Low-level Software Security: Attacks and Countermeasures

Prof. Frank PIESSENS

These slides are based on the paper: “Low-level Software Security by Example” by Erlingsson, Younan and Piessens
Overview

- Introduction
- Example attacks
  - Stack-based buffer overflow
  - Heap-based buffer overflow
  - Return-to-libc attacks
  - Data-only attacks
- Example defenses that prevent / detect exploitation
  - Stack canaries
  - Non-executable data
  - Control-flow integrity
  - Layout randomization
- Other defenses
- Conclusion
Introduction

• An *implementation-level software vulnerability* is a bug in a program that can be exploited by an attacker to cause harm

• Example vulnerabilities:
  o SQL injection vulnerabilities (discussed in other part of the course)
  o XSS vulnerabilities (discussed in other part of the course)
  o Buffer overflows and other memory corruption vulnerabilities

• An *attack* is a scenario where an attacker triggers the bug to cause harm

• A *countermeasure* is a technique to counter attacks

• These lectures will discuss memory corruption vulnerabilities, common attack techniques, and common countermeasures for them
Memory corruption vulnerabilities

• Memory corruption vulnerabilities are a class of vulnerabilities relevant for *unsafe* languages
  o i.e. Languages that do not check whether programs access memory in a correct way
  o Hence buggy programs may mess up parts of memory used by the language run-time

• In these lectures we will focus on memory corruption vulnerabilities in C programs
  o These can have *devastating* consequences
Example vulnerable C program

```c
#include <stdio.h>

int main() {
    int cookie = 0;
    char buf[80];
    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);
    if (cookie == 0x41424344)
        printf("you win!\n");
}
```
Example vulnerable C program

```c
#include <stdio.h>

int main() {
    int cookie;
    char buf[80];
    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);
}
```
Background: Memory management in C

- Memory can be allocated in many ways in C
  - Automatic (local variables in functions)
  - Static (global variables)
  - Dynamic (malloc and new)

- Programmer is responsible for:
  - Appropriate use of allocated memory
    - E.g. bounds checks, type checks, …
  - Correct de-allocation of memory
Process memory layout

- High addresses:
  - Arguments/ Environment
  - Stack
  - Unused and Mapped Memory
    - Heap (dynamic data)
    - Static Data
    - Program Code

- Low addresses:

- Stack grows down
- Heap grows up
Memory management in C

• Memory management is very error-prone
• Some typical bugs:
  o Writing past the bound of an array
  o Dangling pointers
  o Double freeing
  o Memory leaks
• For efficiency, practical C implementations don’t detect such bugs at run time
  o The language definition states that behavior of a buggy program is *undefined*
Overview

• Introduction

• Example attacks
  o Stack-based buffer overflow
  o Heap-based buffer overflow
  o Return-to-libc attacks
  o Data-only attacks

• Example defenses
  o Stack canaries
  o Non-executable data
  o Control-flow integrity
  o Layout randomization

• Other defenses

• Conclusion
Stack based buffer overflow

• The stack is a memory area used at run time to track function calls and returns
  o Per call, an activation record or stack frame is pushed on the stack, containing:
    • Actual parameters, return address, automatically allocated local variables, …

• As a consequence, if a local buffer variable can be overflowed, there are interesting memory locations to overwrite nearby
  o The simplest attack is to overwrite the return address so that it points to attacker-chosen code (shellcode)
Stack based buffer overflow

```
f0:
  ...
  call f1
  ...

f1:
  buffer[]
  overflow()
  ...
```

Stack

- Return address f0
- Saved Frame Ptr f0
- Local variables f0
Stack based buffer overflow

```
f0:
    ...
    call f1
    ...
f1:
    buffer[
    overflow()
    ...
```

Stack
- Return address f0
- Saved Frame Ptr f0
- Local variables f0
- Arguments f1
- Return address f1
- Saved Frame Ptr f1
- Space for buffer
Stack based buffer overflow

f0:
...  
  call f1
...  

f1:
  buffer[]
  overflow()
...  

