

COMP5111 – Fundamentals of Software Testing and Analysis

Software Vulnerabilities



Shing-Chi Cheung

Computer Science & Engineering

HKUST



Some slides are adapted from CS155 course notes by John Mitchell, Stanford in his 2018 offering

Vulnerability Example

■ Missed lower-bound check:

```
/* 2.4.5/drivers/char/drm/i810_dma.c */  
  
if(copy_from_user(&d, arg, sizeof(arg)))  
    return -EFAULT;  
if(d.idx > dma->buf_count)    // upper-bound check  
    return -EINVAL;  
buf = dma->buflist[d.idx];  
Copy_from_user(buf_priv->virtual, d.address, d.used);
```

Memory Related Vulnerabilities

- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code
- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocation of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

Slide credit: Andy Chou

VULNERABILITY ANALYSIS APPROACHES

Three Major Approaches

Proposed by Barton Miller in 1990s
Inputs may be malformed or unmeaningful

- Dynamic analysis

- ❑ Fuzzing (testing by randomly generated inputs)
 - ❑ Slicing and tainting

- Static analysis

- ❑ Slicing and tainting
 - ❑ Pointer analysis

- Statistical analysis / Machine learning

- Can be hybrid ...

Origin of Fuzzing

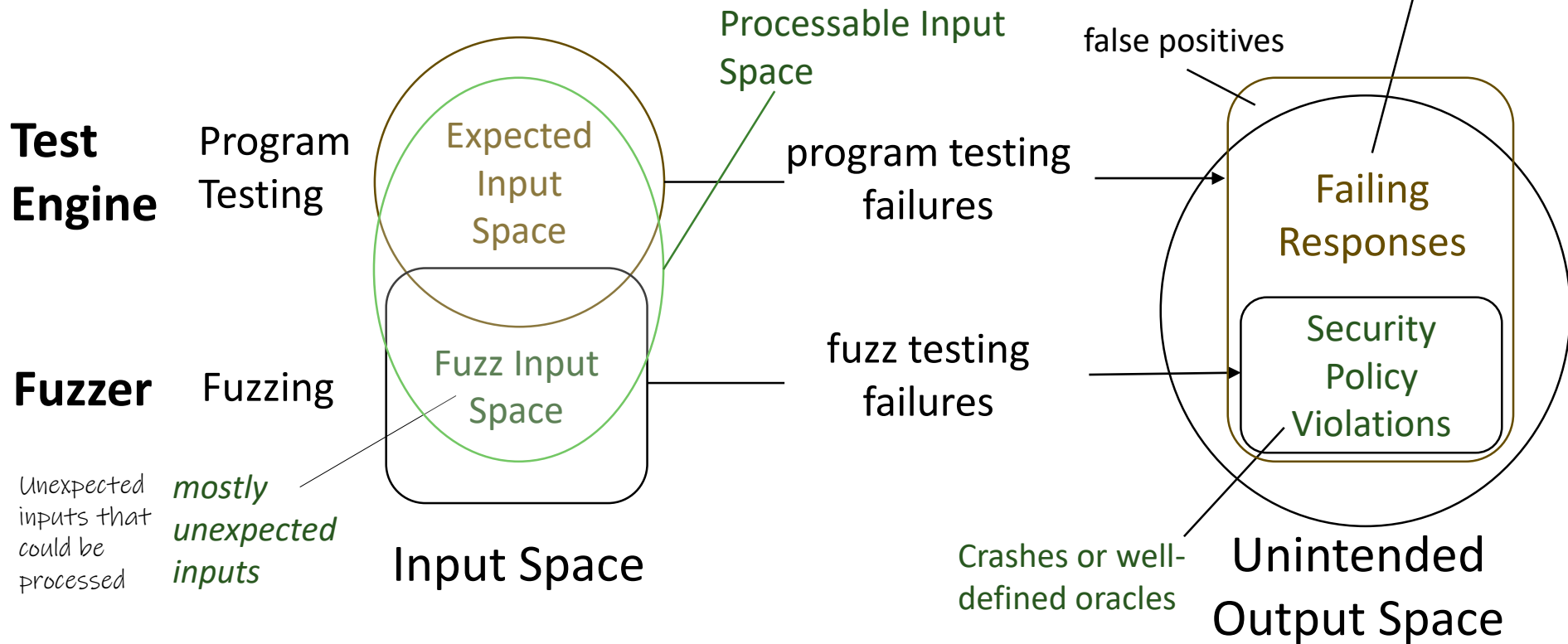
- The term “fuzz” originates from a class project in a graduate level course on Advanced Operating Systems taught by Prof Barton Miller in Fall 1988
- The project was to test the reliability of UNIX command line programs by feeding them with a large number of random inputs until they crash
- The class was able to crash 33% of the utilities fuzzed



Miller et al., An Empirical Study of the Reliability of UNIX Utilities. CACM, Dec 1990.

Program Testing vs Fuzzing

*Design of test oracles
is a key challenge*



Fuzzing Input Space

- Mostly (semi-)malformed inputs/data

- ❑ GEEEE...EET /index.html HTTP/1.1
- ❑ GET //////////index.html HTTP/1.1
- ❑ GET %n%n%n%n%n%n.html HTTP/1.1
- ❑ GET /AAAAAAAAAAAAAAAAA.html HTTP/1.1

Examples of malformed
http GET request

- Generated randomly

- ❑ Mutate a population of seeds (e.g., GET index.html HTTP/1.1)
- ❑ Quality of the seeds greatly affects fuzzing performance
 - Smart selection of seeds is an outstanding research problem

Fuzzing vs Random Testing (more details)

| Fuzzing | Random Testing |
|--|---|
| Detect faults exploitable by hackers | Detect faults unintentionally triggered by users |
| Target well-defined fault types (fault-based) | Target all unexpected program behaviors |
| Generic well-defined test (vulnerability) oracles; No need for regression test oracles | Test oracles vary across test cases; Regression test oracles are meaningful |
| Fuzz anything that can be manipulated (including binaries and memories) | Generate sensible test inputs or test scripts |
| Fuzzed inputs can be meaningless (e.g., a successful attack triggered by hitting buttons hundred times in a second is likely considered legitimate) | Test inputs should be sensible; otherwise faults detected can be false positives (e.g., a failure caused by hitting buttons hundred times in a second is likely considered false negative) |
| Prefer higher recall for vulnerability detection | Prefer higher precision for bug detection |

Crashes vs Security Policy Violations

| Bug ID | Program | | Bug | |
|--------------------|-----------|--------|------|-------|
| | Project | Size | Type | Crash |
| giflib-bug-74 | GIFLIB | 59 Kb | DF | ✗ |
| CVE-2018-11496 | lrzip | 581 Kb | UAF | ✗ |
| yasm-issue-91 | yasm | 1.4 Mb | UAF | ✗ |
| CVE-2016-4487 | Binutils | 3.8 Mb | UAF | ✓ |
| CVE-2018-11416 | jpegoptim | 62 Kb | DF | ✗ |
| mjs-issue-78 | mjs | 255 Kb | UAF | ✗ |
| mjs-issue-73 | mjs | 254 Kb | UAF | ✗ |
| CVE-2018-10685 | lrzip | 576 Kb | UAF | ✗ |
| CVE-2019-6455 | Recutils | 604 Kb | DF | ✗ |
| CVE-2017-10686 | NASM | 1.8 Mb | UAF | ✓ |
| gifsicle-issue-122 | Gifsicle | 374 Kb | DF | ✗ |
| CVE-2016-3189 | bzip2 | 26 Kb | UAF | ✓ |

DF: Double Free

UAF: Use-After-Free

A recent benchmark
in Nguyen et al.,
Binary-level Directed
Fuzzing for Use-After-
Free Vulnerabilities.
USENIX RAID 20.

Can Fuzzing detect deep vulnerabilities?

YES! For example, AFL, a commonly used fuzzer, can uncover the Heartbleed vulnerability in the OpenSSL implementation in 3 minutes.

The vulnerability, reported on 7 April 2014, allows hackers to steal sensitive information including encryption keys and passwords over Websites using OpenSSL connections

Affected applications included those run by Google, Yahoo, Dropbox, Netflix, Facebook, Amazon Web Services, Apple, Microsoft, PayPal, LinkedIn, eBay, Twitter, AOL, Bank of America, Chase, Fidelity, Schwab, US Bank, and Wells Fargo.

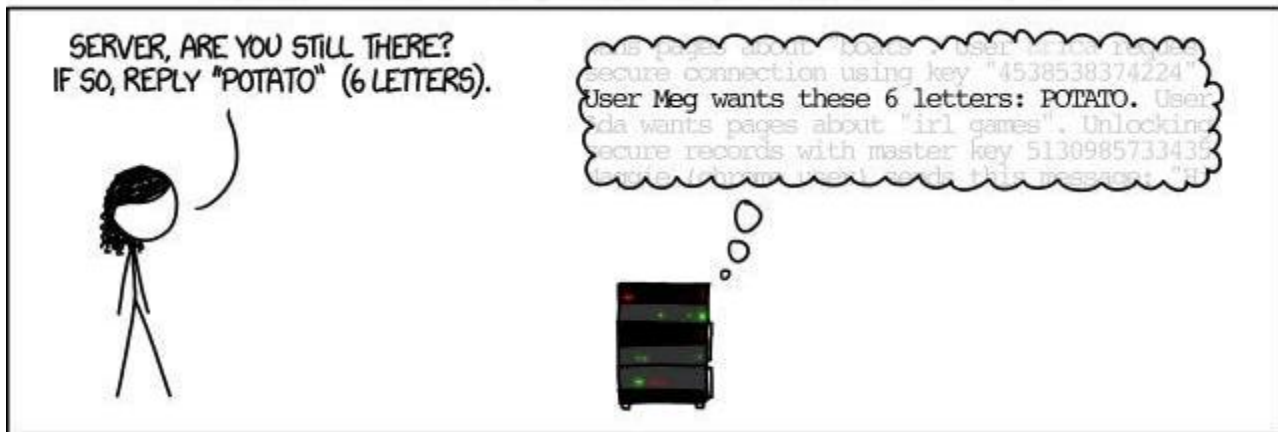
The mistake was traced to a single line of code in the OpenSSL implementation:

