CSE509 System Security

Software Security

Nick Nikiforakis
nick@cs.stonybrook.edu
Things we are going to discuss

• Basic x86 assembly instructions
• Stack workings
• GDB syntax
• Overflows
  – Stack
  – Heap
• Shellcode
• Format string vulnerabilities
• Integer overflows
• Dangling pointers
• Countermeasures
• Evolution of attackers
X86 Processor

• Most common processor type in desktop/laptop/server environments

• X86 Instruction set
  – The CPU’s language
  – Operation <destination>, <source>

• Native programs (C, C++, etc.) are eventually compiled to x86 instructions

• 32bit, 64bit
X86 Registers

- X86 has several registers
  - Internal fast memory
  - Can be viewed as variables available to programs
X86: General Purpose Registers

- EAX: Accumulator
- ECX: Counter
- EDX: Data
- EBX: Base
- ESP: Stack Pointer
- EBP: Base Pointer
- ESI: Source Index
- EDI: Destination Index

Figure 1
Special Registers

- EIP: Instruction Pointer
- EFLAGS: Status Register

[Diagram of EFLAGS register with explanations for each bit:]
- ID — Identification Flag
- VIP — Virtual Interrupt Pending
- VIF — Virtual Interrupt Flag
- AC — Alignment Check
- VM — Virtual-8086 Mode
- RF — Resume Flag
- NT — Nested Task Flag
- IOPL — I/O Privilege Level
- IF — Interrupt Enable Flag
- TF — Trap Flag

Reserved (set to 0)
X86: Common Instructions

• Moving data from one “place” to another
  – mov eax, 0x1234
  – mov ebx, ecx
  – mov DWORD PTR[esp + 0x1c], 0x0

• Arithmetics
  – add esp, 0x1
  – sub ebp, 0x2
  – inc eax
  – ...

X86: Common Instructions

• Comparisons
  – cmp eax, ebx
  – cmp ecx, 0x45

• Branching
  – Jumps: jle, jeq, jne, jmp
    • jmp 0x80abc44
  – Function calls: call

• Misc:
  – lea eax, [ebp – 4]
GDB

- GDB: Gnu Project Debugger
- Start/Stop/Pause an executing program
- Investigate/Alter
  - Registers
  - Memory locations
  - Call functions
  - ...
GDB: Common commands

• **break**: Sets a breakpoint to pause the execution when a specific statement is reached
  – break main
  – break *0x80abc44

• **x**: Examines the contents of memory locations/registers
  – x/20wx $esp (print 20 4-byte memory locations and print them in hexadecimal notation)
  – x/s 0xbffffff321 (print whatever is at that address as a string)

• **disas foo**: disassemble function foo
GDB: Common commands

• `nexti`: execute next instruction after a breakpoint
• `stepi N`: execute N steps after a breakpoint
• `continue`: continue execution after hitting a breakpoint
• `info registers`: show the contents of all registers
• `info frame`: show interesting information about current stack frame
### Process Memory Layout

<table>
<thead>
<tr>
<th>Module</th>
<th>High Mem</th>
<th>Low Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv, env</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>heap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>text</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Argv/Env**: CLI args and environment
- **Stack**: generally grows downwards
- **Heap**: generally grows upwards
- **BSS**: uninitialized global data
- **Data**: initialized global data
- **Text**: read-only program code
Memory Layout Example

/* data segment: initialized global data */
int a[] = { 1, 2, 3, 4, 5 };
/* bss segment: uninitialized global data */
int b;

/* text segment: contains program code */
int main(int argc, char **argv) /* ptr to argv */
{
    /* stack: local variables */
    int *c;
    /* heap: dynamic allocation by new or malloc */
    c = (int *)malloc(5 * sizeof(int));
}
What is the Call Stack?

LIFO data structure: push/pop
- Stack grows downwards in memory.
- SP (esp) points to top of stack (lowest address)

What’s on the call stack?
- Function parameters
- Local variables
- Return values
- Return address
```c
b() {
    ...
}

a() {
    b();
}

main() {
    a();
}
```
Accessing the Stack

Pushing an item onto the stack.

1. Decrement SP by 4.
2. Copy 4 bytes of data to stack.

Example: `push 0x12`

Popping data from the stack.

