Software Security
Dynamic analysis and fuzz testing

Bhargava Shastry

Prof. Jean-Pierre Seifert
Security in Telecommunications
TU Berlin

SoSe 2016
What is this lecture about?

- How do we find bugs in programs?
  - ... by **executing** an open-source program
  - ... applies to binary code in principle

- Let’s recap
  - Last lecture, we discussed static analysis → Find bugs without program execution
  - How does it compare with Dynamic analysis? Pros and cons?
Motivation

Why do this?

- Find an input to the program that leads to undefined/insecure behavior
  - Typically, we are looking for an input that crashes a program
  - Program crashes provide an entry point to craft exploits → Not all program crashes are exploitable
- Technically, this is just another way to find bugs in programs
Outline

- Fuzz testing
- Black-box vs. White-box testing
- Dynamic analysis
- Limitations
- Conclusion
Example 1

Which input make this program crash?

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, charargv[]) {
    char buf[20];

    while (_AFL_LOOP(1000)) {
        memset(buf, 0, 20);
        if (read(0, buf, 19) < 0) {
            perror("read");
            return 1;
        }

        if (buf[0] != 'p')
            printf("first letter is not p\n");
        else if (buf[1] != 'w')
            printf("second letter is not w\n");
        else if (buf[2] != 'n')
            printf("third letter is not n\n");
        else
            abort();

        printf("buf contains %s\n", buf);
    }

    return 0;
}
```
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char buf[20];

    while (AFL_LOOP(1000)) {
        memset(buf, 0, 20);
        if (read(0, buf, 19) < 0) {
            perror("read");
            return 1;
        }
        if (buf[0] != 'p')
            printf("first letter is not p\n");
        else if (buf[1] != 'w')
            printf("second letter is not w\n");
        else if (buf[2] != 'n')
            printf("third letter is not n\n");
        else
            abort();

        printf("buf contains %s\n", buf);
    }
    return 0;
}
```

### pwn

Pwn is a leetspeak slang term derived from the verb own, as meaning to appropriate or to conquer to gain ownership. - Wikipedia
Fuzz testing aka Fuzzing

- Idea attributed to Prof. Barton Miller (Miller, Fredriksen, & So, 1990)

  ...in the Fall of 1988, there was a wild midwest thunderstorm... With the heavy rain, there was noise on the (dial-up) line and that noise was interfering with my ability to type sensible commands to the shell. — Prof. Miller

- Feed random inputs to a program → No knowledge of program internals

- Idea bore very good initial results → crash or hang between 25-33% of the Unix utility programs
Let's fuzz!

### American Fuzzy Lop 2.10b (fuzz-example1)

<table>
<thead>
<tr>
<th>Process Timing</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time: 0 days, 0 hrs, 0 min, 21 sec</td>
<td>Cycles done: 4</td>
</tr>
<tr>
<td>Last new path: 0 days, 0 hrs, 0 min, 20 sec</td>
<td>Total paths: 5</td>
</tr>
<tr>
<td>Last uniq crash: 0 days, 0 hrs, 0 min, 17 sec</td>
<td>Uniq crashes: 2</td>
</tr>
<tr>
<td>Last uniq hang: none seen yet</td>
<td>Uniq hangs: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle Progress</th>
<th>Map Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now processing: 1 (20.00%)</td>
<td>Map density: 19 (0.03%)</td>
</tr>
<tr>
<td>Paths timed out: 0 (0.00%)</td>
<td>Count coverage: 1.00 bits/tuple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Progress</th>
<th>Findings in Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now trying: havoc</td>
<td>Favored paths: 3 (60.00%)</td>
</tr>
<tr>
<td>Stage execs: 4340/15.0k (28.93%)</td>
<td>New edges on: 5 (100.00%)</td>
</tr>
<tr>
<td>Total execs: 252k</td>
<td>Total crashes: 22 (2 unique)</td>
</tr>
<tr>
<td>Exec speed: 10.8k/sec</td>
<td>Total hangs: 0 (0 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fuzzing Strategy Yields</th>
<th>Path Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit flips: 1/144, 1/139, 0/129</td>
<td>Levels: 3</td>
</tr>
<tr>
<td>Byte flips: 0/18, 0/13, 0/5</td>
<td>Pending: 0</td>
</tr>
<tr>
<td>Arithmetics: 1/1008, 0/0, 0/0</td>
<td>Pend fav: 0</td>
</tr>
<tr>
<td>Known ints: 0/105, 0/364, 0/220</td>
<td>Own finds: 4</td>
</tr>
<tr>
<td>Dictionary: 0/0, 0/0, 0/0</td>
<td>Imported: n/a</td>
</tr>
<tr>
<td>Havoc: 3/180k, 0/66.0k</td>
<td>Variable: n/a</td>
</tr>
<tr>
<td>Trim: 18.18%/2, 0.00%</td>
<td></td>
</tr>
</tbody>
</table>

