Program Security and Vulnerabilities

Class 7
Secure Programs

- Programs
  - Operating System
  - Device Drivers
  - Network Software (TCP stack, web servers ...)
  - Database Management Systems ...

Integrity
Confidentiality
Availability

Secure Programs
Security Properties

- Confidentiality
  - Information about system or its users cannot be learned by an attacker

- Integrity
  - The system continues to operate properly, only reaching states that would occur if there were no attacker

- Availability
  - Actions by an attacker do not prevent users from having access to use of the system
Security Properties (cont’d)

- Security is about
  - Honest user (e.g., Alice, Bob, …)
  - Dishonest Attacker
  - How the Attacker
    - Disrupts honest user’s use of the system (Integrity, Availability)
    - Learns information intended for Alice only (Confidentiality)
What is Security?

- **System correctness**
  - If user supplies expected input, system generates desired output
  - Good input $\Rightarrow$ Good output
  - More features: better

- **Security**
  - If attacker supplies unexpected input, system does not fail in certain ways
  - Bad input $\Rightarrow$ Bad output
  - More features: can be worse
In This Section

- Buffer Overflow
- SQL Injection Attack
- Incomplete Mediation
- Time-of-Check to Time-of-Use Errors
- Malicious Code
Ethical Use of Security Information

- We discuss *vulnerabilities* and *attacks*
  - Most vulnerabilities have been fixed
  - Some attacks may still cause harm
  - Do not try these at home or anyplace else

- Purpose of this class
  - Learn to prevent malicious attacks
  - Use knowledge for good purposes
Famous Buffer Overflow Attacks

- **Morris worm**: overflow in fingerd
  - 6,000 machines infected (10% of existing Internet)

- **CodeRed**: overflow in MS-IIS web server
  - Internet Information Services (IIS)
  - Web server application
  - The most used web server after Apache HTTP Server
  - 300,000 machines infected in 14 hours

- **SQL Slammer**: overflow in MS-SQL server
  - 75,000 machines infected in 10 minutes (!!)
Famous Buffer Overflow Attacks

- **Sasser**: overflow in Windows LSASS
  - **Local Security Authority Subsystem Service**
    - Process in Windows OS
    - Responsible for enforcing the security policy on the system.
    - Verifies users logging on to a Windows computer or server, handles password changes, and creates access tokens
    - *Around 500,000 machines infected*

- **Conficker**: overflow in Windows Server
  - *Around 10 million machines infected (estimates vary)*
Memory Exploits

- **Buffer** is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of data
- If executable code is supplied as “data”, victim’s machine may be fooled into executing it
- Code will give attacker control over machine
Stack Buffers

- Suppose Web server contains this function

  ```c
  void func(char *str) {
    char buf[126];
    strcpy(buf, str);
  }
  ```

- When this function is invoked, a new frame with local variables is pushed onto the stack
Stack Buffers (cont’d)

- When `func` returns
  - The local variables are popped from the stack
  - The old value of the stack frame pointer (sfp) is recovered
  - The return address is retrieved
  - The stack frame is popped
  - Execution continues from return address (calling function)

![Stack Diagram]

- buf: Local variables
- sfp: Pointer to previous frame
- ret addr: Execute code at this address after `func()` finishes
- str: Arguments
- Frame of the calling function

Stack grows this way
What If Buffer Is Overstuffed

- Memory pointed to by str is copied onto stack...

```c
void func(char *str) {
    char buf[126];
    strcpy(buf, str);
}
```

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

```
Top of stack
Stack grows this way
```

This will be interpreted as return address!
Attack 1: Smashing the Stack

- Suppose buffer contains attacker-created string
  - For example, \*str contains a string received from the network as input to some network service daemon

- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root
Buffer Overflow Difficulties

- Executable attack code is stored on stack, inside the buffer containing attacker’s string
  - Stack memory is supposed to contain only data, but...

- For the basic attack, overflow portion of the buffer must contain *correct address of attack code* in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
  - Otherwise application will give segmentation violation
  - Attacker must correctly guess in which stack position his buffer will be when the function is called
Problem: No Range Checking

- strcpy does not check input size
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until “\0” is encountered, ignoring the size of area allocated to buf

- Many C library functions are unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, ...)
  - printf(const char *format, ...)
Does Range Checking Help?

