Week 6: Lecture B
Automated Bug Finding

Thursday, September 28, 2023
Announcements

- **Project 2: AppSec** released
  - **Deadline:** Thursday, October 19th by 11:59PM

---

**Project 2: Application Security**

**Deadline:** Thursday, October 19 by 11:59PM.

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

You may optionally work alone, or in teams of at most two and submit one project per team. If you have difficulties forming a team, post on Piazza's Search for Teammates forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). **Don't risk your grade and degree by cheating!**

Complete your work in the CS 4440 VM—we will use this same environment for grading. You may not use any external dependencies. Use only default Python 3 libraries and/or modules we provide you.

**Helpful Resources**
- The CS 4440 Course Wiki
- VM Setup and Troubleshooting
- Terminal Cheat Sheet
- GDB Cheat Sheet

**Table of Contents:**
- Helpful Resources
- Introduction
- Objectives
- Start by reading this!
  - Setup Instructions
  - Important Guidelines
- Part 1: Beginner Exploits
  - Target 0: Variable Overwrite
  - Target 1: Execution Redirect
  - What to Submit
- Part 2: Intermediate Exploits
  - Target 2: Shellcode Redirect
  - Target 3: Indirect Overwrite
  - Target 4: Beyond Strings
  - What to Submit
- Part 3: Advanced Exploits
  - Target 5: Bypassing DEP
  - Target 6: Bypassing ASLR
  - What to Submit
- Part 4: Super L33T Pwnage
  - Extra Credit: Target 7
  - Extra Credit: Target 8
  - What to Submit
- Submission Instructions
Project 2 Progress Update!

Finished Target 0! 0%
Finished Target 1! 0%
Finished Target 2! 0%
Finished Target 3! 0%
Finished Target 4! 0%
Haven't started :( 0%
Announcements

- **Project 1** grades are now available on Canvas

- **Statistics:**
  - Average score: 85%

- **Fantastic job!**
Announcements

- **Project 1** grades are now available on **Canvas**

- Think we made an error? Request a regrade!
  - Valid regrade requests:
    - You have verified your solution is correct (i.e., we made an error in grading)

---

**Project 1 Regrade Requests** (see [Piazza](#) pinned link):
Submit by **11:59 PM on Monday 10/02 via Google Form**
See Discord for meeting info!

www.utahsec.com
Questions?
Last time on CS 4440...

Advanced Exploitation Techniques
ASLR, DEP, and Workarounds
Other Application-level Defenses
Recap: Spawning Shells

- **Attacker goal:** make program open a **root shell**
  - Root-level permissions = **total system ownage**
  - You’ll do this in **Project 2!**

- **Shellcode** = code to open a root shell
  - Inject this somewhere and **direct execution to it**
  - Basic structure:
    1. Call `setuid(0)` to set user ID to “root”
    2. Open a shell with `execve("/bin/sh")`
#include <stdio.h>

void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}

Shell inherits same **privileges** as the original “parent” process

If the original process **run as root**, shell gives ???? access
Shell Spawning in C

#include <stdio.h>

void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
Shell Spawning in C

```c
#include <stdio.h>

void main() {
    char *argv[1];
    argv[0] = "/bin/sh";
    execve(argv[0], NULL, NULL);
}
```

Invoking a Shell

main:
pushl %ebp
movl %esp, %ebp
pushl $0
pushl $0
pushl $.LC0
call execve
leave
ret

main()’s locals

????????????????????????

????????????????????

????????????????????
main:
    pushl  %ebp
    movl  %esp, %ebp
    pushl  $0
    pushl  $0
    pushl  $.LC0
    call  execve
    leave
    ret

.LC0:
    .string "/bin/sh"

main()'s locals

<table>
<thead>
<tr>
<th>arg3</th>
<th>= NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg2</td>
<td>= NULL</td>
</tr>
<tr>
<td>addr to</td>
<td>/bin/sh</td>
</tr>
</tbody>
</table>
main:
  pushl %ebp
  movl %esp, %ebp
  pushl $0
  pushl $0
  pushl $.LC0
  call    execve
  leave
  ret

.LC0:
  .string "/bin/sh"

execve("/bin/sh", NULL, NULL);

main()'s locals

arg3 = NULL
arg2 = NULL
addr to "/bin/sh"

execve()’s ret addr
Invoking a Shell

- **Project 2:** we give you shellcode to set up and call `execve(`/bin/sh`)`
  - This will initialize the correct call frame accordingly

- **Key idea:** ???

