Introduction to Fuzzing
What is Fuzzing?

• A form of vulnerability analysis

• Process:
  • Many slightly anomalous test cases are input into the application
  • Application is monitored for any sign of error
Example

Standard HTTP GET request
• § GET /index.html HTTP/1.1

Anomalous requests
• § AAAAAA...AAAA /index.html HTTP/1.1
• § GET //////////index.html HTTP/1.1
• § GET %n%n%n%n%n.html HTTP/1.1
• § GET /AAAAAAAAAAAAAA.html HTTP/1.1
• § GET /index.html HTTTTTTTTTTTTTTP/1.1
• § GET /index.html HTTP/1.1.1.1.1.1.1.1
• § etc...
User Testing vs Fuzzing

• User testing
  • Run program on many normal inputs, look for bad things to happen
  • Goal: Prevent normal users from encountering errors

• Fuzzing
  • Run program on many abnormal inputs, look for bad things to happen
  • Goal: Prevent attackers from encountering exploitable errors
Types of Fuzzers

• Mutation Based – “Dumb Fuzzing”
  • mutate existing data samples to create test data

• Generation Based – “Smart Fuzzing”
  • define new tests based on models of the input

• Evolutionary
  • Generate inputs based on response from program
Mutation Based Fuzzing

• Little or no knowledge of the structure of the inputs is assumed
• Anomalies are added to existing valid inputs
• Anomalies may be completely random or follow some heuristics
• Requires little to no set up time
• Dependent on the inputs being modified
• May fail for protocols with checksums, those which depend on challenge response, etc.

• Example Tools:
  • Taof, GPF, ProxyFuzz, Peach Fuzzer, etc.
Mutation Based Example: PDF Fuzzing

• Google .pdf (lots of results)
• Crawl the results and download lots of PDFs

• Use a mutation fuzzer:
  1. Grab the PDF file
  2. Mutate the file
  3. Send the file to the PDF viewer
  4. Record if it crashed (and the input that crashed it)

<table>
<thead>
<tr>
<th></th>
<th>Super easy to setup and automate</th>
<th>Little to no protocol knowledge required</th>
<th>Limited by initial corpus</th>
<th>May fail for protocols with checksums, or other complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation-based</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Generation Based Fuzzing

• Test cases are generated from some description of the format: RFC, documentation, etc.

• Anomalies are added to each possible spot in the inputs.

• Knowledge of protocol should give better results than random fuzzing.

• Can take significant time to set up.

• Examples
  • SPIKE, Sulley, Mu-4000, Codenomicon, Peach Fuzzer, etc...
Example Specification for ZIP file

```xml
<!-- A. Local file header -->
<Block name="LocalFileHeader">
  <String name="lfh_Signature" valueType="hex" value="504b0304" token="true" mutable/>
  <Number name="lfh_Ver" size="16" endian="little" signed="false"/>
  ...
  [truncated for space]
  ...
  <Number name="lfh_CompSize" size="32" endian="little" signed="false"/>
    <Relation type="size" of="lfh_CompData"/>
  </Number>
  <Number name="lfh_DecompressSize" size="32" endian="little" signed="false"/>
  <Number name="lfh_FileNameLen" size="16" endian="little" signed="false"/>
    <Relation type="size" of="lfh_FileName"/>
  </Number>
  <Number name="lfh_ExtraFldLen" size="16" endian="little" signed="false"/>
    <Relation type="size" of="lfh_FldName"/>
  </Number>
  <String name="lfh_FileName"/>
  <String name="lfh_FldName"/>
</Block>
```

## Mutation vs Generation

<table>
<thead>
<tr>
<th></th>
<th>Mutation-based</th>
<th>Generation-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy to setup</td>
<td>Super easy to setup and automate</td>
<td>Writing generator is labor intensive for complex protocols</td>
</tr>
<tr>
<td>Protocols knowledge</td>
<td>Little to no protocol knowledge required</td>
<td>have to have spec of protocol (frequently not a problem for common ones http, snmp, etc...)</td>
</tr>
<tr>
<td>Limited corpus</td>
<td>Limited by initial corpus</td>
<td>Completeness</td>
</tr>
<tr>
<td>Fail protocols</td>
<td>May fail for protocols with checksums, or other complexity</td>
<td>Can deal with complex checksums and dependencies</td>
</tr>
</tbody>
</table>

