Program Errors
Buffer Overflow
TOCTTOU
Why Do We Have Security Vulnerabilities?

• Some contributing factors
  – Few courses in computer security 😊
  – Programming text books do not emphasize security
  – Few security audits
  – *C is an unsafe language*
  – Programmers have many other things to worry about
  – *Consumers do not care about security*
  – Security is expensive and takes time
Trends

Vulnerability Disclosures
2000-2009

Percentage of Vulnerability Disclosures
Attributed to Top 10 Vendors
2009

Source: IBM X-Force®
OS Vulnerabilities

Vulnerability Disclosures Affecting Operating Systems
2005-2009

Critical and High Vulnerability Disclosures Affecting Operating Systems
2005-2009

Source: IBM X-Force®
Non-malicious Errors

• How to determine *quality* of program?
  – Testing …
  – Number of faults in requirements, design and code inspections

• Example
  – Module A had 100 faults discovered and fixed
  – Module B had only 20
  – Which one is better?
  – *Software testing result:* software with more faults is likely to have even more !!!
Fixing Faults

• Penetrate and Patch
  – Special teams test programs and find faults
  – If no attack found, the program was OK
  – Otherwise, not – More frequently
  – Then fix faults

• Problem: *The system became less secure!*
  – Focus on fixing the fault and not its context
  – Fault had side effects in other places
  – Fixing fault generated faults somewhere else
  – Fixing fault would affect functionality or performance
Buffer Overflows Hall of Fame

• **Morris worm (1988):** overflow in fingerd
  – 6,000 machines infected (10% of existing Internet)

• **CodeRed (2001):** 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 (2003):** overflow in MS-SQL server
  – 75,000 machines infected in 10 minutes (!!)
Buffer Overflows Hall of Fame (2)

- **Sasser (2004):** 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 (2008-09):** overflow in Windows Server
  - *~10 million machines infected*
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

```
Allocate local buffer (126 bytes reserved on stack)
Copy argument into local buffer
```

Stack grows this way

```
buf    sfp    ret addr    str
```

Local variables

Pointer to previous frame

Execute code at this address after func() finishes

Arguments

Frame of the calling function

Top of stack
Stack Buffers (2)

- 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)
What if Buffer is Over-stuffed?

- 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.

  ![Diagram](diagram.png)

- `strcpy` does NOT check whether the string at *str contains fewer than 126 characters.

  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); ...
  ```

  Copies username (“user”) into buffer (“record”), then appends “:“ and hashed password (“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); ...
  ```
strncpy misuse in htpasswd "fix"

Published "fix" for Apache htpasswd overflow:

```
... 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 ":" characters into buffer
- Again put up to MAX_STRING_LEN-1 characters into buffer
**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`
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);
}
```
void func(char *s){
    char buf[80];
    int (*fnptr)();
    gets(buf);
}

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!
Attack 4: Frame Pointer

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!

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

Checks that “len” is not too big
Negative “len” will always pass this check...

Copies “len” bytes from kernel memory to user space

... interpreted as a huge unsigned integer here

... will copy up to 4G of kernel memory
Time of Check to Time of Use (TOCTTOU) Errors

• **Concurrency issue**
  – Successive instructions may not execute serially
  – Other processes may be given control

• **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 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 …
Combination of Flaws

• Can be used together
• Example: Attacker can
  – Use buffer overflow to disrupt code execution
  – Use TOCTTOU to add a new user to system
  – Use incomplete mediation to achieve privileged status
  – …