Fuzzing

CSC 790

WAKE FOREST
UNIVERSITY
Department of Computer Science

Spring 2014

Software and Security Issues

- Programming bugs often become security issues
  - Buffer overflows, setUID programs, etc...
- If bugs are the problem, then eliminate them...
  - Formal verification is hard, impossible for big projects

```
1  // CWE-126 Buffer Over-Read
2  int processMessageFromSocket(int socket)
3  {
4     int success;
5     char buffer[BUFFER_SIZE];
6     char message[MESSAGE_SIZE];
7
8     if (getMessage(socket, buffer, BUFFER_SIZE) > 0)
9     {
10       // place contents of the buffer into message struct
11       ExMessage *msg = recastBuffer(buffer);
12       // copy message body into string for processing
13       int index;
14       for (index = 0; index < msg->msgLength; index++)
15          message[index] = msg->msgBody[index];
16       message[index] = '\0';
17       success = processMessage(message);
18    }
19    return success;
20 }
```

Safe-Type Languages

- How about Java, Haskell, or *your favorite safe language*
  - Does not eliminate all problems... *for example*
  - Performance, existing code base, flexibility, programmer competence, etc.
  - Perhaps not cost effective (*just catch same bugs as bad guys*)

Typical Bugs

- Most bugs can fit in a few categories
  - See Mike Howards The 19 Deadly Sins of Software Security

> 1. Buffer Overflows, 2. Format String problems, 3. SQL injection, 4. Command injection, 5. Failure to handle errors, 6. Cross-site scripting, 7. Failing to protect network traffic, 8. Use of "magic" URLs and hidden forms, 9. Improper use of SSL, 10. Use of weak password-based systems, 11. Failing to store and protect data, 12. Information leakage, 13. Improper file access, 14. Integer range errors, 15. Trusting network address information, 16. Signal race conditions, 17. Unauthenticated key exchange, 18. Failing to use cryptographically strong random numbers, 19. Poor usability 19.5 Steven, click here `http://goo.gl/QMET`

- Categories vary primarily by language and application domain
  - C/C++ - memory safety, overflows, double `free()` ...
  - Web applications, cross-site scripting

Sin Specifics

Sin	Language	Example	CVE Count
Buffer Overflows	C, C++	A buffer overrun occurs when a program allows input to write beyond the end of the allocated buffer. Results in anything from a crash to the attacker gaining complete control of the operating system. Many famous exploits are based on buffer overflows, such as the Morris worm.	3,326
Format String Problems	C, C++	The standard format string libraries in C/C++ include some potentially dangerous commands (particularly %n). If you allow untrusted user input to pass through a format string, this can result in anything from arbitrary code execution to spoofing user output.	411
Integer Overflows	C, C++, others	Failure to range check on integer types. This can cause integer overflow crashes and logic errors. In C/C++, integer overflows can be turned into a buffer overrun and arbitrary code execution, but all languages are prone to denial of service and logic errors.	288
SQL Injection	All	Forming SQL statements with untrusted user input means users can "inject" their own commands into your SQL statements. This puts your data at risk, and can even lead to complete server and network compromise.	2,225
Command Injection	All	Occurs when untrusted user input is passed to a compiler or interpreter, or worse, a command line shell. Potential risk depends on the context.	193
Failing to Handle Errors	Most	A broad category of problems related to a program's error handling strategy; anything that leads to the program crashing, aborting, or restarting is potentially a denial of service issue and therefore can be a security problem, particularly on servers.	80
Cross-Site Scripting (XSS)	Any web-facing	A web application takes some input from the user, fails to validate it, and echoes that input directly back to the web page. Because this code is running in the context of your web site, it can do anything your website could do, including retrieving cookies, modifying the HTML DOM, and so forth.	2,996
Failing to Protect Network Traffic	All	Most programmers underestimate the risk of transmitting data over the network, even if that data is not private. Attackers can eavesdrop, replay, spoof, tamper with, or otherwise hijack any unprotected data sent over the wire.	26
Use of Magic URLs and Hidden Form Fields	Any web-facing	Passing sensitive or secure information via the URL querystring or hidden HTML form fields, sometimes with lousy or ineffectual "encryption" schemes. Attackers can use these fields to hijack or manipulate a browser session.	33
Improper use of SSL and TLS	All	Using most SSL and TLS APIs requires writing a lot of error-prone code. If programmers aren't careful, they will have an illusion of security in place of the actual security promised by SSL. Attackers can use certificates from lax authorities, subtly invalid certificates, or stolen/revoked certificates, and it's up to the developer to write the code to check for that.	123
Use of Weak Password-Based Systems	All	Anywhere you are using passwords, you need to seriously consider the risks inherent to all password-based systems. Risks like phishing, social engineering, eavesdropping, keyloggers, brute force attacks, and so on. And then you have to worry about how users choose passwords, and where to store them securely on the server. Passwords are a necessary evil, but tread carefully.	1,235
Failing to Store and Protect Data Securely	All	Information spends more time stored on disk than in transit. Consider filesystem permissions and encryption for any data you're storing. And try to avoid hardcoding "secrets" into your code or configuration files.	56
Information Leakage	All	The classic trade-off between giving the user helpful information, and preventing attackers from learning about the internal details of your system. Was the password invalid, or the username?	26
Trusting Network Name Resolution	All	It's simple to override and subvert DNS on a server or workstation with a local HOSTS file. How do you really know you're talking to the real "secureserver.com" when you make a HTTP request?	20
Race Conditions	All	A race condition is when two different execution contexts are able to change a resource and interfere with each other. If attackers can force a race condition, they can execute a denial of service attack. Unfortunately, writing properly concurrent code is incredibly difficult.	139
Unauthenticated Key Exchange	All	Exchanging a private key without properly authenticating the entity/machine/service that you're exchanging the key with. To have a secure session, both parties need to agree on the identity of the opposing party. You'd be shocked how often this doesn't happen.	1
Cryptographically Strong Random Numbers	All	Imagine you're playing poker online. The computer shuffles and deals the cards. You get your cards, and then another program tells you what's in everybody else's hands. Random numbers are similarly fundamental to cryptography; they're used to generate things like keys and session identifiers. An attacker who can predict numbers– even with only a slight probability of success– can often leverage this information to breach the security of a system.	5
Poor Usability	All	Security is always extra complexity and pain for the user. It's up to us software developers to go out of our way to make it as painless as it can reasonably be. Security only works if the secure way also happens to be the easy way.	All

