Low-level Software Security: Vulnerabilities, Attacks and Countermeasures

Prof. Frank PIESSENS

Reading material:
• Frank Piessens, Ingrid Verbauwhede, *Software security: Vulnerabilities and countermeasures for two attacker models*. DATE 2016: 990-999

SECAPPDEV 2017
Introduction

• A significant fraction of software attacks are “layer below” attacks
  o The attack essentially relies on details of the execution infrastructure of the program (hardware / operating system / compiler / ...)
  o The most common examples are attacks against software in C-like languages (so-called unsafe languages)
• The purpose of this lecture is to explain vulnerabilities and countermeasures relating to these attacks
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");
}
```
#include <stdio.h>

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

Example vulnerable C program
Overview

• Understanding execution of C programs
• Memory safety vulnerabilities
• The attacker-defender race
  o Attack 1: Stack-based buffer overflow
  o Defense 1: Stack canaries
  o Attack 2: Heap-based buffer overflow
  o Defense 2: Non-executable data
  o Attack 3: Return-to-libc attacks
  o Defense 3: Layout randomization
• Other defenses
• Conclusion
Compilation

- C code is compiled to machine code
- Each function can be compiled separately
- The control flow through the program is tracked by means of the call-stack
- Variables used in the program are allocated in a number of ways:
  - On the call-stack for local variables
  - Statically for global variables
  - Using a memory management library for dynamically allocated variables (malloc / new)
## Process memory layout

<table>
<thead>
<tr>
<th>High addresses</th>
<th>Arguments/ Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stack</td>
</tr>
<tr>
<td></td>
<td>Unused and Mapped Memory</td>
</tr>
<tr>
<td></td>
<td>Heap (dynamic data)</td>
</tr>
<tr>
<td></td>
<td>Static Data</td>
</tr>
<tr>
<td></td>
<td>Program Code</td>
</tr>
</tbody>
</table>

- Stack grows down
- Heap grows up
Example

```c
int s = 12;

main()
{
    int l = 13;
    int *d = malloc(100);
    printf("Address of l : %8x\n", &l);
    printf("Address of s : %8x\n", &s);
    printf("Address of d : %8x\n", d);
    printf("Address of main: %8x\n", &main);
    f(); g();
}
```

Output:
Address of l : b80008cc
Address of d : 1993010
Address of s : 601028
Address of main: 400544

• Note: This is OS/compiler dependent
The call-stack (or stack)

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

Stack

- Return address f0
- Saved Frame Ptr f0
- Local variables f0

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

```
f1:
    ...
```
The call-stack (or stack)

Stack

- Return address f0
- Saved Frame Ptr f0
- Local variables f0

Arguments f1
- Return address f1
- Saved Frame Ptr f1
- Local variables f1

f0:
...
call f1
...

f1:
...

IP
FP
SP
The call-stack (or stack)

```
Stack

Return address f0
Saved Frame Ptr f0
Local variables f0
Arguments f1
Return address f1
Saved Frame Ptr f1
Local variables f1

f0:
  ...
call f1
  ...

f1:
  ...
```

IP -> FP -> SP
The call-stack (or stack)

```
foo:
    ...
    call bar
    ...
```

```
bar:
    ...
```

Stack

- Return address f0
- Saved Frame Ptr f0
- Local variables f0
Mapping registers to memory

- Intel processors are *little-endian*

```
<table>
<thead>
<tr>
<th>Register</th>
<th>Memory</th>
<th>Register</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A0B0C0D</td>
<td>0D</td>
<td>0A0B0C0D</td>
<td>0A</td>
</tr>
<tr>
<td></td>
<td>0C</td>
<td></td>
<td>0B</td>
</tr>
<tr>
<td></td>
<td>0B</td>
<td></td>
<td>0C</td>
</tr>
<tr>
<td></td>
<td>0A</td>
<td></td>
<td>0D</td>
</tr>
</tbody>
</table>

Little-endian

```

```
<table>
<thead>
<tr>
<th>0x1010</th>
<th>0x13</th>
<th>0x12</th>
<th>0x11</th>
<th>0x10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100C</td>
<td>0x0f</td>
<td>0x0e</td>
<td>0x0d</td>
<td>0x0c</td>
</tr>
<tr>
<td>0x1008</td>
<td>0x0b</td>
<td>0x0a</td>
<td>0x09</td>
<td>0x08</td>
</tr>
<tr>
<td>0x1004</td>
<td>0x07</td>
<td>0x06</td>
<td>0x05</td>
<td>0x04</td>
</tr>
<tr>
<td>0x1000</td>
<td>0x03</td>
<td>0x02</td>
<td>0x01</td>
<td>0x00</td>
</tr>
</tbody>
</table>