- :green: Positive point
- :red: Negative point
White box vs. black box fuzzing

- **Black box fuzzing**: sending the malformed input without any verification of the code paths traversed.
- **White box fuzzing**: sending the malformed input and verifying the code paths traversed. Modifying the inputs (via Symbolic Execution) to attempt to cover all code paths.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Effort</th>
<th>Code coverage</th>
<th>Defects Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>black box + mutation</td>
<td>10 min</td>
<td>50%</td>
<td>25%</td>
</tr>
<tr>
<td>black box + generation</td>
<td>30 min</td>
<td>80%</td>
<td>50%</td>
</tr>
<tr>
<td>white box + mutation</td>
<td>2 hours</td>
<td>80%</td>
<td>50%</td>
</tr>
<tr>
<td>white box + generation</td>
<td>2.5 hours</td>
<td>99%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Evolutionary Fuzzing

• Attempts to generate inputs based on the response of the program

• Autodafe
  • Prioritizes test cases based on which inputs have reached dangerous API functions

• EFS
  • Generates test cases based on code coverage metrics

• AFL
  • Most popular choice in DARPA CGC
Challenges

• Mutation based – can run forever. When do we stop?
• Generation based – stop eventually. Is it enough?
• How to determine if the program did something “bad”?

• These are the standard problems we face in most automated testing.
Code Coverage

• Some of the answers to our problems are found in code coverage

• To determine how well your code was tested, code coverage can give you a metric.

• But it’s not perfect (is anything?)

• Code coverage types:
  • Statement coverage – which statements have been executed
  • Branch coverage – which branches have been taken
  • Path coverage – which paths were taken.
Code Coverage - Example

```java
if (a > 2)
    a = 2;
if (b > 2)
    b = 2
```

How many test cases for 100% line coverage?
How many test cases for 100% branch coverage?
How many test cases for 100% paths?
Code Coverage Tools

• If you have source: gcov, Bullseye, Emma

• If you don’t:
  • Binary instrumentation: PIN, DynamoRIO, QEMU
  • Valgrind : instrumentation framework for building dynamic analysis tools
  • Pai Mei : a reverse engineering framework consisting of multiple extensible components.

Lots more to discuss on Code Coverage in a Software Engineering class.. but lets move on.
Lessons about Fuzzing

• Protocol knowledge is helpful
  • Generational beats random, better specification make better fuzzers

• Using more fuzzers is better
  • Each one will vary and find different bugs

• The longer you run (typically) the more bugs you’ll find

• Guide the process, fix it when it break or fails to reach where you need it to go

• Code coverage can serve as a useful guide
AFL – American Fuzzy Lop

• Fuzzer developed by Michal Zalewski (lcamtuf), Project Zero, Google
  • He's on holiday today 😞
• http://lcamtuf.coredump.cx/afl/
(2) Seed Mutation