1. Copy 4 bytes of data from the top of the stack.
2. Increment SP by 4.

Example: `pop eax`

Retrieve data without pop: `mov eax, esp`
What is a Stack Frame?

Block of stack data for one procedure call.

Frame pointer (FP) points to frame:

- Use offsets to find local variables.
- SP continually moves with push/pops.
- FP only moves on function call/return.
- Intel CPUs use ebp register for FP.
C Calling Convention

1. Push all params onto stack in reverse order.
   Parameter #N
   ...
   Parameter #2
   Parameter #1

2. Issues a call instruction.
   1. Pushes address of next instruction (the return address) onto stack.
   2. Modifies IP (eip) to point to start of function.
Stack before Function Executes

<table>
<thead>
<tr>
<th>old stack frame</th>
<th>Frame Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter #N</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>parameter #1</td>
<td></td>
</tr>
<tr>
<td>return address</td>
<td>Stack Pointer</td>
</tr>
</tbody>
</table>
1. Function pushes FP (ebp) onto stack.
   Save FP for previous function.
   push ebp

2. Copies SP to FP.
   Allows function to access params as fixed indexes from base pointer.
   mov ebp,esp

3. Reserves stack space for local vars.
   subl esp, 0x12
**Stack at Function Start**

<table>
<thead>
<tr>
<th>Mem Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBP (Base Pointer)</td>
<td>old stack frame</td>
</tr>
<tr>
<td></td>
<td>parameter #N</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>parameter #1</td>
</tr>
<tr>
<td></td>
<td>return address</td>
</tr>
<tr>
<td></td>
<td>old FP</td>
</tr>
<tr>
<td></td>
<td>Space for local vars</td>
</tr>
<tr>
<td></td>
<td>Space for local vars</td>
</tr>
</tbody>
</table>

EBP (Base Pointer) and ESP (Stack Pointer) are pointers used to navigate the stack.
C Calling Convention

1. After execution, stores return value in $eax$.
   mov eax, 0x1
   Resets stack to pre-call state.
   Destroys current stack frame; restores caller’s frame.
   mov esp, ebp
   pop ebp

2. Returns control back to where called from.
   ret pops top word from stack and sets $eip$ to that value.
```c
#include <stdio.h>

int confirm(char *exp_answer) {
    char buf[80];
    gets(buf);
    int result = strcmp(buf, exp_answer);
    if (result == 0)
        return 1;
    else
        return 0;
}
```
Stack based buffer overflow

Stack

Return address f0
Saved Frame Ptr f0
Local variables f0

f0:
  ...
  call f1
  ...

f1:
  buffer[]
  overflow()
  ...

IP
FP
SP
Stack based buffer overflow

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

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

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return address f0</td>
</tr>
<tr>
<td>Saved Frame Ptr f0</td>
</tr>
<tr>
<td>Local variables f0</td>
</tr>
<tr>
<td>Arguments f1</td>
</tr>
<tr>
<td>Return address f1</td>
</tr>
<tr>
<td>Saved Frame Ptr f1</td>
</tr>
<tr>
<td>Space for buffer</td>
</tr>
</tbody>
</table>
Stack based buffer overflow

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

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

Stack Diagram:
- Return address f0
- Saved Frame Ptr f0
- Local variables f0
- Arguments f1
  - Overwritten address
  - Overwritten address
  - Injected Code

IP → SP → FP
Attacker-injected code

• Attacker-injected code was named “shellcode” because that’s what the typical attacker wanted... a shell

• However, the injected code can do anything that the attacker wants

• Example shellcode:

```
LINUX on Intel:
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\xe0\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\x8d\xc0\xff\xff\xff/bin/sh";
```
Attacker-injected code

• The shellcode is typically a series of assembly instructions that prepare the stack for a specific system call

• Calling a system call:
  – Place system call number into eax
  – Place arguments to system call into ebx, ecx, etc.
  – int 0x80

• List of system calls for a 32-bit system:
  /usr/include/i386-linux-gnu/asm/unistd_32.h
Response: Stack canaries

• Basic idea
  – Insert a value right in a stack frame right before the stored base pointer/return address
  – Verify on return from a function that this value was not modified
  – If the canary is modified, exit the program

• Why call it a canary?
Stack canaries

Stack

Return address f0
Canary
Saved Frame Ptr f0

IP
f0:
  ...
  call f1
  ...

f1:
  buffer[]
  overflow()
  ...