- `strncpy`(char *dest, const char *src, size_t n)
  - If `strncpy` is used instead of `strcpy`, no more than n characters will be copied from *src to *dest
  - Programmer has to supply the right value of n
- Potential overflow in htpasswd.c (Apache 1.3):
  ```c
  ... strcpy(record, user);
  strcat(record, ":");
  strcat(record, cpw); ...
  ```
- Published “fix” (do you see the problem?):
  ```c
  ... strncpy(record, user, MAX_STRING_LEN-1);
  strcat(record, ":");
  strncpy(record, cpw, MAX_STRING_LEN-1); ...
  ```

Copies username (“user”) into buffer (“record”), then appends “:" and hashed password (“cpw”)
Strncpy Missuse in htpasswd “Fix”

- Published “fix” for Apache htpasswd overflow:

```c
... strncpy(record,user, MAX_STRING_LEN-1); 
    strcat(record,":"); 
    strcat(record,cpw, MAX_STRING_LEN-1); ...
```

MAX_STRING_LEN bytes allocated for record buffer

- Put up to MAX_STRING_LEN-1 characters into buffer
- Put “:”
- Again put up to MAX_STRING_LEN-1 characters into buffer

contents of *user : contents of *cpw
Attack 2: Variable Overflow

- Somewhere in the code `authenticated` is set only if login procedure is successful
  - Other parts of the code test `authenticated` to provide special access

```c
char buf[80];
int authenticated = 0;
void vulnerable() {
    gets(buf);
}
```

- Attacker passes 81 bytes as input to `buf`
Attack 3: Pointer Variables

- \texttt{fnptr} is invoked somewhere else in the program
  - This is only the definition

```c
void func(char *s){
    char buf[80];
    int (*fnptr)();
    gets(buf);
}
```

- \texttt{fnptr} is invoked somewhere else in the program
- This is only the definition
Attack 3: Pointer Variables (cont’d)

- Send malicious code in `s`
- Overflow fnptr
  - Pass more than 80 bytes in `gets`
  - `fnptr` now points to malicious code
- When `fnptr` is executed, malicious code is executed!

```c
void func(char *s){
    char buf[80];
    int (*fnptr)();
    gets(buf);
}
```

![Diagram of local variables and frame of the calling function with pointers to previous frame and arguments](image-url)
void func(char *s){
    char buf[80];
    gets(buf);
}

- Send malicious code in s
- Change the caller’s saved frame ptr.
  - Pass more than 80 bytes in gets
  - sfp now points to malicious code
- Caller’s return address read from sfp
- When func returns, mal. code runs!
static int getpeername1(p, uap, compat) {
    // In FreeBSD kernel, retrieves address of peer to which a socket is connected
    ... 
    struct sockaddr *sa;
    ...
    len = MIN(len, sa->sa_len);
    ...
    copyout(sa, (caddr_t)uap->asa, (u_int)len);
    ...
}  

Copies “len” bytes from kernel memory to user space

Checks that “len” is not too big

Negative “len” will always pass this check...

... interpreted as a huge unsigned integer here

... will copy up to 4G of kernel memory
Buffer Overflow Prevention

- Canary
- Bounds checking
- Tagging
Canary

- Canary words
  - Known values placed between a buffer and control data on the stack
  - When the buffer overflows, the first data to be corrupted will be the canary
  - Failed verification of the canary data: overflow alert!
Bounds Checking

- Compiler based technique
- For each allocated memory block
  - Add run-time bounds information
  - Checks all pointers against bounds at run-time
Tagging

- Tag the type of each piece of data in memory
  - Used for type checking
- Mark data buffers as non-executable
  - Prevent them from storing executable code
In this lecture

- Nonmalicious Program Errors
- Buffer Overflow
- SQL Injection Attack
- Incomplete Mediation
- Time-of-Check to Time-of-Use Errors
SQL in Web Pages

- SQL can be used to display data on a web page
- Web users can input their own search values
- Dynamically change SQL statements to provide the user with selected data:
  - Example (Server side code):
    - `txtUserId = getRequestString("UserId");`
    - `txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId + ";`  
    - `txtSQL` is a select statement
    - Fetch data from “Users” database for “txtUserId” to web page
SQL Injection Attack