<table>
<thead>
<tr>
<th>Vulnerable Function’s RetAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Saved EBP, local vars, etc.</strong></td>
</tr>
<tr>
<td><strong>Vulnerable Buffer</strong></td>
</tr>
</tbody>
</table>
Invoking a Shell

- **Project 2:** we give you shellcode to set up and call `execve(`/bin/sh`)`
  - This will initialize the correct call frame accordingly

- **Key idea:** place the shellcode in an **executable buffer**
  - “Executable” means you are able to **execute code** inside of it
  - ... then direct execution to it, and **BOOM!**

<table>
<thead>
<tr>
<th>Start addr of buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Padding to reach RetAddr</td>
</tr>
<tr>
<td><strong>NOP, NOP, NOP, NOP, NOP, NOP, NOP</strong></td>
</tr>
<tr>
<td><strong>NOP, NOP, NOP, NOP, NOP, NOP, NOP</strong></td>
</tr>
<tr>
<td><code>setuid(0) + execve(</code>/bin/sh<code>)</code></td>
</tr>
</tbody>
</table>
Our provided shellcode requires an **executable buffer**

What if the buffer is **relocated** on every new run?

Start addr of buffer = ?????

Padding to reach RetAddr

WHERE?
Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of NOPs

- We cannot know buffer’s **exact start**...
  - But we can **guess an address inside of it**
    - It is a really large buffer, after all

- **Idea: ?????**

```c
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
setuid(0) + execve("/bin/sh")
```
Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**

- We cannot know buffer’s **exact start**...
  - But we can **guess an address inside of it**
    - It is a really large buffer, after all

- **Idea:** spam **“guessed” buffer addr** up the stack

```bash
setuid(0) + execve("/bin/sh")
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
```
Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**

- We cannot know buffer’s **exact start**...
  - But we can **guess an address inside of it**
    - It is a really large buffer, after all

- **Idea:** spam “**guessed**” **buffer addr** up the stack
  - Eventually we’ll overwrite some **return address**

```
setuid(0) + execve("/bin/sh")
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
NOP, NOP, NOP, NOP, NOP, NOP, NOP
```
Defeating ASLR

- Suppose the buffer is **sufficiently large**
  - We can still place our shellcode there
  - Prepend it with a ton of **NOPs**

- We cannot know buffer’s **exact start**...
  - But we can **guess an address inside of it**
    - It is a really large buffer, after all

- **Idea**: spam “**guessed**” buffer addr up the stack
  - Eventually we’ll overwrite some **return address**
  - When that function returns, jump inside buffer
  - **Hit the huge NOP sled → BOOM!**
Our provided shellcode requires an **executable buffer**

**What if the buffer is prohibited from being executable?**

<table>
<thead>
<tr>
<th>Start addr of buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Padding to reach RetAddr</td>
</tr>
<tr>
<td><strong>NOP, NOP, NOP, NOP, NOP, NOP, NOP</strong></td>
</tr>
<tr>
<td><strong>NOP, NOP, NOP, NOP, NOP, NOP, NOP</strong></td>
</tr>
<tr>
<td><strong>setuid(0) + execve(&quot;/bin/sh&quot;)</strong></td>
</tr>
</tbody>
</table>
Pesky Defenses

- Our provided shellcode requires an **executable buffer**

- **What if the buffer is prohibited** from being executable?

```plaintext
<table>
<thead>
<tr>
<th>Start addr of buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Padding to reach RetAddr</td>
</tr>
<tr>
<td>NOPE</td>
</tr>
</tbody>
</table>
```
Defeating DEP

