Buffer Overflow Attacks

• > 50% of security incidents reported at CERT are related to buffer overflow attacks

• Problem is access control but at a very fine level of granularity

• C and C++ programming languages don’t do array bounds checks
Buffer overflows in library code

• Basic problem is that the library routines look like this:

```c
void strcopy(char *src, char *dst) {
    int i = 0;
    while (src[i] != "\0") {
        dst[i] = src[i];
        i = i + 1;
    }
}
```

• If the memory allocated to `dst` is smaller than the memory needed to store the contents of `src`, a buffer overflow occurs.
Attack Targets and Locations

• Targets
  – Return address
  – Function pointer
  – Longjmp buffer
  – A flag relative to control flow …

• Locations
  – Stack
  – Heap or Data segment
• Linux Kernel 2.6, GCC 4.1.2
• Without stack protection
• Word size = 32 bits (4 bytes)
• Little Endian
  – *(0x0004) = 0x01234567
• Stack grows to smaller addresses
Data Representations

- Sizes of C Objects (in Bytes)
  - C Data Type
    - int 4
    - char 1
    - pointer 4

- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Character “0” has code 0x30
      - Digit i has code 0x30+i
  - String should be null-terminated
    - Final character = 0

```c
char str[5] = "1234";
```

<table>
<thead>
<tr>
<th>Linux</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>bf00</td>
</tr>
<tr>
<td>32</td>
<td>bf01</td>
</tr>
<tr>
<td>33</td>
<td>bf02</td>
</tr>
<tr>
<td>34</td>
<td>bf03</td>
</tr>
<tr>
<td>00</td>
<td>bf04</td>
</tr>
</tbody>
</table>
3 parts of C memory model

- The code & data (or "text") segment
  - contains compiled code, constant strings, etc.
- The Heap
  - Stores dynamically allocated objects
  - Allocated via "malloc"
  - Deallocated via "free"
  - C runtime system
- The Stack
  - Stores local variables
  - Stores the return address of a function
C’s Control Stack

f() {
    g(parameter);
}

g(char *args) {
    int x;
    // more local
    // variables
    ...
}
C’s Control Stack

```c
f() {
    g(parameter);
}

g(char *args) {
    int x;
    // more local
    // variables
    ...
}
```

Larger Addresses
 ESP

EBP

f’s stack frame
C’s Control Stack

```c
f() {
    g(parameter);
}

g(char *args) {
    int x;
    // more local
    // variables
    ...
}
```

Larger Addresses

ESP

EBP

Input parameter

f’s stack frame
C’s Control Stack

f() {
    g(parameter);
}

g(char *args) {
    int x;
    // more local
    // variables
...
}
C’s Control Stack

```c
f() {
    g(parameter);
}

g(char *args) {
    int x;
    // more local // variables
    ...
}
```

Diagram:
- ESP
- EBP
- Larger Addresses
- Base Pointer
- Return Addr.
- Input parameter
- f’s stack frame

Larger Addresses
Buffer Overflow Example

```c
f() {
    g(parameter);
}

g(char *text) {
    char buffer[128];
    strcpy(buffer, text);
    text
}
```
Buffer Overflow Example

```c
f() {
    g(parameter);
}

g(char *text) {
    char buffer[128];
    strcpy(buffer, text);
}
```
Assembly Instructions

• Register
  – %esp : stack pointer
  – %ebp : frame base pointer
  – %eax : function return value
  – ...