Stack

- Return address f0
- Saved Frame Ptr f0
- Local variables f0
- Arguments f1
- Overwritten address
- Injected Code
Very simple shell code

- In examples further on, we will use:

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xcd 0x2e</td>
<td>int 0x2e ; system call to the operating system</td>
</tr>
<tr>
<td>0xeb 0xfe</td>
<td>L: jmp L ; a very short, direct infinite loop</td>
</tr>
</tbody>
</table>

- Real shell-code is only slightly longer:

```c
LINUX on Intel:
char shellcode[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xeb\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";
```
Side-note: endianness

- Intel processors are *little-endian*
Stack based buffer overflow

• Example vulnerable program:

```c
int is_file_foober( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one);
    strcat( tmp, two);
    return strcmp( tmp, "file://foobar" );
}
```
Stack based buffer overflow

• Or alternatively:

```c
int is_file_foobar_using_loops( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    char* b = tmp;
    for( ; *one != '\0'; ++one, ++b ) *b = *one;
    for( ; *two != '\0'; ++two, ++b ) *b = *two;
    *b = '\0';
    return strcmp( tmp, "file://foobar" );
}
```
Stack based buffer overflow

- Snapshot of the stack before the return:

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff5c</td>
<td>0x00353037</td>
<td>argument two pointer</td>
</tr>
<tr>
<td>0x0012ff58</td>
<td>0x0035302f</td>
<td>argument one pointer</td>
</tr>
<tr>
<td>0x0012ff54</td>
<td>0x00401263</td>
<td>return address</td>
</tr>
<tr>
<td>0x0012ff50</td>
<td>0x0012ff7c</td>
<td>saved base pointer</td>
</tr>
<tr>
<td>0x0012ff4c</td>
<td>0x00000072</td>
<td>tmp continues 'r'’int' 0‘ int’ 0‘ int’ 0‘</td>
</tr>
<tr>
<td>0x0012ff48</td>
<td>0x61626f6f</td>
<td>tmp continues 'o’ 'o' 'b' 'a'</td>
</tr>
<tr>
<td>0x0012ff44</td>
<td>0x662f2f3a</td>
<td>tmp continues ':' '/' '/' 'f'</td>
</tr>
<tr>
<td>0x0012ff40</td>
<td>0x656c6966</td>
<td>tmp array: 'f' 'i' 'l' 'e'</td>
</tr>
</tbody>
</table>
Stack based buffer overflow

- Snapshot of the stack before the return:

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff5c</td>
<td>0x00353037</td>
<td>argument two pointer</td>
</tr>
<tr>
<td>0x0012ff58</td>
<td>0x0035302f</td>
<td>argument one pointer</td>
</tr>
<tr>
<td>0x0012ff54</td>
<td>0x00401263</td>
<td>return address</td>
</tr>
<tr>
<td>0x0012ff50</td>
<td>0x0012ff7c</td>
<td>saved base pointer</td>
</tr>
<tr>
<td>0x0012ff4c</td>
<td>0x00000072</td>
<td>tmp continues 'r' '\0' '\0' '\0' 'o' 'o' 'b' 'a'</td>
</tr>
<tr>
<td>0x0012ff48</td>
<td>0x61626f6f</td>
<td>tmp continues ':' '/' '/' 'f'</td>
</tr>
<tr>
<td>0x0012ff44</td>
<td>0x662f2f3a</td>
<td>tmp continues 'f' 'i' 'l' 'e'</td>
</tr>
<tr>
<td>0x0012ff40</td>
<td>0x656c6966</td>
<td>tmp array: 'f' 'i' 'l' 'e'</td>
</tr>
</tbody>
</table>
Stack based buffer overflow

- Snapshot of the stack before the return:

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff5c</td>
<td>0x00353037</td>
<td>argument two pointer</td>
</tr>
<tr>
<td>0x0012ff58</td>
<td>0x0035302f</td>
<td>argument one pointer</td>
</tr>
<tr>
<td>0x0012ff54</td>
<td>0x0012ff4c</td>
<td>return address</td>
</tr>
<tr>
<td>0x0012ff50</td>
<td>0x66666666</td>
<td>saved base pointer</td>
</tr>
<tr>
<td>0x0012ff4c</td>
<td>0xfeeb2ecd</td>
<td>tmp continues</td>
</tr>
<tr>
<td>0x0012ff48</td>
<td>0x66666666</td>
<td>tmp continues</td>
</tr>
<tr>
<td>0x0012ff44</td>
<td>0x662f2f3a</td>
<td>tmp continues</td>
</tr>
<tr>
<td>0x0012ff40</td>
<td>0x656c6966</td>
<td>tmp array: 'f' 'i' 'l' 'e'</td>
</tr>
</tbody>
</table>
Stack based buffer overflow