More Sources of Bugs

- Programmers often repeat bugs
  - Copy/paste
  - Confusion with API
  - Individuals repeat bugs (*that's how I learn't it*)
- Bugs come from bad assumptions, pre and post comments...
  - Trusted inputs become untrusted
- Others bugs are often yours
  - Open source, third party code

Finding Bugs

- Threat modeling, look at design, write out/diagram and find what could go wrong
- Manual code auditing or code review
- Automated tools
  - **Static Analysis**, compile time/source code level and compare code with abstract model
  - **Dynamic Analysis**, run program, give inputs, see what happens

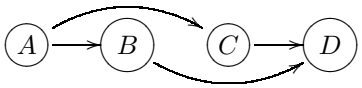
Static Analysis Basics

- Simple parsing programs, looking for potential bugs
  - Look for unsafe functions `strncpy()`, `sprintf()`, `gets()`, ...
  - Look for unsafe functions in your source base
  - Look for recurring problem code (problematic interfaces, copy/paste of bad code, etc.)
- Data flow analysis
  - Control Flow Graph (CFG) and determine how data changes

    "set up dataflow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes"

Control Flow Graphs

```
1 num = rand();
2 if(!num%2)
3    cout << num << "␣is␣even␣\n";
4 else
5    cout << num << "␣is␣odd␣\n";
6 return 0;
```



- Graph represents code blocks and flow
  - Nodes represent a *basic block*, code with any jumps
  - Blocks above are $A = \{1, 2\}$, $B = 3$, $C = \{4, 5\}$ and $D = 6$
- Several graph properties to consider
  - **Reachability**, subgraph is not connected then code is unreachable
  - **Dominate block**, block m dominates block n if every path to n must pass through m

gcc and Static Checks

- gcc provides some static checks

Level	Compiler Options	Description
1	`-std=`	Verify the language dialect to which you are writing
2	`-ansi -pedantic`	Enforce extra language-level checks and compliance
3	`-Wall`	Report the most common programming errors
4	`-Wall`, `-Wmost` or `-Wextra`	Report the most common programming errors and less-serious but potential problems
5	`-Wno-div-by-zero`, `-Wsystem-headers`, `-Wfloat-equal`, `-Wtraditional` (C only), `-Wdeclaration-after-statement` (C only), `-Wundef`, `-Wno-endif-labels`, `-Wshadow`, `-Wlarger-than-len`, `-Wpointer-arith`, `-Wbad-function-cast` (C only), ...	Provide additional checks not covered by `-Wall`, `-Wextra`, or `-Wmost`

  - Sub-item
  - Blah, blah, this is some crap...