- Technique where malicious users can inject SQL commands into an SQL statement, via web page input
- Injected SQL commands can alter SQL statement and compromise the security of a web application
SQL Injection Attack Type 1

- 1=1 is always true
- `txtUserId = getRequestString("UserId");
  txtSQL = "SELECT UserId, Name, Password FROM Users WHERE UserId = " + txtUserId + ";

- Malicious user can enter smart (but wrong) input as `txtUserId`
  
  ```
  UserId:
  105 or 1=1
  ```

- Server code:
  - `SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1`
  - Valid: will return all rows from the table “Users”
**SQL Injection Attack Type 2**

- SQL Injection Based on ""=""" is Always True
- Server code:

uName = getRequestString("UserName");
uPass = getRequestString("UserPass");

sql = "SELECT * FROM Users WHERE Name ="" + uName + "" AND Pass ="" + uPass + """

User Name:

Password:
SQL Injection Attack Type 2 (cont’d)

- SQL Injection Based on ""="" is Always True
- Server code:
  
  ```
  uName = getRequestString("UserName");
  uPass = getRequestString("UserPass");
  sql = "SELECT * FROM Users WHERE Name ="" + uName + 
  "" AND Pass ="" + uPass + """";
  ```

- Attacker can insert " or "=" into the name and password box
- Server code becomes
  
  ```
  SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""="";
  ```
SQL Injection Attack Type 3

- **SQL Injection Based on Batched SQL Statements**
- Batched SQL statements: separated by semicolon
  - SELECT * FROM Users; DROP TABLE Suppliers
  - Return all rows in the Users table, then delete the table called Suppliers
SQL Injection Attack Type 3 (cont’d)

- SQL Injection Based on Batched SQL Statements
- Server code

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

User id:

```
105; DROP TABLE Suppliers
```

- Server code becomes

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers
```
In this lecture

- Nonmalicious Program Errors
- Buffer Overflow
- SQL Injection Attack
- Incomplete Mediation
- Time-of-Check to Time-of-Use Errors
Incomplete Mediation

- What if par2 is
  - 1800Jan01 (outside of range)
  - 2000Feb30 (non-existent)
  - 2048Min32 (undefined)
  - 1Aardvark2Many ?!?
- How to fix such errors?
  - Have client side code to verify input correctness
  - Restrict choices to only possible ones, e.g., drop-down menus ...
Incomplete Mediation (cont’d)

- *Still vulnerable!*
  - The results of the verification are accessible in the URL
  - The (malicious) user can access and modify fields
  - Only then send to the server
  - The server cannot tell if URL came directly from the user browser or from malicious user
In this lecture

- Nonmalicious Program Errors
- Buffer Overflow
- Incomplete Mediation
- Time-of-Check to Time-of-Use Errors (TOCTTOU)
TOCTTOU Errors

- Concurrency issue
  - Successive instructions may not execute serially
  - Other processes may be given control
- Access control
  - Only users with *rights* can access objects
- **TOCTTOU**: control is given to other process *between* access control check and access operation
TOCTTOU Example

```c
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISREG(s.st_mode)) {
        error("only allowed to regular files");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

Path to file

Extract file meta-data

- Between check and open attacker can change `path`
  - Initial `path` is regular file
  - Later `path` is not
- Adversary by-passes security

Open file

No symlink, directory, special file
TOCTTOU: How an Attack Works

- openfile is being run within the kernel (at the OS)
- At the user space level, there is a program P
  - Controlled by adversary
  - Program P defines path variable
- Program P also launches two threads T1 and T2
  - T1 and T2 share the path variable
  - If T2 changes path, T1 also sees the change
- T1 runs openfile where path is set to a file
- T2 sets path to a directory
TOCTTOU Prevention

1. Ensure critical parameters are not exposed during pre-emption
   - openfile “owns” path

2. Ensure serial integrity
   - openfile is atomic
   - No pre-emption during its execution

3. Validate critical parameters
   - Compute checksum of path before pre-emption
   - Compare to checksum of path after ...