Introduction of Software Security

Attacks Are Staggeringly Expensive

• “Cybercrime proceeds in 2004 were $105 billion, greater than those of illegal drug sales” --- Valerie McNiven

• “Identity fraud reached $52.6 billion in 2004.” --- Javelin Strategy & Research

• “Dealing with viruses, spyware, PC theft, and other computer-related crimes costs U.S. businesses a staggering $67.2 billion a year --- FBI

• “Over 130 major intrusions exposed more than 55 million Americans to the growing variety of fraud as personal data like Social Security and credit card numbers were left unprotected” --- USA Today
The Changing Threats to Computer Security

• Vulnerable programs
  – Coding bugs, buffer overflows, parsing errors
• Malicious programs
  – Spyware, trojans, rootkits
• Misconfigured programs
  – Security features not turned on
  – Complex configuration
• Social engineering
  – Phishing/pharming

Causes

• Complexity
  – One security-related bug per thousand lines of source code
• Homogeneity
  – Same operating systems, software, libraries and hardware
• Connectivity
  – Everything is connected in the Internet
• Fundamental OS design flaws
  – Monolithic design
  – Insufficient access control
Software Security

• Common vulnerabilities:
  – Buffer overflow
  – Dangling pointer
  – Format string bugs
  – Time-of-check-to-time-of-use bugs
  – Symbolic link races
  – SQL injection
  – Directory traversal
  – Cross-site scripting
  – Cross-site request forgery
  – ...

Vulnerabilities discovered per year (CERT)
Days from patch to exploit (information security, July 2004)

Software vulnerabilities in C/C++ programs

- String
- Integer
- Formatted IO
- Race Condition
Strings

- Strings—such as command-line arguments, environment variables, and console input—are of special concern in secure programming because they comprise most of the data exchanged between an end user and a software system. Graphic and Web-based applications make extensive use of text input fields and, because of standards like XML, data exchanged between programs is increasingly in string form as well. As a result, weaknesses in string representation, string management, and string manipulation have led to a broad range of software vulnerabilities and exploits.

Examples

```
1. int main(void) {
    2. char Password[80];
    3. puts("Enter 8 character password:");
    4. gets(Password); ...
    5. }

Reading unbounded stream from standard input
```

```
1. int main(int argc, char *argv[]) {
    2. char name[2048];
    3. strcpy(name, argv[1]);
    4. strcat(name, " = ");
    5. strcat(name, argv[2]); ...
    6. }

Unbounded string copy and concatenation
```

```
1. #include <iostream>
2. int main(void) {
    3. char buf[12];
    4. cin >> buf;
    5. cout << "echo: " << buf << endl;
    6. }

Extracting characters from cin into a character array
```
Preparation

• echo 0 > proc/sys/kernel/randomize_va_space
• gcc –fno-stack-protector example01.c –o example-01

Problem 1

• Craft a malicious input to bypass the authentication:
  – Print “access granted” instead of “access denied”
Problem 2

• Inject arbitrary code to execute
  – A shell code template is given
  – Make a working exploit that runs “ps”

Problem 3

• Return into an existing function in libc
  – Make a working exploit that runs “ps”
Mitigations

- Secure Coding practice
- Compiler Enhancement
- OS/Hardware Enhancement

Secure Coding: Input Validation

```c
1. int myfunc(const char *arg) {
2.     char buff[100];
3.     if (strlen(arg) >= sizeof(buff)) {
4.         abort();
5.     }
6. }
```
Secure Coding: gets vs. fgets vs. gets_s

1. #define BUFFSIZE 9
2. int main(int argc, _TCHAR* argv[]){
3.     char buff[BUFFSIZE];
4.     // insecure use of fgets()
5.     fgets(buff);
6.     printf("gets: %s
", buff);
7.     abort();
8. }
9. printf("fgets: %s
", buff);
10. if (fgets(buff, BUFFSIZE, stdin) == NULL) {
11.     printf("invalid input.
");
12.     abort();
13. }
14. printf("gets_s: %s
", buff);
15. return 0;
16. }

Secure Coding: strcpy & strcat

• Standard: strncpy, strncat
  strncpy(dest, source, dest_size - 1);
  dest[dest_size - 1] = '\0';

• Nonstandard: strcpy_s, strcat_s, strlcpy, strlcat
Compiler Enhancement: Canary

- Location (A) has no array or pointer variables.
- Location (B) has arrays or structures that contain arrays.
- Location (C) has no arrays.

