ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Detection

Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, Bin Liang

Email: {you58, ma229, xyzhang}@purdue.edu, {xw48, xw7}@indiana.edu, {hjj, liangb}@ruc.edu.cn
Mutation-based Fuzzing

- Starts from a set of valid input instances as seeds
- Continuously modify to explore various execution paths

Store mutated input if new coverage is hit
Effectiveness of AFL

More than 1.5 million mutations are performed on the 31st byte (0x1F), which is ineffective for coverage improvement.

Observation 1: More than 60% of the mutations are performed on the input bytes that are ineffective.

More than 1.5 million mutations are performed on the 31st byte (0x1F), which is ineffective for coverage improvement.
Effectiveness of AFL

Observation 2: effective mutation ratio (EMR) drops very quickly.

\[
\text{EMR} = \frac{\text{# mutations that increase coverage}}{\text{# total mutations}}
\]

Code coverage is hardly improved after 8 hours.
Existing Works

• Improve the breadth
  • Seed selection: Rebert et al. [SEC 14], Moonshine [SEC 18]
  • Seed prioritization: AFLFast [CCS 16], Steelix [FSE 17], FairFuzz [ASE 18]

• Improve the depth
  • Taint analysis: BuzzFuzz [ICSE 09], TaintScope [S&P 12], VUzzer [NDSS 17]
  • Symbolic execution: Driller [NDSS 16], QSYM [SEC 18], T-Fuzz [S&P 18]
  • Gradient-based search: Angora [S&P 18], NEUZZ [S&P 19]
ProFuzzer

• Basic idea: on-the-fly input structure understanding & utilizing

• Probe input types in a light-weight manner
  • Per-byte mutation observation
  • Field identification
  • Type discovery

• Leverage type information to guide further mutations
  • Explore valid values for better code coverage
  • Exploit specific values that may lead to a vulnerability

• Application-agnostic v.s. application-specific types
  • Application-agnostic: raw data, size, etc.
  • Application-specific: ip address, pdf data structure, etc.
Fuzzing-related Input Types

i. Assertion

ii. Raw Data

iii. Enumeration

iv. Offset

v. Size

vi. Loop Count

```c
header->biBitCount = get2Bytes(IN);
switch (header->biBitCount) {
    case 0x08: bmp8toimage(pData, ...); break;
    case 0x10: bmp16toimage(pData, ...); break;
    case 0x18: bmp24toimage(pData, ...); break;
    case 0x20: bmp32toimage(pData, ...); break;
    default: exit_error();
}

header->bfOffBits = get4Bytes(IN);
fseek(IN, header->bfOffBits, SEEK_SET);
if (fread(pData, ..., stride * header->biHeight, IN) != (stride * height)) exit_error();
```
Probing: observing per-byte mutation effect

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
00000000h: FF 4D 3A 00 00 00 00 00 00 00 36 00 00 00 28 00
00000010h: 00 00 01 00 00 00 02 00 00 00 01 00 18 00 00 00
00000020h: 00 00 04 00 00 00 4F 00 00 00 4F 00 00 00 00 00
00000030h: 00 00 00 00 00 00 FF FF FF 00 FF FF FF 00
```

original execution trace
mutated execution trace
trace difference

execution profiles

```
[ 0x00 0x01 ...... 0xFF ]
```
Field Identification: group consecutive bytes

0 1 2 3 4 5 6 7 8 9 a b c d e f

00000000h: 42 4D 3A 00 00 00 00 00 00 00 36 00 00 00 28 00
00000010h: 00 00 01 00 00 00 02 00 00 00 01 00 18 00 00 00
00000020h: 00 00 04 00 00 00 4F 00 00 00 4F 00 00 00 00 00
00000030h: 00 00 00 00 00 00 FF FF FF 00 FF FF FF 00

execution profile of byte 0x00
[ 0x00 0x01 ...... 0xFF ]

execution profile of byte 0x01
[ 0x00 0x01 ...... 0xFF ]

execution profile of byte 0x02
[ 0x00 0x01 ...... 0xFF ]
Field Identification: group consecutive bytes

- Group bytes at offsets from $i$ to $j$ together as a field if they share the same invalid execution profile (i.e., equal minimum similarity)

