Software Security
Building Security in CMSC330 Summer 2021
Security breaches

- TJX (2007) - 94 million records*
- Adobe (2013) - 150 million records, 38 million users
- eBay (2014) - 145 million records
- Equifax (2017) – 148 millions consumers
- Yahoo (2013) – 3 billion user accounts
- Twitter (2018) – 330 million users
- First American Financial Corp (2019) – 885 million users
- Anthem (2014) - Records of 80 million customers
- Target (2013) - 110 million records
- Heartland (2008) - 160 million records

*containing SSNs, credit card nums, other private info

2017 Equifax Data Breach

• 148 million consumers’ personal information stolen

• They collect every details of your personal life
  • Your SSN, Credit Card Numbers, Late Payments…

• You did not sign up for it

• You cannot ask them to stop collecting your data

• You have to pay to credit freeze/unfreeze
Vulnerabilities: Security-relevant Defects

• The causes of security breaches are varied, but many of them owe to a defect (or bug) or design flaw in a targeted computer system's software.

• Software defect (bug) or design flaw can be exploited to affect an undesired behavior.
Defects and Vulnerabilities

- The **use of software** is growing
  - So: more bugs and flaws

- Software is large (**lines of code**)
  - **Boeing** 787: 14 million
  - **Chevy volt**: 10 million
  - **Google**: 2 billion
  - **Windows**: 50 million
  - **Mac OS**: 80 million
  - **F35 fighter Jet**: 24 million
Quiz 1

Program testing can show that a program has no bugs.

A. True
B. False
Quiz 1

Program testing can show that a program has no bugs.

A. True
B. False

Program testing can be used to show the presence of bugs, but never to show their absence!

--Edsger Dijkstra
In this Lecture

• The basics of threat modeling.

• Two kinds of exploits: buffer overflows and command injection.

• Two kinds of defense: type-safe programming languages, and input validation.

You will learn more in CMSC414, CMSC417, CMSC456
Considering Correctness

• All software is buggy, isn’t it? Haven’t we been dealing with this for a long time?

• A normal user never sees most bugs, or figures out how to work around them

• Therefore, companies fix the most likely bugs, to save money
Exploit the Bug

• A typical interaction with a bug results in a **crash**

• An **attacker** is not a normal user!
  - The attacker **will actively attempt to find defects**, using unusual interactions and features

• An attacker will work to **exploit** the bug to do **much worse**, to achieve his goals
Exploitable Bugs

- **Many kinds of exploits** have been developed over time, with technical names like
  
  - Buffer overflow
  - Use after free
  - Command injection
  - SQL injection
  - Privilege escalation
  - Cross-site scripting
  - Path traversal
  - …
Buffer Overflow

- A buffer overflow describes a family of possible exploits of a vulnerability in which a program may incorrectly access a buffer outside its allotted bounds.
  - A buffer overwrite occurs when the out-of-bounds access is a write.
  - A buffer overread occurs when the access is a read.
Example: Out-of-Bounds Read/write in C

```
#include <stdio.h>

void incr_arr(int *x, int len, int i) {
    if (i >= 0 && i < len) {
        x[i] = x[i] + 1;
        incr_arr(x, len, i+1);
    }
}

int y[10] = {1,1,1,1,1,1,1,1,1,1};
int z = 20;

int main(int argc, char **argv) {
    incr_arr(y, 11, 0);
    printf("%d =? 20\n", z);
    return 0;
}
```

Output: 21 =? 20

The value of z changed from 20 to 21. Why?
Example: Out-of-Bounds Read/write in C

```c
#include <stdio.h>

void incr_arr(int *x, int len, int i) {
    if (i >= 0 && i < len) {
        x[i] = x[i] + 1;
        incr_arr(x, len, i+1);
    }
}

int y[10] = {1,1,1,1,1,1,1,1,1,1};
int z = 20;

int main(int argc, char **argv) {
    incr_arr(y, 11, 0);
    printf("%d =? 20\n", z);
    return 0;
}
```

Output: 21 =? 20

- array `y` has length **10**
- but the second argument of `incr_arr` is **11**, which is one more than it should be.
- As a result, line 5 will be allowed to read/write past the end of the array.
Example: Out-of-Bounds Read/write in OCaml

Consider the same program, written in OCaml:

```ocaml
let rec incr_arr x i len =  
  if i >= 0 && i < len then  
    (x.(i) <- x.(i) + 1;  
    incr_arr x (i+1) len)  
  ;;

let y = Array.make 10 1;;
incr_arr y 0 (1 + Array.length y);;
```

Exception: Invalid_argument "index out of bounds".