- Suppose we can still overwrite buffer
  - We cannot place our shellcode there
  - But, we can overwrite other stack items

- Suppose the program calls a function that can execute arbitrary commands
  - execve()
  - system()

- Idea #1: overwrite ????
Defeating DEP

- Suppose we can still overwrite buffer
  - We cannot place our shellcode there
  - But, we can overwrite other stack items

- Suppose the program calls a function that can execute arbitrary commands
  - execve()
  - system()

- Idea #1: overwrite argument to system()
  - Replace it with our shell command ("/bin/sh")

```
main:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
pushl /bin/ls
call system
leave
ret
```

arg1 = "/bin/ls"

system()’s ret addr

Buffer (non-executable)
Defeating DEP

- Suppose we can still overwrite buffer
  - We **cannot** place our shellcode there
  - But, we can **overwrite other stack items**

- Suppose the program calls a function that can **execute arbitrary commands**
  - `execve()`
  - `system()`

- **Idea #1:** overwrite argument to `system()`
  - Replace it with our shell command ("/bin/sh")
  - Will now execute `system("/bin/sh")`!

```c
main:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
pushl "/bin/ls"
call system
leave
ret
```

```
arg1 = "/bin/sh"
```

```
AAAAAA
```

```
AAAAAAA
```
Defeating DEP

- Suppose we can still overwrite buffer
  - We cannot place our shellcode there
  - But, we can overwrite other stack items
- Suppose the program calls a function that can execute arbitrary commands
  - `execve()`
  - `system()`
- Idea #1: overwrite argument to `system()`
  - Replace it with our shell command (`"/bin/sh"`)
  - Will now execute `system("/bin/sh")`

```
main:
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
pushl   "/bin/ls"
call    system
leave
ret
```

```
arg1 = "/bin/sh"
```

`system()`'s ret addr

`AAAAAAAAAAAAAAAAAAAAAA`
Defeating DEP

- Suppose we can still overwrite buffer
  - We cannot place our shellcode there
  - But, we can overwrite other stack items

- Suppose the program calls a function that can execute arbitrary commands
  - execve()
  - system()

- Idea #1: Overwrite argument to system()
  - Replace it with our shell command ("/bin/sh")
  - Will now execute system("/bin/sh")!

- main:
  - pushl   %ebp
  - movl    %esp, %ebp
  - subl    $16, %esp
  - pushl   "/bin/ls"
  - call    system
  - leave
  - ret

- arg1 = "/bin/sh"
- system()'s ret addr
- AAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAA

- segmentation fault.
  (Core dumped)
Defeating DEP

- Suppose `system()` isn’t executed, but a **call to it exists somewhere**
  - You can examine the `objdump` to look for “interesting” functions in the program

```c
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}
void main() {
    char buf[256];
    memset(buf, 'A', 255);
    buf[255] = '\x00';
    foo(buf);
}
```

<table>
<thead>
<tr>
<th>previous frame ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AAAAAAAAA...\0</code></td>
</tr>
<tr>
<td><code>foo()</code>'s first arg</td>
</tr>
<tr>
<td><code>foo()</code>'s return addr</td>
</tr>
<tr>
<td><code>main()</code>'s frame ptr</td>
</tr>
</tbody>
</table>

Buffer (**non-executable**)
Defeating DEP

- **Idea #2**: create a ????