`memcpy(bp, pl, payload);`

<https://www.vox.com/2014/6/19/18076318/heartbleed>

How Heartbleed works - 1

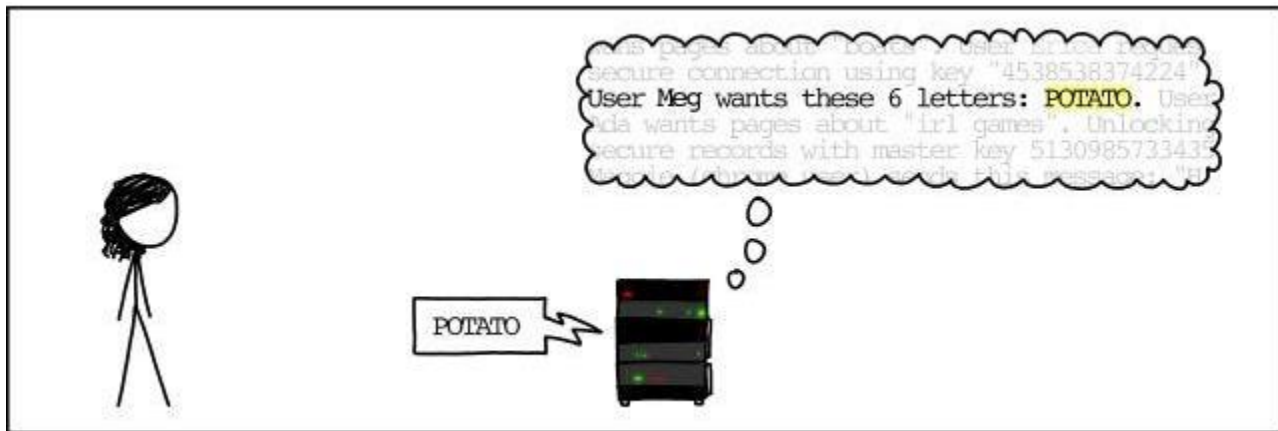
<https://www.howtogeek.com/186735/htg-explains-what-the-heartbleed-bug-is-and-why-you-need-to-change-your-passwords-now/>



OpenSSL keeps a **side channel** allowing users to send heartbeats to a server, checking if it is still awake. As **heartbeats** do not contain sensitive information, they **are not encrypted** for the sake of performance.

How Heartbleed works - 2

<https://www.howtogeek.com/186735/htg-explains-what-the-heartbleed-bug-is-and-why-you-need-to-change-your-passwords-now/>



A **heartbeat** request contains an insensitive **payload** (e.g., POTATO) and an integer indicating the payload size (e.g., 6). After receiving the request, server replies users with a heartbeat message containing the payload content.

How Heartbleed works - 3

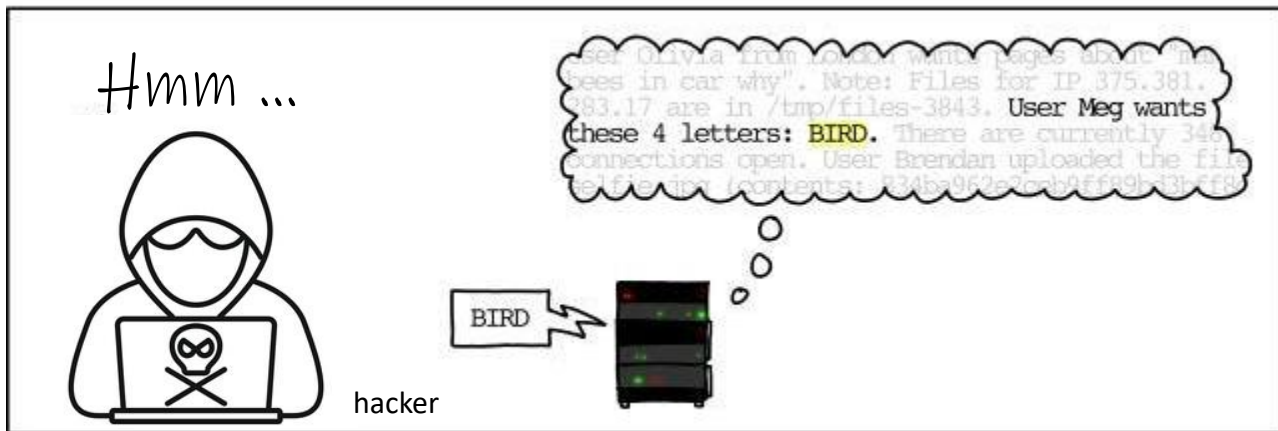
<https://www.howtogeek.com/186735/htg-explains-what-the-heartbleed-bug-is-and-why-you-need-to-change-your-passwords-now/>



After a while, users can send another heartbeat request with different payload (e.g., BIRD) and an integer indicating the payload size (e.g., 4).

How Heartbleed works - 4

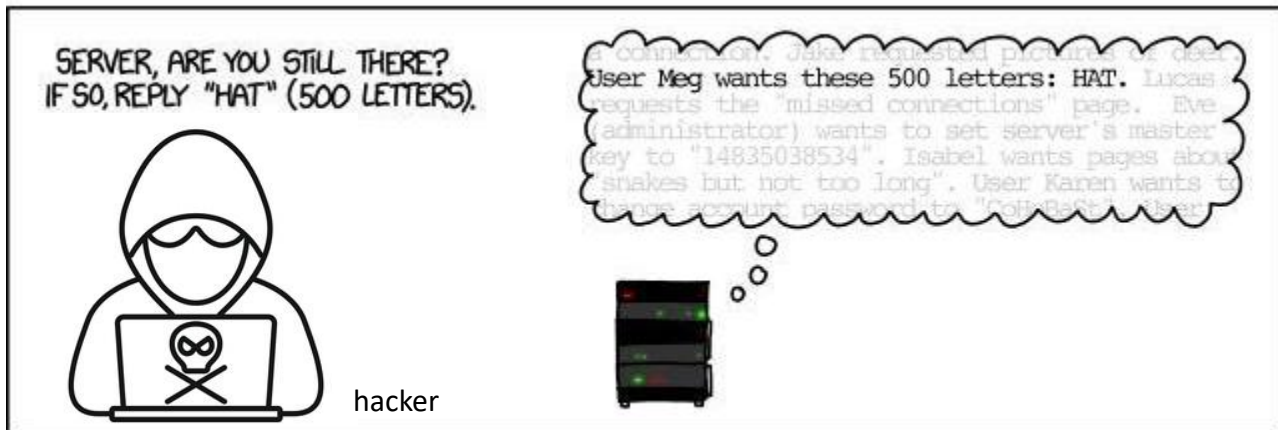
<https://www.howtogeek.com/186735/htg-explains-what-the-heartbleed-bug-is-and-why-you-need-to-change-your-passwords-now/>



Server replies with a heartbeat message containing the payload (e.g., BIRD).

How Heartbleed works - 5

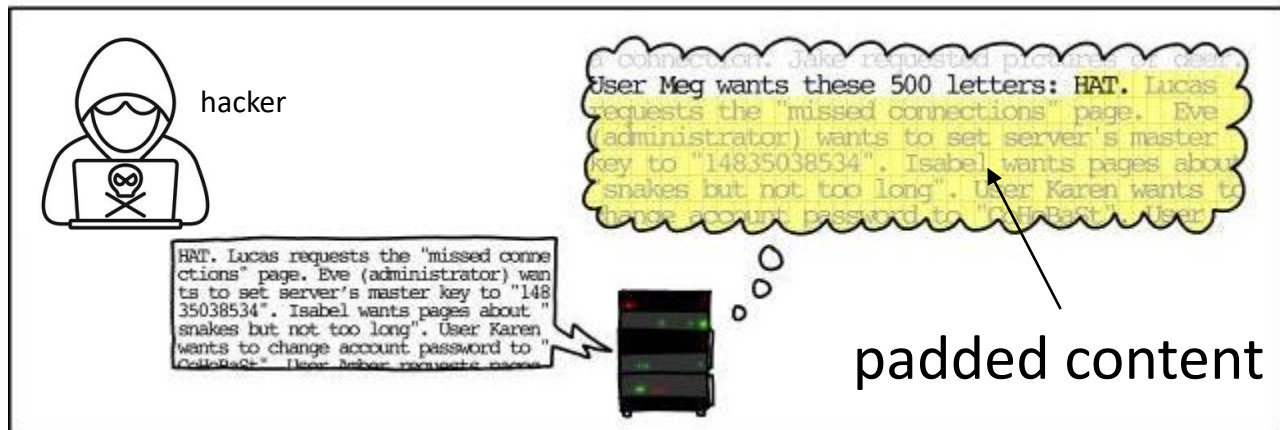
<https://www.howtogeek.com/186735/htg-explains-what-the-heartbleed-bug-is-and-why-you-need-to-change-your-passwords-now/>



However, a hacker can send a heartbeat request with little payload content (e.g., HAT) and tell that it has a large payload size (e.g., 500).

How Heartbleed works - 6

<https://www.howtogeek.com/186735/htg-explains-what-the-heartbleed-bug-is-and-why-you-need-to-change-your-passwords-now/>



The server does not check the consistency but constructs a reply padded with the content after HAT in the memory. The padded content likely contains sensitive information, and it is not encrypted!