Big-endian

```

0x1003 0x03
0x1002 0x02
0x1001 0x01
0x1000 0x00
Putting it all together …

(a) Program source code

```c
void get_request(int fd, char buf[]) {
    read(fd,buf,16);
}

void process(int fd) {
    char buf[16];
    get_request(fd,buf);
    // Process the request (code not shown)
}

void main() {
    int fd;
    // Initialize server, wait for a connection
    // Accept connection, with file descriptor fd
    // Finally, process the request:
    process(fd);
}
```

(b) Machine code for process() function

```
55 push %ebp ; save base pointer
89 e5 mov %esp,%ebp ; set new base pointer
83 ec 18 sub $0x18,%esp ; allocate stack record
8d 45 f0 lea -0x10(%ebp),%eax; put buf in %eax
89 44 24 04 mov %eax,0x4(%esp) ; put fd parameter in %eax
89 04 24 mov %eax,(%esp) ; put fd parameter in %eax
89 e3 ff ff call 0x80483ed ; call get_request
C9 leave ; deallocate stack frame
C3 ret ; return
```

(c) Run-time machine state on entering get_request()
Overview

- Understanding execution of C programs
- Memory safety vulnerabilities
- The attacker-defender race
  - Attack 1: Stack-based buffer overflow
  - Defense 1: Stack canaries
  - Attack 2: Heap-based buffer overflow
  - Defense 2: Non-executable data
  - Attack 3: Return-to-libc attacks
  - Defense 3: Layout randomization
- Other defenses
- Conclusion
Memory safety vulnerabilities

• Memory safety 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 safety vulnerabilities in C programs
Memory safety vulnerabilities

- **Spatial** safety errors:
  - Index an array out of bounds
  - Invalid pointer arithmetic

- Accessing uninitialized memory

- **Temporal** safety errors
  - Use-after-free
  - Double free

- Unsafe libc API functions
  - Format string vulnerabilities
Memory safety vulnerabilities

• Manual memory management is very error-prone
  o Hence memory safety vulnerabilities are common in C
• But what happens on triggering such a vulnerability?
  o For efficiency, practical C implementations do not detect such errors at run time
    • The language definition states that behavior of a buggy program is *undefined*
  o So what happens depends on the compiler / operating system / processor architecture / …
  o The trick of exploiting these vulnerabilities is to use knowledge of these lower layers to make the program do what you want
Overview

• Understanding execution of C programs
• Memory safety vulnerabilities
• The attacker-defender race
  o Attack 1: Stack-based buffer overflow
  o Defense 1: Stack canaries
  o Attack 2: Heap-based buffer overflow
  o Defense 2: Non-executable data
  o Attack 3: Return-to-libc attacks
  o Defense 3: Layout randomization

• Other defenses
• Conclusion
Stack based buffer overflow

• Remember the purpose of the call-stack:
  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
Stack based buffer overflow

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

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

Stack Diagram:
- 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>machine code</th>
<th>opcode bytes</th>
<th>assembly-language version of the machine code</th>
</tr>
</thead>
<tbody>
<tr>
<td>char shellcode[ ] =</td>
<td>0xcd 0x2e</td>
<td>int 0x2e ; system call to the operating system</td>
</tr>
<tr>
<td></td>
<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:

  LINUX on Intel:
  ```
  char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\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";
  ```
Stack based buffer overflow

- Example vulnerable program:

```c
int is_file_foobar( 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>Note</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 \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>Description</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</td>
</tr>
<tr>
<td>0x0012ff48</td>
<td>0x61626f6f</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>

\r"'0' '0' '0' 'r' '0' 'o' 'b' 'a'

\r": '/' '/' '?' 'f'
Stack based buffer overflow

- Snapshot of the stack before the return:

<table>
<thead>
<tr>
<th>address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff5c</td>
<td>0x00353037</td>
</tr>
<tr>
<td>0x0012ff58</td>
<td>0x0035302f</td>
</tr>
<tr>
<td>0x0012ff54</td>
<td>0x0012ff4c</td>
</tr>
<tr>
<td>0x0012ff50</td>
<td>0x66666666</td>
</tr>
<tr>
<td>0x0012ff4c</td>
<td>0xfeeb2ecd</td>
</tr>
<tr>
<td>0x0012ff48</td>
<td>0x66666666</td>
</tr>
<tr>
<td>0x0012ff44</td>
<td>0x662f2f3a</td>
</tr>
<tr>
<td>0x0012ff40</td>
<td>0x656c6966</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 (Elias Levy)
Exploitation challenge (from the SYSSEC 10K challenge)

```c
char gWelcome[] = "Welcome to our system!";

void echo (int fd) {
    int len;
    char name[64], reply[128];

    len = strlen(gWelcome);
    memcpy(reply, gWelcome, len); /* copy the welcome string to reply */
    write_to_socket(fd, "Type your name:");
    read(fd, name, 128);

    /* copy the name into the reply buffer (starting at offset len so
     * that we do not overwrite the welcome message) */
    memcpy(reply+len, name, 64); write(fd, reply, len + 64);
    /* send full welcome message to client */
    return;
}

void server(int socketfd) {
    while(1) echo(socketfd);
}
```
Overview

• Understanding execution of C programs
• Memory safety vulnerabilities
• The attacker-defender race
  o Attack 1: Stack-based buffer overflow
  o Defense 1: Stack canaries
  o Attack 2: Heap-based buffer overflow
  o Defense 2: Non-executable data
  o Attack 3: Return-to-libc attacks
  o Defense 3: Layout randomization
• Other defenses
• Conclusion
Stack canaries

• Basic idea
  o Insert a value 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

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

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

Stack:
- Return address f0
- Saved Frame Ptr f0
- Canary

IP → FP → SP
Stack canaries

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

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

Stack

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

IP
FP
SP
Stack canaries

f0:
  ...
  call f1
  ...

f1:
  buffer[]
  overflow()
  ...

Stack

Return address f0
Saved Frame Ptr f0
Canary
Arguments f1
Overwritten address
Canary
Exploitation challenge (from the SYSSEC 10K challenge)

```c
char gWelcome[] = "Welcome to our system!";
void echo (int fd) {
    int len;
    char name[64], reply[128];

    len = strlen(gWelcome);
    memcpy(reply, gWelcome, len); /* copy the welcome string to reply */
    write_to_socket(fd, "Type your name: ");
    read(fd, name, 128);

    /* copy the name into the reply buffer (starting at offset len so * that we do not overwrite the welcome message) */
    memcpy(reply+len, name, 64); write(fd, reply, len+64);
    /* send full welcome message to client */
    return;
}

void server(int socketfd) {
    while(1) echo(socketfd);
}
```
Overview

- Understanding execution of C programs
- Memory safety vulnerabilities
- The attacker-defender race
  - Attack 1: Stack-based buffer overflow
  - Defense 1: Stack canaries
  - Attack 2: Heap-based buffer overflow
  - Defense 2: Non-executable data
  - Attack 3: Return-to-libc attacks
  - Defense 3: Layout randomization
- Other defenses
- Conclusion
Heap based buffer overflow

- Stack canaries only protect the stack, but there are also buffers on the heap.
- 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.
Overwriting a function pointer

- Example vulnerable program:

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

int is_filefoobar_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 “asdfasdfasdf”.

(c) After a malicious buffer overflow caused by attacker-chosen inputs.
Overview

• Understanding execution of C programs
• Memory safety vulnerabilities
• The attacker-defender race
  o Attack 1: Stack-based buffer overflow
  o Defense 1: Stack canaries
  o Attack 2: Heap-based buffer overflow
  o Defense 2: Non-executable data
  o Attack 3: Return-to-libc attacks
  o Defense 3: 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 this countermeasure may break certain legacy applications
- How would you break this?
Overview

• Understanding execution of C programs
• Memory safety vulnerabilities
• The attacker-defender race
  o Attack 1: Stack-based buffer overflow
  o Defense 1: Stack canaries
  o Attack 2: Heap-based buffer overflow
  o Defense 2: Non-executable data
  o Attack 3: Return-to-libc attacks
  o Defense 3: Layout randomization
• Other defenses
• Conclusion
Return-into-libc

• *Direct code injection*, where an attacker injects code as data is not always feasible
  o 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
  o Usually, interesting code is available, e.g. libc
Return-into-libc: overview