Static Analysis Advantages and Disadvantages

- Strengths
  - Complete code coverage (*in theory*)
  - Potentially verify absence/report instances of whole bug class
  - Catches different bugs than dynamic analysis
- Weaknesses
  - High false positive rates
  - Some properties cannot be easily modeled
  - Difficult to build
  - Almost never have all source code in real systems (operating system, shared libraries, dynamic loading, etc.)

Normal Dynamic Analysis

- Run program in an instrumented execution environment
  - Look for use of invalid memory, race conditions, null pointer dereferencing, etc...
- Regression versus fuzzing
  - Regression, run program on many normal inputs, look for badness (*prevent normal users from encountering errors*)
  - Fuzzing, run program on many abnormal inputs, look for badness (*prevent attackers from encountering exploitable errors*)
  - *"Danny, are you for goodness or badness?"*

WTFuzzing

- Fuzzing can be traced back to the University of Wisconsin in 1988
  - Professor Barton Millers "Operating System Utility Program Reliability The Fuzz Generator" assignment
- 1999 Oulu University starts PROTOS
- 2002 Dave Aitels SPIKE
- 2004 Mangleme by Michael Zalewski
- 2005 FileFuzz, SPIKEfile, Codenomicon
- 2006 ActiveX fuzzers COMRaider and AxMan
- 2014 - Fletcher gets Rick-Roll'd by reading web source, not pretty...

Fuzzing Process and Targets

- Process steps
  1. Identify Targets
  2. Identify Inputs
  3. Generate Fuzzed Data
  4. Execute Fuzzed Data
  5. Monitor for Exceptions
  6. Determine Exploitability
- Possible targets
  - File formats, network protocols, command line arguments, environment variables, and web applications

Fuzzing Made Simple

- Pipe `/dev/urandom` to a target executable

```
cat /dev/urandom | ./a.out
```

How effective is this?

- Can also try fuzzing to `getenv()`

```
1 #include <string.h>
2 int main (int argc, char **argv)
3 { char buffer[10]; strcpy(buffer, getenv("HOME")); }
```

- Can also try leveraging `getopt()`
- Consider trying `argv[0]`
  - Professor Barton Millers "Operating System Utility Program Reliability – The Fuzz Generator" assignment

Simple Web Fuzzing

- Consider a standard HTTP/1.0 request
  - Acceptable...

```
GET /index.html HTTP/1.1
```

  - Following may cause a problem

```
AAAAAA...AAAA /index.html HTTP/1.1
GET ///////index.html HTTP/1.1
GET %n%n%n%n%n%n.html HTTP/1.1
GET /AAAAAAAAAAAAA.html HTTP/1.1
GET /index.html HTTTTTTTTTTTTTP/1.1
GET /index.html HTTP/1.1.1.1.1.1.1.1
GET goo.gl/QMET
```

- Two methods of generating inputs
  - **Mutation based**, dumb fuzzing (kinda random)
  - **Generation based**, smart fuzzing (use insight about program)

Mutation Fuzzing

- Given limited knowledge of the input structure
  - Anomalies are added to existing valid inputs
  - Anomalies may be completely random or follow some heuristics
  - Results in a **very** large number of test cases...
- For example, fuzzing a pdf viewer
  - Google for `.pdf` (*and get 1 kquatrillion results*)
  - Crawl pages to build a corpus
  - Use fuzzing tool (or script) to grab a file, mutate that file, input to the program, record if it crashed, repeat...

PDF Viewer Fuzzing

- Assume a pdf file has 248,000 bytes
  - Changing a certain byte to a certain value causes crash
- A single random mutation has a low probability of causing crash

$$\frac{1}{248000} = 4.0323 \times 10^{-6}$$

- On average, need 124,000 test cases to find it
  - Assuming 2 seconds a test case, thats just under 3 days...

Generation Fuzzing

- Test cases are based on protocol/application knowledge
  - Source code, RFC, documentation, etc...
  - *"key to making effective test cases is to make each case differ from valid data so as to (hopefully) cause a problem in the application, but not to make the data too invalid, or else the target application may quickly discard the input as invalid"*
- Anomalies are added to each possible input
- Number of test cases should be smaller
  - Will take time to create the test cases
- *Should give better results than random fuzzing...*

Fuzzing Input

- Existing generational fuzzers for common protocols
  - Protocols like `FTP`, `HTTP`, `SNMP`, etc....
  - Mu-4000, Codenomicon, PROTOS, FTPFuzz
- Fuzzing frameworks
  - You provide a specification, they provide a fuzz set
  - SPIKE, Peach, Sulley
- Dumb fuzzing automated
  - You provide files or packet traces, they provide the fuzz sets
  - Filep, Taof, GPF, ProxyFuzz, and PeachShark
- Many special purpose fuzzers already exist as well
  - ActiveX (AxMan), regular expressions, etc...