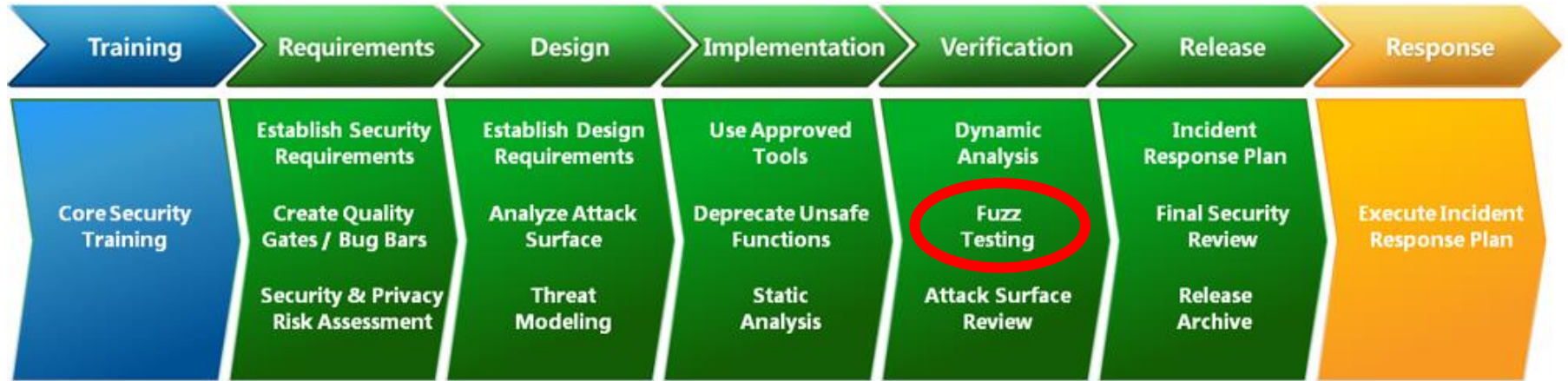
Detection of Heartbleed

```
shadowsword@shadowswordpl ~/HEARTBLEED/openssl-1.0.1f $ ./runError.sh afl_out/master/crashes/id\:\:000000\,sig\:\:06\,src\:\:000032\,op\:\:arith8\,pos\:\:0\,val\:\:+3
=====
==16345== ERROR: AddressSanitizer: unknown-crash on address 0x60820001220b at pc 0x7f4a98de23f7 bp 0x7fffa0b72380 sp 0x7fffa0b71b40
READ of size 25600 at 0x60820001220b thread T0
#0 0x7f4a98de23f6 (/usr/lib/x86_64-linux-gnu/libasan.so.0+0xe3f6)
#1 0x410dad in memcpy /usr/include/x86_64-linux-gnu/bits/string3.h:51
#2 0x410dad in tls1_process_heartbeat /home/shadowsword/HEARTBLEED/openssl-1.0.1f/ssl/t1_lib.c:2586
#3 0x49d16c in ssl3_read_bytes /home/shadowsword/HEARTBLEED/openssl-1.0.1f/ssl/s3_pkt.c:1092
#4 0x4a0ca3 in ssl3_get_message /home/shadowsword/HEARTBLEED/openssl-1.0.1f/ssl/s3_both.c:457
#5 0x465a2d in ssl3_get_client_hello /home/shadowsword/HEARTBLEED/openssl-1.0.1f/ssl/s3_srvr.c:941
#6 0x4764b3 in ssl3_accept /home/shadowsword/HEARTBLEED/openssl-1.0.1f/ssl/s3_srvr.c:357
#7 0x402487 in main /home/shadowsword/HEARTBLEED/openssl-1.0.1f/selftls.c:95
#8 0x7f4a9882cec4 (/lib/x86_64-linux-gnu/libc.so.6+0x21ec4)
#9 0x40299c in start (/home/shadowsword/HEARTBLEED/openssl-1.0.1f/selftls+0x40299c)
0x608200016748 is located 0 bytes to the right of 17736-byte region [0x608200012200,0x608200016748)
allocated by thread T0 here:
#0 0x7f4a98de941a (/usr/lib/x86_64-linux-gnu/libasan.so.0+0x1541a)
#1 0x4d7946 in CRYPTO_malloc /home/shadowsword/HEARTBLEED/openssl-1.0.1f/crypto/mem.c:308
```

After 3 minutes, a seed generated by the AFL fuzzer contains 0x18037F00020164 and causes a crash at the AddressSanitizer.

<http://9livesdata.com/fuzzing-how-to-find-bugs-automagically-using-afl/>

Do Companies use Fuzzing?

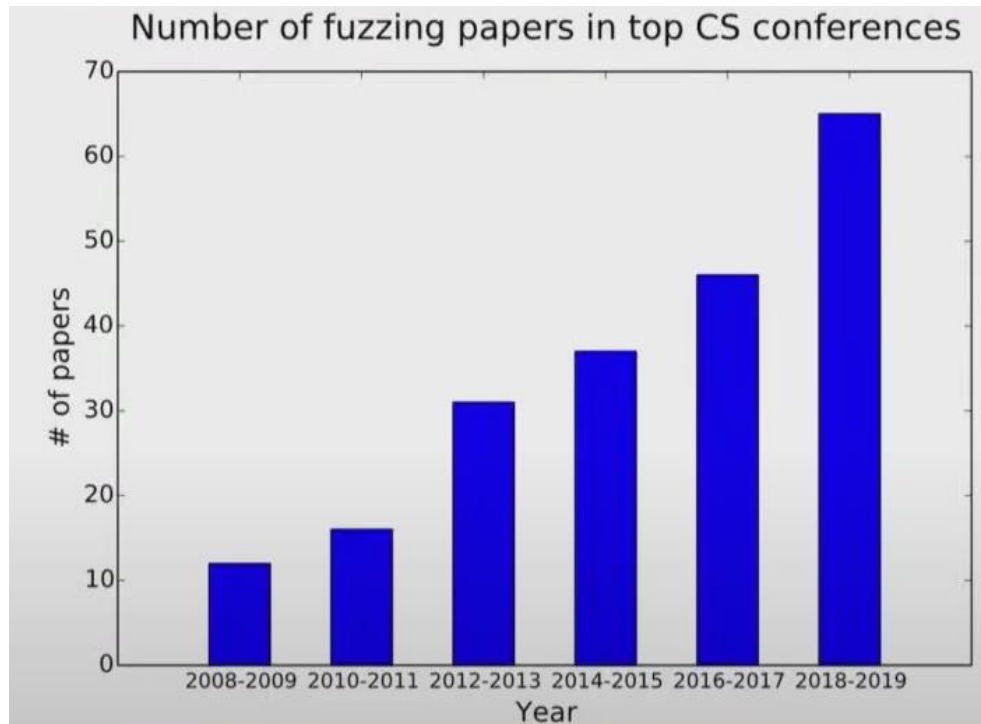


Microsoft Security Development Lifecycle

Industry Adoption

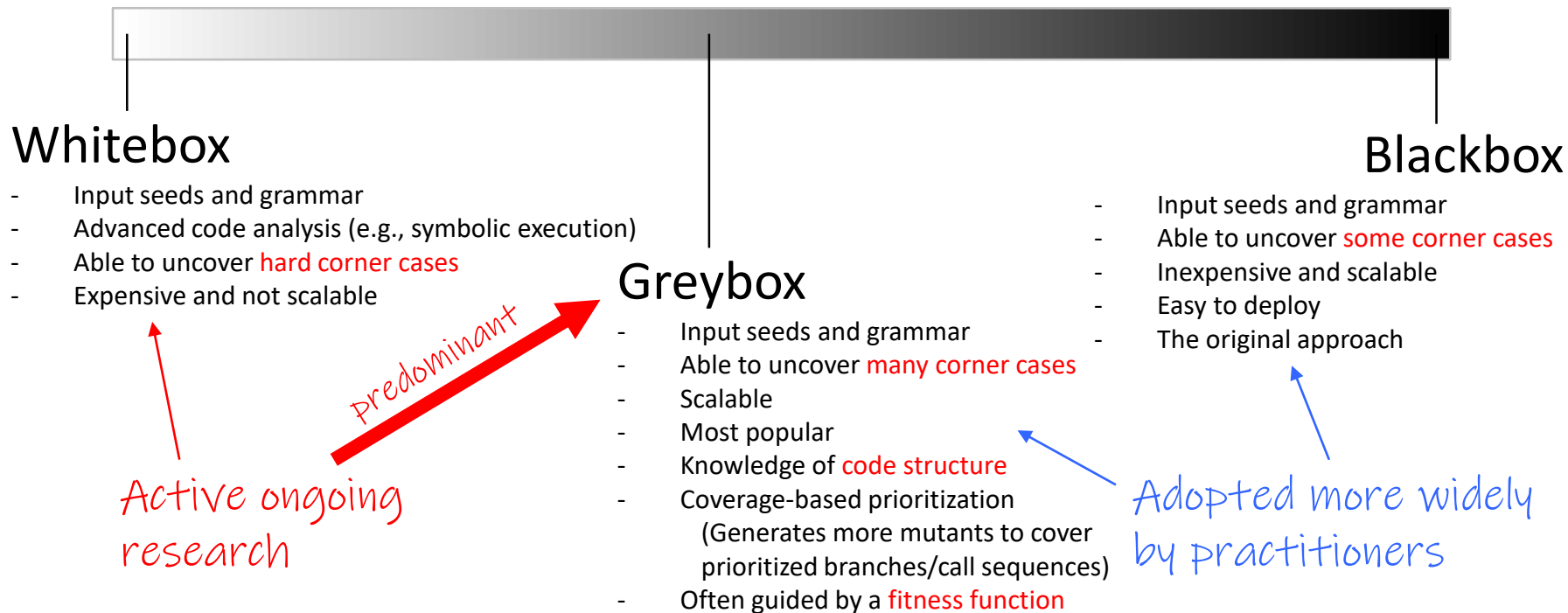
- Google has discovered 16,000+ bugs in Chrome and 11,000+ bugs in 160+ open-source projects using a greybox fuzzer
 - <https://google.github.io/clusterfuzz/#trophies>
- Microsoft saved millions of dollars in Win 7 using SAGE, a whitebox fuzzer [Godefroid et al. NDSS 2008]
- Trail of Bits provides the DeepState framework facilitating C/C++ developers to fuzz testing their systems
 - <https://github.com/trailofbits/deepstate>
- ...