Stack

- Return addr
- 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

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

- 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
Return-into-libc: overview

Stack

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

Code Memory

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

SP

Return addr

IP
Return-into-libc: overview

Stack

- Params for f1
- Return addr

Code Memory

- IP
  - f1
    - return
  - f2
    - return
  - f3
  - return
  - 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:

... 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_lessthan ; 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 stack contents</th>
<th>benign stack overflow contents</th>
<th>malicious stack overflow contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0012ff38</td>
<td>0x004013e0</td>
<td>0x1111110d</td>
<td>0x7c971649; cmp argument</td>
</tr>
<tr>
<td>0x0012ff34</td>
<td>0x00000001</td>
<td>0x1111110c</td>
<td>0x1111110c; len argument</td>
</tr>
<tr>
<td>0x0012ff30</td>
<td>0x00353050</td>
<td>0x1111110b</td>
<td>0x1111110b; data argument</td>
</tr>
<tr>
<td>0x0012ff2c</td>
<td>0x00401528</td>
<td>0x1111110a</td>
<td>0xfeeb2ecd; return address</td>
</tr>
<tr>
<td>0x0012ff28</td>
<td>0x0012ff4c</td>
<td>0x11111109</td>
<td>0x70000000; saved base pointer</td>
</tr>
<tr>
<td>0x0012ff24</td>
<td>0x00000000</td>
<td>0x11111108</td>
<td>0x70000000; tmp final 4 bytes</td>
</tr>
<tr>
<td>0x0012ff20</td>
<td>0x00000000</td>
<td>0x11111107</td>
<td>0x00000040; tmp continues</td>
</tr>
<tr>
<td>0x0012ff1c</td>
<td>0x00000000</td>
<td>0x11111106</td>
<td>0x00003000; tmp continues</td>
</tr>
<tr>
<td>0x0012ff18</td>
<td>0x00000000</td>
<td>0x11111105</td>
<td>0x0001000; tmp continues</td>
</tr>
<tr>
<td>0x0012ff14</td>
<td>0x00000000</td>
<td>0x11111104</td>
<td>0x70000000; tmp continues</td>
</tr>
<tr>
<td>0x0012ff10</td>
<td>0x00000000</td>
<td>0x11111103</td>
<td>0x7c80978e; tmp continues</td>
</tr>
<tr>
<td>0x0012ff0c</td>
<td>0x00000000</td>
<td>0x11111102</td>
<td>0x7c809a51; tmp continues</td>
</tr>
<tr>
<td>0x0012ff08</td>
<td>0x00000000</td>
<td>0x11111101</td>
<td>0x11111101; tmp buffer starts</td>
</tr>
<tr>
<td>0x0012ff04</td>
<td>0x00000004</td>
<td>0x00000040</td>
<td>0x00000040; memcpy length argument</td>
</tr>
<tr>
<td>0x0012ff00</td>
<td>0x00353050</td>
<td>0x00353050</td>
<td>0x00353050; memcpy source argument</td>
</tr>
</tbody>
</table>
| 0x0012fefe    | 0x0012ff08            | 0x0012ff08                    | 0x0012ff08; 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
0x00010000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts

Code Memory

VirtualAlloc

IP

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

Code Memory

VirtualAlloc
.
return
.
.
.

InterlockedExchange
return
.
.
.

SP
IP
Unwinding the fake stack

malicious overflow contents

VirtualAlloc

Return

Code Memory

InterlockedExchange

Return

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
0x7c809f8e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts
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
0x00010000 ; tmp continues
0x70000000 ; tmp continues
0x7c80978e ; tmp continues
0x7c809a51 ; tmp continues
0x11111101 ; tmp buffer starts
Exploitation challenge (from the SYSSEC 10K challenge)

```c
char gWelcome[] = "Welcome to our system!";
void echo (int fd) {
    int len;
    char name[64], reply[128];

    len = strlen(gWelcome);
    memcpy(reply, gWelcome, len); /* copy the welcome string to reply */
    write_to_socket(fd, "Type your name: ");
    read(fd, name, 128);

    /* copy the name into the reply buffer (starting at offset len so
     * that we do not overwrite the welcome message) */
    memcpy(reply + len, name, 64); write(fd, reply, len + 64);
    /* send full welcome message to client */
    return;
}

void server(int socketfd) {
    while(1) echo(socketfd);
}
```
Overview

