Software Security (III): Other types of software vulnerabilities

Dawn Song

Format Functions

- Used to convert simple C data types to a string representation
- Variable number of arguments
  - Including format string
- Example
  - printf(“%s number %d”, “block”, 2)
  - Outputs: “block number 2”

Format String Parameters

Parameter	Output	Passed as
%d	Decimal (int)	Value
%u	Unsigned decimal (unsigned int)	Value
%x	Hexadecimal (unsigned int)	Value
%s	String ((const) (unsigned) char *)	Reference
%n	# bytes written so far, (* int)	Reference

Stack and Format Strings

- Function behavior is controlled by the format string
- Retrieves parameters from stack as requested: “%”
- Example:

printf(“Number %d has no address, number %d has: %08x\n”, I, a, &a)

stack top
…
<&a>
<a>
<i>
A
…
stack bottom

A	Address of the format string
i	Value of variable I
a	Value of variable a
&a	Address of variable a

Example V: Format string problem

```
int func(char *user)   {
   fprintf( stderr, user);
}
```

- Problem: what if   *user = “%s%s%s%s%s%s%s” ??
  - %s displays memory
  - Likely to read from an illegal address
  - If not, program will print memory contents. Privacy?

Correct form: `fprintf( stdout, “%s”, user);`

View Stack

- printf(“%08x. %08x. %08x. %08x\n”)
  - 40012983.0806ba43.bfffff4a.0802738b

- display 4 values from stack

Read Arbitrary Memory

- printf("\x10\x01\x48\x08_%08x. %08x. %08x. %08x|%s|")
  - Will display memory from 0x08480110
- Uses reads to move stack pointer into format string
- %s will read at 0x08480110 till it reaches null byte

Writing to arbitrary memory

- printf(“hello %n”, &temp)
  - writes ‘6’ into temp.

- printf(“%08x.%08x.%08x.%08x.%n”)

Vulnerable functions

Any function using a format string.

Printing:

- printf, fprintf, sprintf, …
- vprintf, vfprintf, vsprintf, …

Logging:

- syslog, err, warn

An Exploit Example

```
syslog("Reading username:");
read_socket(username);
syslog(username);
```

⇓

`Welcome to InsecureCorp. Please login.`

`Login:` ***`EvilUser%s%s…%400n…%n`***

`root@server> _`

Why The Bug Exists

- C language has poor support for variable-argument functions
  - Callee doesn't know the number of actual args
- No run-time checking for consistency between format string and other args
- Programmer error

Real-world Vulnerability Samples

- First exploit discovered in June 2000.
- Examples:
  - wu-ftpd 2.* : remote root
  - Linux rpc.statd: remote root
  - IRIX telnetd: remote root
  - BSD chpass: local root

⋮

What are software vulnerabilities?

- Flaws in software
- Break certain assumptions important for security
  - What assumptions broken in buffer overflow?

Dawn Song

Software is Buggy

- Real-world software is complex
  - Windows: tens of millions LoC
  - Google Chrome, FireFox: millions LoC

- Estimate: 15-50 bugs/thousand LoC
  - McConnell, Steve. *Code Complete*.