Fuzzing is hot!



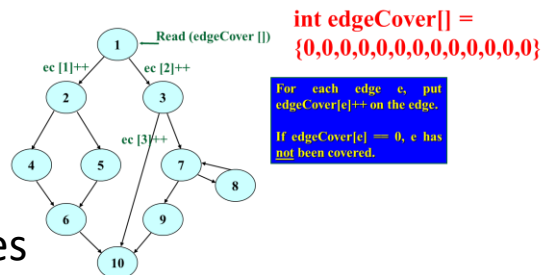
- Increasing attention from academics
- Key problem: Huge input space of fuzzing
- She et al. S&P 2019
<https://www.youtube.com/watch?v=j4ynjsA5CEQ>

Three Fuzzing Approaches

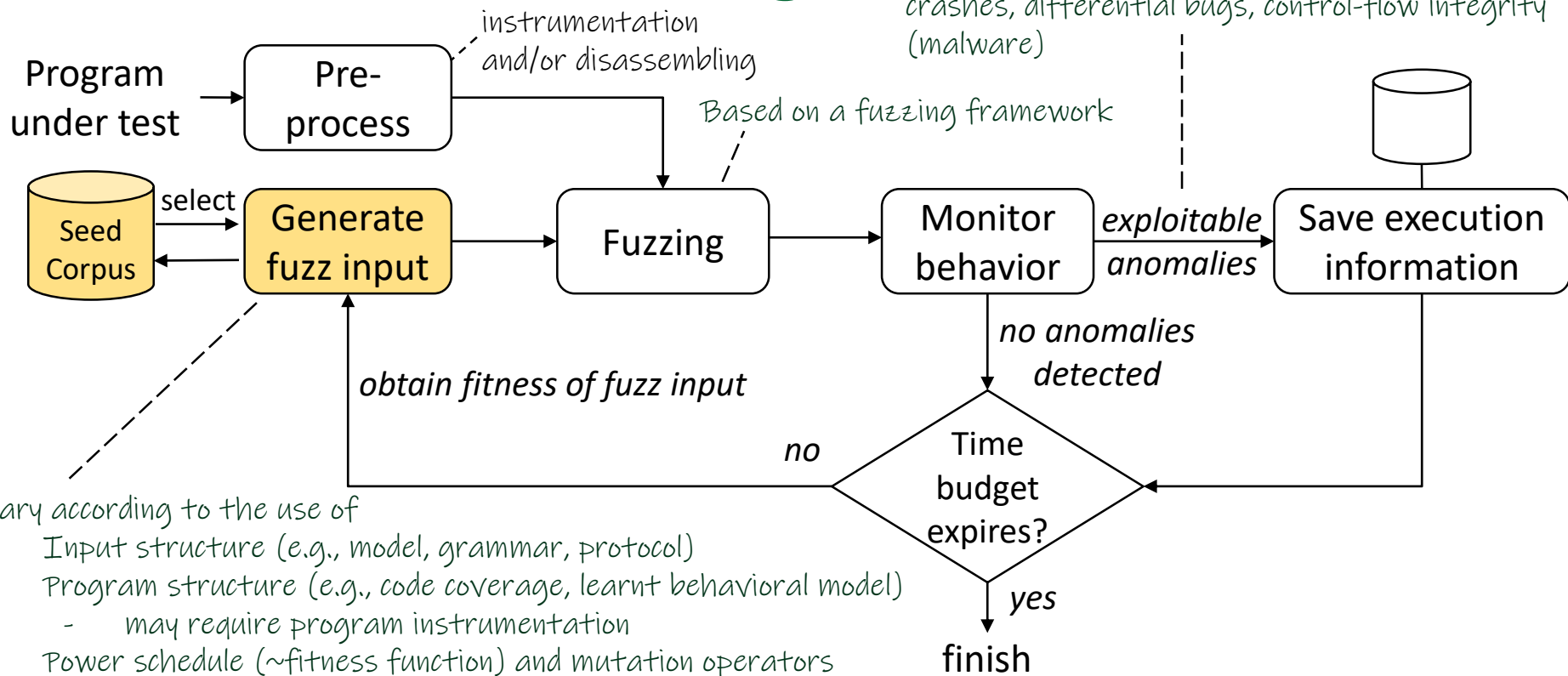


Frameworks of Fuzzing

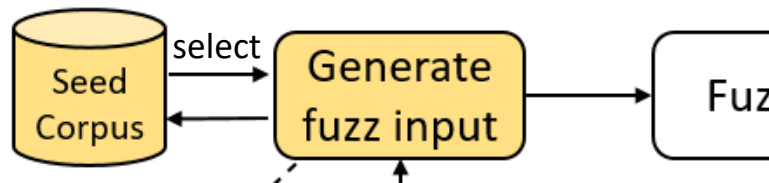
- Driven by a Fuzzer over a fuzzing framework (cf. JUnit)
 - Generate massive fuzzed data randomly (with an optional fitness function)
 - Based on either branch coverage or branch hit count (e.g., AFL) **Edge Coverage Instrumentation**
 - Use of distance-based fitness function is possible:
V. Manes, S. Kim, S. Cha. Ankou: Guiding Grey-box Fuzzing towards Combinatorial Differences. ICSE 2020.
 - Monitor crashes and security policy violations by means of resource (e.g., registry, memory, cpu) anomalies
 - Unlike conventional testing, test oracles are usually application-generic predefined by a fuzzer
- Many frameworks are publicly available
 - Peach Fuzz, **AFL**, LibFuzzer, Honggfuzz, Radamsa, Zzuf, BFF, TAOF, KLEE, SAGE, ...
 - AFL is a popular coverage-based greybox fuzzing framework used by academics



Workflow of Fuzzing



Generate Fuzz Input

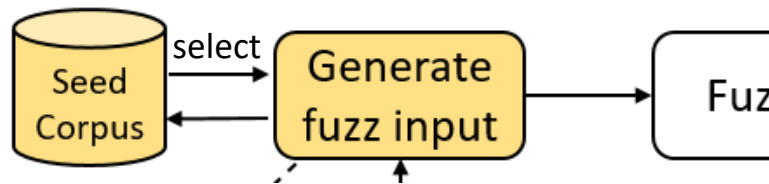


■ Seed selection

- ❑ Search strategy: Decides the order in which seeds are chosen from the corpus based on a **fitness function**
- ❑ Energy assignment (Power Schedule): Determine the number of mutants generated from a seed
 - AFL spends more energy fuzzing seeds that are small and executed quickly
 - Some fuzzers simply generate a small random number of mutants from a selected seed

... based on AFL

Generate Fuzz Input



- Seed mutation (assuming the seed is a string)
 - ❑ Insert a character randomly
 - ❑ Delete a character randomly
 - ❑ Select a character randomly and flip a random bit of it
 - ❑ Add the generated seed to the Seed Corpus

... based on AFL

Common Types of Faults Found by Fuzzing

- Memory corruption in native code
 - ❑ Stack and heap buffer overflows
 - ❑ Un-validated pointer arithmetic (offset can be controlled by attackers)
 - ❑ Integer overflows
 - ❑ Resource exhaustion (memory, disk, CPU)

Common Types of Faults Found by Fuzzing

- Unhandled exceptions
 - ❑ Illegal format exceptions (due to illegal inputs/data)
 - ❑ Memory exceptions
 - ❑ Null reference exceptions
- Exposure to injection
 - ❑ SQL injection
 - ❑ LDAP injection
 - ❑ HTML injection

What Types of Inputs/Data can be Fuzzed?

- No restriction
- Primitive data: bit, byte, word, dword, qword, ...

```
for each {byte | word | dword | qword} aligned location in file
  for each bad_value in bad_valueset
  {
    file[location] := bad_value
    apply_test()
  }
```

What Types of Inputs/Data can be Fuzzed?

- Language specific data: strings, structs, arrays, ...
- High level data: text, xml, header, trailer, ...

```
...  
o_jpeg = fz3AddObjectToList( NULL, TYPE_BYTE, PTR(0xff), 1 ); // new header  
fz3AddObjectToList( o_jpeg, TYPE_BYTE, PTR(0xd8), 1 ); // unknown type (start of file?)  
fz3AddObjectToList( o_jpeg, TYPE_BYTE, PTR(0xff), 1 ); // new header  
fz3AddObjectToList( o_jpeg, TYPE_BYTE, PTR(0xe0), 1 ); // extension marker segment  
o_jfif_len = fz3AddObjectToList( o_jpeg, TYPE_WORD, BE_W(0x10), 2 ); // length  
...
```

- Abstract level data: state machine transitions, ...

LLM Vulnerabilities

- Recent research starts studying the vulnerabilities of LLMs.
- Prompt injection
<https://www.youtube.com/watch?v=Sv5OLj2nVAQ>

OPTIONAL SLIDES

FUZZING APPROACHES

Fuzzing – Three Common Approaches

Mutation-based fuzzing

- ❑ Mutate existing good input seeds (e.g., test suite)

Generation-based fuzzing

- ❑ Generate inputs from specification of format, protocol, ...

Evolutionary (responsive) fuzzing

- ❑ Leverage program instrumentation, code analysis
- ❑ Use program responses to build input sets

Recap: Unexpected inputs that can be processed by the target software

Source: <https://crypto.stanford.edu/cs155/papers/fuzzing.pdf>

Mutation-based fuzzing

Basic idea

- ❑ Mutate **existing valid inputs randomly** or according to some **heuristics**

Standard HTTP GET request

- ❑ GET /index.html HTTP/1.1 ←———— a given valid input seed

Malformed requests

- ❑ GEEEE...EET /index.html HTTP/1.1
- ❑ GET //////////index.html HTTP/1.1
- ❑ GET %n%n%n%n%n%n%.html HTTP/1.1
- ❑ GET /AAAAAAAAAAAAAAAAA.html HTTP/1.1
- ❑ GET /index.html HTTTTTTTTTTTTTTTP/1.1
- ❑ GET /index.html HTTP/1.1.1.1.1.1.1.1

Mutation-based fuzzing

Advantages

- ❑ Little or no knowledge of the structure of the inputs is assumed
- ❑ Requires little or no set up time

Disadvantages

- ❑ Dependent on the inputs being modified
- ❑ May fail for protocols with checksums, challenge-response, ...