<table>
<thead>
<tr>
<th>previous frame ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAAAAAA...\0</td>
</tr>
<tr>
<td>foo()'s first arg</td>
</tr>
<tr>
<td>foo()'s return addr</td>
</tr>
<tr>
<td>main()’s frame ptr</td>
</tr>
<tr>
<td>Buffer (non-executable)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>previous frame ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAAAAAA...\0</td>
</tr>
<tr>
<td>foo()'s first arg</td>
</tr>
<tr>
<td>foo()'s return addr</td>
</tr>
<tr>
<td>main()’s frame ptr</td>
</tr>
<tr>
<td>Buffer (non-executable)</td>
</tr>
</tbody>
</table>
Idea #2: create a "fake" call frame for `system()` with our desired arg

- `system()`'s return addr
- `foo()`'s first arg
- `foo()`'s return addr
- `main()`'s frame ptr

Buffer (non-executable)
Defeating DEP

- Idea #2: create a "fake" call frame for `system()` with our desired arg

  ```
  string /bin/sh
  system()
  ```

  `system()`'s return addr

  Address of `system()`

  previous frame ptr

  foo()’s return addr

  foo()’s first arg

  main()’s frame ptr

  Buffer (non-executable)
What happens when `system()` returns (i.e., the spawned shell is closed)?

```
previous frame ptr
AAAAAAAAAAAAA...
\0
foo()'s first arg
foo()'s return addr
main()'s frame ptr
Buffer (non-executable)
```

```
string " /bin/sh"

Address of " /bin/sh"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Address of system()
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Buffer (non-executable)
What happens when `system()` returns (i.e., the spawned shell is closed)?

<table>
<thead>
<tr>
<th>previous frame ptr</th>
<th>string &quot;/bin/sh&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAAAAAAA...\0</td>
<td>Address of &quot;/bin/sh&quot;</td>
</tr>
<tr>
<td>foo()'s first arg</td>
<td>AAAAAAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
<tr>
<td>foo()'s return addr</td>
<td>Address of <code>system()</code></td>
</tr>
<tr>
<td>main()'s frame ptr</td>
<td>AAAAAAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
<tr>
<td>Buffer (non-executable)</td>
<td>AAAAAAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
</tbody>
</table>

The spawned shell is closed, and the control returns to the previous frame pointer.
Defeating DEP

- What happens when `system()` returns (i.e., the spawned shell is closed)?

```
string
/bin/sh
```

`system()`'s first arg

```
AAAAAAAAAAAAAAAAAAAAAA
```

Address of `system()`

```
AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA
```

Previous frame ptr

```
foo()'s first arg
foo()'s return addr
```

```
main()'s frame ptr
```

Buffer (non-executable)

```
returns to ????
```

**segmentation fault.**

(Core dumped)
Case Study: Drive-by-Downloads

1. **Embedded Malicious Elements**
   - User browses a legitimate website which has been hijacked.

2. **Exploit Kit**
   - Automatically downloads to PC.
   - Probes the system for vulnerabilities.

3. **Outdated Software**
   - Outdated software are most vulnerable and perfect target of exploit.

4. **Malware**
   - Malware pours through the security hole and takes over the system.
Case Study: Drive-by-Downloads

- **Web browser crashing** = a dead giveaway you’re being **exploited**!
Defeating DEP

- How can we make this stealthy (i.e., **not segfault when `system()` returns**)?

![Diagram showing the flow of control and data](image)

- Previous frame ptr
  - `AAAAAAAAAAAA...\0`
- `foo()`'s first arg
  - `system()`'s first arg
- `foo()`'s return addr
  - `system()`'s return addr
- Main()'s frame ptr
  - Address of `system()`
- Buffer (non-executable)