Software is Buggy

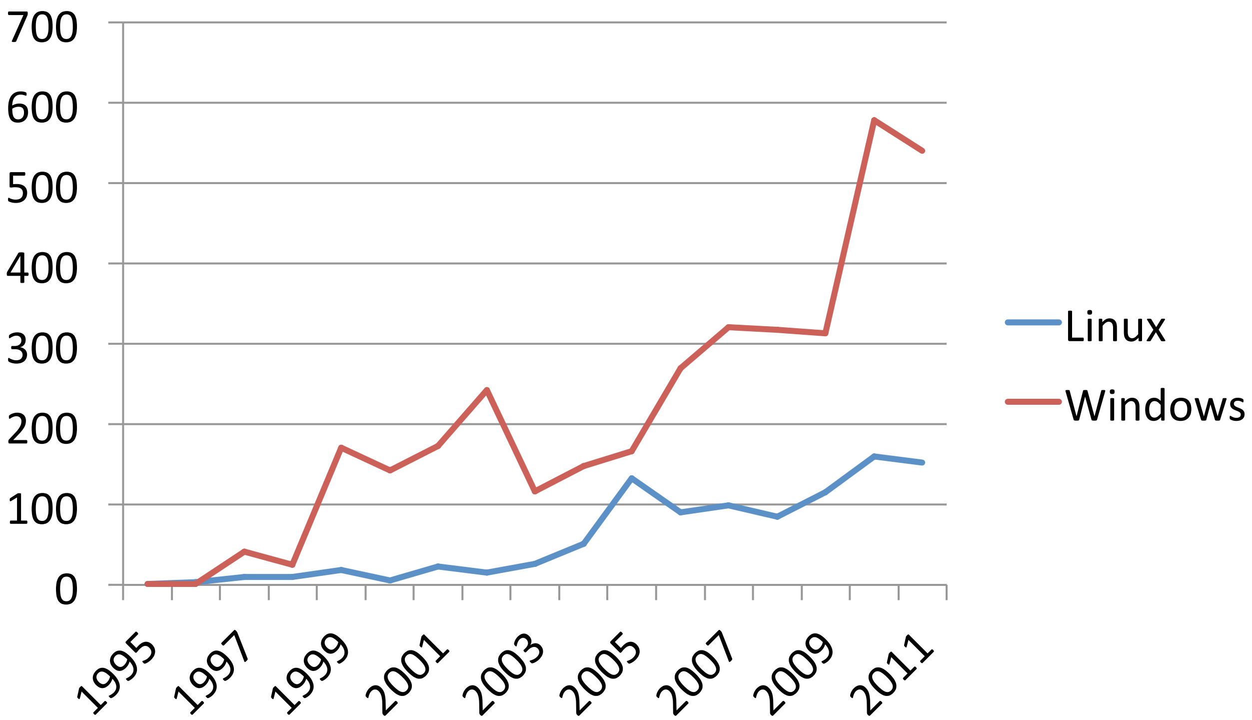


Figure: Number of Vulnerabilities (CVE IDs) [8]

Why does software have vulnerabilities?

- Programmers are humans!
  – Humans make mistakes!
- Programmers were not security aware

- Programming languages are not designed well for security

Dawn Song

What can you do?

- Programmers are humans!
  - Humans make mistakes!
  - Use tools!
- Programmers were not security aware
  - Learn about different common classes of coding errors
- Programming languages are not designed well for security
  - Pick better languages

Software Security: Vulnerability Analysis

Finding Bugs/Vulnerabilities

- Attackers:
  - Find vulnerabilities
  - Weaponize them (Exploit the vulnerabilities)
  - Use exploits to compromise machines & systems
  - Exploits are worth money

Find Vulnerability → Create Exploit → Compromise → $$$

Market for 0days

- Sell for $10K-100K

GLEG

ZERO DAY INITIATIVE

0day Market
@0daybid
This is the place where you can buy unpatched and undisclosured vulnerabilities.

eEye Digital Security®

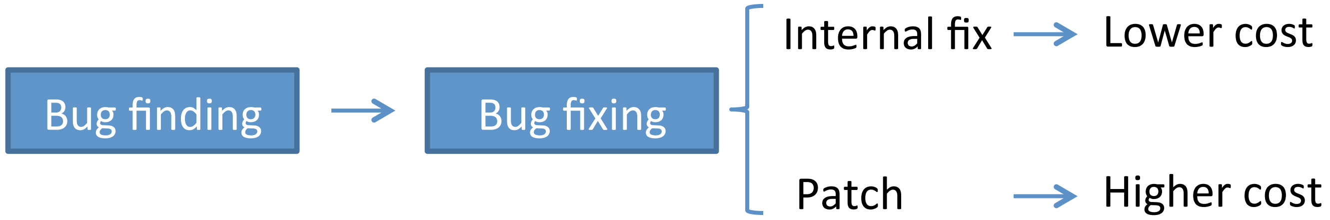
BreakingPoin
Find it before they do.™

iDEFENSE
A VeriSign Company

Square Security

Finding Bugs/Vulnerabilities

- Defenders:
  - Find vulnerabilities & eliminate them
    - Improve security of software
    - Easier and cheaper to fix a vulnerability before software deployed
    - After deployed: patching is expensive
  - Ideally prove a program is free of vulnerabilities



Static Device Verifier

- Verifies that drivers are not making illegal function calls or causing system corruption
  - SLAM project at Microsoft
  - http://research.microsoft.com/en-us/projects/slam

- "The requirements for the Windows logo program (*now Windows Hardware Certification Program*) state that a driver must not fail while running under Driver Verifier."

