Week 3B: Introduction to Fuzzing

Stefan Nagy

University of Utah
Reminders

- Paper presentations start next week

- **Audience:** not required to read the paper
  - Required to participate in discussion

- **Presenters:** your job is to **teach us the paper**
  - Contextualize
  - Pros vs. cons
  - Contribution
  - Summarizing
  - Identify assumptions
  - **Prepare & submit a short slide deck** (get “inspired” from existing presentations)
  - **10 – 20 minute presentation** (with 10 minute audience discussion to follow)
Questions?
Background
Programs run on inputs

- Modern applications accept many sources of input:
  - Files
  - Arguments
  - Environment variables
  - Network packets
  - ...

- Nowadays: multiple sources of inputs
Software Bugs
Software Bugs
When bugs go bad

- Improper input validation leads to **security vulnerabilities**
  - Bugs that violate the system’s confidentiality, integrity, or availability

- **Exploitation**: leveraging a vulnerability to perform unauthorized actions
Exploitation

Common Vulnerabilities
- Missed initialization check
- Free’d pointers not NULL’d
- Unchecked memory writes

Consequences
- Use uninitialized memory
- Use non-owned memory
- Overflowing a data buffer

Attacker Exploitation
- Software denial of service
- Leak sensitive information
- Inject & run arbitrary code

Race against time to find & fix vulnerabilities before they are exploited
With so many vulnerabilities today...

Source: cvedetails.com
... exploits are getting more and more sophisticated

- **1997**: Function ptr hijacking
- **1997**: Ret-2-Libc attacks
- **1996**: Stack overflows
- **1972**: First known overflows
- **1998**: Heap overflows
- **1999**: Format strings
- **2002**: Integer overflows
- **2005**: Heap grooming
- **2005**: Ret oriented programming
- **2002**: ASLR bypasses
- **2007**: Null pointer dereference
- **2007**: Double frees
- **2007**: Heap spraying
- **2010**: JIT spraying
- **2009**: Heap overflows
- **2015**: Double frees
- **2016**: Data oriented programming
- **2014**: Call oriented programming
- **2010**: Heap grooming
- **2007**: StackGuard bypasses
- **2005**: Format strings
- **2002**: Integer overflows
- **2005**: Ret oriented programming
- **2002**: ASLR bypasses
- **2007**: Double frees
- **2007**: Heap spraying
- **2010**: JIT spraying
- **2011**: Jmp oriented programming
- **2011**: Double frees
- **2021**: Zero-click exploits
- **2016**: Data oriented programming
- **2014**: Call oriented programming
- **2010**: Heap grooming
- **2007**: StackGuard bypasses
- **2005**: Format strings
- **2002**: Integer overflows
- **2005**: Ret oriented programming
- **2002**: ASLR bypasses
- **2007**: Double frees
- **2007**: Heap spraying
- **2010**: JIT spraying
- **2011**: Jmp oriented programming

What's next?
Proactive Vulnerability Discovery

Static Analysis:
- Analyze program **without running it**
- Accuracy a major concern
  - **False negatives** (vulnerabilities missed)
  - **False positives** (results are unusable)
- As code size grows, **speed drops**

Dynamic Testing:
- Analyze program **by executing it**
- Better accuracy: **no false positives**
  - Execution reveals only what exists
  - Program crashed? You found a bug!
- Capable of very **high throughput**
Proactive Vulnerability Discovery

Static Analysis:
- Analyze program **without running it**
- Accuracy a major concern
  - False negatives (vulnerabilities missed)
  - False positives (results are unusable)
- As code size grows, speed drops

Dynamic Testing:
- Analyze program **by executing it**
- Better accuracy: **no false positives**
  - Execution reveals only what exists
  - Program crashed? You found a bug!
- Capable of very **high throughput**
Fuzzing
One dark and stormy night...
in the era of dial-up internet

One dark and stormy night...
in the era of dial-up internet

One dark and stormy night…
in the era of dial-up internet

- Shouldn’t programs do much better with glitched or invalid input?

Bart’s idea: test programs on *random* inputs

<table>
<thead>
<tr>
<th>Listing 1</th>
<th>Simple Fuzzer in Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>import random</td>
<td></td>
</tr>
<tr>
<td>def fuzzer(max_length=100, char_start=32, char_range=32):</td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;&quot;Generate a string of up to <code>max_length</code> characters</td>
<td></td>
</tr>
</tbody>
</table>
| in the range `[`char_start` `, `char_start` + `char_range` - 1`"""
| string_length = random.randint(0, max_length + 1) |
| out = "" |
| for i in range(0, string_length): |
| out += chr(random.randint(char_start, char_start + char_range)) |
| return out |

```python
!7%"*#0=)$;%6*;>638:*">80"=</>(/*
:-(2<4 !:5*6856&?"""11<7+%<%7,4.8+
```
Bart’s idea: test programs on *random* inputs

- Quickly generate lots and lots of *random inputs*
- Execute each on the target program
- See what happens
  - Crash
  - Hang
  - Nothing at all
Random inputs work!

