Fuzzing

Lecture 10: CPEN 400P

Karthik Pattabiraman, UBC

(Slides are based on Gary Tan's CSE597: Topics in Software Testing at Penn State)

Outline

What is fuzzing ?

Black box Fuzzing

Gray Box and White-box Fuzzing

Fuzz Testing

- Run program on many random, abnormal inputs and look for bad behavior in the responses
  - Bad behaviors such as crashes or hangs

Fuzz Testing (Bart Miller, U. Of Wisconsin)

- A night in 1988 with thunderstorm and heavy rain
- Connected to his office Unix system via a dial up connection
- The heavy rain introduced noise on the line
- Crashed many UNIX utilities he had been using everyday
- He realized that there was something deeper
- Asked three groups in his grad-seminar course to implement this idea of fuzz testing
    - Two groups failed to achieve any crash results!
    - The third group succeeded! Crashed 25-33% of the utility programs on the seven Unix variants that they tested

Fuzz Testing

- Approach
  - Generate random inputs
  - Run lots of programs using random inputs
  - Identify crashes of these programs
  - Correlate random inputs with crashes
- Errors found: Not checking returns, Array indices out of bounds, not checking null pointers, …

Example Found

```
format.c (line 276):
...
while (lastc != '\n') {
    rdc();
}
...

input.c (line 27):
rdc()
{ do { readchar(); }
  while (lastc == ' ' || lastc == '\t');
  return (lastc);
}
```

When end of file, readchar() sets lastc to be 0; then the program hangs (infinite loop)

Fuzz Testing Types

- Black-box fuzzing
  - Treating the system as a blackbox during fuzzing; not knowing details of the implementation
- Grey-box fuzzing
- White-box fuzzing
  - Design fuzzing based on internals of the system

Outline

What is fuzzing ?

Black box Fuzzing

Gray Box and White-box Fuzzing

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

- Example that would be hard for black box fuzzing to find the error

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

Mutation-Based Fuzzing

- User supplies a well-formed input
- Fuzzing: Generate random changes to that input
- No assumptions about input
  - Only assumes that variants of well-formed input may be problematic
- Example: zzuf
  - https://github.com/samhocevar/zzuf
  - Reading: The Fuzzing Project Tutorial

Mutation-Based Fuzzing

- Easy to set up, 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)

Mutation-Based Fuzzing

- The Fuzzing Project Tutorial
  - zzuf -s 0:1000000 -c -C 0 -q -T 3 objdump -x win9x.exe
  - Fuzzes the program `objdump` using the sample input `win9x.exe`
  - Try 1M seed values (-s) from command line (-c) and keep running if crashed (-C 0) with timeout (-T 3)
  - Can be combined with address sanitizers such as valgrind

Generation-Based Fuzzing

- Generate inputs “from scratch” rather than using an initial input and mutating
- However, require the user to specify a format or protocol spec to start
  - Equivalently, write a generator for generating well-formatted input
- Examples include
  - SPIKE, PeachFuzz

Generation-Based Fuzzing

- Can be more accurate, but at a cost
- Pros: More complete search
  - Values more specific to the program operation
  - Can account for dependencies between inputs
- Cons: More work
  - Get the specification
  - Write the generator – ad hoc
  - Need to specify a format for each program

PeachFuzzer: Generation-based Fuzzer

```xml
<DataModel name="Header">
    <String name="Header" />
    <String value=": " />
    <String name="Value" />
    <String value="\r\n" />
</DataModel>

<DataModel name="HttpRequest">

    <!-- The HTTP request line: GET http://foo.com HTTP/1.0 -->
    <Block name="RequestLine">

        <String name="Method"/>
        <String value=" " type="char"/>
        <String name="RequestUri"/>
        <String value=" "/>
        <String name="HttpVersion"/>
        <String value="\r\n"/>
    </Block>


    <Block name="HeaderHost" ref="Header">
        <String name="Header" value="Host" isStatic="true"/>
    </Block>

    <Block name="HeaderContentLength" ref="Header">
        <String name="Header" value="Content-Length" isStatic="true"/>
        <String name="Value">
            <Relation type="size" of="Body"/>
        </String>
    </Block>

    <String value="\r\n"/>

    <Blob name="Body" minOccurs="0" maxOccurs="1"/>

</DataModel>
```