Techniques & Approaches

Automatic test
Case generation

Static analysis

Program
verification

Fuzzing

Dynamic
Symbolic
Execution

Lower coverage
Lower false positive
Higher false negative

Higher coverage
Lower false negative
Higher false positive

Fuzzing

Finding bugs in pdf viewer

Pdf viewer

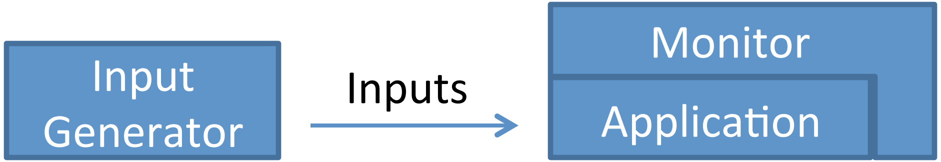
?

Black-box Fuzz Testing

- Given a program, simply feed it random inputs, see whether it crashes
- Advantage: really easy
- Disadvantage: inefficient
  - Input often requires structures, random inputs are likely to be malformed
  - Inputs that would trigger a crash is a very small fraction, probability of getting lucky may be very low

Fuzzing

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target interface
- Application is monitored for errors
- Inputs are generally either file based (.pdf, .png, .wav, .mpg)
- Or network based…
  - http, SNMP, SOAP

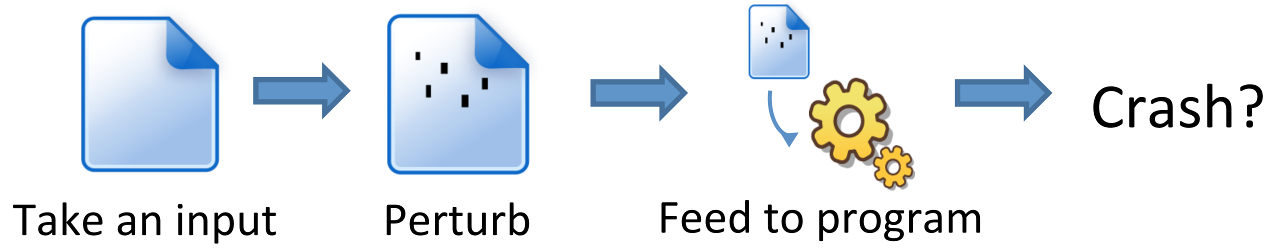


Regression vs. Fuzzing

	Regression	Fuzzing
Definition	Run program on many **normal** inputs, look for badness.	Run program on many **abnormal** inputs, look for badness.
Goals	Prevent **normal users** from encountering errors (e.g. assertion failures bad).	Prevent **attackers** from encountering **exploitable** errors (e.g. assertion failures often ok).

Enhancement I: Mutation-Based Fuzzing

- Take a well-formed input, randomly perturb (flipping bit, etc.
- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
- Anomalies may be completely random or follow some heuristics (e.g. remove NUL, shift character forward)
- Examples:
  - E.g., ZZUF, very successful at finding bugs in many real-world programs, http://sam.zoy.org/zzuf/
  - Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.



Example: fuzzing a pdf viewer

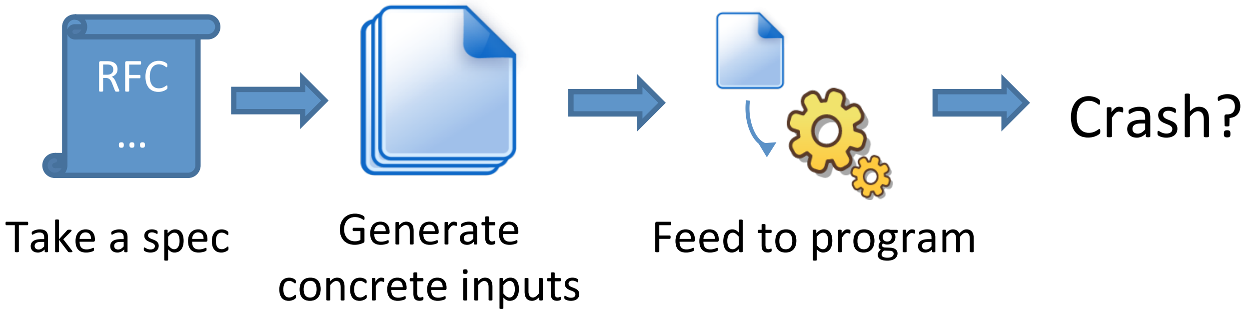
- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus
- Use fuzzing tool (or script)
  1. Grab a file
  2. Mutate that file
  3. Feed it to the program
  4. Record if it crashed (and input that crashed it)