No-Execute Protection

- In new hardware:
  - NX (No-Execute) by AMD
  - XD (eXecute-Disabled) by Intel
  - DEP (Data Execution Prevention) by Microsoft
- A bit in the page table entry indicates if this page can be executed.
- Software emulation: PaX, W^X
- Prevent code injection attack
OS Enhancement: ASLR

• ASLR: Address Space Layout Randomization
  – Stack, heap, executable, library, etc
  – Executable/library need to be compiled to be PIE
    (e.g. position-independent executable)
  – On 32-bit architecture
    • 5-10% performance overhead
    • Not enough entropy: brute force can still succeed
  – On 64-bit architecture
    • Very low performance overhead
    • Enough entropy

Integer Vulnerabilities

• Integer Overflow
• Sign Error
• Truncation Error
Integer Overflow

1. int i;
2. unsigned int j;
3. i = INT_MAX;  // 2,147,483,647
4. i++;
5. printf("i = %d\n", i);  /* i = -2,147,483,648 */
6. j = UINT_MAX;  // 4,294,967,295
7. j++;
8. printf("j = %u\n", j);  /* j = 0 */
9. i = INT_MIN;  // -2,147,483,648;
10. i--;
11. printf("i = %d\n", i);  /* i = 2,147,483,647 */
12. j = 0;
13. j--;
14. printf("j = %u\n", j);  /* j = 4,294,967,295 */

Integer Overflow Vulnerability

A real world vulnerability in handling comments in JPEG files
Sign Error

1. int i = -1;
2. unsigned short u;
3. u = i;
4. printf("u = %hu\n", u); /* u = 65533 */

Sign Error Vulnerability

1. #define BUFF_SIZE 10
2. int main(int argc, char* argv[1]){
3.   int len;
4.   char buf[BUFF_SIZE];
5.   len = atoi(argv[1]);
6.   if (len < BUFF_SIZE){
7.     strcpy(buf, argv[2], len);
8.   }
9.   else
10.   printf("Too much data\n");
11.   }
Truncation Errors

1. unsigned short int u = 32769;
2. short int i;
3. i = u;
4. printf("i = %d\n", i); /* i = -32768 */
5. u = 65535;
6. i = u;
7. printf("i = %d\n", i); /* i = -1 */

Truncation Error Vulnerability

1. int main(int argc, char *const *argv) {
2.   unsigned short int total;
3.   total = strlen(argv[1])+strlen(argv[2])+1;
4.   char *buff = (char *) malloc(total);
5.   strcpy(buff, argv[1]);
6.   strcat(buff, argv[2]);
7. }

Mitigations for Integer Vulnerabilities

• Type range checking
  – In Pascal & Ada: type day is new INTEGER range 1..31
  – In C: we need to explicitly check at runtime

• Compiler checking
  – Warning for “possible loss of data”
  – Runtime checks
    • VC++: /RTCc  GCC: -ftrapv
    • Performance overhead is high, only good for debugging

• Safe library: SafeInt

• Research Ideas
  – Static Binary Analysis
  – Dynamic Testing

Format String Vulnerabilities

• Buffer Overflow
• Read Memory Content
• Write Memory Content
Format String: Buffer Overflow

1. char buffer[512];
2. sprintf(buffer, "Wrong command: %s\n", user);

When user is too large

1. char outbuf[512];
2. char buffer[512];
3. sprintf(outbuf, "ERG Wrong command: %s.400\n", user);
4. sprintf(buffer, outbuf);

user = $497d\x3c\xd3\xff\x1f\n<shellcode>

Format String: View Stack Content

```c
char format [32];
strcpy(format, "%08x.%08x.%08x%08x%08x%08x%08x%08x\n");
...
print(format, 1, 2, 3);
```

How to view arbitrary memory content?