Examples fuzzing tools:

- ❑ Taof, GPF, ProxyFuzz, etc.

(Mutation-based) Fuzzing - Example

- Find bugs in pdf viewer
 - Difficult to randomly generate valid pdf files from scratch



PDF Viewer

Crash viewer and isolate causes

Modify valid pdf files

Generation-based Fuzzing

Request for Comments

Basic idea

- ❑ Tests generated from description of format: RFC, spec, etc.
- ❑ Anomalies are added to each possible spot in the test inputs

Advantages

- ❑ Knowledge of protocol may give better results than random fuzzing

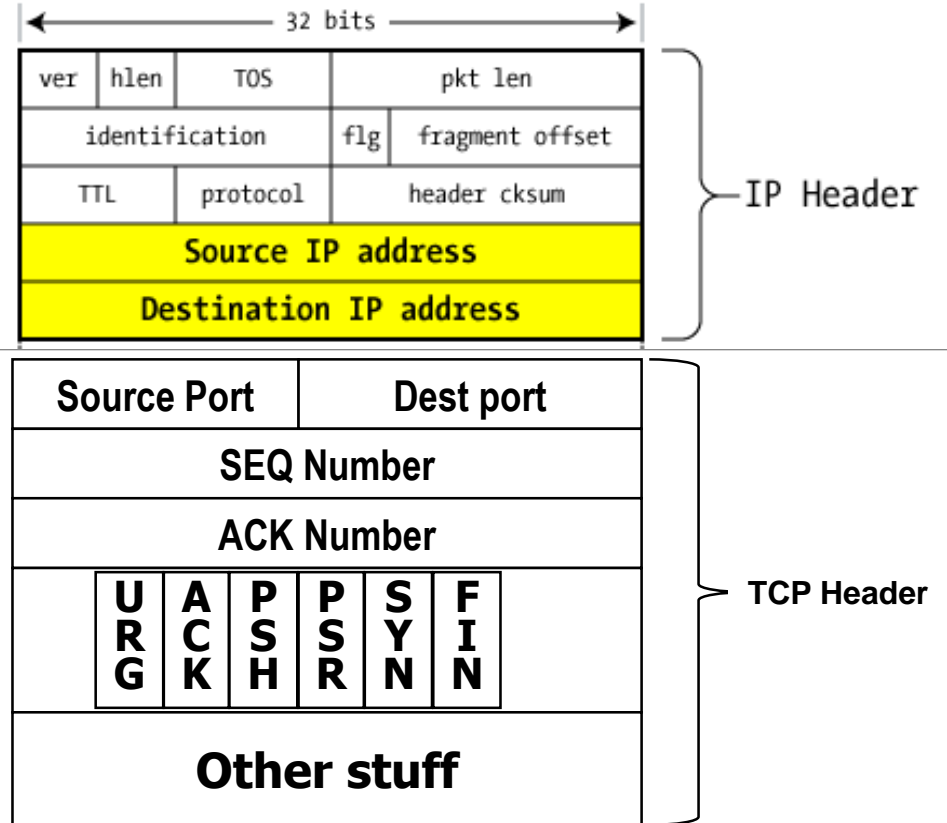
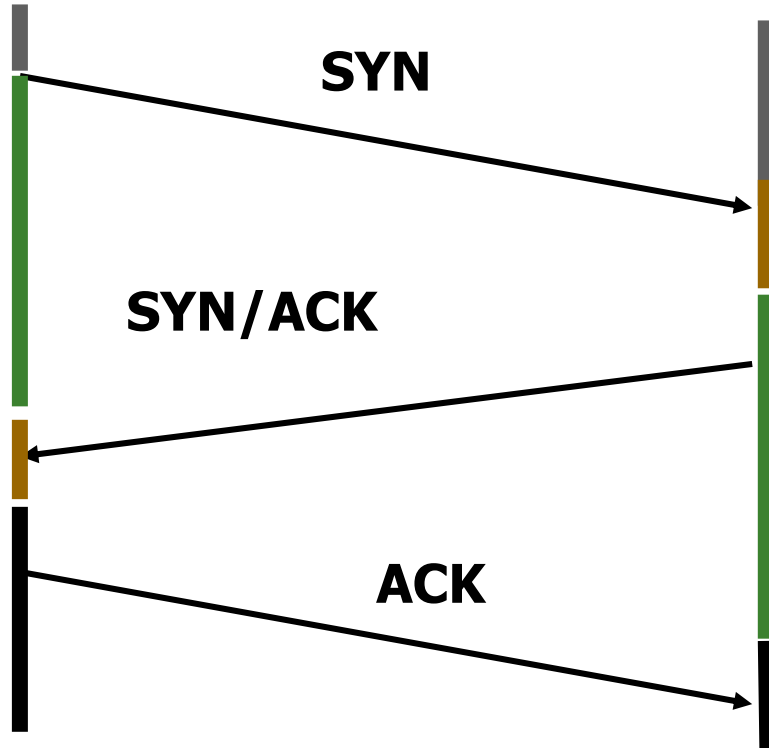
Disadvantages

- ❑ Can take significant time to set up

Example tools:

- ❑ SPIKE, Sulley, Mu-4000, Codenomicon

Generation-based Fuzzing for Protocol Bugs - Example



Generation-based Fuzzing for Zip file - Example

```
1  <!-- A. Local file header -->
2  <Block name="LocalFileHeader">
3    <String name="lfh_Signature" valueType="hex" value="504b0304" token="true" mut
4    <Number name="lfh_Ver" size="16" endian="little" signed="false"/>
5    ...
6    [truncated for space]
7    ...
8    <Number name="lfh_CompSize" size="32" endian="little" signed="false">
9      <Relation type="size" of="lfh_CompData"/>
10   </Number>
11   <Number name="lfh_DecompSize" size="32" endian="little" signed="false"/>
12   <Number name="lfh_FileNameLen" size="16" endian="little" signed="false">
13     <Relation type="size" of="lfh_FileName"/>
14   </Number>
15   <Number name="lfh_ExtraFldLen" size="16" endian="little" signed="false">
16     <Relation type="size" of="lfh_FldName"/>
17   </Number>
18   <String name="lfh_FileName"/>
19   <String name="lfh_FldName"/>
20   <!-- B. File data -->
21   <Blob name="lfh_CompData"/>
22 </Block>
```


Evolutionary (Responsive) Fuzzing

Basic idea

- Generate inputs based on the structure, response of the program

Advantages

- Exploits detailed knowledge of program
- Prioritizes inputs that may lead to vulnerability

Disadvantages

- Requires more effort and domain knowledge

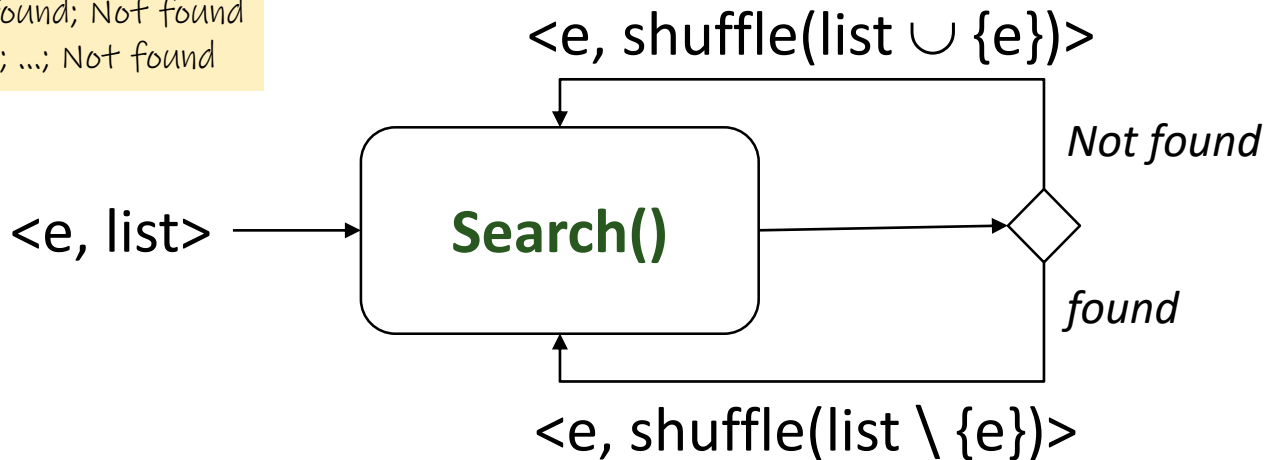
Examples:

- Autodafe
 - Prioritizes test cases based on which inputs have reached dangerous API functions
- Metamorphic testing
 - Generates test cases based on metamorphic relations
- AFL
 - Fitness function based on branch hit count

Evolutionary (Responsive) Fuzzing - MT Example

Two example scenarios:

- Not found; found; Not found
- found; found; ...; Not found



Have knowledge of the search program structure that shuffling a list does not alter the search result.

DIRECTED GREYBOX FUZZING (EVOLUTIONARY BASED)

Directed Greybox Fuzzing

- Direct the fuzzing to expose vulnerabilities at known dangerous locations or critical system calls
- Augment energy assignment of seeds with information about the targets
- Typical application scenarios include:
 - Patch testing: set changed statements as targets [Böhme ESEC/FSE13; Marinescu ESEC/FSE13]
 - Crash reproduction: set method calls in stack trace as targets [Jin ICSE12; Pham ICSE15]
 - Information flow detection: set sensitive sources and sinks as targets [Matthis ASE17]
 - ☞ □ Static analysis report verification: set statements, method call sequences or traces reported in a warning as targets [Christakis ICSE16; Nyugen RAID20]



Fuzzing Algorithm (based on AFL)

Input: Seed Corpus C

repeat

$s = \text{ChooseNext}(C)$ // Selection strategy

$p = \text{AssignEnergy}(s)$ // Power schedule

Scheduler

for i from 1 to p **do**

$s' = \text{MutateInput}(s)$ // Which mutators are used?