• Understanding execution of C programs
• Memory safety vulnerabilities
• The attacker-defender race
  o Attack 1: Stack-based buffer overflow
  o Defense 1: Stack canaries
  o Attack 2: Heap-based buffer overflow
  o Defense 2: Non-executable data
  o Attack 3: Return-to-libc attacks
  o Defense 3: 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</th>
<th>stack two</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>address</td>
</tr>
<tr>
<td>0x0022feac</td>
<td>0x0013f750</td>
</tr>
<tr>
<td>0x0022fe8a</td>
<td>0x0013f74c</td>
</tr>
<tr>
<td>0x0022fe4a</td>
<td>0x0013f748</td>
</tr>
<tr>
<td>0x0022fe0a</td>
<td>0x0013f744</td>
</tr>
<tr>
<td>0x0022fe9c</td>
<td>0x0013f740</td>
</tr>
<tr>
<td>0x0022fe98</td>
<td>0x0013f73c</td>
</tr>
<tr>
<td>0x0022fe94</td>
<td>0x0013f738</td>
</tr>
<tr>
<td>0x0022fe90</td>
<td>0x0013f734</td>
</tr>
<tr>
<td>0x0022fe8c</td>
<td>0x0013f730</td>
</tr>
<tr>
<td>0x0022fe88</td>
<td>0x0013f72c</td>
</tr>
<tr>
<td>0x0022fe84</td>
<td>0x0013f728</td>
</tr>
<tr>
<td>0x0022fe80</td>
<td>0x0013f724</td>
</tr>
<tr>
<td>0x0022fe7c</td>
<td>0x0013f720</td>
</tr>
<tr>
<td>0x0022fe78</td>
<td>0x0013f71c</td>
</tr>
<tr>
<td>0x0022fe74</td>
<td>0x0013f718</td>
</tr>
<tr>
<td>0x0022fe70</td>
<td>0x0013f714</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 buffer starts
; memcpy length argument
; memcpy source argument
; memcpy destination arg.
Exploitation challenge (from the SYSSEC 10K challenge)

```c
char gWelcome[] = "Welcome to our system!";

void echo (int fd) {

    int len;
    char name[64], reply[128];

    len = strlen(gWelcome);
    memcpy(reply, gWelcome, len); /* copy the welcome string to reply */

    write_to_socket(fd, "Type your name:");
    read(fd, name, 128);

    /* copy the name into the reply buffer (starting at offset len so
     * that we do not overwrite the welcome message) */

    memcpy(reply+len, name, 64); write(fd, reply, len + 64);
    /* send full welcome message to client */
    return;
}

void server(int socketfd) {
    while(1) echo(socketfd);
}
```
The attacker-defender race continues…

• Attack technique: Return-Oriented-Programming attacks
  • Generalization of return-2-libc, chaining “gadgets” instead of function calls into libc
• Defense technique: Control-Flow-Integrity (CFI)
  • Instrument the program to check that control flow at runtime follows the expected control-flow graph
  • Many variants have been proposed, several have been broken
• Attack technique: Data-only attacks
  • Influence the behavior of the program while only tampering with memory locations that contain program data
  • Recently shown to allow arbitrary attacks against a significant fraction of programs
• Should we give up on these “mitigate the exploit” countermeasures?
Overview

• Understanding execution of C programs
• Memory safety vulnerabilities
• The attacker-defender race
  o Attack 1: Stack-based buffer overflow
  o Defense 1: Stack canaries
  o Attack 2: Heap-based buffer overflow
  o Defense 2: Non-executable data
  o Attack 3: Return-to-libc attacks
  o Defense 3: Layout randomization

• Other defenses
• Conclusion
# Overview of automatic defenses

<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>Partial defense</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>Partial defense</td>
</tr>
<tr>
<td><strong>Control-flow integrity (D3)</strong></td>
<td>Partial defense</td>
<td>Partial defense</td>
<td>Partial defense</td>
<td>Partial defense</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.
  - They can still be “exploited” for denial-of-service purposes.
  - Exploitable vulnerabilities can still be present in native parts of the application.
  - There is a cost associated with using safe languages.
- There are currently interesting recent developments.
  - E.g. The Rust language from Mozilla, the Go language from Google.
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
    • Many variants: random, directed, model-based, ...
  o Run-time memory safety checkers
    • E.g. AddressSanitizer
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# / Rust / Go is from the point of view of security preferable