• Constant Numbers: $0x0, $-16
  • Moving Data: movl src dest
  • Arithmetic/Logical Operation:
    • addl $12, %esp
    • subl , andl
    • Stack Operation: pushl, popl
    • Control Flow: jmp, call, leave, ret…
Details: C calling conventions

```c
int function(int a, int b, int c) {
    char buffer1[4];
    int ans = a + b + c;
    char buffer2[10];
    return ans;
}

int main() {
    return function(1,2,3);
}

Compile with: gcc -S -o example.s example.c
```
Resulting Assembly (1)

```
.file "example.c"
.text
.globl function
...
.globl main
    .type main, @function
main:
    pushl %ebp
    movl %esp, %ebp
    // Set up stack frame
    subl $8, %esp
    andl $-16, %esp
    subl $16 %esp
    // Align the stack on 16-byte boundary,
    // reserve some space on the stack
    pushl $3
    pushl $2
    pushl $1
    // Push arguments onto the stack
    call function
    // Push return address, jump to function:
    addl $12, %esp
    // Pop arguments off the stack
    leave
    // Tear down stack frame, Undo stack alignment
    ret
```
## Resulting Assembly (2)

```assembly
.globl function
.type function, @function

function:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl %ebp</td>
<td>Set up stack frame</td>
</tr>
<tr>
<td>movl %esp, %ebp</td>
<td></td>
</tr>
<tr>
<td>subl $32, %esp</td>
<td>Allocate local storage</td>
</tr>
<tr>
<td>movl 12(%ebp), %eax</td>
<td></td>
</tr>
<tr>
<td>addl 8(%ebp), %eax</td>
<td>ans = a + b + c</td>
</tr>
<tr>
<td>addl 16(%ebp), %eax</td>
<td></td>
</tr>
<tr>
<td>movl %eax, -4(%ebp)</td>
<td>%eax holds the return value</td>
</tr>
<tr>
<td>movl -4(%ebp), %eax</td>
<td></td>
</tr>
<tr>
<td>leave</td>
<td>Tear down stack frame</td>
</tr>
<tr>
<td>ret</td>
<td>Pop return address &amp; jump to it</td>
</tr>
</tbody>
</table>

.size function, .-function
```

### Smaller Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td></td>
</tr>
<tr>
<td>-18</td>
<td>buf2</td>
</tr>
<tr>
<td>-8</td>
<td>buf1</td>
</tr>
<tr>
<td>-4</td>
<td>ans</td>
</tr>
<tr>
<td>ebp</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Constructing a Payload

- Idea: Overwrite the return address on the stack
  - Value overwritten is an address of some code in the "payload"
  - The processor will jump to the instruction at that location
  - It may be hard to figure out precisely the location in memory

- You can increase the size of the "target" area by padding the code with no-op instructions
- You can increase the chance over overwriting the return address by putting many copies of the target address on the stack

[NOP]…[NOP]{attack code} {attack data}[ADDR]…[ADDR]
More About Payloads

• How do you construct the attack code to put in the payload?
  – You use a compiler!
  – Gcc + gdb + options to spit out assembly (hex encoded)

• What about the padding?
  – NOP on the x86 has the machine code 0x90

• How do you guess the ADDR to put in the payload?
  – Some guesswork here
  – Figure out where the first stack frame lives: OS & hardware
    platform dependent, but easy to figure out
  – Look at the program -- try to guess the stack depth at the point of
    the buffer overflow vulnerability.
Project hints

• Use mod-l.seas.upenn.edu
  – minus.seas.upenn.edu still has stack protection turned on
  – 'uname -a' will give you some useful information about which machine you're connected to

• GCC has changed significantly since the Aleph One tutorial was written:
  – 16 bit vs. 32 bit architecture
  – GCC now automatically reserves bytes of "free" space in main() frame.
  – Syntax of inline assembly is different
  – GCC (>=4.1) supports canaries in main() frame, extra credit
If you must use C/C++

• Avoid the (long list of) broken library routines:
  – strcpy, strcat, sprintf, scanf, sscanf, `gets`, read, …

• Use (but be careful with) the "safer" versions:
  – e.g. strncpy, snprintf, fgets, …

• *Always* do bounds checks
  – One thing to look for when reviewing/auditing code

• Be careful to manage memory properly
  – Dangling pointers often crash program
  – Deallocate storage (otherwise program will have a memory leak)

• Be aware that doing all of this is difficult.
Defeating Buffer Overflows

• Use a typesafe programming language
  – Java/C# are not vulnerable to these attacks

• Some operating systems patch, E.g. eniac-l
  – move the start of the stack on a per-process basis
  – turn on non-executable stack
  – Gcc (>=4.1) supports Stack-Smashing-Protection (SSP)
    • Padding canaries between local vars and return address