Mutation-based Fuzzing In Short

Mutation-based	Super easy to setup and automate ✚	Little to no protocol knowledge required ✚	Limited by initial corpus −	May fail for protocols with checksums, those which depend on challenge −

Enhancement II: Generation-Based Fuzzing

- Test cases are generated from some description of the format: RFC, documentation, etc.
  - Using specified protocols/file format info
  - E.g., SPIKE by Immunity http://www.immunitysec.com/resources-freesoftware.shtml
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing



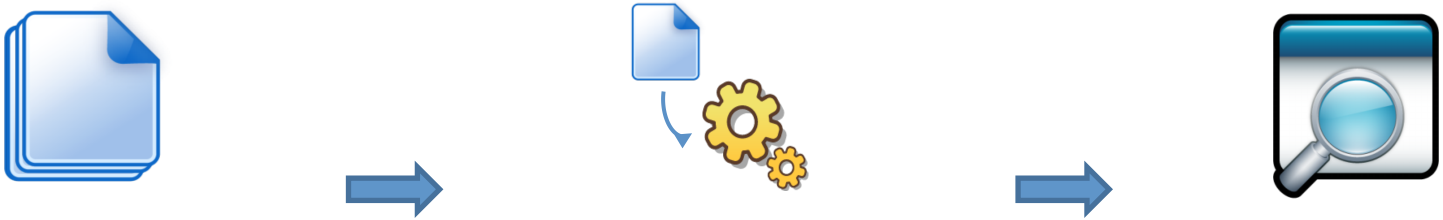
Example: Protocol Description

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);    // Width
                s_push_int(0x14, 1);    // Height
                s_push_int(0x8, 3);     // Bit Depth - should be 1,2,4,8,16, based on colortype
                s_push_int(0x3, 3);     // ColorType - should be 0,2,3,4,6
                s_binary("00 00");      // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);     // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Generation-Based Fuzzing In Short

Mutation-based	Super easy to setup and automate +	Little to no protocol knowledge required +	Limited by initial corpus −	May fail for protocols with checksums, those which depend on challenge −
Generation-based	Writing generator can be labor intensive for complex protocols −	Have to have spec of protocol (Often can find good tools for existing protocols e.g. http, SNMP) −	Completeness +	Can deal with complex dependencies e.g. checksums +

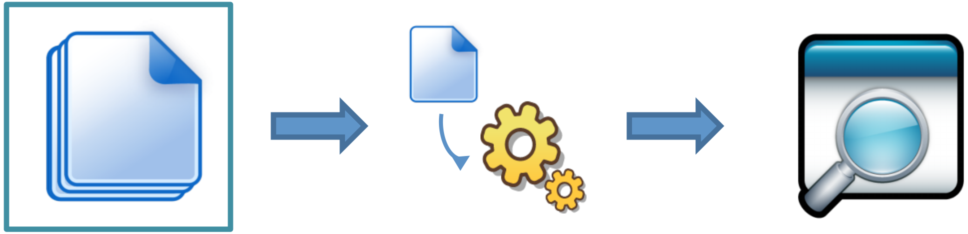
Fuzzing Tools & Frameworks



Input generation → Input injection → Bug detection

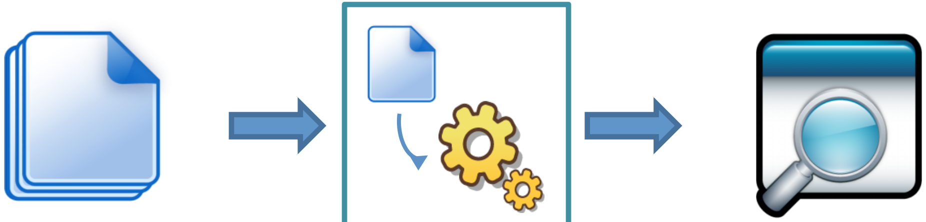
Input Generation

- Existing generational fuzzers for common protocols (ftp, http, SNMP, etc.)
  - Mu Dynamics, Codenomicon, PROTOS, FTPFuzz, WebScarab
- Fuzzing Frameworks: providing a fuzz set with a given spec
  - SPIKE, Peach, Sulley
- Mutation-based fuzzers
  - Taof, GPF, ProxyFuzz, PeachShark
- Special purpose fuzzers
  - ActiveX (AxMan), regular expressions, etc.