Generator

if s' crashes **then**

add s' to F

else if $\text{IsInteresting}(s')$ **then** // s' introduces new coverage?

Updater

add s' to C

end if

end for

until *timeout* reached or *abort* signal

Output: Crashing Inputs F

Enhancing the Scheduler

GreyOne [USENIX Security 2020]

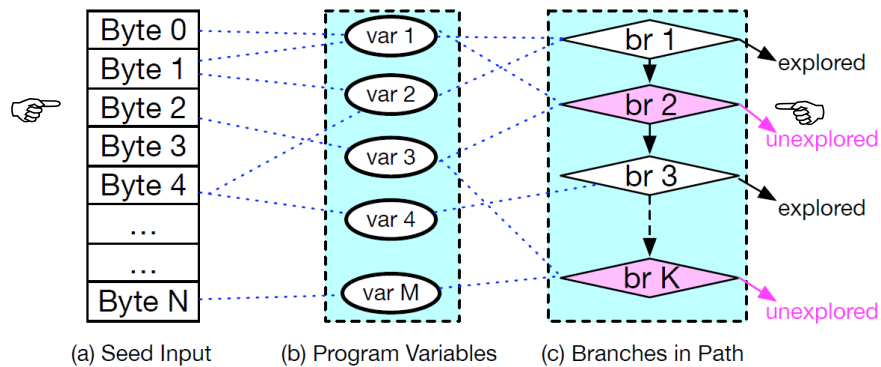
Enhancing the seed scheduling for data flow sensitive fuzzing

Problem

Uncertain if a seed after mutation will likely expose vulnerability bugs

Solution

Apply taint analysis to record the impact of a seed on branch coverage



Pre-fuzzing: Mutate each byte of the initial seed input in turn and record which variables and branches are affected

During fuzzing: Prioritize the mutation of a byte that can affect more unexplored branches, and prioritize the exploration of an unexplored branch that is affected by more prioritized bytes

GreyOne [USENIX Security 2020]

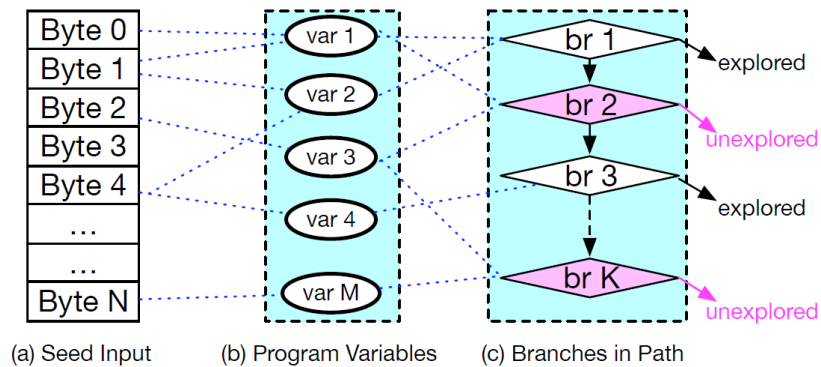
Enhancing the seed scheduling for data flow sensitive fuzzing

Problem

Uncertain if a seed after mutation will likely expose vulnerability bugs

Solution

Apply taint analysis to record the impact of a seed on branch coverage



Intuition:

- A variable depends on an input byte if the variable's value changes after the byte is mutated.
- Mutating this input byte could change the constraints of branches that use this variable, leading to new branch exploration

GreyOne [USENIX Security 2020]

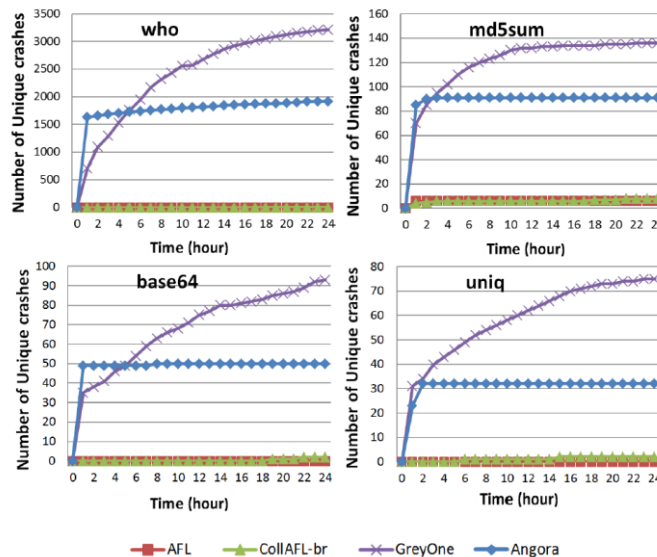
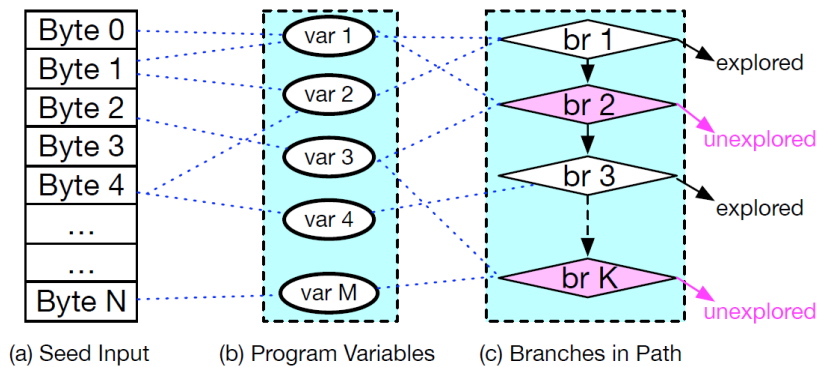
Enhancing the seed scheduling for data flow sensitive fuzzing

Problem

Uncertain if a seed after mutation will likely expose vulnerability bugs

Solution

Apply taint analysis to record the impact of a seed on branch coverage



Double the number of crashes detected

UAFuzz [USENIX RAID 2020]

Source code is given here for the purpose of explanation.
Greybox fuzzer works on binary code or its disassembled version

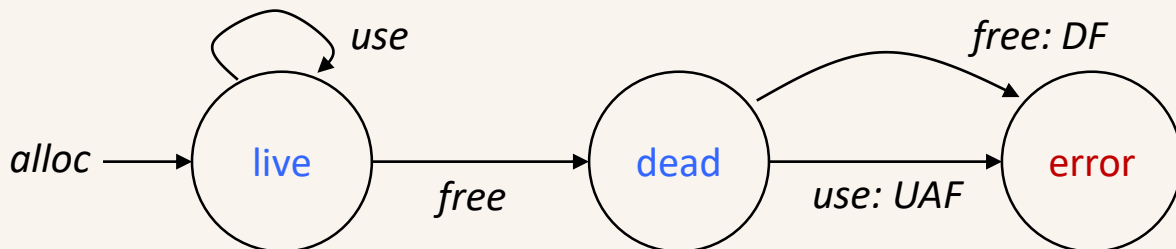
```
1  int *p, *p_alias;
2  char buf[10];
3  void bad_func(int *p) {free(p);} /* exit() is missing */
4  void func() {
5      if (buf[1] == 'F')
6          bad_func(p);
7      else /* lots more code ... */
8      }
9  int main (int argc, char *argv[]) {
10     int f = open(argv[1], O_RDONLY);
11     read(f, buf, 10);
12     p = malloc(sizeof(int));
13     if (buf[0] == 'A'){
14         p_alias = malloc(sizeof(int));
15         p = p_alias;
16     }
17     func();
18     if (buf[2] == 'U')
19         *p_alias = 1;
20     return 0;
21 }
```

- Input seed: 'AFU'
- Bug target: 14 (alloc) → 17 → 6 → 3 (free) → 19 (use)
- Execution: No crash but times out after 6 hours
- Many use-after-free bugs do not result in crash

Nguyen et al., Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. USENIX RAID 20

Fuzzing Use-After-Free (UAF) is Hard

Variable
lifecycle



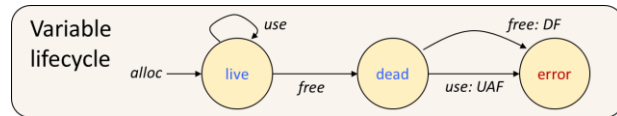
Problem

UAF bugs are rarely detected by blackbox fuzzers:

- **Complexity.** Involves 3 events in sequence spanning multiple functions
- **Temporal & spatial constraints.** Hard to reach 3 different code locations sequentially with no events occurring in between invalidating the sequence
- **Often silence.** No segmentation fault but the consequence can be severe because the program continues to run, contaminating other data

Directed Greybox Fuzzing

```
1 int *p, *p_alias;
2 char buf[10];
3 void bad_func(int *p) {free(p);} /* exit() is missing */
4 void func() {
5     if (buf[1] == 'F')
6         bad_func(p);
7     else /* lots more code ... */
8 }
9 int main (int argc, char *argv[]) {
10     int f = open(argv[1], O_RDONLY);
11     read(f, buf, 10);
12     p = malloc(sizeof(int));
13     if (buf[0] == 'A'){
14         p_alias = malloc(sizeof(int));
15         p = p_alias;
16     }
17     func();
18     if (buf[2] == 'U')
19         *p_alias = 1;
20     return 0;
21 }
```



Derivation of a target trace:

- Target trace: UAF events + call sites
- UAF events: 14 (alloc) → 3 (free) → 19 (use)
- Simple dependency analysis: 17 → 6 → 3
- A target trace: 14 (alloc) → 17 → 6 → 3 (free) → 19 (use)