• Lots of details to get right before it works:
  o No nulls in (character-)strings
  o Filling in the correct return address:
    • Fake return address must be precisely positioned
    • Attacker might not know the address of his own string
  o Other overwritten data must not be used before return from function
  o …

• More information in
  o “Smashing the stack for fun and profit” by Aleph One
Overview

• Introduction

• Example attacks
  o Stack-based buffer overflow
  o Heap-based buffer overflow
  o Return-to-libc attacks
  o Data-only attacks

• Example defenses
  o Stack canaries
  o Non-executable data
  o Control-flow integrity
  o Layout randomization

• Other defenses

• Conclusion
Heap based buffer overflow

• If a program contains a buffer overflow vulnerability for a buffer allocated on the heap, there is no return address nearby

• So attacking a heap based vulnerability requires the attacker to overwrite other code pointers

• We look at two examples:
  o Overwriting a function pointer
  o Overwriting heap metadata
Overwriting a function pointer

• Example vulnerable program:

```c
typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
```
Overwriting a function pointer

• And what happens on overflow:

(a) A structure holding “file://foobar” and a pointer to the `strcmp` function.

(b) After a buffer overflow caused by the inputs “file://” and “asdfasdfsdf”.

(c) After a malicious buffer overflow caused by attacker-chosen inputs.
Overwriting heap metadata

• The heap is a memory area where dynamically allocated data is stored
  o Typically managed by a memory allocation library that offers functionality to allocate and free chunks of memory (in C: malloc() and free() calls)

• Most memory allocation libraries store management information in-band
  o As a consequence, buffer overruns on the heap can overwrite this management information
  o This enables an “indirect pointer overwrite”-like attack allowing attackers to overwrite arbitrary memory locations
Heap management in dlmalloc

Dlmalloc maintains a doubly linked list of free chunks.

When chunk c gets unlinked, c’s backward pointer is written to *(forward pointer + 12)*.

Or: green value is written 12 bytes above where red value points.

Free chunk

Top Heap grows with brk()

User data

Chunk in use
Exploiting a buffer overrun

Top Heap grows with brk()

Green value is written 12 bytes above where red value points

A buffer overrun in d can overwrite the red and green values

• Make Green point to injected code
• Make Red point 12 bytes below a function return address
Exploiting a buffer overrun

Top Heap grows with brk()

Green value is written 12 bytes above where red value points

Net result is that the return address points to the injected code
Indirect pointer overwrite

• This technique of overwriting a pointer that is later dereferenced for writing is called *indirect pointer overwrite*

• This is a broadly useful attack technique, as it allows to selectively change memory contents

• A program is vulnerable if:
  - It contains a bug that allows overwriting a pointer value
  - This pointer value is later dereferenced for writing
  - And the value written is under control of the attacker
Overview

• Introduction

• Example attacks
  o Stack-based buffer overflow
  o Heap-based buffer overflow
  o Return-to-libc attacks
  o Data-only attacks

• Example defenses
  o Stack canaries
  o Non-executable data
  o Control-flow integrity
  o Layout randomization

• Other defenses

• Conclusion
Return-into-libc

- *Direct code injection*, where an attacker injects code as data is not always feasible
  - E.g. When certain countermeasures are active
- *Indirect code injection* attacks will drive the execution of the program by manipulating the stack
- This makes it possible to execute fractions of code present in memory
  - Usually, interesting code is available, e.g. libc
Return-into-libc: overview

Stack
- Params for f1
- Return addr
- Params for f2
- Return addr
- Params for f3
- Return addr

Code Memory
- f1
  - return
- f2
  - return
- f3
  - return

SP
IP
Return-into-libc: overview

Stack

- Return addr
- Params for f2
- Return addr
- Params for f3
- Return addr

Code Memory

- f1
  - return
- f2
  - return
- f3
  - return

SP

IP
Return-into-libc: overview

Stack

- Params for f1
- Return addr
- Params for f2
- Return addr
- Params for f3
- Return addr

Code Memory

- f1
  - return
- f2
  - return
- f3
  - return

SP

IP
Return-into-libc: overview

Stack

- Params for f1
- Return addr
- Params for f2
- Return addr
- Params for f3
- Return addr