- OCaml detects the attempt to write one past the end of the array and signals by throwing an exception.
Exploiting a Vulnerability

```c
#include <stdlib.h>
int main(int argc, char **argv) {
    int len = 10;
    if (argc == 2) len = atoi(argv[1]);
    incr_arr(y,len,0);
    printf("%d == 20
",z);
    return 0;
}
```

If an attacker can force the argument to be 11 (or more), then he can trigger the bug.

a.out

a.out 11

If an attacker can force the argument to be 11 (or more), then he can trigger the bug.
Quiz 2

If you declare an array as `int a[100];` in C and you try to write 5 to `a[i]`, where `i` happens to be 200, what will happen?

A. Nothing
B. The C compiler will give you an error and won’t compile
C. There will always be a runtime error
D. Whatever is at `a[200]` will be overwritten
Quiz 2

If you declare an array as `int a[100];` in C and you try to write 5 to `a[i]`, where `i` happens to be 200, what will happen?

A. Nothing
B. The C compiler will give you an error and won’t compile
C. There will always be a runtime error
D. Whatever is at `a[200]` will be overwritten
What Can Exploitation Achieve?

• **Buffer Overread: Heartbleed**
  - Heartbleed is a bug in the popular, open-source OpenSSL codebase, part of the HTTPS protocol.
  - The attacker can read the memory beyond the buffer, which could contain secret keys or passwords, perhaps provided by previous clients.
What Can Exploitation Achieve?

- Buffer Overwrite: Morris Worm
What happened?

- For C/C++ programs
  - A buffer with the password could be a local variable
- Therefore
  - The attacker’s input (includes machine instructions) is too long, and overruns the buffer
    - The overrun rewrites the return address to point into the buffer, at the machine instructions
  - When the call "returns" it executes the attacker’s code
Which kinds of operation is most likely to not lead to a buffer overflow in C?

A. Floating point addition
B. Indexing of arrays
C. Dereferencing a pointer
D. Pointer arithmetic
Which kinds of operation is most likely to not lead to a buffer overflow in C?

A. Floating point addition
B. Indexing of arrays
C. Dereferencing a pointer
D. Pointer arithmetic
Code Injection

- Attacker tricks an application to treat attacker-provided data as code

- This feature appears in many other exploits too
  - SQL injection treats data as database queries
  - Cross-site scripting treats data as Javascript commands
  - Command injection treats data as operating system commands
  - Use-after-free can cause stale data to be treated as code
  - Etc.
Use After Free (bug, no exploit)

```c
#include <stdlib.h>

struct list {
    int v;
    struct list *next;
};

int main() {
    struct list *p = malloc(sizeof(struct list));
    p->v = 0;
    p->next = 0;
    free(p); // deallocates p
    int *x = malloc(sizeof(int)*2); // reuses p's old memory
    x[0] = 5; // overwrites p->v
    x[1] = 5; // overwrites p->next
    p = p->next; // p is now bogus
    p->v = 2; // CRASH!
    return 0;
}
```
Trusting the Programmer?

• Buffer overflows rely on the ability to read or write outside the bounds of a buffer

• Use-after-free relies on the ability to keep using freed memory once it’s been reallocated

• C and C++ programs expect the programmer to ensure this never happens
  • But humans (regularly) make mistakes!