![Initial argument pointer](image1)

![Final argument pointer](image2)

Output: 00000000.00000002.00000003.25303878

Format string:

```
% 08 x . % 08 x . % 08 x . % 08 x
```

Initial argument pointer: `e0f84201`
Format String: Write Arbitrary Memory

```c
int i;
printf("hello\n", (int *)&i);
```

After printf, i=5

A malicious case:

```c
printf("\xdc/xf5\xd0\x08\x08\x08\n");
```

Mitigations

- Making format string static/constant
- Dynamic use of static content
- `snprintf` versus `sprintf`

```c
1. #include <stdio.h>
2. #include <string.h>
3. int main(int argc, char * argv[]) {
4.     int x, y;
5.     static char format[256] = "{x} \x{y} \x{y}";
6.     x = atoi(argv[1]);
7.     y = atoi(argv[2]);
8.     if (strcmp(argv[3], "hex") == 0) {
9.         strcat(format, "\x{x}\n");
10.    } else {
11.        strcat(format, "\x{y}\n");
12.    }
13.    printf(format, x, y, x * y);
14.    exit(0);
15. }
```
Mitigations (cont’d)

- Compiler checks
  - GNU C compiler flags include -Wformat, -Wformat-noliteral, and -Wformat-security

- Research Ideas:
  - Static taint analysis
  - Dynamic taint analysis
Race Condition

- Race Condition:
  - An unanticipated execution ordering of concurrent flows that results in undesired behavior

- Three Properties:
  - Concurrency
  - Shared Object
  - Change State

- TOCTOU Race Condition
  - Time of check, time of use

Exploiting Symbolic Links

```c
1. if (stat("/some_dir/some_file", &statbuf) == -1) {
2.   err1, "stat");
3. }
4. if (statbuf.st_size >= MAX_FILE_SIZE) {
5.   err2, "file size");
6. }
7. }
8. if ((fd=open("/some_dir/some_file", O_RDONLY)) == -1) {
9.   err3, "open - /some_dir/some_file");
10. }
11. // process file
```

An attacker that has appropriate permission could exploit this vulnerability by executing the following commands during the race window (between lines 1 and 8):

- `rm /some_dir/some_file`
- `ln -s attacker_file /some_dir/some_file`
Exploiting Temporary Files

- If a /tmp/some_file file already exists, then that file is opened and truncated.
- If /tmp/some_file is a symbolic link, then the target file referenced by the link is truncated.
- This call to open fails whenever /tmp/some_file already exists, including when it is a symbolic link.
- The test for file existence and the file creation are guaranteed to be atomic.

```c
int fd = open("/tmp/some_file",
    O_WRONLY | O_CREAT | O_TRUNC, 0600);
```

Mitigation

- No easy solution
- Use file descriptor instead of filename
  - fchown vs. chown, fstat vs. stat, fchmod vs. chmod
  - Use caution with link, unlink, symlink, mkdir, rmdir, mount, umount, etc.
- Avoid shared objects, if possible
- Least privilege
- Temporary files
  - Never reuse filenames
  - Randomize filename generation
  - Use mkstemp, rather than mktemp, tempnam, or tempnam_s

```c
int fd = open("/tmp/some_file",
    O_WRONLY | O_CREAT | O_EXCL | O_TRUNC, 0600);
```
Demo --- Exploit String Vulnerability

1. int IsPasswordOkay(void) {
2.   char Password[12];
3.   gets(Password);
4.   if (!strcmp(Password, "goodpass"))
5.     return(true);
6.   else return(false);
7. }
8. void main(void) {
9.   int PwStatus;
10.  puts("Enter password:");
11.  PwStatus = IsPasswordOkay();
12.  if (PwStatus == false) {
13.    puts("Access denied");
14.    exit(-1);
15.  }
16.  else puts("Access granted");
17. }

Introduction of Malware
Outline

• Malware Taxonomy & Overview
• Code Obfuscation
• Rootkit Techniques
• New Trends

Malware Taxonomy

• Virus vs. Worm
  – Propagate itself or human involved
• Adware/Spyware
• Keylogger
• Password thief
• Network sniffer
• Mass mailer
• Backdoor
• Bot
• Driveby-download
  – Exploit browser vulnerabilities
• Rootkit
Malicious Code Problem

Malware is everywhere.
Source: Symantec Internet Security Threat Report (vol. VII)

- Large malware families.

Obfuscation Techniques

- Metamorphism
  - Upon replication, the malware generates a new (equivalent) version of itself

- Polymorphism
  - The malware encrypts its malicious payload, to be decrypted for execution
  - The encryptor and decryptor functions mutate with each replication

- Emulation
  - The malicious payload is converted into a virtual instruction set
  - An interpreter is imbedded in the malware to emulate each virtual instruction at runtime
Metamorphism

• Code Transposition (changing order of instructions)
  – Version 1 and 2 are semantically equivalent:

  **Version 1:**
  
  mov eax, ebx  
  mov ecx, 5  
  jmp +14  

  ...

