

Software Security

CMSC 433
Bill Pugh

Software Security

- Making sure that if your software is misused, it doesn't do any of the vast number of things you didn't intend for the software to do

On trusting trust

- You can hide a trojan horse in a compiler
 - or in the operating system

Compiler

- Code generateCode(AST method) {
 if (method.getName()
 .equals("authenticateLogin")) {
 return ... code with trap door ...;
 }
 .. generate code normally

Slightly cool, but not very interesting

- Get spotted in a code audit

Compiler

- Code generateCode(AST method) {
 if (method.getName()
 .equals("authenticateLogin")) {
 return .. code with trap door.. }
 if (method.getName()
 .equals("generateCode")) {
 return ... code with special code gen ...;
 .. generate code normally
}

Trusted code base

- Trusted code base is the code that, if compromised, causes all of your security to fail
- Typically, includes all your software, your compiler, your operating system, ...
- Feeling comfy?

Software defects

- Traditional approach to correctness
 - define precondition
 - show that if precondition satisfied, output satisfied postcondition
- Didn't examine what happened if input didn't satisfy precondition

#1 source of security defects

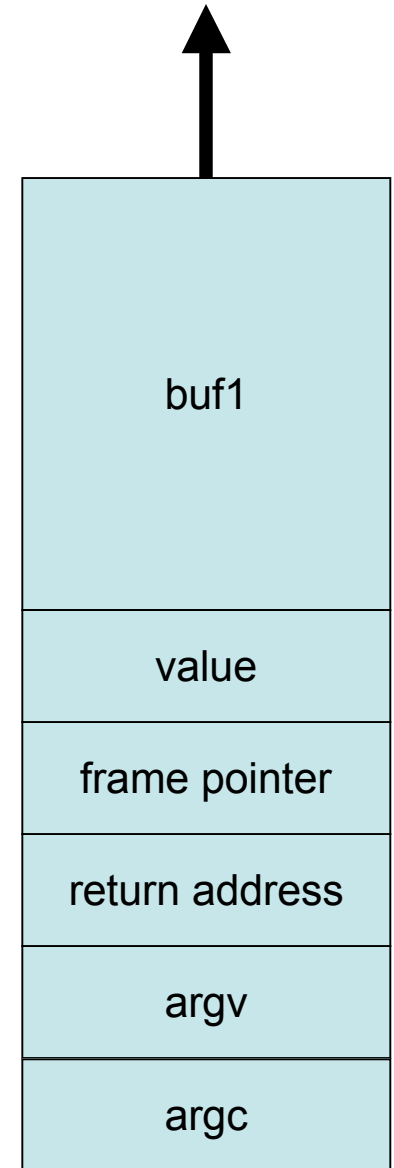
- Untrusted, unverified and unexpected input leading to a program doing something completed unexpected
 - unexpected by developer
 - intended by attacker
- of all the untrusted input problems, # 1 is buffer overruns in C/C++.

Buffer overflows

- In C, arrays are just locations in memory
- if you write past the allocated end of the array, you write into something else
- possibly other variables, return address
- can both rewrite return address and deliver payload
- <http://insecure.org/stf/smashstack.html>

Stack layout

```
int main(int argc,  
        char *argv[]) {  
    int value;  
    char buf1[80];  
    ...  
}
```



gets() is evil

- Impossible to use gets() correctly

```
char buf[20];  
gets(buf);
```

C String functions

```
char buf[20];
```

```
char * prefix = "http://";
```

```
strcpy(buf, prefix);
```

```
strncat(buf, path, sizeof(buf));
```

C String functions

```
char buf[20];
```

```
char * prefix = "http://";
```

```
strcpy(buf, prefix);
```

```
strncat(buf, path, sizeof(buf) - strlen(buf));
```

sprintf

- `char buf[80];`
`sprintf(buf, "%s - %d\n", path, errno);`

safe copy

```
#define MAX_BUF 256

void doStuff(char * in) {
    short len;
    char buf[MAX_BUF];
    len = strlen(in);
    if (len > MAX_BUF) return;
    strcpy(buf, in);
    .. do stuff with buf ...
}
```

Huh...?

