targeted program slower. Moreover, a large file reduces the likelihood that important mutations are found. The recommendation is to keep files under 1kb. AFL attempts to remove blocks of data from test cases. If the resulting trimmed test case doesn't affect the execution trace, the trimmed test case replaces the original test case.

Some mutated test cases are very similar to their parents. To further optimize the fuzzing process, AFL also evaluates the entire set of test cases and retains a subset of favoured test cases. This subset contains those test cases that still cover all edges seen so far and are easy to process. The following algorithm is followed to obtain this subset.

1. Test cases are scored based on their execution latency and file size.
2. Tuples are coupled with the test cases with the best score that contain that tuple in their execution path.
3. A tuple is picked at random, the accompanying test case is added to the subset, and all the tuples in the execution path of that test case are marked as processed.
4. If there are tuples left that have not been processed yet, go back to #3.

The resulting subset is usually around 5-10x smaller than the starting set of test cases. As the name implies, favoured test cases are processed before non-favoured test cases. If a non-favoured test case has never been processed before, there's a 75% chance of it getting skipped in the input queue. If it has been used before, this chance increases to 95%. These non-favoured test cases are not deleted, but their usefulness seems limited. This process is repeated periodically.

3.1.3 Mutation strategies

One of the most important parts of AFL is its mutation engine. As mentioned in the introduction of section 3, AFL uses a collection of techniques that have found to be effective. The mutation engine applies six techniques, starting from the most basic deterministic techniques to the more complex random strategies, in order to arrive at new interesting test cases.

1. **Walking Bit Flips**: The first and simplest strategy is to flip bits sequentially. AFL first flips single bits at a time. After that, it flips two bits in a row, and finally four bits in a row.

2. **Walking Byte Flips**: The natural followup to bit flips. AFL applies what are essentially 8-, 16-, or 32-bit flips.

3. **Arithmetics**: The next step is to increment or decrement integers. The range by which it increments/decrements is $[-35, 35]$. Experiments have shown that using values beyond this range does not yield significant advantages.

4. **Known integers** The last deterministic step is to overwrite values with integers that are known to be interesting, e.g., -1, 256, MAX_INT.

5. **Stacked Tweaks** In this step, the input file is processed with a number of operations that are applied randomly:

   - Single bit flips
   - Setting "interesting" bytes, words, or dwords
   - Addition or subtraction of small integers to bytes, words, or dwords
   - Completely random single-byte sets
   - Block deletion
   - Block duplication via overwrite or insertion
   - Block memset

   The probability of that each operation applies is roughly the same. Typically, this step is as effective or even more effective than the deterministic stages.

6. **Test case splicing**: The final strategy involves taking two test cases from the queue, splicing them at a random location, and running them through the previously mentioned "stacked tweaks" algorithm. This strategy tends to find paths that are unlikely to be triggered using the previous strategies.

The mutation engine has found some very interesting cases, e.g., being able to conjure jpeg files with a text file as starting input [7], or being able to discover the CDATA syntax in XML [8]. This shows AFL's surprising power and effectiveness.

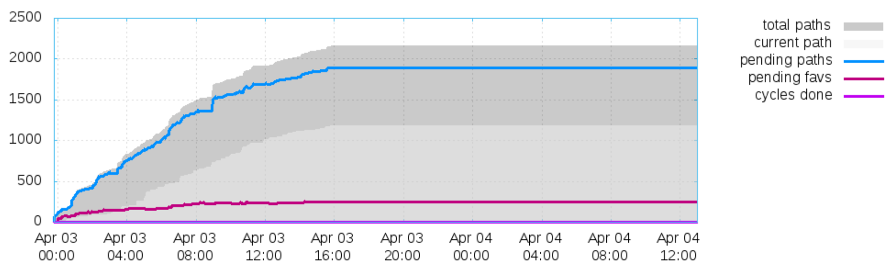
3.2 AFL vs. Imagemagick

ImageMagick has been the target of many fuzzing attempts. It's a widely used library for processing images from the command line. However, after a security audit performed by google [9], most of the trivial bugs have been fixed after version 6.8.9-10. To confirm this, we ran afl against two versions of ImageMagick: 6.8.9-10 and the most recent version, 6.9.3-7.

