

CS165 – Computer Security

Dynamic (Fuzz) Testing

November 4, 2024

Our Goal

- We want to develop techniques to detect vulnerabilities automatically before they are exploited
 - ▣ What's a vulnerability?
 - ▣ How to find them?



Vulnerability

- How do you define computer ‘vulnerability’?
 - *Flaw*
 - *Accessible to an adversary*
 - *Adversary has ability to exploit*



Problem



- How do we know if your **program has a flaw?**
 - ▣ May be likely, but not guaranteed
- More importantly, how do we **locate a flaw?**
 - ▣ To assess whether it is vulnerable
 - ▣ Or better yet, to fix the flaw

Example

□ Can you find the flaw(s)?

```
1  int
2  im_vips2dz( IMAGE *in, const char *filename ){
3      char *p, *q;
4      char name[FILENAME_MAX];
5      char mode[FILENAME_MAX];
6      char buf[FILENAME_MAX];
7      ...
8
9      im_strncpy( name, filename, FILENAME_MAX );
10     if( (p = strchr( name, ':' )) ){
11         *p = '\0';
12         im_strncpy( mode, p + 1, FILENAME_MAX );
13     }
14
15     strcpy( buf, mode );
16     p = &buf[0];
17     ...
18 }
```

Flaw Evidence



- What indicates that your program has a flaw?

Flaw Evidence

- What indicates that your program has a flaw?
- A **crash** (i.e., memory error)
 - ▣ Means that an instruction accessed an illegal memory location
 - ▣ **First example** – read/write beyond bounds
- A **hang** (i.e., infinite loop)
 - ▣ Some loop condition check has an error
 - ▣ **Second example** - Not check for EOF

Find Flaws



- How can we find flaws?
 - ▣ Run the program
 - ▣ When it hangs/crashes, we have found a flaw
- Challenge
 - ▣ Flaw may only be triggered by particular inputs
 - ▣ The task of producing inputs to test your program for flaws is called **dynamic analysis**

Dynamic Analysis Options



□ Regression Testing

- ▣ Run program on many **normal** inputs and look for bad behavior in the responses
 - Typically looking for behavior that differs from expected – e.g., a previous version of the program

□ Fuzz Testing

- ▣ Run program on many **abnormal** inputs and look for bad behavior in the responses
 - Looking for behaviors that may cause the program to stop executing at all – crash or hang

Fuzz Testing



- Fuzz Testing
 - ▣ Idea proposed by Bart Miller at Wisconsin in 1988
- **Problem:** People assumed that utility programs could correctly process any input values
 - ▣ But, untrusted programs could run them
 - ▣ Supply any inputs they wanted (command line)
- Found that they could crash 25-33% of UNIX utility programs

Fuzz Testing



- Fuzz Testing
 - ▣ Idea proposed by Bart Miller at Wisconsin in 1988
- Approach
 - ▣ Generate random inputs
 - ▣ Run lots of programs using random inputs
 - ▣ Identify crashes of these programs
 - ▣ Correlate with the random inputs that caused the crashes
- **Problems:** Crashes and hangs

Example Found

□ Fuzz Testing

▣ Produce random inputs for processing

```
format.c (line 276):
```

```
...  
while (lastc != '\n') {  
    rdc();  
}  
...
```

```
input.c (line 27):
```

```
rdc()  
{ do { readchar(); } // assigns 'lastc' to 0  
while (lastc == ' ' || lastc == '\t'); return (lastc);  
}
```

▣ Eventually produce line with EOF in the middle

Fuzz Testing



- **Idea:** Search for flaws in a program by running the program under a variety of inputs
- **Challenge:** Selecting input values for the program
 - ▣ What should be the goals in choosing input values for fuzz testing?