Solution Schedule seeds that fuzz target traces more

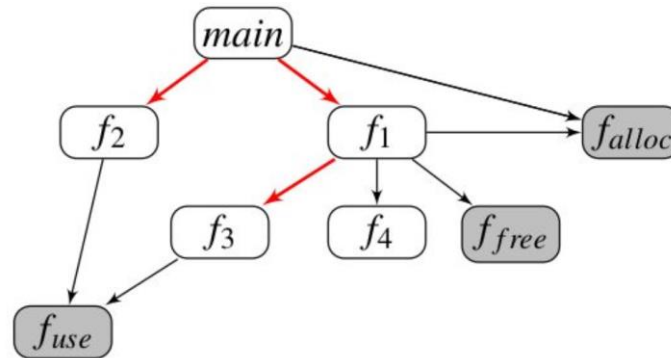
UAFuzz: Power Schedule

- UAFuzz: A state-of-the-art fuzzer for use-after-free (UAF) bugs
- Presented at the **blackhat** conference in August 2020.
- Energy (E) = $E_f * E_e * E_b$
- Function level (E_f): Prioritize function call relations that cover UAF events
- Edge level (E_e): Prioritize edges in a control flow graph that can reach targets more often
- Block level (E_b): Prioritize traces that are similar to the target trace

Nguyen et al., Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. USENIX RAID 20

UAFuzz: Function-level Energy (E_f)

- UAF events: f_{alloc} , f_{free} and f_{use}
- Select function call relations that cover more than one UAF event
 - $\text{main} \rightarrow f_2$ covers f_{alloc} and f_{use}
 - $\text{main} \rightarrow f_1$ covers f_{alloc} and f_{free}
 - $f_1 \rightarrow f_3$ covers f_{alloc} , f_{free} and f_{use}
- Assign higher energy to a seed whose execution triggers the selected call relations

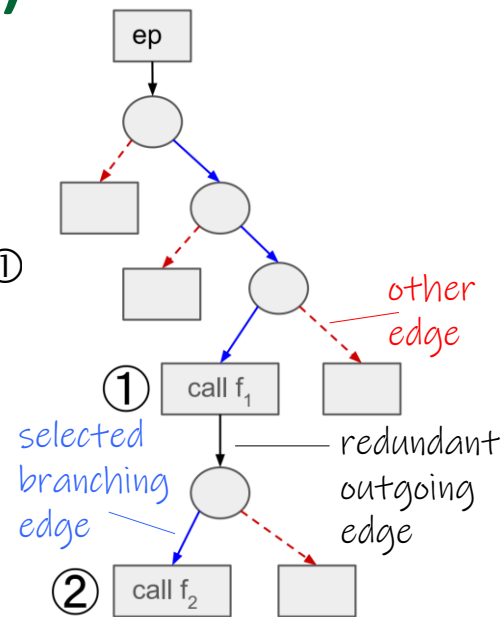


Call Graph Example, **selected call relations (caller, callee) in red**

$\text{Covers}(x \rightarrow y) = \text{union of child events of } x \text{ and child events of } y$

UAFuzz: Edge-level Energy (E_e)

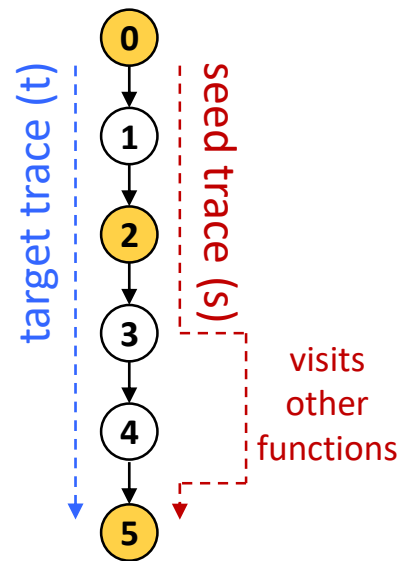
- Targets: Next node in a target trace
 - Target trace: ① → ②
 - From entry point (ep), select branching edges that can reach ①
 - After reaching ①, select edges that can reach ②
- Assign higher energy to a seed whose execution (a) visits the selected edges (blue colored ones) more often and (b) visits the other edges (red colored ones) less often
- Ignore the redundant outgoing edges (black colored) from ep and targets



Control flow graph, **selected branching edges are in blue**
black edges are redundant

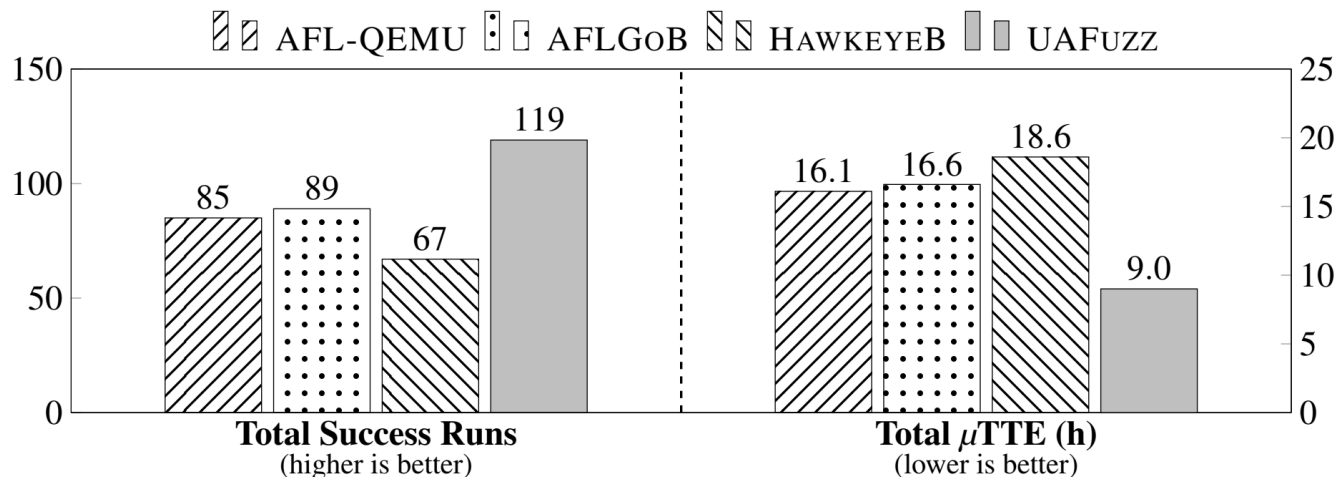
UAFuzz: Block-level Energy (E_b)

- Similarity between target and seed traces
 - Common prefix: common trace before divergence
 - E.g., $\textcircled{0} \rightarrow \textcircled{1} \rightarrow \textcircled{2} \rightarrow \textcircled{3}$
 - Target prefix ($s1$): #nodes in the common prefix, e.g., 4
 - UAF prefix ($s2$): #UAF events in the common prefix, e.g., 2
 - Target bag ($s3$): #nodes common, e.g., 5
 - UAF bag ($s4$): #UAF events common, e.g., 3
- Assign higher energy to a seed whose trace has higher similarity with weights $s1 > s2 > s3 > s4$



Trace similarity (UAF events are colored in yellow, where $\textcircled{0}$ alloc, $\textcircled{2}$ free and $\textcircled{5}$ use)

Comparison on 13 UAF Vulnerabilities



Assign a time budget to each of the 13 vulnerable projects. Add up the #success runs that trigger the vulnerability in each project.

Average the time to trigger the vulnerability in each project within the assigned time budget.

AFL-QEMU: AFL emulation mode

AFL-GoB: Binary version of Aflgo. <https://github.com/aflgo/aflgo>, 2020.

HawkeyeB: Binary version of Hawkeye. Chen et al., Hawkeye: towards a desired directed grey-box fuzzer. CCS18.

UAFuzz: Nguyen et al., Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. USENIX RAID 20