[cpu: 69%]
Black-box testing

- Black-box testing treats the program under test as a black-box (opaque to tester)
- This means the testing framework feeds random inputs to the program
- However, random inputs alone are unlikely to trigger bugs
  - \[ x = \frac{1000}{(input + 100000)}, \text{ how long will it take the fuzzer to feed} \]
  - \[ input = -100000? \]
- Fuzzing, in its initial form, was conceived as a black-box testing technique
  - Ever since, it has evolved into a white box testing technique
  - State of the art fuzzers like afl (Zalewski, n.d.) “learn” from program behavior
While-box testing treats the PUT as a white-box (transparent to tester)

Program internals can be used to reduce input search space

- Techniques: Program coverage guided fuzzing, Concolic execution etc.
- However, these need a model that abstracts program behavior for generating inputs
- Uses program instrumentation to realize the model \( \Rightarrow \) Access to program binary or source code
- Instrumentation may be static (compile-time) or dynamic (run-time)
Coverage guided fuzzing

- Fuzz inputs based on program coverage information
- For instance, retain only those input mutations that lead to new coverage
Each colour indicates a different program path

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    ...
    ...
    if (buf[0] != 'p')
        // buf[0] != 'p'
        printf("first letter is not p\n");
    else if (buf[1] != 'w')
        // buf[0] == 'p' && buf[1] != 'w'
        printf("second letter is not w\n");
    else if (buf[2] != 'n')
        printf("third letter is not n\n");
    else
        abort();
    ...
    ...
}
```
But how does the fuzzer **know** which program path was taken?

Answer: Instrument the program

What does the instrumentation look like?

```c
cur_location = random-number;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Each branch point indexes a byte in global state

State captures number of times branch point taken

Comparing state changes per input, it is possible to filter seemingly redundant inputs
Thought experiment

- Is the state captured by the instrumentation good enough?
- In other words, can you think of an example where the instrumentation might discard potentially crashing inputs

  Only a **heuristic** → But, turns out to be very good in practice
Program instrumentation can be done at compile-time or at run-time

Compile-time instrumentation incurs low performance overhead
- Checking logic can be optimized before-hand
- Slow down only because of additional instructions and associated I/O
- But, hard to encode checking logic for dynamic language features in general

Run-time instrumentations incurs relatively high overhead
- Checking logic needs to be inserted at program run time
- Typically implemented using binary translation \(\rightarrow\) very slow
- Since instrumentation occurs at run time, dynamism (e.g., indirect branches) can be handled relatively easily

Rest of the lecture looks at dynamic analyses that use compile-time instrumentation
Dynamic analysis

- Why bother if a good fuzzer gives us crashing inputs?
  - A **normal** program crash happens only when there is a serious problem
  - Example: Segmentation fault, assertion failure, stack canary failures etc.
  - But, it does not uncover latent faults
  - Example: Heap corruption, stack corruption but canary intact etc.

- Dynamic analysis can help uncover **latent faults**
- Can be used in conjunction with a fuzzer for proactive security audits
AddressSanitizer (Serebryany, Bruening, Potapenko, & Vyukov, 2012) is an instrumentation framework for detecting multiple classes of memory corruption faults:

- Heap/stack buffer overflows, use-after-free bugs

Idea is similar to coverage guided fuzzing i.e., insert bounds checks by instrumenting the program

Each word of addressable heap maps to a byte of shadow memory

Uses custom memory allocation functions to set shadow byte during allocation

Uses shadow lookups during load and store instructions to detect faults e.g. loading from/storing to unaddressable memory location
ASan Instrumentation

- Every 8-byte aligned word on x86_64 has 9 states
  - First \( n \) bytes \( 0 \leq n \leq 8 \) are addressable, rest not \( \implies \) 1 shadow byte for 8 bytes of application memory

- Instrumentation looks like

  ```c
  char *shadow = MemToShadow(addr);
  if (*shadow && *shadow <= func(addr))
     ReportError(addr)
  ...
  ```
Fuzz testing and dynamic analysis covered in the lecture are program-input centric. This means, if a fuzzed input cannot uncover a program path, bugs in that program path cannot be found. Data on false negatives (bugs missed) by fuzzers is hard to come by. Exception: Recent research on bug insertion (Dolan-Gavitt et al., 2016). Preliminary results show that only 10% of inserted bugs found by a SotA fuzzer. Future: Use symbolic execution to maximize path coverage? Awaiting a public study of efficacy.
Conclusions

- Dynamic analysis attempts to uncover bugs during program execution
- We discussed fuzz testing and program instrumentation
- Discussed techniques and tools remain relevant for real-world security audits
  - Anecdotal evidence shows that over 90% of C/C++ bugs found using a white-box fuzzer
- Recent data on bugs missed by fuzzing tells us that there is room for improvement


http://lcamtuf.coredump.cx/afl/.