Challenges



- **Idea:** Search for flaws in a program by running the program under a variety of inputs
- **Challenge:** Selecting input values for the program
 - ▣ What should be the goals in choosing input values for fuzz testing?
 - ▣ ***Find as many exploitable flaws as possible***
- **Implies**
 - ▣ Maximize code coverage (branches)
 - ▣ Generate inputs that cause crash/hang

Black Box Fuzzing

- Like Miller – Feed the program random inputs and see if it crashes
- **Pros:** Easy to configure
- **Cons:** May not search efficiently
 - ▣ May re-run the same path over again (low coverage)
 - ▣ May be very hard to generate inputs for certain paths (checksums, hashes, restrictive conditions)
 - ▣ May cause the program to terminate for logical reasons – fail format checks and stop

Black Box Fuzzing

- May be difficult to pass “authenticate_user” with random inputs

```
function( char *name, char *passwd, char *buf )
{
    if ( authenticate_user( name, passwd ) ) {
        if ( check_format( buf ) ) {
            update( buf );
        }
    }
}
```


Mutation-Based Fuzzing

- Supply a **well-formed input**
 - ▣ Generate random changes to that input
- No assumptions about modified input
 - ▣ Only assumes that variants of the well-formed input will be effective in fuzzing
- Example: **zzuf**
 - ▣ <https://fuzzing-project.org/tutorial1.html>
 - ▣ **Reading:** The Beginners' Guide to Fuzzing

Mutation-Based Fuzzing

- Example: zzuf
 - ▣ <https://fuzzing-project.org/tutorial1.html>
- The Beginners' Guide to Fuzzing
 - ▣ `zzuf -s 0:1000000 -c -C 0 -q -T 3
objdump -x win9x.exe`
 - ▣ Fuzzes the program `objdump` using the sample input executable `win9x.exe`
 - ▣ Try IM seed values (-s) from command line (-c) and keep running if crashed (-C 0) with timeout (-T 3)

Mutation-Based Fuzzing

- Easy to setup, and not dependent on program details
- But may be strongly biased by the initial input
- Still prone to some problems
 - ▣ May re-run the same path over again (same test)
 - ▣ May be very hard to generate inputs for certain paths (checksums, hashes, restrictive conditions)
 - ▣ **May not generate a legal value for executable** (e.g., not constrained to legal instruction)

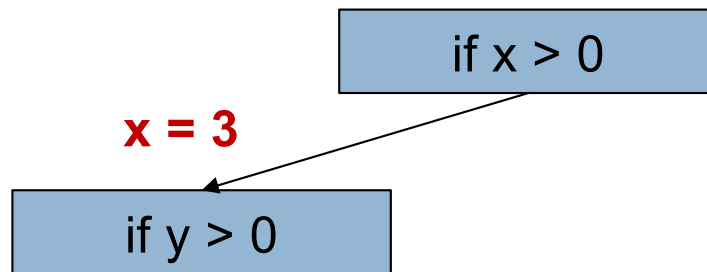
Grey Box Fuzzing



- Rather than treating the program as a black box, instrument the program to track the paths run
- Save inputs that lead to new paths
 - ▣ Associated with the paths they exercise
 - ▣ To bias toward running new paths
- Example
 - ▣ **American Fuzzy Lop** (AFL)
- “State of the practice” at this time

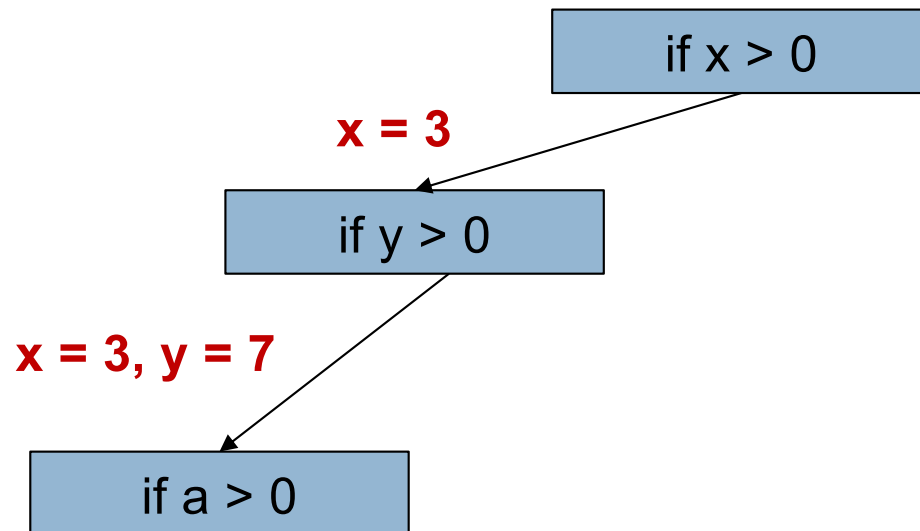
Grey Box Fuzzing

- Logical operation – instrument conditionals to record inputs that caused particular branches to run