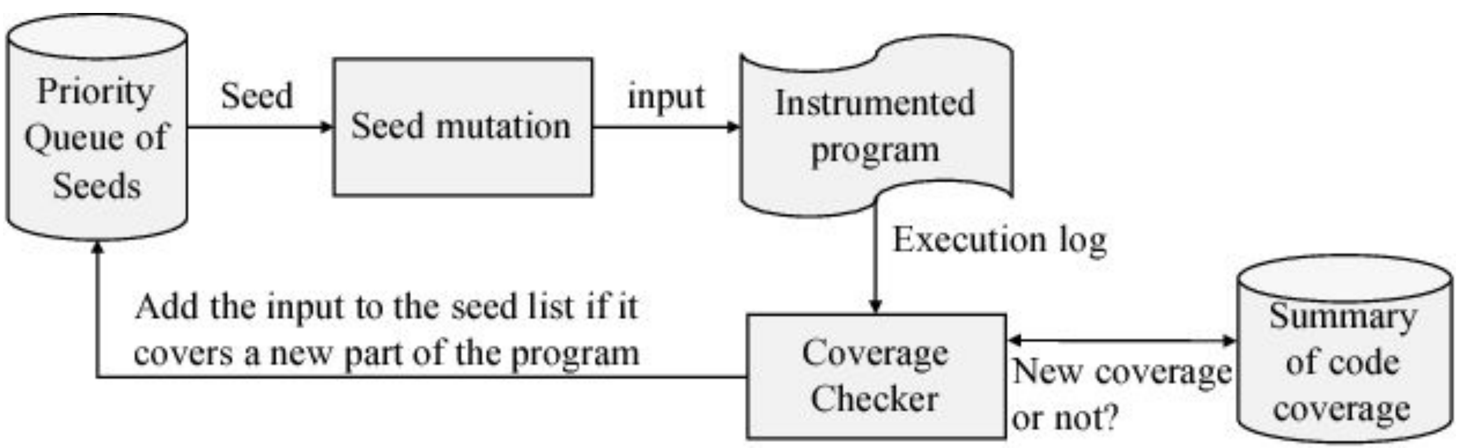
```
C:\peach>peach -1 --debug HTTP.xml

] Peach 2.3.8 Runtime
] Copyright (c) Michael Eddington

[*] Performing single iteration
[*] Optmizing DataModel for cracking: 'HttpRequest'
[*] Optmizing DataModel for cracking: 'HttpRequest'
[*] Starting run "DefaultRun"
[-] Test: "HttpGetRequestTest" (HTTP Request GET Test)
[1:?:?] Element: N/A
        Mutator: N/A

StateEngine.run: State1
StateEngine._runState: Initial

StateEngine._runAction: Named_38
Actiong output sending 105 bytes
>>>>>>>>>>>>>>>>
tcp.Tcp.send():
0000   47 45 54 20 68 74 74 70 3A 2F 2F 31 30 2E 31 37   GET http://.....
0010   38 2E 31 34 36 2E 32 35 35 20 48 54 54 50 2F 31              HTTP/1
0020   2E 31 0D 0A 48 6F 73 74 3A 20 68 74 74 70 3A 2F   .1..Host: http:/
0030   2F 31 30 2E 31 37 38 2E 31 34 36 2E 32 35 35 0D   /...............
0040   0A 43 6F 6E 74 65 6E 74 2D 4C 65 6E 67 74 68 3A   .Content-Length:
0050   20 31 38 0D 0A 0D 0A 54 65 73 74 20 46 75 7A 7A    18....Test Fuzz
0060   7A 69 6E 67 67 67 67 67 20                        zinggggg

-- Completed our iteration range, exiting
[-] Test "HttpGetRequestTest" completed
[-] Test: "HttpOptionsRequestTest" (HTTP Request OPTIONS Test)
[1:?:?] Element: N/A
        Mutator: N/A

tateEngine.run: State2
tateEngine._runState: Initial

StateEngine._runAction: Named_40
Actiong output sending 68 bytes
>>>>>>>>>>>>>>>>
tcp.Tcp.send():
0000   4F 50 54 49 4F 4E 53 20 2A 20 48 54 54 50 2F 31   OPTIONS * HTTP/1
0010   2E 31 0D 0A 48 6F 73 74 3A 20 0D 0A 43 6F 6E 74   .1..Host: ..Cont
0020   65 6E 74 2D 4C 65 6E 67 74 68 3A 20 31 38 0D 0A   ent-Length: 18..
0030   0D 0A 54 65 73 74 20 46 75 7A 7A 7A 69 6E 67 67   ..Test Fuzzzingg
0040   67 67 67 20                                       ggg

-- Completed our iteration range, exiting
[-] Test "HttpOptionsRequestTest" completed
[*] Run "DefaultRun" completed
C:\peach>_
```