  **Version 2:**
  
  mov ecx, 5  
  mov eax, ebx  
  jmp +14


Metamorphism 2

• “nop” insertion
  – Version 1 and 2 are semantically equivalent:

  **Version 1:**
  
  mov eax, ebx  
  mov ecx, 5  
  call [ebp]  

  ...

  **Version 2:**
  
  mov eax, ebx  
  mov eax, ecx  
  test eax, eax  
  nop  
  inc eax  
  dec eax  
  mov ecx, 5  
  call [ebp]
Metamorphism 3

• Register re-assignment
  — Version 1 and 2 are semantically equivalent, calling function at 0x2020 with parameter ‘5’ and clearing both ebx and eax:

<table>
<thead>
<tr>
<th>Version 1:</th>
<th>Version 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, 5</td>
<td>mov ebx, 5</td>
</tr>
<tr>
<td>push eax</td>
<td>push ebx</td>
</tr>
<tr>
<td>call 0x2020</td>
<td>call 0x2020</td>
</tr>
<tr>
<td>xor eax, eax</td>
<td>xor ebx, ebx</td>
</tr>
<tr>
<td>xor ebx, ebx</td>
<td>xor eax, eax</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Metamorphism 4

• Substitution of equivalent instruction sequences
  — Version 1 and 2 are semantically equivalent:

<table>
<thead>
<tr>
<th>Version 1:</th>
<th>Version 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, 5</td>
<td>mov eax, 5</td>
</tr>
<tr>
<td>shl eax, 1</td>
<td>mul eax, eax, 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Metamorphism 5

• Modifying condition jumps
  – Version 1, 2, and 3 are semantically equivalent:

<table>
<thead>
<tr>
<th>Version 1:</th>
<th>Version 2:</th>
<th>Version 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, 5</td>
<td>mov eax, 5</td>
<td>mov eax, 5</td>
</tr>
<tr>
<td>test eax, eax</td>
<td>push 0x2020</td>
<td>jmp 0x2020</td>
</tr>
<tr>
<td>jnz 0x2020</td>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Polymorphism (Packed Executable)
Emulator-Based Obfuscation

Obfuscated Program

$P_L$

Bytecode Program (written in language L)

ISA for L

$EM_{x86}^L$

Emulator

Original Malware Program (written in x86)

$x86$ ISA

Obfuscation Method

Sharf et al. 
Automatically Reverse Engineering Malware Emulators

Impacts on Existing Malware Analysis

- **Unknown Language L**
  - $L$ can be randomly generated

- **Pure Static Analysis (whitebox)**
  - Completely thwarted
  - Only emulator code is analyzable
  - $P_L$ is considered as data by analyzer

- **Greybox methods**
  - Includes instruction level analyzers, information-flow, dynamic tainting, multi-path exploration etc.
  - Analysis is inaccurate
  - For example, paths may explored in the emulator, but not the malware

Sharf et al. 
Automatically Reverse Engineering Malware Emulators
Rootkits

- Replace system utility tools
  - E.g., ls, ps, netstat
- Hooking user-level APIs
  - Hot patching
  - Modify IAT, EAT
- Kernel hooking
  - System call table, IDT
  - Function pointers on heap (stealthier)
- Direct Kernel Object Manipulation (DKOM)
  - Unlike a process object from the active process list
  - Set pid to 0
- Virtual Machine Monitor based rootkit
  - Using hardware virtualization technology
  - Bluepill
- BIOS, Firmware rootkit ...

Trend for Attackers

- From Virus to Worm to Driveby Downloads
  - No exploit -> simple exploits -> complex exploits
- From user to kernel to even lower level
  - It become harder to detect and has higher privilege
- Code obfuscation is common practice
  - Metamorphism, Polymorphism, Built-in emulator
- From hobby to profit driven
  - Economy chain
  - E.g, exploits infrastructure, botnet, black market
Trend for Defenders

- Traditional malware detection is failing
  - Signature checking: byte sequence, regular expression
  - Semantic-aware: too expensive, not practice
  - Whitelisting is a promising approach

- More security mechanism should be implemented in OS
  - Finer-grained access control
  - E.g., UAC
  - More kernel code integrity protection