- Crash or hang 25–33% of utility programs in seven UNIX variants
- Results reveal several common mistakes made by programmers
- They called this fuzz testing
  - Known today as fuzzing

An Empirical Study of the Reliability of UNIX Utilities

Barton P. Miller
bart@cs.wisc.edu

Lars Fredriksen
L.Fredriksen@att.com

Bryan So
so@cs.wisc.edu
The Evolution of Fuzzing
Fuzzing like it’s 1989

- Random inputs

- **Black-box:** only check program’s end result
  - Signals
  - Return values
  - Program-specific output

- Save inputs that trigger **weird behavior**
  - SIGSEGV, SIGFPE, SIGILL, etc.
  - Assertion failures
  - Other reported errors
Finding Bugs with Fuzzing

Source: https://blog.trailofbits.com/2020/10/22/lets-build-a-high-performance-fuzzer-with-gpus/
Black-box fuzzing only gets you so far
How can fuzzing exploration be guided?

- Idea: track some measure of exploration “progress”
  - Coverage of program code
  - Stack traces
  - Memory accesses

- Pinpoint inputs that further progress over the others

- Mutate only those inputs
Code Coverage

- **Code coverage**: program regions reached by each test case

- Horse racing analogy: breed only the winning inputs
  - New coverage? **Keep the input**
  - Old coverage? **Discard it**

```javascript
function fib(n) {
  if (n === 0) {
    return 0
  } else if (n === 1) {
    return 1
  } else if (n > 1) {
    return fib(n - 1) + fib(n - 2)
  } else {
    thrower()
  }
}

console.log('fib(10):', fib(10))
```
Coverage-guided Fuzzing

Inputs

Program

Trace code coverage, Monitor execution

New coverage

Triggers bugs

No new coverage
Coverage-guided Fuzzing

1) Run the seed input through the program to produce a CFG

2) Mutate the input, test the new inputs, and look for changes in the CFG

3) Rinse and repeat!

Source: https://blog.trailofbits.com/2020/10/22/lets-build-a-high-performance-fuzzer-with-gpus/
Modern Fuzzing
Fuzzing across the industry

- Fuzzing = today’s most popular bug-finding technique
  - Most real-world fuzzing is coverage-guided

Google: We've open-sourced ClusterFuzz tool that found 16,000 bugs in Chrome

New fuzzing tool finds 26 USB bugs in Linux, Windows, macOS, and FreeBSD
Taxonomy of Fuzzers

Fuzzer component

Test case generation
- Grammar-based
  - dharma [13]
  - gramfuzz [14]
  - Peach [15]
- Mutational
  - Directed
    - TaintScope [16]
  - Coverage-guided
    - AFL [5]
    - honggfuzz [4]
    - libFuzzer [6]
    - VUzzer [7]

Execution monitoring
- Black-box
  - Autodafe [17]
  - dharma [13]
  - Peach [15]
- White-box
  - Driller [18]
  - QSYM [19]
  - KLEE [20]
  - Mayhem [21]
  - S2E [22]
  - SAGE [23]
  - TaintScope [16]
- Grey-box
  - AFL [5]
  - honggfuzz [4]
  - libFuzzer [6]
  - VUzzer [7]
  - TriforceAFL [24]
Tools of the trade: AFL

- Most historically significant fuzzer ever developed

- Authors: Michal Zalewski (2013)
  - Google (2019–2022)
  - The AFL++ team (2020–onwards)

- Versatile, easy to spin up & modify
  - Spawned probably ~100 PhD & MS theses
  - (mine included)

- Mix of carefully chosen trade-offs
What AFL aims to be

- **Primary goal:** high test case throughput
- **Sacrifice precision in most areas**
  - Lightweight, simple mutators
  - Coarse, approximated code coverage
  - Little reasoning about seed selection
- **Revolutionary & still insanely effective**
  - Ideas ported over to honggFuzz, libFuzzer
  - and nearly all other fuzzers
Tools of the trade: AFL++

- **By far today’s most popular fuzzer**

- **Official successor to vanilla AFL**
  - Started out as a community-led fork
  - Google has since archived vanilla AFL

- **A platform for trying-out new features**
  - Integrated lots of academic prototypes
  - Easily tailorable to your target’s needs

https://github.com/AFLplusplus/AFLplusplus
Trade-offs are target-dependent

Building a good fuzzer is all about finding the right balance of performance & precision.
Any fuzzing is better than not fuzzing

If something has not been fuzzed before, *any* fuzzing will probably find *lots* of bugs.
Questions?
Lab 1
Lab 1: Beginner Fuzzing

- **Assignment:** familiarize yourself with AFL++
  - Read its documentation in `docs/`

- **Pick three features, try them out, and discuss your findings**
  - E.g., impacts on code coverage, speed, crash discovery
  - What insights do you have?
  - Why did one feature work better than another?

- **Deliverable:** a 1–3 page report detailing your findings
  - Feel free to make it your own (e.g., pictures, text, etc.)

- Need a Linux environment
  - See me if you don’t have one
Lab 1: Introduction to Fuzzing

- **Primary goal:** *prepare you for the semester project*

- **Other goals:**
  - Give you experience with industry-standard tools
  - Put you in the “research” mindset
  - Improve your debugging skills
Questions?