Seed Scheduling

Seed

Testcase

Seed

(3) Seed Selection

Coverage Bitmap

Instrumented Program

New Seed

Queue of Seeds

new coverage!
Why use AFL?
It finds bugs

IJG jpeg 1 libjpeg-turbo 1 2 libpng 1 libtiff 1 2 3 4 5 mozjpeg 1 libbpg (1) Mozilla Firefox 1 2 3 4 5 Google Chrome 1 Internet Explorer 1 2 (3) (4) LibreOffice 1 2 3 poppler 1 freetype 1 2 GnuTLS 1 GnuPG 1 2 (3) OpenSSH 1 2 3 bash (post-Shellshock) 1 2 tcpdump 1 2 3 4 5 6 7 Adobe Flash / PCRE 1 2 JavaScriptCore 1 2 3 4 pdfium 1 ffmpeg 1 2 3 4 libmatroska 1 libarchive 1 2 3 4 5 6 ... wireshark 1 ImageMagick 1 2 3 4 5 6 7 8 ... Icms (1) PHP 1 2 lame 1 FLAC audio library 1 2 libsndfile 1 2 3 less / lesspipe 1 2 3 strings (+ related tools) 1 2 3 4 5 6 7 file 1 2 dpkg 1 rcs 1 systemd-resolved 1 2 sqlite 1 2 3 libyaml 1 Info-Zip unzip 1 2 OpenBSD pfctl 1 NetBSD bpf 1 man & mandoc 1 2 3 4 5 ... IDA Pro clamav 1 2 libxml2 1 glibc 1 clang / llvm 1 2 3 4 5 6 nasm 1 2 ctags 1 mutt 1 procmail 1 fontconfig 1 pdksh 1 2 Qt 1 wavpack 1 redis / lua-cmsgpack 1 taglib 1 2 3 privoxy 1 perl 1 2 3 4 5 6 libxmp radare2 1 2 fwknop metacam 1 exifprobe 1 capnproto 1
It's spooky

• Michal gave djpeg (IJG jpeg library) to AFL
• Plus a non-jpeg file as an input
  • $ echo 'hello' > in_dir/hello
• AFL started to produce valid jpeg files after a day or two
More reasons

• It's dead simple
• No configuration of AFL necessary, robust
• It's cutting edge
• It's fast
• Produces very very good input files (corpus) that can be used in other fuzzers
• Many targets that were never touched by AFL (and it will crush them)
You won't believe what you are reading

• Source: http://lcamtuf.coredump.cx/afl/demo/
• afl-generated, minimized image test sets (partial) [...] 
• JPEG XR jxrlib 1.1 JxrDecApp¹ IE ➞ Ditched ²
• ² Due to the sheer number of exploitable bugs that allow the fuzzer to jump to arbitrary addresses.
When to use AFL
The usual use case

• You have the source code and you compile with gcc or clang
• Your are on 32bit or 64bit on Linux/OSX/BSD
• The to-be-fuzzed code (e.g. parser) reads it's input from stdin or from a file
• The input file is usually only max. 10kb
• This covers *a lot* of Linux libraries
What if something does not apply?

• No source code?
  – Try the experimental QEMU instrumentation

• Not on 32/64 bit?
  – There is an experimental ARM version

• Not reading from stdin or file?
  – Maybe your project has a utility command line tool that does read from file
  – Or you write a wrapper to do it
  – Same if you want to test (parts of) network protocol parsers
How to use AFL
Steps of fuzzing

1. Compile/install AFL (once)
2. Compile target project with AFL
   • afl-gcc / afl-g++ / afl-clang / afl-clang++ / (afl-as)
3. Chose target binary to fuzz in project
   • Chose its command line options to make it run fast
4. Chose valid input files that cover a wide variety of possible input files
   • afl-cmin / (afl-showmap)
Steps of fuzzing

5. Fuzzing
   • afl-fuzz

6. Check how your fuzzer is doing
   • command line UI / afl-whatsup / afl-plot / afl-gotcpu

7. Analyze crashes
   • afl-tmin / triage_crashes.sh / peruvian were rabbit
   • ASAN / valgrind / exploitable gdb plugin / ...