Fuzzing Ain't Easy

- Mutation can create an infinite number of test cases...
  - *When should you stop?*
- Generation can create a finite number of test cases
  - *What if no bugs are found?*
- *How do you monitor the target application such that you know when something "bad" has happened?*
- *What happens when you find too many bugs?*
  - Or every anomalous test causes same (boring) bug
- *How do you figure out which test case caused the fault?*
- *Given a crash, how do you find the actual vulnerability?*

Coverage

- Many of the questions can be answered using coverage
  - **Line coverage** measures how many lines of code (source code lines or assembly instructions) have been executed
  - **Branch coverage** measures how many branches in code have been taken (conditional jumps)
- Consider the following

```
if( x > 2 )
   Fletcher.friendlyHandShake(Chukar);
Class.highFive(Chukar);
```

  - Full line coverage in one test case (`x = 3`)
  - Two test cases for total branch coverage (`x = 1` and `x = 2`)

More Coverage

- Path coverage is the number of paths executed, for example

```
if(a > 2)
   Steven.awesomePatOnBack(Ben);
if(b > 2)
   Tomithy.awkwardComment(Hobo);
Hobo.reply("I got nothing");
```

- Consider the coverage...
  - One case for line coverage
  - Two cases for branch coverage
  - Path coverage is (a,b)={(0,0),(3,0),(0,3),(3,3)}

Coverage Issues

- Generally, a program with n reachable branches will
  - Require $2n$ test cases for branch coverage
  - Require 2^n test cases for path coverage

    What about loops?

- Some paths are not possible...

```
if(x > 2)
   Fulp.tellJoke();
if(x < 0)
   Class.laughs();
```

  - Cannot satisfy both of these conditionals
  - There are only three paths through this code, not four

Problems with Coverage

- Code can be covered without revealing bugs

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

- Error checking code mostly missed (*perhaps we don't particularly care about it*)

```
ptr = malloc(DEFINED_COSTANT);
if(!ptr) noMoreMemory();
```

- Only the *attack surface* reachable
  - Code processing user controlled data
  - No easy way to measure the attack surface
  - This sub-item is here for your enjoyment

gcc Coverage Example

- Consider the following program with x paths

```
int main(int argc, char *argv[])
{
   if(argc == 2)
   {
      if(strstr(argv[1], "hi"))
        printf("Hello world \n");
   }
   else
      printf("Come on Flecther, wrong number of arguments \n");
   return 0;
}
```

- Compile with gcc coverage flags

```
gcc -g -fprofile-arcs -ftest-coverage -o hello hello.c
```

  - Generates a .gcno file for each object file
  - Contains static info about branches, functions, etc...

Execute the Program

Terminal

```
> ./hello there
> ./hello hi_there
Hello world
```

- When program runs, code coverage info stored in .gcda files
- To process these files, run gcov

Terminal

```
> gcov hello.c
File 'hello.c'
Lines executed:83.33% of 6
hello.c:creating 'hello.c.gcov'
```


.gcov file

```
    -:    0:Source:hello.c
    -:    0:Graph:hello.gcno
    -:    0:Data:hello.gcda
    -:    0:Runs:2
    -:    0:Programs:1
    -:    1:#include<stdio.h>
    -:    2:#include<string.h>
    -:    3:
    -:    4:int main(int argc, char *argv[])
    2:    5:{
    2:    6:   if(argc == 2)
    -:    7:   {
    2:    8:      if(strstr(argv[1], "hi"))
    1:    9:        printf("Hello world \n");
    -:   10:   }
    -:   11:   else
#####:   12:      printf("Come on Flecther, wrong number of arguments \n");
    2:   13:   return 0;
    -:   14:}
    -:   15:
```

Cover This

- Spent some time talking about code coverage... so what

  Is there a correlation between code coverage and finding a security vulnerability?

How to Detect a Problem

- See if program crashed
  - Type of crash can tell a lot (SEGV vs. assert fail)
- Run program under dynamic memory error detector (valgrind/purify)
  - Can catch more bugs, but more expensive
- See if program locks up