```c
header->bfType = get2Bytes(IN);
if (header->bfType != 0x4d42) exit_error();
```
Type Inference: determine type of each field

- **Enumeration**

  If there exists a valid value set \( V_S \), such that:
  
  - values in \( V_S \) correspond to large similarity;
  - other values correspond to small similarity.

- **Size**

  If there exists a bound value \( b_v \), such that:
  
  - values within \( b_v \) correspond to large similarity;
  - values beyond \( b_v \) correspond to small similarity.

profile similarity graph of the 28th byte (0x1C)  
profile similarity graph of the 22nd byte (0x16)
Type Inference: determine type of each field

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Raw Data</th>
<th>Enumeration</th>
<th>Loop Count</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
</table>

By matching execution profiles with different feature patterns, the type of each input field is identified.
Type-guided Exploration (for better coverage)

Limit mutation to all the valid values of the field type.
For size field: increase its value by X and appends X bytes data
Type-guided Exploitation (for bug detection)

Exploit a set of special values that may lead to potential vulnerabilities.

```
location_end - location_current = 0x27
```
Evaluation

- Generality of Assumptions
- Input Size and Path Coverage
- Probing Accuracy
- Finding Zero-day Vulnerabilities
- Evaluation on Standard Benchmarks
- Exposing Known Vulnerabilities
- Performance
## Probing Accuracy

### Results Table

<table>
<thead>
<tr>
<th>Product</th>
<th>Actual</th>
<th>ProFuzzer</th>
<th></th>
<th></th>
<th>afl-analyze</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Inferred</td>
<td>Wrong (FP*)</td>
<td>Missed (FN**)</td>
<td>Inferred</td>
<td>Wrong (FP*)</td>
<td>Missed (FN**)</td>
</tr>
<tr>
<td>exiv2</td>
<td>20</td>
<td>21</td>
<td>3 (14%)</td>
<td>0 (0%)</td>
<td>16</td>
<td>11 (69%)</td>
<td>15 (75%)</td>
</tr>
<tr>
<td>graphicsmagick</td>
<td>17</td>
<td>19</td>
<td>1 (5%)</td>
<td>2 (12%)</td>
<td>7</td>
<td>4 (57%)</td>
<td>14 (82%)</td>
</tr>
<tr>
<td>libtiff</td>
<td>20</td>
<td>23</td>
<td>2 (9%)</td>
<td>3 (15%)</td>
<td>17</td>
<td>9 (53%)</td>
<td>12 (60%)</td>
</tr>
<tr>
<td>openjpeg</td>
<td>17</td>
<td>17</td>
<td>1 (6%)</td>
<td>0 (0%)</td>
<td>9</td>
<td>4 (44%)</td>
<td>12 (71%)</td>
</tr>
<tr>
<td>libav</td>
<td>14</td>
<td>14</td>
<td>1 (7%)</td>
<td>0 (0%)</td>
<td>4</td>
<td>2 (50%)</td>
<td>12 (86%)</td>
</tr>
<tr>
<td>libming</td>
<td>14</td>
<td>14</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>3</td>
<td>1 (33%)</td>
<td>12 (86%)</td>
</tr>
<tr>
<td>mupdf</td>
<td>52</td>
<td>53</td>
<td>2 (4%)</td>
<td>1 (2%)</td>
<td>34</td>
<td>13 (38%)</td>
<td>31 (60%)</td>
</tr>
<tr>
<td>podofo</td>
<td>52</td>
<td>53</td>
<td>1 (2%)</td>
<td>2 (4%)</td>
<td>25</td>
<td>11 (44%)</td>
<td>38 (73%)</td>
</tr>
<tr>
<td>lrzip</td>
<td>39</td>
<td>39</td>
<td>0 (0%)</td>
<td>5 (13%)</td>
<td>30</td>
<td>3 (10%)</td>
<td>12 (31%)</td>
</tr>
<tr>
<td>zziplib</td>
<td>36</td>
<td>36</td>
<td>2 (6%)</td>
<td>0 (0%)</td>
<td>14</td>
<td>4 (29%)</td>
<td>26 (72%)</td>
</tr>
</tbody>
</table>

- **ProFuzzer:** 5.3% FP, 4.6% FN
- **AFL-analysis:** 42.7% FP, 69.6% FN
# Finding Zero-day Vulnerabilities

<table>
<thead>
<tr>
<th>Category</th>
<th>Product</th>
<th>SLOC</th>
<th>Bugs</th>
<th>CVEs</th>
<th>Fixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>exiv2</td>