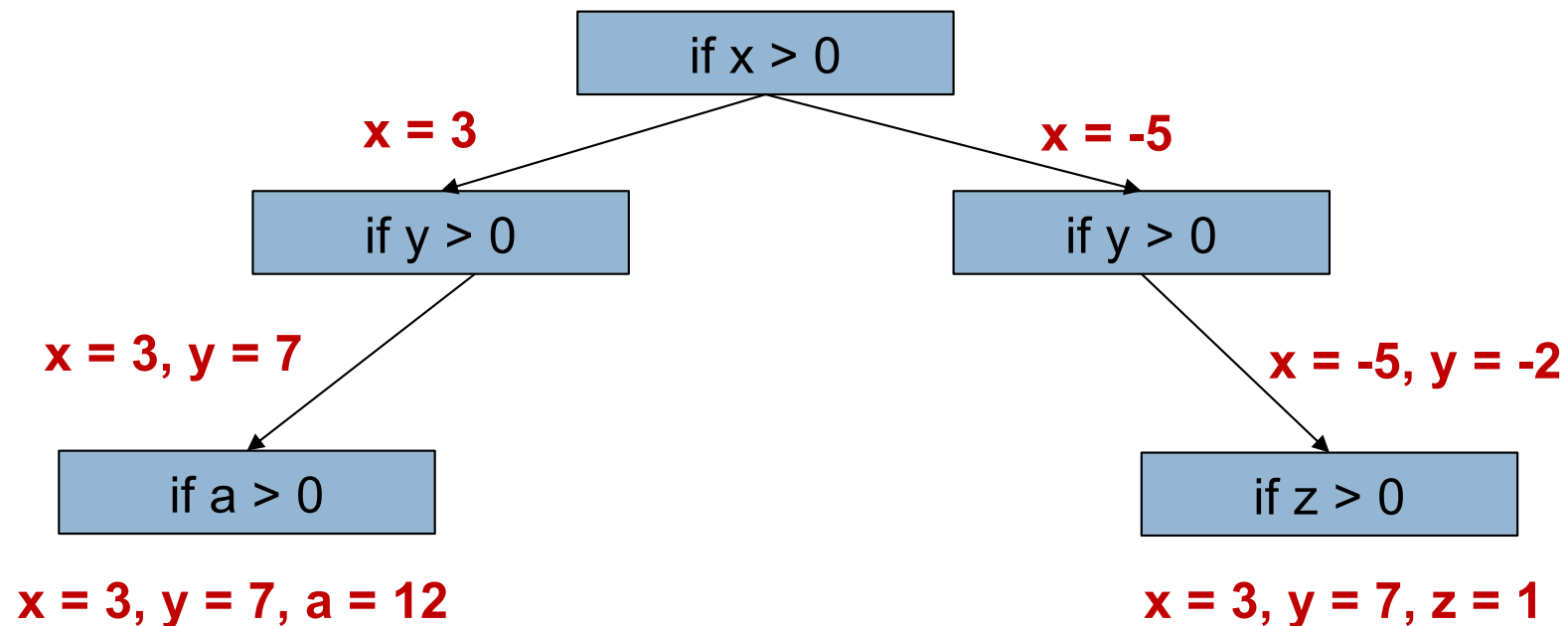
Grey Box Fuzzing

- Logical operation – instrument conditionals to record inputs that caused particular branches to run



Grey Box Fuzzing

- Logical operation – instrument conditionals to record inputs that caused particular branches to run



Track the branch coverage and generate inputs to explore new branches

AFL

- Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats



AFL



- Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats
- See
 - ▣ <http://lcamtuf.coredump.cx/afl/>

AFL Build

- ❑ Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats
- ❑ Replace the gcc compiler in your build process with afl-gcc
- ❑ For example, in the Makefile
 - ❑ `CC=path-to/afl-gcc`
- ❑ Then build your target program with afl-gcc
 - ❑ Generates a binary instrumented for AFL fuzzing

AFL Use

- Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats

- Run the fuzzer using afl-fuzz

```
path-to/afl-fuzz -i <input-dir> -o <output-dir> <path-to-bin> [args]
```

- For example

```
path-to/afl-fuzz -i input/ -o output/ ./cmpsc447-p3 set user passwd @@
```

- Where

- ▣ `input/` directory with the input file
- ▣ `output/` is the directory where the AFL results will be placed