8. Have a lot more work than before
   • CVE assignment / responsible disclosure / ...
# Download & compile new AFL version:
wget http://lcamtuf.coredump.cx/afl.tgz

tar xfz afl.tgz
rm afl.tgz

cd `find . -type d -iname "afl-*"|sort|head -1`
mke

echo "Provide sudo password for sudo make install"
sudo make install
AFL binaries

```
/opt/afl-1.56b$ ./afl-
afl-as      afl-fuzz      afl-plot
afl-clang   afl-g++       afl-showmap
afl-clang++ afl-gcc       afl-tmin
afl-cmin    afl-gotcpu    afl-whatsup
```

```
/opt/afl-1.56b$ ./afl-gcc
[...]
This is a helper application for afl-fuzz. It serves as a drop-in replacement for gcc or clang, letting you recompile third-party code with the required runtime instrumentation.
[...]
Instrumenting a project (step 2) – example: libtiff from CVS repository

/opt/libtiff-cvs-afl$ export CC=afl-gcc
/opt/libtiff-cvs-afl$ export CXX=afl-g++
/opt/libtiff-cvs-afl$ ./configure --disable-shared
/opt/libtiff-cvs-afl$ make clean
/opt/libtiff-cvs-afl$ make
Choosing the binary to fuzz (step 3) – they are all waiting for it

```
/opt/libtiff-cvs-afl$ ./tools/

bmp2tiff  fax2tiff  ppm2tiff  raw2tiff
thumbnail  tiff2pdf  tiff2rgba  tiffcp
tiffdither  tiffinfo  tiffset  fax2ps
gif2tiff  pal2rgb  ras2tiff  rgb2ycbcr
tiff2bw  tiff2ps  tiffcmp  tiffcrop
tiffdump  tiffmedian  tiffspli
```

```
/opt/libtiff-cvs-afl$ ./tools/bmp2tiff
LIBTIFF, Version 4.0.3
Copyright (c) 1988-1996 Sam Leffler
[...]
usage: bmp2tiff [options] input.bmp [input2.bmp ...]
output.tif
```
Chose initial input files (step 4)

/opt/libtiff-cvs-afl$ mkdir input_all
/opt/libtiff-cvs-afl$ scp host:/bmps/ input_all/
/opt/libtiff-cvs-afl$ ls -l input_all | wc -l
886
Chose initial input files (step 4)

```
/opt/libtiff-cvs-afl$ afl-cmin -i input_all -o input
-- /opt/libtiff-cvs-afl/tools/bmp2tiff @@ /dev/null
```
corpus minimization tool for afl-fuzz by
<lcamtuf@google.com>
[*] Testing the target binary...
[+] OK, 191 tuples recorded.
[*] Obtaining traces for input files in
'input_all'...
Processing file 886/886...
[*] Sorting trace sets (this may take a while)...
[+] Found 4612 unique tuples across 886 files.
[*] Finding best candidates for each tuple...
Processing file 886/886...
[*] Sorting candidate list (be patient)...
[*] Processing candidates and writing output files...
Processing tuple 4612/4612...
[+] Narrowed down to 162 files, saved in 'input'.
Chose initial input files (step 4)

```
/opt/libtiff-cvs-afl$ ls -l input | wc -l
162
```
Fuzzing (step 5)

