Week 3: Lecture B

Introduction to Fuzzing

Wednesday, January 24, 2024
Recap: Paper Presentations

- Two paper presentations per lecture, followed by 5–10 minute discussions

- **Audience:** you are **not** required to read the paper
  - ... but you are required to **participate** in the discussion!

- **Presenters:** your job is to **teach us the paper**
  - Summarizing
  - Contextualize
  - Pros vs. cons
  - Contributions
  - Key assumptions
  - **Prepare a short slide deck** (you can get “inspired” from existing presentations)
  - **15 – 20 minute presentation** (with a 5–10 minute audience discussion to follow)
Recap: Hands-on Labs

- Three (relatively easy) labs to be completed solo
  - Lab 1: Beginner fuzzing
  - Lab 2: Crash triage
  - Lab 3: Target harnessing

- Paced with the introductory content from Weeks 4–9
  - Apply the techniques you’ve learned in class
  - Get familiar with state-of-the-art tools like AFL and ASAN
  - Deliverables: a short report (1–3 pages) of what you’ve learned

- Designed to prepare you for the Semester Final Project
Recap: Key Dates

- **Jan. 24**  
  Lab 1 released

- **Feb. 07**  
  Lab 1 due

- **Feb. 14**  
  Lab 2 due

- **Feb. 19**  
  No class (President’s Day)

- **Feb. 28**  
  Lab 3 due

- **Feb. 28**  
  5-minute project proposals

- **Mar. 04 & 06**  
  No class (Spring Break)

- **Apr. 17 & 22**  
  Final project presentations

*cs.utah.edu/~snagy/courses/cs5963/schedule*
Questions?
Background
Programs and 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
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...

- Denial of Service: 25%
- Code Execution: 13%
- Overflow: 13%
- Cross Site Scripting: 16%
- Directory Traversal: 8%
- Bypass Something: 6%
- Gain Information: 4%
- Gain Privilege: 3%
- Memory Corruption: 3%
- SQL Injection: 2%
- File Inclusion: 2%
- Cross Site Request Forgery: 0%
- HTTP Response Splitting: 0%

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

1998
StackGuard bypasses

1999
Format strings

2002
Integer overflows

2005
Heap grooming

2005
Ret oriented programming

2002
ASLR bypasses

2002
Integer overflows

2005
Hardware DEP bypasses

2007
Null pointer dereference

2007
Double frees

2007
Heap spraying

2009
Heap spraying

2010
JIT spraying

2011
Jmp oriented programming

2014
Call oriented programming

2016
Data oriented programming

2021
Zero-click exploits

2005
Format strings

2010
ASLR bypasses

2011
JIT spraying

2014
Hardware DEP bypasses

2016
Double frees

2021
Null pointer dereference

2021
ASLR bypasses

2005
StackGuard bypasses

2007
Ret oriented programming

2007
Integer overflows

2010
Null pointer dereference

2021
Function ptr hijacking

2014
StackGuard bypasses

2021
Ret-2-Libc attacks

2021
First known overflows

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!

**Listing 1  Simple Fuzzer in Python**

```python
import random
def fuzzer(max_length=100, char_start=32, char_range=32):
    """Generate a string of up to `max_length` characters in the range `[char_start, char_start + char_range - 1]`""
    string_length = random.randrange(0, max_length + 1)
    out = ""
    for i in range(0, string_length):
        out += chr(random.randrange(char_start, char_start + char_range))
    return out
```

!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

The space of possible program behaviors

Detective Fuzz

Badly behaved edge cases

The Land of Expected Behavior

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
Coverage-guided Fuzzing

Inputs

Program

Execute and Collect Feedback
(e.g., code coverage)
Coverage-guided Fuzzing

Inputs → Program → Execute and Collect Feedback (e.g., code coverage) → Interesting!

(new code)
Coverage-guided Fuzzing

Inputs

Program

Execute and Collect Feedback
(e.g., code coverage)

Interesting!
(new code)

Uninteresting
(no new code)
Coverage-guided Fuzzing

Inputs

Program

Execute and Collect Feedback
(e.g., code coverage)

Interesting!
(new code)

Crashes
(SEGFAULT)

Uninteresting
(no new code)
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 in 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]

Black-box
- Autodafe [17]
- dharma [13]
- Peach [15]

Execution monitoring

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** 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
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: Beginner Fuzzing
Lab 1: Beginner Fuzzing

- See Assignments tab on course website
  - Click the drop-down link for Lab 1

- Deadline: Wednesday, February 7
  - Submit on Canvas by 11:59 PM MST
  - Late assignments are not accepted

Lab Exercises (collected via Canvas)

Instructions: There will be three introductory fuzzing exercises that will count for 45% of your course grade (15% each). Unless otherwise indicated, you must work alone. You may consult general reference material, but you may not collaborate with other students. The material you turn in must be entirely your own work, and you are bound by the Student Code.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Deadline (by 11:59PM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lab 1: Beginner Fuzzing</td>
<td>Wednesday, February 7</td>
</tr>
</tbody>
</table>

Overview: In preparation for the semester course project, this lab will familiarize you with AFL++ (the world’s most popular and extensible fuzzing platform).

Your task: Select three of AFL++’s user-configurable features and evaluate their impacts on fuzzing:
- What led you to explore these fundamental features and why?
- How do these features impact speed, coverage, and crash discovery?
- Do certain features work better in tandem, or individually?
- Do these features perform as you expected, or unexpectedly?

Other Notes:
- Information on AFL++’s available features can be found in its documentation.
- Linux is recommended. You are welcome to use the Lubuntu VM from CS 4440.
- For issues troubleshooting AFL++, you can ask for help on the Course Plaza, or reach its authors via GitHub or the fuzzplusplus-issues-questions channel in the Awesome Fuzzing Discord. It is recommended that you start early.

Recommended Readings:
- AFL++ Combining Incremental Steps of Fuzzing Research.
- Dissecting American Fuzzy Lop: A FuzzBench Evaluation.

What to Submit:
Submit a 1-3 page report detailing your experimental findings. There are no “right” or “wrong” answers—your work will be assessed by your overall effort. You have full creative liberty—feel free to use images, tables, etc.
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**
  - Use the [CS 4440 Lubuntu VM](#) 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?