FP
SP
Stack based buffer overflow

f0:
  ...
  call f1
  ...

f1:
  buffer[]
  overflow()
  ...

Stack

Return address f0
Canary
Saved Frame Ptr f0
Arguments f1
Return address f1
Canary
Saved Frame Ptr f1
Stack based buffer overflow

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

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

Stack Diagram:
- Return address f0
- Saved Frame Ptr f0
- Canary
- Arguments f1
- Overwritten address
- Canary
0x08048494 <+0>:    push   ebp
0x08048495 <+1>:    mov    ebp,esp
0x08048497 <+3>:    sub    esp,0x88
0x0804849d <+9>:    mov    eax,DWORD PTR [ebp+0x8]
0x080484a0 <+12>:   mov    DWORD PTR [ebp-0x6c],eax
0x080484a3 <+15>:   mov    eax,gs:0x14
0x080484a9 <+21>:   mov    DWORD PTR [ebp-0xc],eax
0x080484ac <+24>:   xor    eax,eax

...
Things missing from Stackguard

• Attackers typically do not roll over and die when you defeat their strategy
• Instead they adapt...
• Attacks
  – Frame-pointer overwriting
  – Indirect pointer overwrite
  – Data-only attacks
Frame-pointer overwriting

• Canary is protecting the return address, so we can’t overflow into that

• But what about the saved-frame pointer?
Frame-pointer overwriting

- Canary is protecting the return address, so we can’t overflow into that

- But what about the saved-frame pointer?
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
Indirect-pointer overwriting

• If there are pointers present between the overflowed buffer and the canary, these can also be abused

• Pointers can be used to write memory (bypass canary)

• Read-out sensitive memory
**Data-only attacks**

- Non-control variables can also lead to exploitation.

- **Examples:**
  - Filenames
  - Filepaths
  - Passwords
  - Integers controlling sensitive operations
Unix password attack

• Old implementations of login program looked like this:

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashed password</td>
</tr>
<tr>
<td>password</td>
</tr>
</tbody>
</table>

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

- Just adding a canary before the return address isn’t enough

- ProPolice:
  - Add a canary before the saved frame pointer
  - Reorder local variables and place buffers right next to canaries
  - Copy sensitive function arguments below the buffers

- Present by default in *nix
Heaps of fun

• While the stack has, historically been, the number one playground for attackers, the heap is always an option
  – Corruption of memory acquired through malloc(), calloc(), realloc() operations

• Added benefit for attackers:
  – Significantly less researched
  – Few or zero countermeasures present in modern operating systems
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:
  – Overwriting a function pointer
  – Overwriting heap metadata
Overwriting a function pointer

• Example vulnerable program:

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 “asdfsdfasdfsdf”.

(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
  – 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
  – As a consequence, buffer overruns on the heap can overwrite this management information
  – 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 \(*\text{(forward pointer}+12)\).

Or: green value is written 12 bytes above where red value points.
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
Buffer over-reads
HOW THE HEARTBLEED BUG WORKS:

SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).

User Ada wants pages about "irl games". User Meg wants these 6 letters: POTATO. User Magie (chrome user) sends this message: "HI
Server, are you still there? If so, reply "bird" (4 letters).

User Olivia from London wants pages about "me bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 340 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2cob9ff89b13hff8).

Hmm...

BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User Amber requests pages.
int main(int argc, char* argv[])
{
    int a, b, total;
    a = b = total = 0;
    scanf("%d", &a);
    scanf("%d", &b);
    total = a + b;
    printf("Sum of %d and %d is %d\n", a, b);
    return 0;
}
Output of program

• Sum of 1 and 2 is -145141211
• Sum of 1 and 2 is -144592347
• Sum of 1 and 2 is -144518619

• What are these numbers?
  – My CS professor back when I was an undergrad called them “garbage”.
One man’s trash is another man’s treasure

• The garbage isn’t really garbage... it’s stack contents

• printf("...%d....%d...%d",a,b,)
  – Format string requires three arguments
  – Only two are given... the third one is just read of the stack. I.e. whatever is before these arguments