Jim Hague’s IOCCC winner program
Defense: Type-safe Languages

- Type-safe Languages (like Python, OCaml, Java, etc.) ensure buffer sizes are respected

  - Compiler **inserts checks** at reads/writes. Such checks can halt the program. But will prevent a bug from being exploited

  - **Garbage collection** avoids the use-after-free bugs. No object will be **freed** if it could be used again in the future.
Why Is Type Safety Helpful?

- Type safety ensures two useful properties that preclude buffer overflows and other memory corruption-based exploits.
  - **Preservation**: memory in use by the program at a particular type T always has that type T.
  - **Progress**: values deemed to have type T will be usable by code expecting to receive a value of that type

- To ensure preservation and progress implies that only non-freed buffers can only be accessed within their allotted bounds, precluding buffer overflows.
  - Overwrites breaks preservation
  - Overreads could break progress
  - Uses-after-free could break both
Quiz 4

Applications developed in the programming languages __________ are susceptible to buffer overflows and uses-after-free.

A. Ruby, Python
B. Ocaml, Haskell
C. C, C++
D. Rust, C#
Quiz 4

Applications developed in the programming languages ________ are susceptible to buffer overflows and uses-after-free.

A. Ruby, Python
B. Ocaml, Haskell
C. C, C++
D. Rust, C#
Costs of Ensuring Type Safety

• Performance
  - Array Bounds Checks and Garbage Collection add overhead to a program's running time.

• Expressiveness
  - C casts between different sorts of objects, e.g., a struct and an array.
    - Need casting in System programming
  - This sort of operation -- cast from integer to pointer -- is not permitted in a type safe language.
Command Injection

• A type-safe language will rule out the possibility of buffer overflow exploits.

• Unfortunately, type safety will not rule out all forms of attack
  • Command Injection: (also known as shell injection) is a security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server that is running an application.
What’s wrong with this Ruby code?

catwrapper.rb:

```ruby
if ARGV.length < 1 then
  puts "required argument: textfile path"
  exit 1
end

# call cat command on given argument
system("cat " + ARGV[0])

exit 0
```
Possible Interaction

> ls
catwrapper.rb
hello.txt

> ruby catwrapper.rb hello.txt
Hello world!

> ruby catwrapper.rb catwrapper.rb
if ARGV.length < 1 then
  puts "required argument: textfile path"
...

> ruby catwrapper.rb “hello.txt; rm hello.txt”
Hello world!

> ls
catwrapper.rb
What Happened?

**catwrapper.rb:**

```ruby
if ARGV.length < 1 then
  puts "required argument: textfile path"
  exit 1
end

# call cat command on given argument
system("cat " + ARGV[0])
exit 0
```

**system()** interpreted the string as having two commands, and executed them both.
When could this be bad?

catwrapper.rb as a web service
Consequences

• If `catwrapper.rb` is part of a web service
  • Input is untrusted — could be anything
  • But we only want requestors to read (see) the contents of the files, not to do anything else
  • Current code is too powerful: vulnerable to
    command injection

• How to fix it?
  Need to validate inputs

[https://www.owasp.org/index.php/Command_Injection](https://www.owasp.org/index.php/Command_Injection)
Defense: Input Validation

- Inputs that could cause our program to do something illegal
- Such atypical inputs are more likely when an untrusted adversary is providing them

We must validate the client inputs before we trust it

- **Making input trustworthy**
  - *Sanitize it* by modifying it or using it in such a way that the result is correctly formed by construction
  - *Check it* has the expected form, and reject it if not
Checking: Blacklisting

• *Reject strings with possibly bad chars:* ’ ; --

```ruby
if ARGV[0] =~ /;/ then
  puts "illegal argument"
  exit 1
else
  system("cat "+ARGV[0])
end
```

> ruby catwrapper.rb "hello.txt; rm hello.txt"

*illegal argument*
Sanitization: Blocklisting

- Delete the characters you don’t want: ’ ; --

```ruby
system("cat "+ARGV[0].tr(";"","")
```
delete occurrences of ; from input string

> ruby catwrapper.rb "hello.txt; rm hello.txt"
```
Hello world!
cat: rm: No such file or directory
Hello world!
> ls hello.txt
```
hello.txt
Sanitization: Escaping