Code Memory

- f1
  - return
- f2
  - return
- f3
  - return
  - return
  - return

SP

IP
Return-into-libc: overview

Stack

- Return addr
- Params for f2
- Return addr
- Params for f1

Code Memory

- f1
  - return
- f2
  - return
- f3
  - return
Return-into-libc: overview

Stack
- Params for f1
- Return addr
- Params for f2
- Return addr

Code Memory
- f1
  -
  - return
- f2
  -
  - return
- f3
  return
  -
  return
Return-into-libc: overview

Stack
- Params for f1
  - Return addr

Code Memory
- IP
  - f1
    - return
  - f2
    - return
  - f3
    - return
Return-to-libc

• What do we need to make this work?
  o Inject the fake stack
    • Easy: this is just data we can put in a buffer
  o Make the stack pointer point to the fake stack right before a return instruction is executed
    • We will show an example where this is done by jumping to a trampoline
  o Then we make the stack execute existing functions to do a direct code injection
    • But we could do other useful stuff without direct code injection
Vulnerable program

```c
int median( int* data, int len, void* cmp )
{
    // must have 0 < len <= MAX_INTS
    int tmp[MAX_INTS];
    memcpy( tmp, data, len*sizeof(int) );  // copy the input integers
    qsort( tmp, len, sizeof(int), cmp );   // sort the local copy
    return tmp[len/2];                    // median is in the middle
}
```
The trampoline

Assembly code of `qsort`:

```assembly
... push edi ; push second argument to be compared onto the stack
push ebx ; push the first argument onto the stack
call [esp+comp_fp] ; call comparison function, indirectly through a pointer
add esp, 8 ; remove the two arguments from the stack
test eax, eax ; check the comparison result
jle label_less-than ; branch on that result
...
```

Trampoline code

<table>
<thead>
<tr>
<th>address</th>
<th>machine code</th>
<th>assembly-language version of the machine code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7c971649</td>
<td>0x8b 0xe3</td>
<td>mov esp, ebx ; change the stack location to ebx</td>
</tr>
<tr>
<td>0x7c97164b</td>
<td>0x5b</td>
<td>pop ebx ; pop ebx from the new stack</td>
</tr>
<tr>
<td>0x7c97164c</td>
<td>0xc3</td>
<td>ret ; return based on the new stack</td>
</tr>
</tbody>
</table>
Launching the attack

<table>
<thead>
<tr>
<th>stack address</th>
<th>normal contents</th>
<th>benign contents</th>
<th>malicious contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff38</td>
<td>0x004013e0</td>
<td>0x11111111d</td>
<td>0x7c971649</td>
</tr>
<tr>
<td>0x0012ff34</td>
<td>0x00000001</td>
<td>0x11111110c</td>
<td>0x11111110c</td>
</tr>
<tr>
<td>0x0012ff30</td>
<td>0x00353050</td>
<td>0x11111110b</td>
<td>0x11111110b</td>
</tr>
<tr>
<td>0x0012ff2c</td>
<td>0x00401528</td>
<td>0x11111110a</td>
<td>0xfeeb2ecd</td>
</tr>
<tr>
<td>0x0012ff28</td>
<td>0x0012ff4c</td>
<td>0x111111109</td>
<td>0x70000000</td>
</tr>
<tr>
<td>0x0012ff24</td>
<td>0x00000000</td>
<td>0x111111108</td>
<td>0x70000000</td>
</tr>
<tr>
<td>0x0012ff20</td>
<td>0x00000000</td>
<td>0x111111107</td>
<td>0x00000040</td>
</tr>
<tr>
<td>0x0012ff1c</td>
<td>0x00000000</td>
<td>0x111111106</td>
<td>0x000003000</td>
</tr>
<tr>
<td>0x0012ff18</td>
<td>0x00000000</td>
<td>0x111111105</td>
<td>0x0001000</td>
</tr>
<tr>
<td>0x0012ff14</td>
<td>0x00000000</td>
<td>0x111111104</td>
<td>0x70000000</td>
</tr>
<tr>
<td>0x0012ff10</td>
<td>0x00000000</td>
<td>0x111111103</td>
<td>0x7c80978e</td>
</tr>
<tr>
<td>0x0012ff0c</td>
<td>0x00000000</td>
<td>0x111111102</td>
<td>0x7c809a51</td>
</tr>
<tr>
<td>0x0012ff08</td>
<td>0x00000000</td>
<td>0x111111101</td>
<td>0x111111101</td>
</tr>
<tr>
<td>0x0012ff04</td>
<td>0x00000004</td>
<td>0x00000040</td>
<td>0x00000040</td>
</tr>
<tr>
<td>0x0012ff00</td>
<td>0x00353050</td>
<td>0x00353050</td>
<td>0x00353050</td>
</tr>
<tr>
<td>0x0012ffec</td>
<td>0x0012ff08</td>
<td>0x0012ff08</td>
<td>0x0012ff08</td>
</tr>
</tbody>
</table>