AFL Use

- Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats

- Run the fuzzer using afl-fuzz

```
path-to/afl-fuzz -i <input-dir> -o <output-dir> <path-to-bin> [args]
```

- For example

```
path-to/afl-fuzz -i input/ -o output/ ./cmpsc497-p1 set user passwd @@
```

- Where

- ▣ @@ shows that the last arg (input file) will be fuzzed
- ▣ Can also fuzz “user” and “passwd”

AFL Display

- Tracks the execution of the fuzzer

```
american fuzzy lop 0.47b (readpng)

process timing
run time      : 0 days, 0 hrs, 4 min, 43 sec
last new path : 0 days, 0 hrs, 0 min, 26 sec
last uniq crash : none seen yet
last uniq hang : 0 days, 0 hrs, 1 min, 51 sec
cycle progress
new processing : 38 (19.49%)
paths timed out : 0 (0.00%)
stage progress
now trying : interest 32/8
stage execs : 0/9990 (0.00%)
total execs : 654k
exec speed : 2306/sec
fuzzing strategy yields
bit flips : 88/14.4k, 6/14.4k, 6/14.4k
byte flips : 0/1804, 0/1786, 1/1750
arithmetics : 31/126k, 3/45.6k, 1/17.8k
known ints : 1/15.8k, 4/65.8k, 6/78.2k
havoc : 34/254k, 0/0
trin : 2876 8/931 (61.45% gain)

overall results
cycles done : 0
total paths : 195
uniq crashes : 0
uniq hangs : 1
map coverage
map density : 1217 (7.43%)
count coverage : 2.55 bits/tuple
findings in depth
favored paths : 128 (65.64%)
new edges on : 85 (43.59%)
total crashes : 0 (0 unique)
total hangs : 1 (1 unique)
path geometry
levels : 8
pending : 178
pend fav : 114
reported : 0
variable : 0
latent : 0
```

- Key information are
 - ▣ “total paths” – number of different execution paths tried
 - ▣ “unique crashes” – number of unique crash locations

AFL Output

- Shows the results of the fuzzer
 - ▣ E.g., provides inputs that will cause the crash
- File “**fuzzer_stats**” provides summary of stats – UI
- File “**plot_data**” shows the progress of fuzzer
- Directory “**queue**” shows inputs that led to paths
- Directory “**crashes**” contains input that caused crash
- Directory “**hangs**” contains input that caused hang

AFL Operation

- How does AFL work?
 - ▣ http://lcamtuf.coredump.cx/afl/technical_details.txt
- The instrumentation captures branch (edge) coverage, along with coarse branch-taken hit counts.
 - ▣ `shared_mem[cur_location ^ prev_location]++;`
- Record branches taken (previous branch to current branch) with low collision rate
- Enables distinguishing unique paths

AFL Operation

- How does AFL work?
 - http://lcamtuf.coredump.cx/afl/technical_details.txt
- “When a mutated input produces an execution trace containing new tuples, the corresponding input file is preserved and routed for additional processing”
 - Otherwise, input is discarded
- “Mutated test cases that produced new state transitions [as above] are added to the input queue and used as a starting point for future rounds of fuzzing”

AFL Operation



- How does AFL work?
 - ▣ http://lcamtuf.coredump.cx/afl/technical_details.txt
- Fuzzing strategies
 - ▣ Highly deterministic at first – bit flips, add/sub integer values, and choose interesting integer values
 - ▣ Then, non-deterministic choices – insertions, deletions, and combinations of test cases

Grey Box Fuzzing



- Finds flaws, but still does not understand the program
- **Pros:** Much better than black box testing
 - ▣ Essentially no configuration
 - ▣ Lots of crashes have been identified
- **Cons:** Still a bit of a stab in the dark
 - ▣ May not be able to execute some paths
 - ▣ Searches for inputs independently from the program
- Need to improve the effectiveness further

Conclusions

- It is important to **detect vulnerabilities** in your programs before adversaries find them
- **Dynamic testing** has long been a way to find problems in your programs
 - ▣ But, we need a more comprehensive form of testing to detect vulnerabilities to memory errors
- **Fuzz testing** is designed to find memory errors in your programs
 - ▣ Generate inputs that: (1) run as much of the program as possible and (2) try values that may cause crash/hang

Questions

39