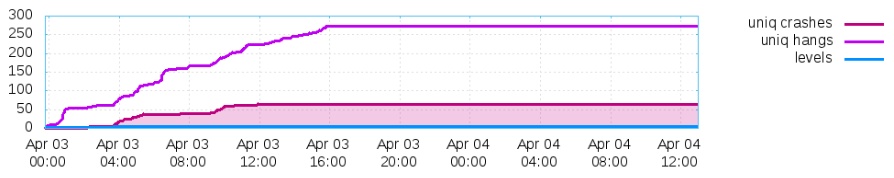
The set of initial input files we used were provided by The Fuzzing Project [10]. It contains 4 files that are known to cause crashes in older versions of ImageMagick. The specific command-line utility we target is the *convert* command, which is mostly used to convert between image formats, but also has other functionalities, such as blurring, cropping, and resizing. We instruct the convert command to convert our test cases to jpeg.

AFL contains a tool for generating basic plots, displayed in figures 3 and 4. Each run has been running for approximately 24 hours. There's a clear difference between the number of crashes found: 66 for 6.8.9-10, 7 for 6.9.3-7. Moreover, we were unable to reproduce the crashes that afl found in 6.9.3-7. The total number of paths found is very similar; It is possible that both runs find the same paths, but where it crashes is fixed in the newer version.

In conclusion, ImageMagick has become much more robust against trivial fuzzing for versions after 6.8.9-10. AFL has proven to be an effective and easy to use tool for finding bugs. Moreover, efforts like The Fuzzing Project motivate software to become more resilient and more bug-proof.

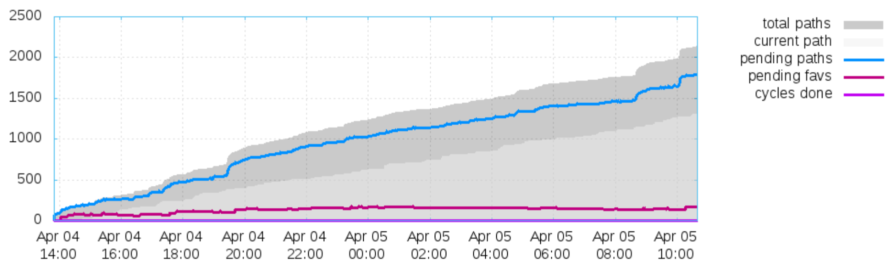


(a) Paths found

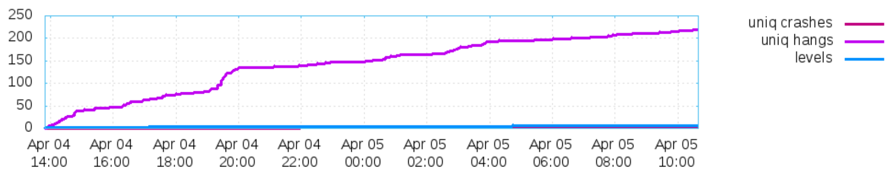


(b) Unique crashes and hangs

Figure 3: Results fuzzing ImageMagick 6.8.9-10



(a) Paths found



(b) Unique crashes and hangs

Figure 4: Results fuzzing ImageMagick 6.9.3-7

4 Sulley

Sulley is an open source fuzzing framework written in python [11]. Sulley is can be used to fuzz applications, network protocols and command line arguments.

Sulley is an extensive framework for generation based fuzzing, that combining fuzzing and multiple ways of monitoring the outcomes.

Sulley offers three ways to help with fuzzing a target application. It provides data representation. Sulley generates values for the parameters in the request to fuzz the application and try generate crashes. Sulley uses a session to keep track of which test cases have been run and communicates the results to the agent that are listening. During a run the data is captured and after the run elaborate results can be seen and replayed.

4.1 Block statement

The idea of the fuzzer is to create requests generated with block statement. These blocks can be determined by the user and are used to generate a valid request to fuzz the target. A block can consist of a static part that is always the same in the request and several primitives that are base fuzz elements that sulley supports. These primitives can be integers, strings, binaries and more. Most of the mutations sulley generated are bound by the primitive and only full random primitives can generate mutations in a specified larger size. With the block statement sulley generates a set of mutations for the block statement and starts fuzzing by sending requests to the target.