- `cmp` argument
- `len` argument
- `data` argument
- return address
- saved base pointer
- tmp final 4 bytes
- tmp continues
- tmp continues
- tmp continues
- tmp continues
- tmp continues
- tmp continues
- tmp continues
- tmp continues
- tmp continues
- tmp continues
- tmp buffer starts
- `memcpy` length argument
- `memcpy` source argument
- `memcpy` destination arg.
Unwinding the fake stack

malicious
overflow
contents

0x7c971649 ; cmp argument
0x1111110c ; len argument
0x1111110b ; data argument
0xfeeb2ecd ; return address
0x70000000 ; saved base pointer
0x70000000 ; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts

Code Memory

VirtualAlloc

return

InterlockedExchange

return

SP
Unwinding the fake stack

malicious
overflow
contents
0x7c971649 ; cmp argument
0x1111110c ; len argument
0x1111110b ; data argument
0xfeeb2ecd ; return address
0x70000000 ; saved base pointer
0x70000000 ; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts

Code Memory

IP

VirtualAlloc

return

InterlockedExchange

return

SP
Unwinding the fake stack

malicious
overflow
contents
0x7c971649 ; cmp argument
0x1111110c ; len argument
0x1111110b ; data argument
0xfeeb2ecd ; return address
0x70000000 ; saved base pointer
0x70000000 ; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x70000100 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts

Code Memory

VirtualAlloc

VirtualAlloc

InterlockedExchange

InterlockedExchange

return

return

return

return
Unwinding the fake stack

malicious
overflow
contents
0x7c971649 ; cmp argument
0x11111110c ; len argument
0x11111110b ; data argument
0xfeeb2ecd ; return address
0x70000000 ; saved base pointer
0x70000000 ; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x111111101 ; tmp buffer starts

Code Memory

VirtualAlloc

- return

- InterlockedExchange

- return
Unwinding the fake stack

malicious
overflow
contents

0x7c971649 ; cmp argument
0x1111110c ; len argument
0x1111110b ; data argument
0xfeeb2ecd ; return address
0x70000000 ; saved base pointer
0x70000000 ; tmp final 4 bytes
0x00000040 ; tmp continues
0x00003000 ; tmp continues
0x00001000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts
Overview

• Introduction

• Example attacks
  o Stack-based buffer overflow
  o Heap-based buffer overflow
  o Return-to-libc attacks
  o Data-only attacks

• Example defenses
  o Stack canaries
  o Non-executable data
  o Control-flow integrity
  o Layout randomization

• Other defenses

• Conclusion
Data-only attacks

- These attacks proceed by changing only data of the program under attack
- Depending on the program under attack, this can result in interesting exploits
- We discuss two examples:
  - The unix password attack
  - Overwriting the environment table
Unix password attack

- Old implementations of login program looked like this:

Stack

<table>
<thead>
<tr>
<th>Password check in login program:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Read loginname</td>
</tr>
<tr>
<td>2. Lookup hashed password</td>
</tr>
<tr>
<td>3. Read password</td>
</tr>
<tr>
<td>4. Check if</td>
</tr>
<tr>
<td>hashed password = hash (password)</td>
</tr>
</tbody>
</table>
Unix password attack

Password check in login program:
1. Read loginname
2. Lookup hashed password
3. Read password
4. Check if
   hashed password = hash (password)

ATTACK: type in a password of the form pw || hash(pw)
void run_command_with_argument( pairs* data, int offset, int value )
{
    // must have offset be a valid index into data
    char cmd[MAX_LEN];
    data[offset].argument = value;
    {
        char valuestring[MAX_LEN];
        itoa( value, valuestring, 10 );
        strcpy( cmd, getenv("SAFECOMMAND") );
        strcat( cmd, " " );
        strcat( cmd, valuestring );
    }
    data[offset].result = system( cmd );
}
Overview

• Introduction

• Example attacks
  o Stack-based buffer overflow
  o Heap-based buffer overflow
  o Return-to-libc attacks
  o Data-only attacks

• Example defenses
  o Stack canaries
  o Non-executable data
  o Control-flow integrity
  o Layout randomization

• Other defenses

• Conclusion
Stack canaries

• Basic idea
  o Insert a value right in a stack frame right before the stored base pointer/return address
  o Verify on return from a function that this value was not modified

• The inserted value is called a *canary*, after the coal mine canaries
Stack canaries

Stack

Return address f0
Saved Frame Ptr f0
Canary

f0:
  ...
  call f1
  ...

f1:
  buffer[]
  overflow()
  ...