```
/opt/libtiff-cvs-afl$ screen -S fuzzing
/opt/libtiff-cvs-afl$ afl-fuzz -i input -o output --
/opt/libtiff-cvs-afl/tools/bmp2tiff @@ /dev/null
```
How is our fuzzer doing? (step 6)

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time : 0 days, 0 hrs, 2 min, 30 sec</td>
<td>Cycles done : 0</td>
</tr>
<tr>
<td>Last new path : 0 days, 0 hrs, 0 min, 3 sec</td>
<td>Total paths : 193</td>
</tr>
<tr>
<td>Last uniq crash : 0 days, 0 hrs, 0 min, 4 sec</td>
<td>Uniq crashes : 2</td>
</tr>
<tr>
<td>Last uniq hang : 0 days, 0 hrs, 0 min, 1 sec</td>
<td>Uniq hangs : 15</td>
</tr>
<tr>
<td>Cycle progress</td>
<td>Map coverage</td>
</tr>
<tr>
<td>Now processing : 3 (1.55%)</td>
<td>Map density : 1344 (2.05%)</td>
</tr>
<tr>
<td>Paths timed out : 0 (0.00%)</td>
<td>Count coverage : 3.53 bits/tuple</td>
</tr>
<tr>
<td>Stage progress</td>
<td>Findings in depth</td>
</tr>
<tr>
<td>Now trying : auto extras (over)</td>
<td>Favored paths : 68 (35.23%)</td>
</tr>
<tr>
<td>Stage execs : 15/72 (20.83%)</td>
<td>New edges on : 79 (40.93%)</td>
</tr>
<tr>
<td>Total execs : 86.9k</td>
<td>Total crashes : 19 (2 unique)</td>
</tr>
<tr>
<td>Exec speed : 71.11/sec (slow!)</td>
<td>Total hangs : 100 (15 unique)</td>
</tr>
<tr>
<td>Fuzzing strategy yields</td>
<td>Path geometry</td>
</tr>
<tr>
<td>Bit flips : 12/704, 1/700, 1/692</td>
<td>Levels : 2</td>
</tr>
<tr>
<td>Byte flips : 0/88, 0/84, 0/76</td>
<td>Pending : 190</td>
</tr>
<tr>
<td>Arithmetics : 4/4840, 0/4068, 0/2495</td>
<td>Pend fav : 65</td>
</tr>
<tr>
<td>Known ints : 1/404, 1/2333, 2/2842</td>
<td>Own finds : 31</td>
</tr>
<tr>
<td>Dictionary : 0/0, 0/0, 0/16</td>
<td>Imported : n/a</td>
</tr>
<tr>
<td>Havoc : 9/65.6k, 0/0</td>
<td>Variable : 0</td>
</tr>
<tr>
<td>Trim : 8.33%/20, 0.00%</td>
<td></td>
</tr>
</tbody>
</table>
How is our fuzzer doing? (step 6)
How is our fuzzer doing? (step 6)

<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time</td>
<td>cycles done: 0</td>
</tr>
<tr>
<td>0 days, 1 hrs, 27 min, 43 sec</td>
<td>total paths: 281</td>
</tr>
<tr>
<td>last new path</td>
<td>uniq crashes: 44</td>
</tr>
<tr>
<td>0 days, 0 hrs, 28 min, 27 sec</td>
<td>uniq hangs: 76</td>
</tr>
<tr>
<td>last uniq crash</td>
<td></td>
</tr>
<tr>
<td>0 days, 0 hrs, 31 min, 10 sec</td>
<td></td>
</tr>
<tr>
<td>last uniq hang</td>
<td></td>
</tr>
<tr>
<td>0 days, 0 hrs, 29 min, 29 sec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>map coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing</td>
<td>map density: 1375 (2.10%)</td>
</tr>
<tr>
<td>57 (20.28%)</td>
<td>count coverage: 3.67 bits/tuple</td>
</tr>
<tr>
<td>paths timed out</td>
<td>findings in depth</td>
</tr>
<tr>
<td>0 (0.00%)</td>
<td>favored paths: 95 (33.81%)</td>
</tr>
<tr>
<td>stage progress</td>
<td>new edges on: 104 (37.01%)</td>
</tr>
<tr>
<td>now trying: arith 32/8</td>
<td></td>
</tr>
<tr>
<td>stage execs: 3480/18.9k (18.37%)</td>
<td>total crashes: 427 (44 unique)</td>
</tr>
<tr>
<td>total execs: 938k</td>
<td>total hangs: 4681 (76 unique)</td>
</tr>
<tr>
<td>exec speed: 18.23/sec (zzzz...)</td>
<td>path geometry</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fuzzing strategy yields</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bit flips</td>
<td></td>
</tr>
<tr>
<td>40/24.8k, 4/24.7k, 4/24.7k</td>
<td></td>
</tr>
<tr>
<td>byte flips</td>
<td></td>
</tr>
<tr>
<td>0/3096, 0/2554, 1/2654</td>
<td></td>
</tr>
<tr>
<td>arithmetics</td>
<td></td>
</tr>
<tr>
<td>22/137k, 6/110k, 0/62.2k</td>
<td></td>
</tr>
<tr>
<td>known ints</td>
<td></td>
</tr>
<tr>
<td>0/10.5k, 6/67.6k, 17/97.3k</td>
<td></td>
</tr>
<tr>
<td>dictionary</td>
<td></td>
</tr>
<tr>
<td>0/0, 0/0, 3/6243</td>
<td></td>
</tr>
<tr>
<td>havoc</td>
<td></td>
</tr>
<tr>
<td>55/356k, 0/0</td>
<td></td>
</tr>
<tr>
<td>trim</td>
<td></td>
</tr>
<tr>
<td>14.63%/1266, 18.73%</td>
<td></td>
</tr>
</tbody>
</table>

[cpu: 304%]
How is our fuzzer doing? (step 6)