```string
"/bin/sh"
```
## Defeating DEP

- How can we make this stealthy (i.e., **not segfault when system() returns**)?
  - Replace the **return address** in our fake `system()` call frame with the **address of `_exit()`**

<table>
<thead>
<tr>
<th>previous frame ptr</th>
<th>string &quot;/bin/sh&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAAAAA...\0</td>
<td>Address of &quot;/bin/sh&quot;</td>
</tr>
<tr>
<td>foo()'s first arg</td>
<td>Address of <code>_exit()</code></td>
</tr>
<tr>
<td>foo()'s return addr</td>
<td>Address of <code>system()</code></td>
</tr>
<tr>
<td>main()'s frame ptr</td>
<td>AAAAAAAAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
<tr>
<td>Buffer (non-executable)</td>
<td>AAAAAAAAAAAAAAAAAAAAAAAAAAAAA</td>
</tr>
</tbody>
</table>

- **Buffer** (non-executable) returns to `_exit`
Defeating DEP

- How can we make this stealthy (i.e., not segfault when system() returns)?
  - Replace the return address in our fake system() call frame with the address of _exit()
Project 2 Tips

- **Targets 0, 1, 2**
  - Relatively simple attacks
  - Should not require too much effort
  - They build up your skills for the others!

- **Suggestion: get these finished ASAP**

- **Having trouble? Come to office hours**
  - See [CS 4440 Wiki](#) for cheat sheets!
### Project 2 Tips: Attack Planning

1. Establish a plan of attack
   - Draw a **before/after stack diagram**

2. What object do you **control**?
   - Vulnerable buffer

3. What objects are **adjacent** to it?
   - `main()`'s frame pointer
   - `foo()`'s return address

4. What do you need to **overwrite**?
   - `foo()`'s return address, etc.

<table>
<thead>
<tr>
<th>Stack that the program gives us</th>
<th>How we want the stack to look</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Pointer</td>
<td>“/bin/sh”</td>
</tr>
<tr>
<td>“whatever”</td>
<td>Arg1 Address</td>
</tr>
<tr>
<td>Arg1 Address</td>
<td>Ret Address</td>
</tr>
<tr>
<td>Ret Address</td>
<td>system()</td>
</tr>
<tr>
<td>Frame Pointer</td>
<td>AAAAAAAAAAAA</td>
</tr>
<tr>
<td>Buffer</td>
<td>AAAAAAAAAAAA</td>
</tr>
</tbody>
</table>
Project 2 Tips: Memory Inspection

1. Get familiar with **memory inspection** in GDB

2. Begin with simple, **easily-identifiable payload**
   - E.g., the string “AAAAAAAAAA...”

3. Set **breakpoint** on payload-inserting function
   - E.g., the function that calls `strcpy()`

4. Single-step to **right before function returns**

5. Inspect memory and **look for payload bytes**
   - At what address does `0x4141414141...` appear?

```
(gdb) x/32bx 0xfff6d8c0
0xfff6d8c0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xfff6d8c8: 0x41 0x41 0x41 0x41
0xfff6d8d0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
```

Buffer probably begins at `0xfff6d8c8 + 4`
Project 2 Tips: Overflowing

- **Segfaults** = you’re on the right track!
  - Means you’ve overwritten something of value
  - E.g., the current function’s return address

- Get a dummy “AAAA” payload down **first**
  - Are you overwriting the objects you want?
  - **How many bytes** do you need to do so?

- **Then** move onto your full shellcode attack
  - **Suggestion:** replace “A”s with \(0x90\)s (NOPs)

RetAddr **Partial Overwrite**
Program received signal SIGSEGV, Segmentation fault.
\(0x08004141\) in ?? ()

RetAddr **Full Overwrite**
Program received signal SIGSEGV, Segmentation fault.
\(0x41414141\) in ?? ()
Questions?
This time on CS 4440...

Automated Bug-Finding
Fuzz Testing
Symbolic Execution
Modern applications accept many sources of input:
- Files
- Arguments
- Environment variables
- Network packets
- ...

Nowadays: multiple sources of inputs
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
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**
Questions?
“Fuzz” Testing (aka Fuzzing)
One dark and stormy night…

One dark and stormy night...

One dark and stormy night...

- 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
```

```bash
!7%"*#0=)%;%6*;>638:*>80"=<//(*
:-c<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
Finding Bugs with Fuzzing

The space of possible program behaviors

Source: https://blog.trailofbits.com/2020/10/22/lets-build-a-high-performance-fuzzer-with-gpus/
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]

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++

- 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
Feedback-driven 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
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**
Feedback-driven Fuzzing
Feedback-driven Fuzzing

Inputs

Program

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

Inputs

Program

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

Interesting!
(new code)
Feedback-driven Fuzzing

Inputs

Program

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

Interesting! (new code)

Uninteresting (no new code)
Feedback-driven Fuzzing

Inputs

Program

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

Interesting! (new code)

Crashes (SEGFAULT)

Uninteresting (no new code)
Types of Feedback-driven Fuzzers

- **Black-box**
  - Zero Introspection

- **Grey-box**
  - Some Introspection

- **White-box**
  - High Introspection
Types of Feedback-driven Fuzzers

- Black-box: Zero Introspection
- Grey-box: Some Introspection
- White-box: High Introspection

ineffective to inefficient
Types of Feedback-driven Fuzzers

- Black-box: Zero Introspection
- Grey-box: Some Introspection
- White-box: High Introspection

 ineffective  

inefficient
Coverage-guided Grey-box Fuzzing

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

- **Horse racing analogy**: “breed” (mutate) only the “winning” (coverage-increasing) inputs
  - New coverage? **Keep and mutate the input**
  - Old coverage? **Discard it and try again**

- Most fuzzing today is **coverage-guided**
  - Good balance of performance and precision