Coverage-Based Fuzzing

- AKA grey-box fuzzing
- Rather than treating the program as a black box, instrument the program to track coverage
  - E.g., the edges covered
- Maintain a pool of high-quality tests
  1) Start with some initial ones specified by users
  2) Mutate tests in the pool to generate new tests
  3) Run new tests
  4) If a new test leads to new coverage (e.g., edges), save the new test to the pool; otherwise, discard the new test

AFL

- Example of coverage-based fuzzing
  - American Fuzzy Lop (AFL)
  - The original version is no longer maintained; afl++ is the newer version



AFL Build

- Provides compiler wrappers for gcc to instrument target program to track test coverage
- Replace the gcc compiler in your build process with afl-gcc
- Then build your target program with afl-gcc
    - Generates a binary instrumented for AFL fuzzing

Toy Example of Using AFL

```
int main(int argc, char* argv[]) {
  …
  FILE *fp = fopen(argv[1],"r");  …
  size_t len;
  char *line=NULL;
  if (getline(&line,&len,fp) < 0) {
      printf("Fail to read the file; exiting...\n");
      exit(-1);
  }

  long pos = strtol(line,NULL,10); …

  if (pos > 100) {if (pos < 150) { abort(); } }
  fclose(fp);  free(line);
  return 0;
}
```

* Omitted some error-checking code in "…"

Test Cases are Important for Speed

- For the toy example,
  - If the only test case is 55, it typically takes 3 to 15 mins to get a crashing input
  - If the test cases are 55 and 100, it typically takes only 1 min
    - Since crashing tests are in (100,150), the test is close to it syntactically; that’s why the fuzzing speed is faster

AFL Display

```
                american fuzzy lop 2.51b (cmpsc497-p1)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 2 hrs, 16 min, 32 sec      │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 13 min, 31 sec      │  total paths : 41     │
│ last uniq crash : 0 days, 0 hrs, 43 min, 58 sec      │ uniq crashes : 11     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 3 (7.32%)         │    map density : 0.11% / 0.40%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.62 bits/tuple      │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 8/8             │ favored paths : 6 (14.63%)            │
│ stage execs : 12.3k/41.9k (29.31%)  │  new edges on : 7 (17.07%)            │
│ total execs : 243k                  │ total crashes : 2479 (11 unique)      │
│  exec speed : 30.98/sec (slow!)     │  total tmouts : 10 (5 unique)         │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 7/15.4k, 32/15.4k, 0/15.4k            │    levels : 3          │
│  byte flips : 0/1929, 0/1926, 0/1920                │   pending : 39         │
│ arithmetics : 8/71.7k, 4/5434, 0/0                  │  pend fav : 5          │
│  known ints : 0/6938, 0/35.5k, 0/56.3k              │ own finds : 40         │
│  dictionary : 0/0, 0/0, 0/1270                      │  imported : n/a        │
│       havoc : 0/178, 0/0                            │ stability : 17.69%     │
│        trim : 0.00%/930, 0.00%                      ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 19%]
```

- 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
- Mutation 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

Outline

What is fuzzing ?

Black box Fuzzing

Gray Box and White-box Fuzzing

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

White Box Fuzzing

- Combines test generation with fuzzing
  - Test generation based on static analysis and/or symbolic execution – more later
  - Rather than generating new inputs and hoping that they enable a new path to be executed, compute inputs that will execute a desired path
    - And use them as fuzzing inputs
- Goal: Given a sequential program with a set of input parameters, generate a set of inputs that maximizes code coverage

One user's experience ….

Source: http://msdn.microsoft.com/en-us/library/cc162782.aspx

Technique	Effort	Code coverage	Defects Found
black box + mutation	10 min	50%	25%
black box + generation	30 min	80%	50%
white box + mutation	2 hours	80%	50%
white box + generation	2.5 hours	99%	100%

Fuzzing: open challenges

Selecting seeds to efficiently achieve high coverage

- Balance between coverage and efficiency
- Duplicate seeds cause redundancy and must be removed
- Small seeds preferred as they're faster for program to process

Branches that are difficult to get past

```
void test (int n) {
        if (n == 0x12345678) crash(); // Need 2^32 attempts to get past
}
```

Solution: Transform into code that produces granular feedback on each byte

Source: Suman Jana's lecture notes at Columbia University

Outline

What is fuzzing ?

Black box Fuzzing

Gray Box and White-box Fuzzing