IP
FP
SP
Stack based buffer overflow

Stack

- Return address f0
- Saved Frame Ptr f0
- Canary
- Arguments f1
- Return address f1
- Saved Frame Ptr f1
- Canary

f0:

... 
call f1 
...

f1:

buffer[]
overflow()
...

IP
FP
SP
Stack based buffer overflow

```
f0:
    ...
    call f1
    ...
```

```
f1:
    buffer[]
    overflow()
    ...
```

Stack:
- Return address f0
- Saved Frame Ptr f0
- Canary
- Arguments f1
  - Overwritten address
  - Canary
Overview

• Introduction

• Example attacks
  o Stack-based buffer overflow
  o Heap-based buffer overflow
  o Return-to-libc attacks
  o Data-only attacks

• Example defenses
  o Stack canaries
  o Non-executable data
  o Control-flow integrity
  o Layout randomization

• Other defenses

• Conclusion
Non-executable data

• Direct code injection attacks at some point execute data
• Most programs never need to do this
• Hence, a simple countermeasure is to mark data memory (stack, heap, ...) as non-executable
• This counters direct code injection, but not return-into-libc or data-only attacks
• In addition, this countermeasure may break certain legacy applications
Overview

• Introduction
• Example attacks
  o Stack-based buffer overflow
  o Heap-based buffer overflow
  o Return-to-libc attacks
  o Data-only attacks
• Example defenses
  o Stack canaries
  o Non-executable data
  o Control-flow integrity
  o Layout randomization
• Other defenses
• Conclusion
Control-flow integrity

• Most attacks we discussed break the control flow as it is encoded in the source program
  o E.g. At the source code level, one always expects a function to return to its call site

• The idea of control-flow integrity is to instrument the code to check the “sanity” of the control-flow at runtime
Example CFI at the source level

• The following code explicitly checks whether the cmp function pointer points to one of two known functions:

```c
int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // ... elided code ...
    if( (s->cmp == strcmp) || (s->cmp == stricmp) ) {
        return s->cmp( s->buff, "file://foobar" );
    } else {
        return report_memory_corruption_error();
    }
}
```
Example CFI with labels

```c
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
sort2(int a[], int b[], int len) {
    sort( a, len, lt );
    sort( b, len, gt );
}
```

Diagram:

- **lt()**: label 17 → ret 23
- **gt()**: label 17 → ret 23
- **sort2()**: call sort (label 55) → ret ...
- **sort()**: call 17,R (label 23) → ret 55
Overview

• Introduction

• Example attacks
  o Stack-based buffer overflow
  o Heap-based buffer overflow
  o Return-to-libc attacks
  o Data-only attacks

• Example defenses
  o Stack canaries
  o Non-executable data
  o Control-flow integrity
  o Layout randomization

• Other defenses

• Conclusion
Layout Randomization

• Most attacks rely on precise knowledge of run time memory addresses
• Introducing artificial variation in these addresses significantly raises the bar for attackers
• Such address space layout randomization (ASLR) is a cheap and effective countermeasure
### Example