4.2 Group

Group are used to couple blocks to a set of primitives. A group is defined with a name and the raw values a mutation of a value can take. Groups can be used to fuzz multiple valid requests option of the application.

4.3 Encoders

Sulley also supports encoders. Encoders are block modifiers. The encoder takes a primitive and transforms it to a defined output format. This can be useful when a protocol accepts a specific It lets Sulley use the fuzz mutation it used and transform it easily to a valid request.

4.4 Agents

Sulley has 3 main components to monitor crashes and violations. These are the process monitor, network monitor and virtual machine control agent. All are driven by PedRPC, a protocol that Sulley uses to communicate the observed results to the different agents.

4.4.1 Network Monitor

The network monitor agent is used for monitoring network communications between sulley and the target program. It logs the results of the tests in PCAP files. Sulley communicates with the network agent prior and after transmitting the test case. This allows the agent to start recording the process of the test and capture the results. The PCAP tests cased are named after the corresponding test case, so they are to find when a test case generates a failure. An advantage of the network agent is that it can run on a separate device as the fuzzed application and can monitor results when using a virtual machine as test environment.

4.4.2 Process Monitor

The process monitor is responsible for monitoring the results of tests when fuzzing of a process. The agent listens to port 26002 and does receive data from the Sulley session. AFter sending a test test case.

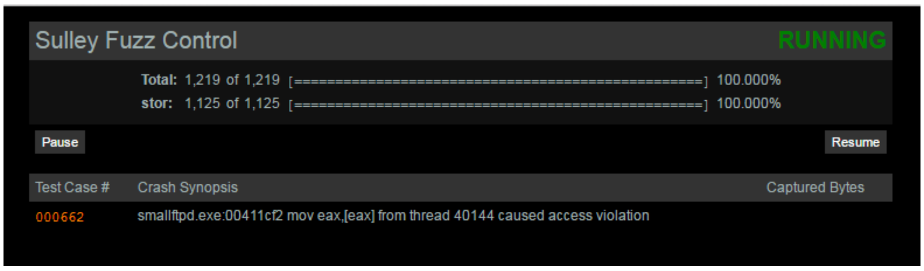
The process of the fuzzing can be viewed on a webserver that shows the status of the session. The fuzzing can be paused and resumed here and when triggered faults will be shown. The found faults will be stored in a crashbin file that can be accessed by the webserver or in the console. When selecting a test case that triggered a fault, a stack trace of the events that happened during the test is shown. This can be used for analysing for kind of bug in the program has occured.

4.4.3 VMWare Control

VMWare control is an API used for setting up a test environment with a virutal machine. VMWare control can start, stop, pause or reset a virutal machine as well as restore a snapshot of the virtual machine to test in a specific scenario of a machine state. This agent exposes an API for interacting with a virtual machine. When a crash occurs it can restore or restart the virtual machine to a known good state.

5 Results

Sulley was harder to use than expected, when trying to fuzz certain applications it would either not show any failures or when using the network agent not capture any data from the session. In the following application used for setting up a ftp server, a bug with the login was quickly found.

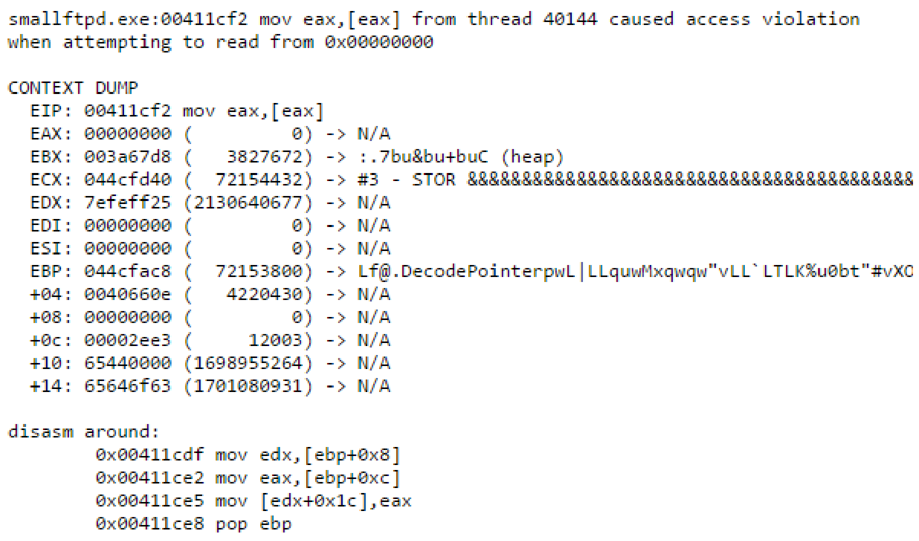


```
Total: 1,219 of 1,219 [================================================] 100.000%
stor:  1,125 of 1,125 [================================================] 100.000%
```

