No Linux, No Problem:
Fast and Correct Windows Binary Fuzzing via Target-embedded Snapshotting

Leo Stone
leo@vt.edu

Rishi Ranjan
rishiranjan@vt.edu

Stefan Nagy
snagy@cs.utah.edu

Matthew Hicks
mdhicks2@vt.edu
Introduction

- **Fuzzing**: most successful method for automated software testing
- Attempt to break program using **many** randomized inputs
Limiting factor

取决于速度：更快的模糊测试 = 更有效的模糊测试
Motivation

● On Linux, execution is very fast ➔ fuzzing is effective
  ○ fork()
  ○ Kernel modifications

● No equivalents on Windows...
  ○ Windows software ecosystem is larger, but fuzzing is orders of magnitude less effective than on Linux!

● Challenge: without kernel support, how can we build an efficient Windows fuzzer?
One Solution: Persistent Mode

- **LOAD**
  - Fuzzer
  - Target

- **INIT**
- **Body**
- **Return**

- **LOOP & EXECUTE**

- **JUMP**

<table>
<thead>
<tr>
<th>Fast, but incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution changes state...</td>
</tr>
<tr>
<td>Many binaries crash!</td>
</tr>
</tbody>
</table>
Our Approach

1. LOAD

Fuzzer
 Target

2. SAVE STATE

Copy state to a snapshot
Process restores own state
Fast like persistent mode, but correct!

3. LOOP & EXECUTE

Persistent
 INIT

Body

Return

4. RESET STATE

Key Insight
All relevant state is controllable with language-level constructs
No kernel support necessary
Implementation - WinFuzz

- Based on Winnie, an existing Windows fuzzer
- Main steps:
  1. DLL injection
  2. DLLMain() - hook setup
  3. Initializing target
  4. Taking state snapshot
  5. Main fuzzing loop: run and restore state
<table>
<thead>
<tr>
<th>Elements of Program State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>Registers</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Global variables</td>
</tr>
</tbody>
</table>
### 1. Snapshot

```
savedBP = BP
savedSP = SP
```

Stack

```plaintext
Local var
...
Local var
Local var
Free
...
Free
Free
```

### 2. Target runs

```
Local var
Local var
Local var
Local var
Free
...
Free
```

### 3. Stack restored

```
BP = savedBP
SP = savedSP
```

Stack

```plaintext
Local var
Local var
Free
...
Free
```
Resetting Registers

- 2 types of parameters:
  - Simple values
  - Pointers
- We reset each register to its original value
- We avoid returning from the target function and destroying any stack frames that could hold target parameters
1. **Snapshot**
   - CPU
     - RSP: a
     - RBP: b
     - RAX: c
     - RCX: d

2. **Target runs - registers changed**
   - `main(RBP-8, RBP-12)`
     - RCX = 12
   - `func1(RCX)`
     - RAX = 42
     - RDX = 'n'
   - `func2(RDX)`
     - RAX = 0

3. **State restored**
   - CPU
     - RSP: a'
     - RBP: b'
     - RAX: c'
     - RCX: d'

   - ...
Resetting the Heap

- Memory leaks are expected - we’re trying to break the program!
- Small memory leak can cause a fuzzer crash
- We use heap API hooks to prevent memory leaks
1 Initialization

- Heap
- Chunk list

2 Target runs

- a. Live pointers tracked
  - main(int, char*)
  - $a = \text{malloc}(64)$
  - $\text{func1}(\text{int})$
  - $b = \text{calloc}(512, 4)$

- Heap
  - $a$
  - $b$

- Chunk list
  - &a
  - &b

3 Remaining chunks freed

- Heap
  - $a$
  - $b$
  - $c$

- Chunk list
  - &a
  - &b
  - &c

- c = realloc(a, 8192)
- func2(buf)
- free(b)
- return 0

- Chunk list
- Heap
- &c
Resetting Global Memory

- Copy correct starting state of all mutable global sections
- Use guard pages to track modifications
- Only restore *modified* pages
1. Snapshot

Target
.rdata
Page 1
Page 2
.data
Page 1
Page 2
Page 3
Page 4

2. Target runs

Target
.rdata
Page 1
Page 2
.data
Page 1
Page 2
Page 3
Page 4

3. Modified pages restored

Target
.rdata
Page 1
Page 2
.data
Page 1
Page 2
Page 3
Page 4
Evaluation

- Criteria: correctness, performance, bug discovery
  - Versus state-of-the-art: Winnie (custom forkserver) and WinAFL (process creation/persistent mode)

- Setup: Azure instances running Windows 10 Pro with single-core 2.1 GHz Intel Xeon CPUs, 3.5 GB RAM