```
/opt/libtiff-cvs-afl$ afl-gotcpu
afl-gotcpu 1.56b (Mar 9 2015 02:50:32) by <lcamtuf@google.com>
[*] Measuring preemption rate (this will take 5.00 sec)...
[+] Busy loop hit 79 times, real = 5001 ms, slice = 2448 ms.
>>> FAIL: Your CPU is overbooked (204%). <<<
```
How is our fuzzer doing? (step 6)

- `afl-plot`
How is our fuzzer doing? (step 6)

- afl-plot
### Other examples

<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time: 87 days, 18 hrs, 25 min, 44 sec</td>
<td>cycles done: 0</td>
</tr>
<tr>
<td>last new path: 0 days, 0 hrs, 21 min, 38 sec</td>
<td>total paths: 16.1k</td>
</tr>
<tr>
<td>last uniq crash: 8 days, 0 hrs, 47 min, 10 sec</td>
<td>uniq crashes: 88</td>
</tr>
<tr>
<td>last uniq hang: 0 days, 11 hrs, 6 min, 1 sec</td>
<td>uniq hangs: 432</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>map coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing: 7570* (47.01%)</td>
<td>map density: 27.4k (41.75%)</td>
</tr>
<tr>
<td>paths timed out: 0 (0.00%)</td>
<td>count coverage: 4.17 bits/tuple</td>
</tr>
<tr>
<td>stage progress</td>
<td>findings in depth</td>
</tr>
<tr>
<td>now trying: havoc</td>
<td>favored paths: 2024 (12.57%)</td>
</tr>
<tr>
<td>stage execs: 69.4k/80.0k (86.80%)</td>
<td>new edges on: 4925 (30.58%)</td>
</tr>
<tr>
<td>total execs: 213M</td>
<td>total crashes: 124 (88 unique)</td>
</tr>
<tr>
<td>exec speed: 32.71/sec (slow!)</td>
<td>total hangs: 24.4k (432 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fuzzing strategy yields</th>
<th>path geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit flips: 629/5.13M, 240/5.13M, 240/5.13M</td>
<td>levels: 9</td>
</tr>
<tr>
<td>byte flips: 29/641k, 34/639k, 44/637k</td>
<td>pending: 15.0k</td>
</tr>
<tr>
<td>arithmetics: 956/44.9M, 286/15.9M, 49/3.99M</td>
<td>pend fav: 1741</td>
</tr>
<tr>
<td>known ints: 119/5.63M, 400/23.6M, 536/31.9M</td>
<td>own finds: 16.1k</td>
</tr>
<tr>
<td>havoc: 12.5k/70.3M, 0/0</td>
<td>imported: 0</td>
</tr>
<tr>
<td>trim: 62.0 kB/252k (9.02% gain)</td>
<td>variable: 0</td>
</tr>
</tbody>
</table>
Crash analysis (step 7)  
minimizing crash input

```
/opt/libtiff-cvs-afl$ afl-tmin -i
output/crashes/id\:000000\,sig\:11\,src\:000003\,op\:int16\,pos\:21\,val\:+1  
-o minimized-crash

```