```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);
  }
}

console.log('fib(10):', fib(10));
```
Code Coverage

- Program represented as **control-flow graphs (CFG)**
  - Directed graph encompassing all program paths
  - Basis of virtually all software analysis techniques

- Various coverage metrics in use today
  - **Instructions:** units that make up basic blocks
  - **Basic blocks:** nodes of the program’s CFG
  - **Edges:** transitions between basic blocks
  - **Hit counts:** frequencies of basic blocks
  - **Paths:** sequences of edges
Tracking Code Coverage

- **Challenge:** coverage-tracing **instrumentation**
  - Modifying program to track test case code coverage

- Target is **open-source? Easy and fast!**
  - Can **compile-in** coverage-tracing instrumentation

- Target is **closed-source? Difficult and slow!**
  - **Dynamic Translation:** modify executable **as it’s running**
    - Easy, but really slows down runtime speed
  - **Static Rewriting:** modify executable **before running it**
    - Conceptually similar to compiler instrumentation
    - Fast, but difficult to do without breaking the program
Questions?
Fuzzing Input Generation
Before you start: choose your seeds

- **Seeds:** starting inputs from which to mutate from

  - Small seeds
    - Smallest-possible PDF file
    - Empty file

  - Large seeds
    - Crawl web for every PDF ever created

- No right answer—it is target-dependent!
  - Smaller seeds = cover earlier code, but struggle to reach deeper code
  - Larger seeds = cover deeper code to start, but are slower to execute
Types of Input Generation

- **Model-agnostic**: brute-force your way to valid inputs
  - Random insertions, deletions, and splicing

- **Model-guided**: follow a pre-defined input specification
  - Follow “rules” to create highly-structured inputs

- **White-box approaches**:
  - **Symbolic execution**: solve branches as *symbolic* expressions
  - **Concolic execution**: solve branches as *concrete* values
  - **Taint tracking**: infer critical input “*parts*” and mutate those

Source: The Art, Science, and Engineering of Fuzzing: A Survey
Model-agnostic Generation

- Brute-force your way to valid inputs
  - Bit and byte “flipping”
  - Addition and subtraction
  - Inserting random chunks
  - Inserting dictionary “tokens”
  - Splicing two inputs together

- The good: super fast
  - Incorporating feedback like coverage enables you to **synthesize valid inputs** (eventually)
- **Surprisingly effective:** valid inputs appear out of thin air
Need a lot of luck to solve magic bytes checks and nested checksums


```c
if (u64(input) == u64("MAGICHDR"))
    bug(1);
```

Listing 3: Fuzzing problem (2): finding valid input to bypass checksums.