<td>131,993</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>graphicsmagick</td>
<td>299,186</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>libtiff</td>
<td>82,484</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>openjpeg</td>
<td>164,284</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Audio &amp; Video</td>
<td>libav</td>
<td>703,369</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>libming</td>
<td>72,747</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PDF</td>
<td>mupdf</td>
<td>102,824</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>podofo</td>
<td>78,195</td>
<td>6</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Compression</td>
<td>lrzip</td>
<td>19,098</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>zziplib</td>
<td>12,898</td>
<td>8</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>10</strong></td>
<td><strong>1,667,078</strong></td>
<td><strong>42</strong></td>
<td><strong>30</strong></td>
<td><strong>27</strong></td>
</tr>
</tbody>
</table>
### Evaluation on Standard Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Location</th>
<th>Reaching Time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ProFuzzer</td>
</tr>
<tr>
<td>guetzli</td>
<td>output_image.cc:398</td>
<td>0.83</td>
</tr>
<tr>
<td>json</td>
<td>fuzzer-..._json.cpp:50</td>
<td>0.05</td>
</tr>
<tr>
<td>lcms</td>
<td>cmsintrp.c:642</td>
<td>0.67</td>
</tr>
<tr>
<td>libarchive</td>
<td>archive_..._warc.c:537</td>
<td>1.34</td>
</tr>
<tr>
<td>libjpeg</td>
<td>jdmaker.c:659</td>
<td>11.68</td>
</tr>
<tr>
<td>libpng</td>
<td>png.c:1035</td>
<td>1.84</td>
</tr>
<tr>
<td></td>
<td>pngread.c:757</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>pngrutil.c:1393</td>
<td>7.63</td>
</tr>
<tr>
<td>vorbis</td>
<td>codebook.c:479</td>
<td>T/O</td>
</tr>
<tr>
<td></td>
<td>codebook.c:407</td>
<td>T/O</td>
</tr>
<tr>
<td></td>
<td>res0.c:690</td>
<td>11.76</td>
</tr>
</tbody>
</table>

- ProFuzzer reaches more target locations than other fuzzers
- ProFuzzer is 2.26 to 8.85 times faster than other fuzzers
Performance

- ProFuzzer archives 27% ~ 227% more path coverage than other fuzzers
- ProFuzzer spends 53% ~ 79% less time to reach the same coverage
- ProFuzzer keeps relatively high effective mutation ratio

Comparison on Path Coverage

Comparison on Effective Mutation Ratio
Closely Related Works

• Input structure reverse engineering
  • Tupni [CCS 08]: identify input bytes relations via symbolic execution
  • Reward [NDSS 10]: propagates program type through syscalls and instructions
  • Howard [NDSS 11]: analyze memory access patterns during program execution

• Field-aware fuzzing
  • Steelix [FSE 17] infers magic value bytes by intercepting string comparisons
  • TIFF [ACSAC 18] infers program type (e.g., int, string) via taint analysis
  • Angora [S&P 18] infers shape and size of input bytes via taint analysis

• Difference:
  • ProFuzzer adopts lightweight mechanism instead of heavyweight analysis
  • ProFuzzer infers application-agnostic and fuzzing-related types
Conclusion

• Leverage on-the-fly type learning to improve fuzzing
  • Probe input fields and types by observing the fuzzing process
  • *Explore* valid values for better code coverage
  • *Exploit* the values that could lead to an vulnerability

• Results:
  • Better performance on code coverage and vulnerability exposure
  • 42 zero-day vulnerabilities, 30 of which are assigned CVEs
Thank you!

Q&A