• This gives rise to a vulnerability called
  – Format-string vulnerability
int main(int argc, char *argv[])
{
    char buffer[120];
    char tmp[40];

    strcpy(buffer,"Welcome user: ");
    fgets(tmp,sizeof(tmp),stdin);
    strcat(buffer, tmp);

    printf(buffer);
    return 0;
}
Output of program

• Nick
  – Welcome user Nick

• Nick %d
  – Welcome user: Nick -6037020

• Nick %x %x %x %x
  – Welcome user: Nick ff9bd6b4 f7771ac0 8048230 ff9bd6a8
There is no application set to open the URL lets-try-format-string:
0x0 0x0 0x7f893b619340
0xffff8076c49c795f 0x7f893b814290 0x7f893b622ff1
0x7f893b6386a0 0x7f893b40bc40 0x0
0x7f893b53dd10 0x0 0x7f893b608cf0
0x7f893b53e1d0 0xffffd5c200000000 0x7ff78a444d0
0x7f893b53e280 0x7f893b62eb0
0x7035322570353225 0x7f893b6258f0
0x7f893b6258f0 0x7f893b6386a0 0x20
0x7ff58a68560 0x7f893b534ff0 0x7ff8e8e1080
0x7ff58a673c0 0x107197b88 0x64
0x0 0x23a 0x7f893b418410
0x7f893b625f10 0x1 0x7ff58a67430
0x7ff92bb0f78 0x7f893b7064a0 0x120680053c6ea
0x7f893b626f60 0x0 0x7ff00000001
0x7f893b534ff0 0x0 0x7ff58a68548
0x7f893b706470 0x7ff58a68560.

Search the App Store for an application that can open this document, or choose an existing application on your computer.

Choose Application... Cancel Search App Store
Integer overflows

```c
int i;
//Will we ever exit this loop?
for(i=1; i > 0;i++){
}
printf(“i = %d\n”, i);
```
# C Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size (bits)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>short int</td>
<td>16</td>
<td>[-32,768; 32,767]</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>16</td>
<td>[0; 65,535]</td>
</tr>
<tr>
<td>unsigned int</td>
<td>16</td>
<td>[0; 4,294,967,295]</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>[-2,147,483,648; 2,147,483,647]</td>
</tr>
<tr>
<td>long int</td>
<td>32/64</td>
<td>(depends on the architecture)</td>
</tr>
<tr>
<td>signed char</td>
<td>8</td>
<td>[-128; 127]</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8</td>
<td>[0; 255]</td>
</tr>
</tbody>
</table>
Where Does an Integer Overflow Matter?

- Allocating spaces using calculation.
- Calculating indexes into arrays
- Checking whether an overflow could occur
I had enough of you and your overflows...
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
  – Write-XOR-Execute, DEP

• This counters direct code injection
  – In principle, this countermeasure may also break certain legacy applications
Randomization of Memory Address Space

If you don’t know where a buffer/function is, then your exploit is unreliable

ASLR (Address Space Layout Randomization)
  – Stack location.
  – Shared library locations.
  – Heap location.

PIE: Position Independent Executable
  – Default format: binary compiled to work at an address selected when program was compiled.
  – Gcc can compile binaries to be freely relocatable throughout address space.
    • gcc flags: -fpie -pie
    • Program loaded at different address for each invocation.
Ways around ASLR

• De-randomize the stack/heap using a buffer over-read or format-string vulnerability
• Heap Spraying many instances of shellcode
• Use existing instructions in non-randomized memory segments instead of providing your own shellcode
Reaction: No code injection necessary

• Instead of injecting malicious code, why not assemble malicious code out of existing code already present in the program
  – *Indirect code injection* attacks will drive the execution of the program by manipulating the stack

• E.g. Just execute `system("/bin/bash")` instead of creating your own interrupts
  – You just need to find where the system function is and call it with the right parameter
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
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

SP -> Code Memory

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

Memory

SP

IP
Return-into-libc: overview

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

Code Memory

- \texttt{f1}
  \hspace{0.5cm} return
- \texttt{f2}
  \hspace{0.5cm} return
- \texttt{f3}
  \hspace{0.5cm} return