```
/opt/libtiff-cvs-afl/tools/bmp2tiff @@ /dev/null
```

```
afl-tmin 1.56b (Mar 9 2015 02:50:31) by <lcamtuf@google.com>
[+] Read 36 bytes from
'output/crashes/id:000000,src:000003,op:int16,
pos:21,val:+1'.
[*] Performing dry run (mem limit = 25 MB, timeout = 1000 ms)...
[+] Program exits with a signal, minimizing in crash mode.
[*] --- Pass #1 ---
[*] Stage #1: Removing blocks of data...
Block length = 2, remaining size = 36
Block length = 1, remaining size = 34
[...]
```
Crash analysis (step 7)  
minimizing malicious input

/opt/libtiff-cvs-afl$ ls -als
output/crashes/id:000000,sig:11,src:000003,op:int16,pos:21,val:+14 -rw------  1 user user 36
Mär  9 04:17
output/crashes/id:000000,sig:11,src:000003,op:int16,pos:21,val:+1

/opt/libtiff-cvs-afl$ ls -als minimized-crash 4 -rw------  1 user user 34  Mär  9 05:51 minimized-crash
Crash analysis (step 7)
example of manual analysis

uncompr_size = width * length;
...
uncomprbuf = (unsigned char *)_TIFFmalloc(uncompr_size);

(gdb) p width
$70 = 65536
(gdb) p length
$71 = 65544
(gdb) p uncompr_size
$72 = 524288

524289 is (65536 * 65544) % MAX_INT
Crash analysis (step 7) peruvian were-rabbit
Crash analysis (step 7) peruvian were-rabbit

• Using crashes as inputs, mutate them to find different crashes (that AFL considers "unique")

/opt/libtiff-cvs-afl$ afl-fuzz -i output/crashes/ -o peruvian_crashes -C /opt/libtiff-cvs-afl/tools/bmp2tiff @@ /dev/null
Crash analysis (step 7)
peruvian were-rabbit

```
<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time : 0 days, 0 hrs, 3 min, 3 sec</td>
<td>cycles done : 0</td>
</tr>
<tr>
<td>last new path : 0 days, 0 hrs, 0 min, 21 sec</td>
<td>total paths : 170</td>
</tr>
<tr>
<td>last uniq crash : 0 days, 0 hrs, 0 min, 20 sec</td>
<td>uniq crashes : 34</td>
</tr>
<tr>
<td>last uniq hang : 0 days, 0 hrs, 0 min, 0 sec</td>
<td>uniq hangs : 29</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>map coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing : 1 (0.59%)</td>
<td>map density : 816 (1.25%)</td>
</tr>
<tr>
<td>paths timed out : 0 (0.00%)</td>
<td>count coverage : 3.39 bits/tuple</td>
</tr>
<tr>
<td>stage progress</td>
<td>findings in depth</td>
</tr>
<tr>
<td>now trying : havoc</td>
<td>favored paths : 30 (17.65%)</td>
</tr>
<tr>
<td>stage execs : 47.5k/60.0k (79.16%)</td>
<td>new edges on : 52 (30.59%)</td>
</tr>
<tr>
<td>total execs : 57.7k</td>
<td>new crashes : 7987 (34 unique)</td>
</tr>
<tr>
<td>exec speed : 374.1/sec</td>
<td>total hangs : 369 (29 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fuzzing strategy yields</th>
<th>path geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetics : 19/1981, 3/1919, 0/1227</td>
<td>pend fav : 30</td>
</tr>
<tr>
<td>known ints : 0/162, 8/944, 4/1252</td>
<td>own finds : 82</td>
</tr>
<tr>
<td>dictionary : 0/0, 0/0, 0/32</td>
<td>imported : n/a</td>
</tr>
<tr>
<td>havoc : 0/0, 0/0</td>
<td>variable : 2</td>
</tr>
<tr>
<td>trim : 0.00%/8, 0.00%</td>
<td></td>
</tr>
</tbody>
</table>
```

[cpu:306%]
Research Questions

• What coverage metric should we use?

• Better mutation strategy than random?

• How to make better use of fuzzing and symbolic execution?