• **Replace problematic characters with safe ones**
  - change ‘ to \’
  - change ; to \;
  - change – to \–
  - change \ to \\

• Which characters are problematic depends on the interpreter the string will be handed to
  - Web browser/server for URIs
    - URI::escape(str, unsafe_chars)
  - Program delegated to by web server
    - CGI::escape(str)
Sanitization: Escaping

def escape_chars(string):
    pat = /(("|\"|\.|\*|\/|\-|\\;|\\\s)/
    string.gsub(pat) { |match| "\\" + match }
end

system("cat " + escape_chars(ARGV[0]))

> ruby catwrapper.rb "hello.txt; rm hello.txt"
cat: hello.txt; rm hello.txt: No such file or directory
> ls hello.txt
hello.txt
Checking: Safelisting

- Check that the user input is known to be safe
  - E.g., only those files that exactly match a filename in the current directory

- **Rationale**: Given an invalid input, **safer to reject than to fix**
  - “Fixes” may result in wrong output, or vulnerabilities
  - **Principle** of fail-safe defaults
files = Dir.entries(".").reject{|f| File.directory?(f)}

if not (files.member? ARGV[0]) then
  puts "illegal argument"
  exit 1
else
  system("cat "+ARGV[0])
end

reject inputs that do not mention a legal file name
Validation Challenges

• **Cannot always delete or sanitize problematic characters**
  • You may want dangerous chars, e.g., “Peter O’Connor”
  • How do you know if/when the characters are bad?
  • Hard to think of all of the possible characters to eliminate

• **Cannot always identify safelist cheaply or completely**
  • May be expensive to compute at runtime
  • May be hard to describe (e.g., “all possible proper names”)
Software Security
Part II: Web Security

CMSC330 Spring 2021
WWW Security

- **Security for the World-Wide Web (WWW)** presents new vulnerabilities to consider:
  - SQL injection
  - Cross-site Scripting (XSS)

- These share some common causes with memory safety vulnerabilities; like **confusion of code and data**
  - **Defense** also similar: validate untrusted input

- New wrinkle: **Web 2.0’s use of mobile code**
  - How to protect your applications and other web resources?
The Internet

(Much) user data is part of the browser

FS/DB is a separate entity, logically (and often physically)

Need to protect this state from illicit access and tampering

(Much) user data is part of the browser

FS/DB is a separate entity, logically (and often physically)
The World Wide Web (WWW)
Interacting with web servers

Resources which are identified by a URL
(Universal Resource Locator)


<table>
<thead>
<tr>
<th>Protocol</th>
<th>Hostname/server</th>
<th>Path to a resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftp</td>
<td>Translated to an IP address by DNS (e.g., 128.8.127.3)</td>
<td>index.html is static content i.e., a fixed file returned by the server</td>
</tr>
<tr>
<td>https</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

http://facebook.com/delete.php?f=joe123&w=16

Here, the file delete.php is dynamic content. i.e., the server generates the content on the fly
HyperText Transfer Protocol (HTTP)

- Requests contain:
  - The **URL** of the resource the client wishes to obtain
  - **Headers** describing what the browser can do

- **Request types** can be **GET** or **POST**
  - **GET**: retrieves data, most of it in URL itself (no server side effects)
  - **POST**: provides data as separate fields (can have side effects)
HTTP GET Requests

http://www.reddit.com/r/security

User-Agent is typically a browser, but it can be wget, JDK, etc.
Referrer URL: the site from which this request was issued.
HTTP POST Requests

Posting on Piazza

Implicitly includes data as a part of the URL

Explicitly includes data as a part of the request’s content
HyperText Transfer Protocol (HTTP)

User clicks

- **Responses** contain:
  - **Status** code
  - **Headers** describing what the server provides
  - **Data**
  - **Cookies** (much more on these later)
    - Represent *state* the server would like the browser to store on its behalf
HTTP Responses

HTTP version

Status code

Reason phrase

Headers

Data

<html> ...... </html>
Relational Databases & Stable Storage

Need to protect this state from illicit access and tampering.
SQL Injection

- SQL injection is a code injection attack that aims to steal or corrupt information kept in a server-side database.
Data as Tables

• A relational database organizes information as tables of records.