<table>
<thead>
<tr>
<th>stack one address</th>
<th>contents</th>
<th>stack two address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0022feac</td>
<td>0x008a13e0</td>
<td>0x0013f750</td>
<td>0x00b113e0</td>
</tr>
<tr>
<td>0x0022fe8</td>
<td>0x00000001</td>
<td>0x0013f74c</td>
<td>0x00000001</td>
</tr>
<tr>
<td>0x0022fe4</td>
<td>0x00a91147</td>
<td>0x0013f748</td>
<td>0x00191147</td>
</tr>
<tr>
<td>0x0022fe0</td>
<td>0x008a1528</td>
<td>0x0013f744</td>
<td>0x00b11528</td>
</tr>
<tr>
<td>0x0022fe9c</td>
<td>0x0022fec8</td>
<td>0x0013f740</td>
<td>0x0013f76c</td>
</tr>
<tr>
<td>0x0022fe98</td>
<td>0x00000000</td>
<td>0x0013f73c</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0022fe94</td>
<td>0x00000000</td>
<td>0x0013f738</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0022fe90</td>
<td>0x00000000</td>
<td>0x0013f734</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0022fe8c</td>
<td>0x00000000</td>
<td>0x0013f730</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0022fe88</td>
<td>0x00000000</td>
<td>0x0013f72c</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0022fe84</td>
<td>0x00000000</td>
<td>0x0013f728</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0022fe80</td>
<td>0x00000000</td>
<td>0x0013f724</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0022fe7c</td>
<td>0x00000000</td>
<td>0x0013f720</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x0022fe78</td>
<td>0x0000000004</td>
<td>0x0013f71c</td>
<td>0x0000000004</td>
</tr>
<tr>
<td>0x0022fe74</td>
<td>0x00a91147</td>
<td>0x0013f718</td>
<td>0x00191147</td>
</tr>
<tr>
<td>0x0022fe70</td>
<td>0x0022fe8c</td>
<td>0x0013f714</td>
<td>0x0013f730</td>
</tr>
</tbody>
</table>

; cmp argument
; len argument
; data argument
; return address
; saved base pointer
; tmp final 4 bytes
; tmp continues
; tmp continues
; tmp continues
; tmp continues
; tmp continues
; tmp continues
; tmp continues
; tmp continues
; tmp continues
; tmp continues
; tmp buffer starts
; memcpy length argument
; memcpy source argument
; memcpy destination arg.
Overview

• Introduction

• Example attacks
  o Stack-based buffer overflow
  o Heap-based buffer overflow
  o Return-to-libc attacks
  o Data-only attacks

• Example defenses
  o Stack canaries
  o Non-executable data
  o Control-flow integrity
  o Layout randomization

• Other defenses

• Conclusion
## Overview

<table>
<thead>
<tr>
<th></th>
<th>Return address corruption (A1)</th>
<th>Heap function pointer corruption (A2)</th>
<th>Jump-to-libc (A3)</th>
<th>Non-control data (A4)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack Canary (D1)</strong></td>
<td>Partial defense</td>
<td></td>
<td>Partial defense</td>
<td>Partial defense</td>
</tr>
<tr>
<td><strong>Non-executable data (D2)</strong></td>
<td>Partial defense</td>
<td>Partial defense</td>
<td>Partial defense</td>
<td></td>
</tr>
<tr>
<td><strong>Control-flow integrity (D3)</strong></td>
<td>Partial defense</td>
<td>Partial defense</td>
<td>Partial defense</td>
<td></td>
</tr>
<tr>
<td><strong>Address space layout randomization (D4)</strong></td>
<td>Partial defense</td>
<td>Partial defense</td>
<td>Partial defense</td>
<td>Partial defense</td>
</tr>
</tbody>
</table>
Need for other defenses

• The “automatic” defenses discussed in this lecture are only one element of securing C software
• Instead of preventing / detecting exploitation of the vulnerabilities at run time, one can:
  o Prevent the introduction of vulnerabilities in the code
  o Detect and eliminate the vulnerabilities at development time
  o Detect and eliminate the vulnerabilities with testing
Preventing introduction

• Safe programming languages such as Java / C# take memory management out of the programmer’s hands
• This makes it impossible to introduce exploitable memory safety vulnerabilities
  o They can still be “exploited” for denial-of-service purposes
  o Exploitable vulnerabilities can still be present in native parts of the application
Detect and eliminate vulnerabilities

• Code review

• Static analysis tools:
  o Simple “grep”-like tools that detect unsafe functions
  o Advanced heuristic tools that have false positives and false negatives
  o Sound tools that require significant programmer effort to annotate the program

• Testing tools:
  o Fuzz testing
  o Directed fuzz-testing / symbolic execution
Overview

• Introduction

• Example attacks
  o Stack-based buffer overflow
  o Heap-based buffer overflow
  o Return-to-libc attacks
  o Data-only attacks

• Example defenses
  o Stack canaries
  o Non-executable data
  o Control-flow integrity
  o Layout randomization

• Other defenses

• Conclusion
Conclusion

• The design of attacks and countermeasures has led to an arms race between attackers and defenders

• While significant hardening of the execution of C-like languages is possible, the use of safe languages like Java / C# is from the point of view of security preferable