Test Case #	Crash Synopsis	Captured Bytes
000662	smallftpd.exe:00411cf2 mov eax,[eax] from thread 40144 caused access violation	

Figure 5: Process monitor agent, that shows captured violations

This violation can be examined. Clicking on the violation will show a stack trace when the problem occurred. Looking at the stack trace the error it looks like it tries to read from an unknown position that can be caused by a buffer-overflow.



```
smallftpd.exe:00411cf2 mov eax,[eax] from thread 40144 caused access violation
when attempting to read from 0x00000000

CONTEXT DUMP
  EIP: 00411cf2 mov eax,[eax]
  EAX: 00000000 (         0) -> N/A
  EBX: 003a67d8 (   3827672) -> :.7bu&bu+buC (heap)
  ECX: 044cfd40 (  72154432) -> #3 - STOR &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
  EDX: 7efeff25 (2130640677) -> N/A
  EDI: 00000000 (         0) -> N/A
  ESI: 00000000 (         0) -> N/A
  EBP: 044cfac8 (  72153800) -> Lf@.DecodePointerpwL|LLquwMxqwqw"vLL`LTLK%u0bt"#vXO
  +04: 0040660e (   4220430) -> N/A
  +08: 00000000 (         0) -> N/A
  +0c: 00002ee3 (     12003) -> N/A
  +10: 65440000 (1698955264) -> N/A
  +14: 65646f63 (1701080931) -> N/A

disasm around:
        0x00411cdf mov edx,[ebp+0x8]
        0x00411ce2 mov eax,[ebp+0xc]
        0x00411ce5 mov [edx+0x1c],eax
        0x00411ce8 pop ebp
```

Figure 6: Crash report of a request

In conclusion Sulley offers a lot of possibilities and support for fuzzing an application. It requires a bit of learning, but when getting to know the tool it can be really comprehensive in finding bugs in a variety of applications.

References

[1] Micha 'lcamtuf' Zalewski. *american fuzzy lop.* URL: `http://lcamtuf.coredump.cx/afl/`.

[2] Jakub Wilk. *python-afl.* URL: `https://bitbucket.org/jwilk/python-afl/src`.

[3] Corey Farwell. *afl.rs.* URL: `https://github.com/frewsxcv/afl.rs`.

[4] Stephen Dolan. *Instrumentation for american fuzzy lop (afl-fuzz) #504.* URL: `https://github.com/ocaml/ocaml/pull/504`.

[5] *QEMU: Quick Emulator.* URL: `http://wiki.qemu.org/Main_Page`.

[6] Micha 'lcamtuf' Zalewski. *Historical Notes.* URL: `http://lcamtuf.coredump.cx/afl/historical_notes.txt`.

[7] Micha 'lcamtuf' Zalewski. *Pulling JPEGs out of thin air.* URL: `https://lcamtuf.blogspot.nl/2014/11/pulling-jpegs-out-of-thin-air.html`.

[8] Micha 'lcamtuf' Zalewski. *afl-fuzz: nobody expects CDATA sections in XML.* URL: `https://lcamtuf.blogspot.nl/2014/11/afl-fuzz-nobody-expects-cdata-sections.html`.

[9] Bastien Roucaries. *[oss-security] Imagemagick fuzzing bug.* URL: `https://www.marc.info/?l=oss-security&m=141942017331222&w=2`.

[10] Hanno Bck. *Invalid input examples.* URL: `https://crashes.fuzzing-project.org/`.

[11] Aaron Portnoy Pedram Amini. *Sulley: Fuzzing Framework.* URL: `http://www.fuzzing.org/wp-content/SulleyManual.pdf`.