```c
if (u64(input) == sum(input + 8, len - 8))
    if (u64(input + 8) == sum(input + 16, len - 16))
        if (input[16] == 'R' && input[17] == 'Q')
            bug(2);
```
Model-guided Generation

- Follow a pre-defined input **specification**
  - Pre-defined input grammars
  - Dynamically-learned grammars
  - Domain-specific generators

- **The good:** many more valid inputs
  - Model-agnostic inputs are often discarded because they fail basic input sanity checks
  - Valid inputs = **higher code coverage**

```xml
[XML_GRAMMAR: Grammar = {
  "<start>" : ["<xml-tree>"],
  "<xml-tree>" : ["<text>",
  "<xml-open-tag><xml-tree><xml-close-tag>",
  "<xml-open-close-tag>",
  "<xml-tree><xml-tree>",
  "<xml-open-tag>" : ["<id>", "<id> <xml-attribute>"],
  "<xml-close-tag>" : ["</id>"],
  "<xml-attribute>" : ["<id>-<id>", "<xml-attribute> <xml-attribute>"]],
  "<id>" : ["<letter>" "<letter>"],
  "<text>" : ["<text>" "<letter_space>" "<letter_space>"]],
  "<letter>" : srange(string.ascii_letters + string.digits + "\"" + ":" + ","),
  "<letter_space>" : srange(string.ascii_letters + string.digits + "\"" + ":" + "," + ":" + ":")},
}]
```
Model-guided Generation Trade-offs

- **Writing or learning specifications is hard**
  - E.g., CSmith written in 40,000+ LoC
  - Domain expertise is critical

- **Seemingly impossible for many inputs**
  - For example, no grammar for x86 binaries

- **Deeper coverage is not always better**
  - Likely to miss bugs hidden in shallow code (e.g., input validity checks)
Symbolic and Concolic Execution

- **Model paths as symbolic expressions**
  - Construct a system of boolean equations
  - Pass this off to an SMT solver
  - Attempt to find all satisfiable assignments
  - **Concolic execution**: test one concrete path

- Many solvers available today
  - E.g., Z3, Yices, CVC4

- **The good**: great for many branches
  - Cuts through magic bytes without much trouble
Symbolic Execution Example

0. def f (x, y):
   1.   if (x > y):
   2.     x = x + y
   3.     y = x - y
   4.     x = x - y
   5.   if (x - y > 0):
   6.     assert false
   7.   return (x, y)
Symbolic Execution Example

0. def f (x, y):
1.   if (x > y):
2.     x = x + y
3.     y = x - y
4.     x = x - y
5.   if (x - y > 0):
6.     assert false
7.   return (x, y)
Symbolic Execution Example

0. `def f (x, y):
1.     if (x > y):
2.         x = x + y
3.     y = x - y
4.     x = x - y
5.     if (x - y > 0):
6.         assert false
7.     return (x, y)

A > B

L2
x : A+B
y : B

L3
x : A+B
y : (A+B) - B = A

L4
x : (A+B) - A = B
y : A

x : A
y : B
Symbolic Execution Example

0. def f (x, y):
   1. if (x > y):
   2.     x = x + y
   3.     y = x - y
   4.     x = x - y
   5.     if (x - y > 0):
   6.         assert false
   7.         return (x, y)

Possible path constraints:
- \((A > B)\) and \((B - A > 0)\) = satisfiable?
Symbolic Execution Example

```
0. def f (x, y):
1.   if (x > y):
2.     x = x + y
3.     y = x - y
4.     x = x - y
5.   if (x - y > 0):
6.     assert false
7.   return (x, y)
```

Possible path constraints:
- \((A > B)\) and \((B - A > 0)\) = unsatisfiable
- \((A > B)\) and \((B - A <= 0)\) = satisfiable?
Symbolic Execution Example

0. def f(x, y):
1.   if (x > y):
2.     x = x + y
3.     y = x - y
4.     x = x - y
5.   if (x - y > 0):
6.     assert false
7.   return (x, y)

Possible path constraints:
- (A > B) and (B - A > 0) = unsatisfiable
- (A > B) and (B - A <= 0) = satisfiable
- (A <= B) = satisfiable?
Symbolic Execution Example

0. `def f(x, y):`
1. `if (x > y):`
2. `x = x + y`
3. `y = x - y`
4. `x = x - y`
5. `if (x - y > 0):`
6. `assert false`
7. `return (x, y)`

Possible path constraints:

- `(A > B) and (B-A > 0)` = unsatisfiable
- `(A > B) and (B-A <= 0)` = satisfiable
- `(A <= B)` = satisfiable
Taint Tracking

- Track input bytes’ flow throughout program
  - Identify input “chunks” that affect program state
    - Chunks that affect branches
    - Chunks that flow to function calls

- Mutate these chunks
  - Random mutation
  - Insert fun or useful tokens

- The good: finding vulnerable buffers, solving branches