- SP
- IP
Return-into-libc: overview

Stack

- Params for f1
- Return addr

Code Memory

- f1
  - return
- f2
  - return
- f3
  - return
  - return
Return-to-libc

• What do we need to make this work?
  – Inject the fake stack
    • Easy: this is just data we can put in a buffer
  – Make the stack pointer point to the fake stack right before a return instruction is executed
  – 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
return-to-libc on Steroids

• Overwritten saved EIP need not point to the beginning of a library routine
  • Any existing instruction in the code image is fine
    – Will execute the sequence starting from this instruction

• What if instruction sequence contains RET?
  – Execution will be transferred... to where?
  – Read the word pointed to by stack pointer (ESP)
    • Guess what? Its value is under attacker’s control! (why?)
  – Use it as the new value for EIP
    • Now control is transferred to an address of attacker’s choice!
  – Increment ESP to point to the next word on the stack
Chaining RETs for Fun and Profit

- Can chain together sequences ending in RET
  - Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique” (2005)

- What is this good for?

- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
  - Attack can perform arbitrary computation using no injected code at all – return-oriented programming
## Return Oriented Programming

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80abdea0</td>
<td>EAX = SMTH</td>
</tr>
<tr>
<td>0x309</td>
<td>EBX = SMTH</td>
</tr>
<tr>
<td>0x80345677</td>
<td>ECX = SMTH</td>
</tr>
<tr>
<td>0x80abddaa</td>
<td></td>
</tr>
<tr>
<td>0x80abddab</td>
<td></td>
</tr>
<tr>
<td>0x80abdeaa0</td>
<td></td>
</tr>
<tr>
<td>0x80abcdee</td>
<td></td>
</tr>
</tbody>
</table>

```
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...
0x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdeaa0: int 0x80;
...```
Return Oriented Programming

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x80abdea0</td>
<td>0x80345677</td>
</tr>
<tr>
<td>0x309</td>
<td>&amp;&quot;/tmp/lala&quot;</td>
</tr>
<tr>
<td>0x80345677</td>
<td>0x80abddaa</td>
</tr>
<tr>
<td>8</td>
<td>0x80abcdee</td>
</tr>
</tbody>
</table>

EAX = SMTH
EBX = SMTH
ECX = SMTH

0x80abdea0: int 0x80;
0x80abddaa: pop $ebx;
0x80abddab: ret;

0x80abdea0: int 0x80;
Return Oriented Programming

\[ EAX = 8 \]
\[ EBX = SMTH \]
\[ ECX = SMTH \]

<table>
<thead>
<tr>
<th>ESP</th>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x80abdea0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x309</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x80345677</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/tmp/lala</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x80abddaa</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0x80abcdee</td>
</tr>
</tbody>
</table>

\[ 0x80345677: \text{pop } \$ecx; \]
\[ 0x80345678: \text{ret}; \]

\[ 0x08abcdee: \text{pop } \$eax; \]
\[ 0x08abcdef: \text{ret}; \]

\[ 0x80abddaa: \text{pop } \$ebx; \]
\[ 0x80abddab: \text{ret}; \]

\[ 0x80abdea0: \text{int } 0x80; \]
\[ \ldots \]
Return Oriented Programming

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x80abdea0</td>
<td></td>
</tr>
<tr>
<td>0x309</td>
<td></td>
</tr>
<tr>
<td>0x80345677</td>
<td></td>
</tr>
<tr>
<td>&amp;”/tmp/lala”</td>
<td></td>
</tr>
<tr>
<td>0x80abddaa</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>0x80abcdee</td>
<td></td>
</tr>
</tbody>
</table>

EAX = 8
EBX = SMTH
ECX = SMTH

0x80abdea0: int 0x80;
0x80abddaa: pop $ebx;
0x80abcdab: ret;
0x80abddda: pop $ebx;
0x80abcdee: pop $eax;
0x80abcdef : ret;
0x80345678: ret;
0x80345677: pop $ecx;
...
Return Oriented Programming

EAX = 8
EBX = "/tmp..."
ECX = SMTH

0x80abdea0
0x309
0x80345677
'/tmp/lala"
0x80abddaa
8
0x80abcdee

...
## Return Oriented Programming

<table>
<thead>
<tr>
<th>ESP</th>
<th>High</th>
<th>EIP</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80abdea0</td>
<td></td>
<td>0x80345677: pop $ecx;</td>
<td></td>
</tr>
<tr>
<td>0x309</td>
<td></td>
<td>0x80345678: ret;</td>
<td></td>
</tr>
<tr>
<td>0x80345677</td>
<td></td>
<td></td>
<td>0x80abddaa: pop $ebx;</td>
</tr>
<tr>
<td>&amp;”/tmp/lala”</td>
<td></td>
<td></td>
<td>0x80abddab: ret;</td>
</tr>
<tr>
<td>0x80abddaa</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80abcdee</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAX = 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBX = &amp;”/tmp...”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECX = SMTH</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Return Oriented Programming