<table>
<thead>
<tr>
<th>Name</th>
<th>Gender</th>
<th>Age</th>
<th>Email</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dee</td>
<td>F</td>
<td>28</td>
<td><a href="mailto:dee@pp.com">dee@pp.com</a></td>
<td>j3i8g8ha</td>
</tr>
<tr>
<td>Mac</td>
<td>M</td>
<td>7</td>
<td><a href="mailto:bouncer@pp.com">bouncer@pp.com</a></td>
<td>a0u23bt</td>
</tr>
<tr>
<td>Charlie</td>
<td>M</td>
<td>32</td>
<td><a href="mailto:aneifjask@pp.com">aneifjask@pp.com</a></td>
<td>0aergja</td>
</tr>
<tr>
<td>Dennis</td>
<td>M</td>
<td>28</td>
<td><a href="mailto:imagod@pp.com">imagod@pp.com</a></td>
<td>1bjb9a93</td>
</tr>
<tr>
<td>Frank</td>
<td>M</td>
<td>57</td>
<td><a href="mailto:armed@pp.com">armed@pp.com</a></td>
<td>zio9gga</td>
</tr>
</tbody>
</table>
SELECT Age FROM Users WHERE Name='Dee';

UPDATE Users SET email='readgood@pp.com'
WHERE Age=32; -- this is a comment

INSERT INTO Users Values('Frank', 'M', 57, ...);

DROP TABLE Users;
Web Server SQL Queries

Website

“Login code” (Ruby)

result = db.execute "SELECT * FROM Users
    WHERE Name='#{user}' AND Password='#{pass}';"

Suppose you successfully log in as user if this returns any results

How could you exploit this?
SQL injection

result = db.execute "SELECT * FROM Users
WHERE Name='#{user}' AND Password='#{pass}';"

result = db.execute "SELECT * FROM Users
WHERE Name='frank' OR 1=1; --' AND Password='whocares';"

Always true
(so: dumps whole user DB)

Commented out
SQL injection

result = db.execute "SELECT * FROM Users
WHERE Name='#{user}' AND Password='#{pass}';"

result = db.execute "SELECT * FROM Users
WHERE Name='frank' OR 1=1;
DROP TABLE Users; --' AND Password='whocares';";

Can chain together statements with semicolon:
STATEMENT 1 ; STATEMENT 2
SQL injection

http://xkcd.com/327/
The Underlying Issue

result = db.execute "SELECT * FROM Users
WHERE Name='#{user}' AND Password='#{pass}';"

• This one string combines the **code and the data**
  • Similar to buffer overflows
  • and command injection

*When the boundary between code and data blurs, we open ourselves up to vulnerabilities*
The Underlying Issue

```
result = db.execute "SELECT * FROM Users
    WHERE Name='#{user}' AND Password='#{pass}';"
```

Intended AST for parsed SQL query

Should be **data**, not **code**
Defense: Input Validation

Just as with command injection, we can defend by validating input, e.g.,
- **Reject** inputs with bad characters (e.g., ; or --)
- **Remove** those characters from input
- **Escape** those characters (in an SQL-specific manner)

These can be effective, but the best option is to avoid constructing programs from strings in the first place
Sanitization: Prepared Statements

- **Treat user data according to its type**
  - Decouple the code and the data

```python
result = db.execute "SELECT * FROM Users
    WHERE Name='#{user}' AND Password='#{pass}';"

stmt = db.prepare("SELECT * FROM Users WHERE
    Name = ? AND Password = ?")

result = stmt.execute (user, pass)
```

**Variable binders parsed as strings**

*Arguments*
Using Prepared Statements

```python
stmt = db.prepare("SELECT * FROM Users WHERE Name = ? AND Password = ?")
result = stmt.execute(user, pass)
```

Binding is only applied to the leaves, so the structure of the AST is fixed.