```python
0. def f (x, y):
1.   if (x > y):
2.     ........
```
Taint Tracking

- **Track input bytes’ flow throughout program**
  - Identify input “chunks” that affect program state
    - Chunks that affect branches
    - Chunks that flow to function calls

- **Mutate these chunks**
  - Random mutation
  - Insert fun or useful tokens

- **The good:** finding vulnerable buffers, solving branches

```python
0. def f(x, y):
1.   if (x > y):
2.     ........
```
Taint Tracking

- Track input bytes’ flow throughout program
  - Identify input “chunks” that affect program state
    - Chunks that affect branches
    - Chunks that flow to function calls
  - Mutate these chunks
    - Random mutation
    - Insert fun or useful tokens

- The good: finding vulnerable buffers, solving branches

```python
0. def f(x, y):
1.   if (x > y):
2.     ...........
```

<table>
<thead>
<tr>
<th>Bytes that comprise X</th>
<th>11</th>
<th>11</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes that comprise Y</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Mutate!
White-box Generation Trade-offs

- **All of these techniques are heavyweight**
  - Too slow to deploy for every input, branch, etc.
  - Must decide *which* problems to feed it
    - Scheduling problem

- **Generally limited to simple software**
  - Good luck doing taint tracking on MS Office...

- **Emerging techniques give us hope!**
  - Fast taint tracking: RedQueen
  - Fast concolic exec: SymCC
Types of Input Generation

- **Model-agnostic**: great on simple, easy-to-solve branches
  - Need a lot of luck to solve **multi-byte conditionals** and **checksums**

- **Model-guided**: more valid inputs leads to higher coverage
  - Out of luck if specification is **not defined** or **hard-to-define**

- **White-box approaches**:
  - **Symbolic / concolic exec**: precise solving of multi-byte conditionals
  - **Taint tracking**: easily identifies key data objects, branch constraints
  - Far too **heavyweight** to deploy on every single generated input

Source: The Art, Science, and Engineering of Fuzzing: A Survey
Questions?
Testing Takeaways
Results?
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.
Interested in fuzzing?

  - Everything you’d ever want to know about fuzzing for finding security bugs!
  - Course project: team up to fuzz a real program (of your choice), and find and report its bugs!
  - [https://cs.utah.edu/~snagy/courses/cs5963/](https://cs.utah.edu/~snagy/courses/cs5963/)

**CS 5963/6963: Applied Software Security Testing**

This special topics course will dive into today's state-of-the-art techniques for uncovering hidden security vulnerabilities in software. Projects will provide hands-on experience with real-world security tools like AFL++ and AddressSanitizer, culminating in a final project where you'll team up to hunt down, analyze, and report security bugs in a real application or system of your choice.

This class is open to graduate students and upper-level undergraduates. It is recommended you have a solid grasp over topics like software security, systems programming, and C/C++.

**Professor**

Stefan Nagy
Interested in fuzzing?

- **Fuzzing Faculty**
  - Lots of us that do research in fuzzing

- **Get involved!**
  - Undergrad research
  - UROP, SPUR
  - Independent study
  - Senior thesis
Questions?
Food for Thought

- Today, we’ve talked about **thwarting bugs** by **proactively** discovering them
  - E.g., run fuzzing and try to catch all the bugs!
  - Hopefully the **attacker** will not beat us to it...

- **Question:** how can we redesign our **systems** to prevent software exploits?
Next time on CS 4440...

Virtualization, Isolation, Sandboxing