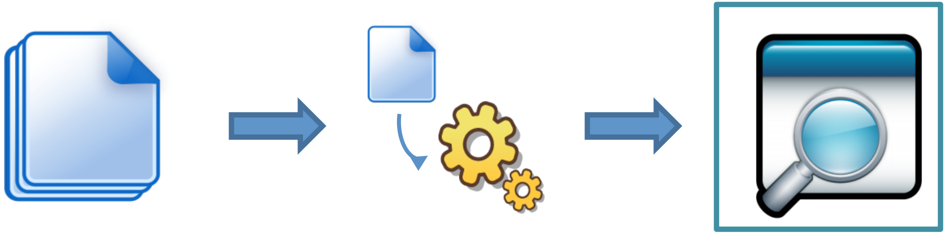
Input Injection

- Simplest
  - Run program on fuzzed file
  - Replay fuzzed packet trace
- Modify existing program/client
  - Invoke fuzzer at appropriate point
- Use fuzzing framework
  - e.g. Peach automates generating COM interface fuzzers



Bug Detection

- See if program crashed
  - Type of crash can tell a lot (SEGV vs. assert fail)
- Run program under dynamic memory error detector (valgrind/purify)
  - Catch more bugs, but more expensive per run.
- See if program locks up
- Write your own checker: e.g. valgrind skins



Workflow Automation

- Sulley, Peach, Mu-4000
  – Provide tools to aid setup, running, recording, etc.
- Virtual machines: help create reproducible workload

How Much Fuzz Is Enough?

- Mutation based fuzzers may generate an infinite number of test cases... When has the fuzzer run long enough?
- Generation based fuzzers may generate a finite number of test cases. What happens when they're all run and no bugs are found?

Quiz

- I have a PDF file with 248,000 bytes. There is one byte that, if changed to a particular value, causes a crash. This byte is 94% of the way through the file
- Any single random mutation to the file has a probability of ______ of finding the crash
- On average, need ______ test cases to find it
- At 2 seconds a test case, that's just under 3 days...
- It could take a week or more...

Code Coverage

- Some of the answers to these questions lie in *code coverage*
- Code coverage is a metric which can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools. e.g. `gcov`

Quiz: Line Coverage

Line/block coverage: Measures how many lines of source code have been executed.

For the code on the right, how many test cases (values of pair (a,b)) needed for full(100%) line coverage?

- 1
- 2
- 3
- 4

```
if( a > 2 )
a = 2;
if( b > 2 )
b = 2;
```

Quiz: Branch Coverage

Branch coverage: Measures how many branches in code have been taken (conditional jmps)

For the code on the right, how many test cases needed for full branch coverage?

```
if( a > 2 )
a = 2;
if( b > 2 )
b = 2;
```

- 1
- 2
- 3
- 4

Quiz: Path Coverage

Path coverage: Measures how many paths have been taken.

For the code on the right, how many test cases needed for full path coverage?

```
if( a > 2 )
a = 2;
if( b > 2 )
b = 2;
```

- ○ 1
- ○ 2
- ○ 3
- ○ 4

Code Coverage

- Benefits:
  - How good is this initial file?
  - Am I getting stuck somewhere?

```
if(packet[0x10] < 7) { //hot path
} else { //cold path

}
```

  - How good is fuzzer X vs. fuzzer Y
  - Am I getting benefits from running a different fuzzer?

Quiz: Problems of code coverage

- For:

```
mySafeCpy(char *dst, char* src){
     if(dst && src)
          strcpy(dst, src);
}
```

- Does full line coverage guarantee finding the bug?

  ○ Yes ○ No

Quiz: Problems of code coverage

- For:

```
mySafeCpy(char *dst, char* src){
     if(dst && src)
          strcpy(dst, src);
}
```

- Does full line coverage guarantee finding the bug?
    - Yes
    - No
- Does full branch coverage guarantee finding the bug?
    - Yes
    - No

Fuzzing Rules of Thumb

- Protocol specific knowledge very helpful
  - Generational tends to beat random, better spec’s make better fuzzers
- More fuzzers is better
  - Each implementation will vary, different fuzzers find different bugs
- The longer you run, the more bugs you may find
- Best results come from guiding the process
  - Notice where your getting stuck, use profiling!
- Code coverage can be very useful for guiding the process
- Can we do better?