- C doesn't seem to give any warnings when invoking a function that returns an unsigned long long
- and assign the result to something smaller
 - like a signed short or a char
- Even with -Wall and -pendantic-errors

Format String

- Using untrusted/unchecked string as a format string
 - `printf(s);` // just print s, no formatting needed
- what if s is “%d”
 - it prints the value of a value on the stack

The little known %n

- One of the least known and most dangerous format specified
 - %n expects the corresponding parameter to be the address of an int value
 - writes the number of characters written so far into that address
- `sprintf(buf, “%d%n”, x, &y)`
 - stores into y the number of characters needed to represent x

Now we have a way to update memory

- Some hackers are very clever
- Figured out how to turn several instances of this into an exploit
 - force a program to execute an arbitrary payload

References

- Newsham, Tim. *Format String Attacks*.
 - <http://muse.linuxmafia.org/lost+found/format-string-attacks.pdf>
- scut. *Exploiting Format String Vulnerabilities*.
 - <http://julianor.tripod.com/teso-fs1-1.pdf>

Integer overflows

- In C/C++/Java, no warnings or exceptions if an integer value overflows the range of values it can hold
- In C (and C++), no warnings when an assignment of a integer value involves two incompatible ranges
 - e.g., stores an unsigned long in an int

Integer overflows, continued

- Even if you are careful, and check to see if $x+y > \text{max}$
 - if $x+y$ overflows, you won't catch it.

Insecure Randomness

- In Java 1.4 and earlier, new Random uses currentTimeMillis() as a seed
- Imagine an on-line Texas hold'em poker game
 - assume you have access to a copy of the implementation
- You see your hole cards and the communal cards
 - Can check: would Random(1165336371231) have generated those cards?

How fast can you check?

- Takes less than 2 seconds to check 3,600,000 possibilities
- handles any Random() created in the last hour

Actual exploit

- There was an actual exploit for this
 - developed by a white hat team
 - <http://seclists.org/bugtraq/1999/Sep/0102.html>
- Poker implemented in Delphi Pascal
 - 32 bit random number generator
 - only 2^{32} possible decks, much less than the $52!$ decks that should be possible
 - Checked 200,000 possible seeds
 - Required 3 community cards (the first flop)

Solution?

- In Java 5 and above, new Random() uses System.nanoTime
 - nanoTime often has only microsecond resolution
 - In 2 seconds, can check 4 seconds worth of possible seeds, assuming microsecond resolution.
 - using one processor

SecureRandom

- **java.security.SecureRandom**
 - been around since at least Java 1.2
- Uses secure seed and secure random number generator
 - as secure as we know how to generate

SQL Injection

```
ResultSet getEmployees(String data) {  
    Statement stmt  
        = connection.createStatement();  
    stmt.execute(  
        "select * from employees where id = "  
        + data);  
    return statement.getResultSet();  
}
```

where does data come from?

- Does it come from a web form?
- Are you expecting something like:
 - “457”
- What happens if you get:
 - “457; delete from employees”

Cross site scripting

- Anything that allows untrusted and unchecked content to be injected into a web page (e.g., the response from a web server)
 - http://en.wikipedia.org/wiki/Cross_site_scripting
- Primarily a problem when browsers act on responses from a web server
 - e.g., execute embedded Javascript

Hypothetical example

- Add a review to a book on Amazon
- The “review” contains executable Javascript
- The “review” is now included verbatim on the Amazon page for that book.
- When a web page containing that “review” is viewed
 - the embedded JavaScript is executed and clicks on “Buy now with 1-click”

3 types of Cross-site scripting

- Local
 - using DOM to inject content into another web page that displays a local HTML file
 - hard to find, but dangerous because JavaScript executing on a local page may be treated as trusted Javascript
- Reflected
 - A URL that when opened in a web browser generates a response with injected code
- Stored
 - Storing untrusted content into a server database such that that untrusted content is generated as part of many page views
 - Easy to turn into lots of page views

Magic URL's

- People sometimes encode secrets in URL's
- Example:
 - I was a reviewer for a conference
 - had web site for reviewing
 - each reviewer can their own unique URL (with lots of random characters in it)
 - allowed reviewer to enter their review and view others

What went wrong

- Someone must have saved their “secret” URL to a web page
 - perhaps a private web page of links they wanted to be able to access from anywhere
- Somehow, Google found the link
- Google crawled the entire conference review site, including the contents of all the reviews (and names of the reviewers)

Untrusted file names

- `InputStream getData(String untrusted) {
return new FileInputStream(
“/fs/project/dataFiles/”+untrusted);`
- What if `untrusted = “../../../../etc/passwd”`?

Some Sins

- Buffer Overflows
- Format String problems
- Integer overflows
- SQL injection
- Command injection
- Failure to handle errors
- Cross-site scripting
- Failing to protect network traffic
- Use of "magic" URLs and hidden forms

More sins

- Improper use of SSL
- Use of weak password-based systems
- Failing to store and protect data
- Information leakage
- Improper file access
- Trusting network address information
- Race conditions
- Unauthenticated key exchange
- Failing to use cryptographically strong random numbers
- Poor usability