Software Errors
Buffer Overflow
TOCTTOU
Why 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
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
How many bugs/line of code

Up to 5% BPLOC!!!
Buffer Overflow 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 Overflow 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
e.g. stack buffer

- 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.

---

**Diagram:**
- Stack grows this way
- Local variables
- Frame of the calling function
- Top of stack
- Execute code at this address after func() finishes
- Arguments
- Pointer to previous frame
- Allocate local buffer (126 bytes reserved on stack)
- Copy argument into local buffer
Stack buffer (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

Stack grows this way

- `buf`
- `overflow`
- `str`
- `Frame of the calling function`
- `Top of stack`

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

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

Attacker puts actual assembly instructions into his input string, e.g., binary code of `execve("/bin/sh")`

In the overflow, a pointer back into the buffer appears in the location where the system expects to find return address

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
Real Problem: No Range Checks

• **strncpy** does **not** check input size
  – `strncpy(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,":");
  strcat(record,cpw, MAX_STRING_LEN-1); ...
  ```
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
- 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);
}
```

---

**Local variables**

- `buf` - Pointer to previous frame
- `fnptr` - Execute code at this address after `func()` finishes
- `sfp` - Arguments
Alter Pointer Variables (2)

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

Send malicious code in \( s \)
Change the caller’s \textit{saved frame ptr.}
Pass more than 80 bytes in \texttt{gets}
\texttt{sfp} now points to malicious code
Caller’s return address read from \texttt{sfp}
When \texttt{func} returns, mal. code runs!

\begin{itemize}
    \item \texttt{buf}
    \item \texttt{s}
    \item \texttt{ret addr}
    \item \texttt{malicious code}
\end{itemize}
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

• 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
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);
}
TOCTTOU Defense

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 ...
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

- *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
Use in Combination

• 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
  – ...
Firmware Supply Chains

- Are long and obscure
- Involve hundreds of modules (300+)
- Many tens of (sub)vendors
- Firmware is often flashed in factory (China)
- Relatively easy to compromise (<$100k)
Firmware

- Is critical
- Is completely overlooked
- Underpins everything on top
Firmware Compromise

- Is virtually **impossible to detect**
- **Much easier than compromising foundries/chips**
- Transforms the machine into an APT zombie
- Any “security” built on top is 100% compromised
- Almost the very definition of “sandcastle”
## Modern Stack: Millions of Bugs. Literally.

<table>
<thead>
<tr>
<th>Data</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM OS Image</td>
<td>15,000,000 - 250,000,000</td>
</tr>
<tr>
<td>VM OS Kernel</td>
<td>2,000,000 - 28,000,000</td>
</tr>
<tr>
<td>Cloud Hypervisor</td>
<td>6,000,000</td>
</tr>
<tr>
<td>Motherboard BIOS/firmware</td>
<td>1,500,000</td>
</tr>
<tr>
<td>Motherboard IPMI/controller</td>
<td>2,000,000</td>
</tr>
<tr>
<td>Intel ME</td>
<td>100,000 - 200,000</td>
</tr>
<tr>
<td>Intel AMT</td>
<td>150,000 - 300,000</td>
</tr>
<tr>
<td>Intel microcode</td>
<td>500,000 - 750,000</td>
</tr>
<tr>
<td>Cloud Network Fabric / SDN</td>
<td>14,000,000 - 25,000,000</td>
</tr>
</tbody>
</table>

- 40,000,000-300,000,000 lines
- 2% bugs/line of code (BLOC)
  - 600,000-6,000,000 bugs
- 5% actual exploits
  - 30,000-300,000 viable exploits
- Zero-day exploit
  - $500k+ on darknet
- Exploit market
  - $150b
Firmware is impossible to fully secure

- Typical BIOS
  - 2-3m lines of code
  - about 60,000 bugs
  - about 3000 exploits

- Smallest custom embedded BIOSes
  - 20,000-100,000 lines of code
  - at least 400-2000 bugs
  - at least 20-100 exploits
  - not really usable in modern servers
80% of Firmware:

- is unnecessary
- is obsolete
- is full of bugs and exploits
- is difficult or impossible to update
- should be removed
- harden remaining core
- this can significantly disrupt supply chain attacks
Results: SuperMicro

DECAF Runtime - SuperMicro A1Sri

*Removed 152/244 modules
**~62% of modules
**~70% of binary
Results: Tyan

DECAF Runtime - Tyan S5533

*Removed 134/194 modules
** ~70% of modules
** ~40% of binary
Results: SuperMicro

DECAF Runtime - SuperMicro A2SDi

*Removed 154/312 modules
** ~50% of modules
** ~50% of binary
Take Home (Firmware)

- Firmware is critical yet very often overlooked
- Everything built on top depends on its security
- Even the best firmware has thousands of bugs
- Firmware supply chains are difficult to trust
- Reducing firmware vulnerability surfaces can significantly disrupt supply chain attacks