30 New Zero-day Vulnerabilities Found

11 new UAF and DF bugs
7 new CVEs assigned
27 confirmations
20 fixes

| Program | Code Size | Version (Commit) | Bug ID | Vulnerability Type | Crash | Vulnerable Function | Status | CVE |
|-----------|-----------|--------------------|---------|--------------------------|-------|--------------------------------|-----------|----------------|
| GPAC | 545K | 0.7.1 (987169b) | #1269 | User after free | ✗ | gf_m2ts_process_pmt | Fixed | CVE-2019-20628 |
| | | 0.8.0 (56eaea8) | #1440-1 | User after free | ✗ | gf_isom_box_del | Fixed | CVE-2020-11558 |
| | | 0.8.0 (56eaea8) | #1440-2 | User after free | ✗ | gf_isom_box_del | Fixed | Pending |
| | | 0.8.0 (56eaea8) | #1440-3 | User after free | ✗ | gf_isom_box_del | Fixed | Pending |
| | | 0.8.0 (5b37b21) | #1427 | User after free | ✓ | gf_m2ts_process_pmt | | |
| | | 0.7.1 (987169b) | #1263 | NULL pointer dereference | ✓ | ilst_item_Read | Fixed | |
| | | 0.7.1 (987169b) | #1264 | Heap buffer overflow | ✓ | gf_m2ts_process_pmt | Fixed | CVE-2019-20629 |
| | | 0.7.1 (987169b) | #1265 | Invalid read | ✓ | gf_m2ts_process_pmt | Fixed | |
| | | 0.7.1 (987169b) | #1266 | Invalid read | ✓ | gf_m2ts_process_pmt | Fixed | |
| | | 0.7.1 (987169b) | #1267 | NULL pointer dereference | ✓ | gf_m2ts_process_pmt | Fixed | |
| | | 0.7.1 (987169b) | #1268 | Heap buffer overflow | ✓ | BS_ReadByte | Fixed | CVE-2019-20630 |
| | | 0.7.1 (987169b) | #1270 | Invalid read | ✓ | gf_list_count | Fixed | CVE-2019-20631 |
| | | 0.7.1 (987169b) | #1271 | Invalid read | ✓ | gf_odf_delete_descriptor | Fixed | CVE-2019-20632 |
| GNU patch | 7K | 0.8.0 (5b37b21) | #1445 | Heap buffer overflow | ✓ | gf_bs_read_data | Fixed | |
| | | 0.8.0 (5b37b21) | #1446 | Stack buffer overflow | ✓ | gf_m2ts_get_adaptation_field | Fixed | |
| | | 2.7.6 (76e7758) | #56683 | Double free | ✓ | another_hunk | Confirmed | CVE-2019-20633 |
| | | 2.7.6 (76e7758) | #56681 | Assertion failure | ✓ | pch_swap | Confirmed | |
| Perl 5 | 184K | 2.7.6 (76e7758) | #56684 | Memory leak | ✗ | xmalloc | Confirmed | |
| | | 5.31.3 (a3c7756) | #134324 | Use after free | ✓ | S_reg | Confirmed | |
| | | 5.31.3 (a3c7756) | #134326 | Use after free | ✓ | Perl_regnext | Fixed | |
| | | 5.31.3 (a3c7756) | #134329 | User after free | ✓ | Perl_regnext | Fixed | |
| | | 5.31.3 (a3c7756) | #134322 | NULL pointer dereference | ✓ | do_clean_named_objs | Confirmed | |
| | | 5.31.3 (a3c7756) | #134325 | Heap buffer overflow | ✓ | S_reg | Fixed | |
| | | 5.31.3 (a3c7756) | #134327 | Invalid read | ✓ | S_regmatch | Fixed | |
| | | 5.31.3 (a3c7756) | #134328 | Invalid read | ✓ | S_regmatch | Fixed | |
| MuPDF | 539K | 5.31.3 (45f8e7b) | #134342 | Invalid read | ✓ | Perl_mro_isa_changed_in | Confirmed | |
| | | 1.16.1 (6566de7) | #702253 | Use after free | ✗ | fz_drop_band_writer | Fixed | |
| Boolector | 79K | 3.2.1 (3249ae0) | #90 | NULL pointer dereference | ✓ | set_last_occurrence_of_symbols | Confirmed | |
| fontforge | 578K | 20200314 (1604c74) | #4266 | Use after free | ✓ | SFDGetBitmapChar | | |
| | | 20200314 (1604c74) | #4267 | NULL pointer dereference | ✓ | SFDGetBitmapChar | | |

Summary of Directed Greybox Fuzzing

UAFuzz demonstrates the major design components of directed greybox fuzzing:

- Information that can be extracted from the target bug type
- Power schedule that smartly assigns energy based on:
 - Distance between a seed's execution from the targets
 - Frequency that the targets are executed by a seed

Enhancing the Generator

MOPT [USENIX Security 2019]

Optimized Mutation Scheduling for Fuzzers

Problem

Generate input mutants to discover more paths and more vulnerability bugs

Solution

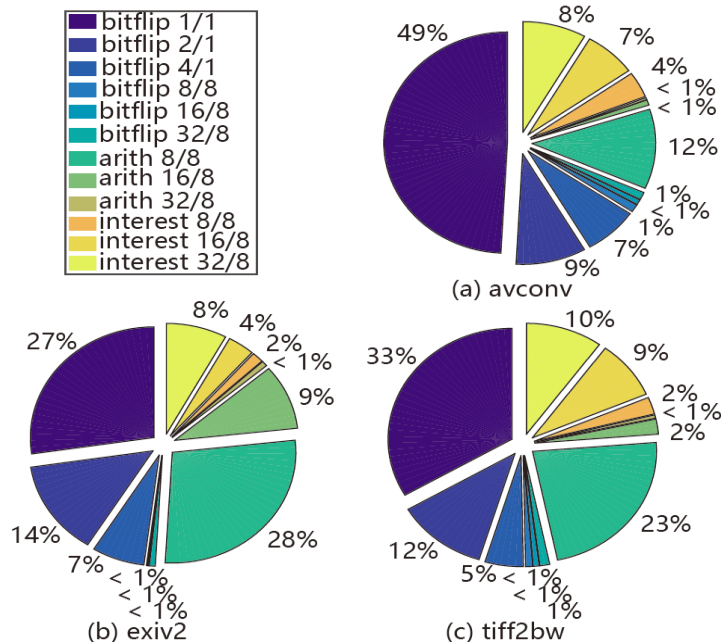
Dynamically find an optimal probability distribution of mutation operators

| Type | Meaning | Operators |
|-------------------------------|---|---|
| bitflip | Invert one or several consecutive bits in a test case, where the stepover is 1 bit. | bitflip 1/1, bitflip 2/1, bitflip 4/1 |
| byteflip | Invert one or several consecutive bytes in a test case, where the stepover is 8 bits. | bitflip 8/8, bitflip 16/8, bitflip 32/8 |
| arithmetic inc/dec | Perform addition and subtraction operations on one byte or several consecutive bytes. | arith 8/8, arith 16/8, arith 32/8 |

Mutation operators
defined by AFL

MOPT [USENIX Security 2019]

Optimized Mutation Scheduling for Fuzzers



- Mutation operators are not equally effective in generating interesting input mutants
- bitflip 1/1 and bitflip 2/1 are more effective than others
- Apply an optimization algorithm to dynamically determine the best next mutation operators based on the generated input mutants' usefulness
- Dynamically optimizing the mutation operators can find 382% more crashes and 118% more unique paths as compared with AFL

Enhancing the Updater

AnKou [ICSE 2020]

Guiding Grey-box Fuzzing towards Combinational Difference

Problem

Bugs may not be triggered even their concerned branches are covered

Solution

Needs to consider also the number of times a branch has been covered

Buffer Overflow Bugs

```
if (...) b1;    t1 = (1, 1, 1)
if (...) b2;    t2 = (1, 0, 2)
while (...) b3; t3 = (1, 1, 0)
               t4 = (0, 1, 8).
```

- AFL: Test input of t1 alone can cover all three branches. Other test inputs are not updated to the seed corpus because they are redundant.
- AnKou: Treat the number of branch visits as a vector. Determine redundancy based on the Euclidean distances of such vectors.

AnKou [ICSE 2022]

Guiding Grey-box Fuzzing t

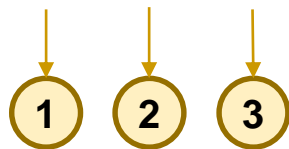
Problem

Bugs may not be tr

Solution

Needs to consider

Buffer Overflow Bugs



$t1 = (1, 0, 2)$

$t2 = (1, 1, 1)$

$t3 = (1, 1, 0)$

$t4 = (0, 1, 8).$

| Package Name | # of Bugs Found | | # of Bugs Found | |
|----------------|-----------------|-----|--------------------|--------|
| | Ankou | AFL | Ankou [†] | Angora |
| binutils | 11 | 36 | 11 | 4 |
| bison | 58 | 71 | 36 | 2 |
| catdoc | 0 | 15 | 0 | 0 |
| cflow | 21 | 18 | 13 | 1 |
| clamav | 0 | 0 | 0 | 0 |
| dwarf | 2 | 2 | 2 | 2 |
| GraphicsMagick | 22 | 27 | 21 | 11 |
| jasper | 42 | 37 | 0 | 0 |
| libav | 1 | 7 | 0 | 0 |
| libexiv2 | 82 | 59 | 16 | 4 |
| libgxps | 5 | 5 | 0 | 0 |
| liblouis | 18 | 11 | 0 | 0 |
| libming | 84 | 60 | 0 | 0 |
| libncurses | 48 | 53 | 0 | 0 |
| libraw | 2 | 4 | 0 | 0 |
| libsass | 155 | 12 | 0 | 0 |
| libtasn1 | 0 | 0 | 0 | 0 |
| libtiff | 3 | 2 | 0 | 0 |
| libtorrent | 0 | 0 | 0 | 0 |
| mpg123 | 0 | 0 | 0 | 0 |
| nasm | 28 | 12 | 0 | 0 |
| pspp | 168 | 99 | 0 | 0 |
| tcpdump | 0 | 0 | 0 | 0 |
| vim | 2 | 2 | 0 | 0 |
| Total | 752 | 532 | 99 | 24 |

[†] As mentioned in §6.6.1, we were not able to compile Angora on all the packages. For fair comparison, we report bugs found by Ankou only on the subjects that Angora was able to run on.

Dynamic Symbolic Execution (a.k.a. Concolic Testing)

WHITEBOX FUZZING

Dynamic Symbolic Execution

- Whitebox fuzzing originates from Patrice Godefroid (Microsoft Research) when deploying dynamic symbolic execution (i.e., concolic testing) to detect software vulnerabilities
- Microsoft Research is maintaining a team that develop a whitebox fuzzer called SAGE
- SAGE has solved 1 Billion+ constraints
- SAGE exposed many Win7 security bugs
- SAGE is now used daily by the Windows and Office teams

Godefroid et al., Grammar-based Whitebox Fuzzing. PLDI 2008

Summary

■ We discussed multiple fuzzing techniques

- ❑ PDF readers: Acrobat and Preview
- ❑ iPhone vulnerability based on browser rendering png files
- ❑ LangFuzz: JavaScript fuzzer
- ❑ GreyOne: Data flow sensitive fuzzing
- ❑ UAFuzz: Directed greybox fuzzer for Use-after-Free vulnerabilities
- ❑ MOPT: Optimized mutation scheduling for fuzzers
- ❑ AnKou: Guiding grey-box fuzzing towards combinatorial difference

■ Some general themes

- ❑ Choosing good inputs is very important
 - Very unlikely to find significant crashes by chance
 - Use many samples and simplify them using information about system
- ❑ Standard tools can diagnose crashes, find vulnerabilities
- ❑ Lots of hard work and only small chances of success in many cases
 - But developer teams should use some of these methods too (easier for them!)