```
EAX = 8
EBX = &"/tmp..."
ECX = 0x309

0x80abdea0
0x309
0x80345677
&"/tmp/lala"
0x80abddaa
8
0x80abcdee
```

```
0x80345677: pop $ecx; 0x80345678: ret;
...
0x08abcdee: pop $eax; 0x08abcdef : ret;
...
0x80abddaa: pop $ebx; 0x80abddab: ret;
...
0x80abdea0: int 0x80;
...
# Return Oriented Programming

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EIP</td>
</tr>
<tr>
<td></td>
<td>EAX = 8</td>
</tr>
<tr>
<td></td>
<td>EBX = &amp;&quot;/tmp...&quot;</td>
</tr>
<tr>
<td></td>
<td>ECX = 0x309</td>
</tr>
<tr>
<td>0x80abdea0</td>
<td>0x309</td>
</tr>
<tr>
<td>0x80345677</td>
<td>&amp;&quot;/tmp/lala&quot;</td>
</tr>
<tr>
<td>0x80abddaa</td>
<td>8</td>
</tr>
<tr>
<td>0x80abcdee</td>
<td>0x80abdeaa0</td>
</tr>
</tbody>
</table>

... 0x80345677: pop $ecx; 0x80345678: ret; ...
... 0x08abcdee: pop $eax; 0x08abcdef : ret; ...
... 0x80abddaa: pop $ebx; 0x80abddab: ret; ...
... 0x80abdeaa0: int 0x80; ...

ESP
Dangling pointers

• A dangling pointer is a pointer that does not point to a valid object of the appropriate type

• Dangling pointers typically occur when memory is freed but not all pointers that used to point to that memory, are invalidated
int dangling()
{
    char *p1;
    char c;
    ...
    p1 = malloc(42);
    ...
    free(p1);
    ...
    if(...){
       ;c = *p1;
    }
}

Dangling Pointer, Stack example

• This is the same reason why you should never return local memory from the stack
  • The variable will stay intact for a while, but as soon as another function is called, that memory location will be overwritten with new data

```c
int *func(void)
{
    int num = 1234;
    /* ... */
    return &num;
}
```
Dangling pointers

• When we use memory that we have freed, we have a use-after-free vulnerability

• These vulnerabilities got a lot of attention in the recent years because of many exploits targeting IE abusing use-after-free errors

CVE-2012-4969 (IE CMshtmlEd UAF)

Image credit: https://securityintelligence.com/use-after-frees-that-pointer-may-be-pointing-to-something-bad/
Solutions for dangling pointers

• Some of the obvious cases can be fixed by immediately setting a freed pointer to NULL

```c
int dangling()
{
    char *p1 = malloc(42);
    ...
    free(p1);
    p1 = NULL;
}

safe_free(void **p)
{
    if (p != NULL)
    {
        free(*p)
        *p = NULL
    }
}

safe_free(&p1);
```
Limitations

• This NULL-ing of pointers will not help if you have created multiple copies of that pointer, or if some pointers point “in” the objects

```c
int dangling()
{
    char   *p1 = malloc(42);
    char   *p2 = p1;

    ...

    free(p1);

    p1 = NULL;
}
```
Recap

• Assembly commands, function prologue and function epilogue
• Workings of the stack
• Buffer overflows (stack, heap, int)
• Format string vulnerabilities
• Countermeasures (Randomization, Non-exec memory, canaries)