- Each fuzzing trial ran at least 5 times to collect statistically significant results
  - Mann-Whitney u-tests used to determine significance
### Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>WINFUZZ</th>
<th>Winnie</th>
<th>WinAFL</th>
<th>Source</th>
<th>File Format</th>
<th>Size (KB)</th>
<th>Basic Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>tar</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>Proprietary</td>
<td>.tar</td>
<td>606</td>
<td>30758</td>
</tr>
<tr>
<td>nconvert</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>Proprietary</td>
<td>.png</td>
<td>2458</td>
<td>91550</td>
</tr>
<tr>
<td>freetype</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>Open Source</td>
<td>.ttf</td>
<td>482</td>
<td>20891</td>
</tr>
<tr>
<td>audiofile</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>Open Source</td>
<td>.wav</td>
<td>45</td>
<td>1504</td>
</tr>
<tr>
<td>flac</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>Open Source</td>
<td>.flac</td>
<td>686</td>
<td>19292</td>
</tr>
<tr>
<td>nanosvg</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>Open Source</td>
<td>.svg</td>
<td>47</td>
<td>1966</td>
</tr>
<tr>
<td>sqlite</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>Open Source</td>
<td>.db</td>
<td>802</td>
<td>46758</td>
</tr>
<tr>
<td>smpdf</td>
<td>✔</td>
<td>No cov</td>
<td>❌</td>
<td>Proprietary</td>
<td>.pdf</td>
<td>3379</td>
<td>39816</td>
</tr>
<tr>
<td>irfanview</td>
<td>✔</td>
<td>No cov</td>
<td>✔</td>
<td>Proprietary</td>
<td>.png</td>
<td>1946</td>
<td>55187</td>
</tr>
<tr>
<td>jq</td>
<td>✔</td>
<td>No cov</td>
<td>❌</td>
<td>Open Source</td>
<td>.json</td>
<td>2662</td>
<td>13965</td>
</tr>
</tbody>
</table>
Correctness Test

● New fuzzer mode that checks for state corruption by comparing program states
● Used to test all benchmarks in corpus
● Available as part of our open source implementation
  ○ New users can test their own targets and saved inputs
1. Run

2. Snapshot

3. Parse many inputs

4. Run

5. Snapshot

6. Compare

Expected State

Actual State
Results: Throughput

Average improvement: 7x vs. Winnie and 182x vs. WinAFL
Results: Edge Coverage

Versus Winnie

Average improvement: 15% vs. Winnie and 5% vs. WinAFL

* $p=0.148$

Versus WinAFL

* $p=0.216$
Bug Discovery Time

- We compared the average time taken to find specific bugs
- Some benchmarks (flac, nanosvg, audiofile) used older versions to increase bug count

- 10 unique bugs found across all fuzzers/trials
  - Winnie: 3/10
  - WinAFL: 3/10
  - WinFuzz: 8/10
<table>
<thead>
<tr>
<th>Binary</th>
<th>Category</th>
<th>WINFUZZ</th>
<th>Winnie</th>
<th>WinAFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>flac</td>
<td>Null ptr deref</td>
<td>12.25 s</td>
<td>15.6 s</td>
<td></td>
</tr>
<tr>
<td>nconvert</td>
<td>Illegal address</td>
<td>2.1 hrs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nconvert</td>
<td>Invalid free</td>
<td>3.6 hrs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nconvert</td>
<td>Invalid ptr deref</td>
<td>7.2 hrs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nconvert</td>
<td>Heap overflow</td>
<td>8.5 hrs</td>
<td>15.8 hrs</td>
<td></td>
</tr>
<tr>
<td>nconvert</td>
<td>Illegal address</td>
<td></td>
<td>4.5 hrs</td>
<td></td>
</tr>
<tr>
<td>nanosvg</td>
<td>Stack overflow</td>
<td>1.4 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nanosvg</td>
<td>Null ptr deref</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nanosvg</td>
<td>Null ptr deref</td>
<td></td>
<td></td>
<td>21.9 hrs</td>
</tr>
<tr>
<td>audiofile</td>
<td>Illegal Address</td>
<td>12 min</td>
<td></td>
<td>5.2 hrs</td>
</tr>
</tbody>
</table>

WINFUZZ’s speedup 1.56x 23x
Undiscovered Bugs

- We ran additional experiments with WinFuzz to find 0-day bugs
- All bugs were reported to authors

<table>
<thead>
<tr>
<th>Binary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nconvert</td>
<td>2 invalid ptr reads, 3 invalid ptr writes</td>
</tr>
<tr>
<td>audiofile</td>
<td>Infinite loop</td>
</tr>
<tr>
<td>jhead</td>
<td>Invalid ptr read</td>
</tr>
<tr>
<td>flvmeta</td>
<td>2 invalid ptr reads</td>
</tr>
<tr>
<td>gpmf-parser</td>
<td>1 invalid ptr read, 1 invalid ptr write</td>
</tr>
<tr>
<td>gpmf-parser</td>
<td>Invalid ptr write</td>
</tr>
<tr>
<td>pdf2json</td>
<td>Stack buffer overrun (ntdll.dll)</td>
</tr>
<tr>
<td>pdf2json</td>
<td>Stack buffer overrun (pdf2json.exe)</td>
</tr>
<tr>
<td>pdf2json</td>
<td>Stack overflow</td>
</tr>
</tbody>
</table>

Total 0-day bugs: 9
Thank you!

Q&A

(Open source release pending)

gitHub.com/FoRTE-Research