Mechanics for Memory Corruption

CSP009_S_E

PRESENTATION BY:
ELIAS ATHANASOPOULOS

Elias Athanasopoulos

Profile of Trainer

- Associate Professor, University of Cyprus
- Research in systems security and privacy
- https://elathan.github.io/srec/

CyberSecPro

Topic-1: Introduction to Software Vulnerabilities

We will cover these skills

- How software is developed?
- What is the difference between memory-safe and memory -unsafe programming systems?
- How memory-safe vulnerabilities look like?

CyberSecPro

Topic-2: Exploiting Vulnerabilities in Native Software

We will cover these skills

- The mechanics of a control-flow attack
- How can someone introduce new code in a vulnerable program
- How modern software exploitation works

Training Practical Exercises

Practical exercises requiring both personal and team effort

	Title	Goal of Exercise
Exercise-1 (Day-1)	Control-flow Attack	Hijack the control flow of a vulnerable program
Exercise-2 (Day-1)	Code injection	Inject code in a vulnerable program
Exercise-3 (Day-1)	ROP attack	Compromise a vulnerable program

Evaluation method

Outline the evaluation elements and assessment process

Evaluation Element	How	Notes
Applied Tasks (Individual)	Problem solution to be submitted later	
Teamwork	Team project to submitted later	
Group discussion	During workshop	

Background Knowledge and Prerequisites

Background knowledge:

Basic understanding of C programming

Basic understanding of Operating Systems

Prerequisites:

None

Technical Tools and Other Requirement

Technical Tools

Linux-based Operating System with
a C compiler toolchain

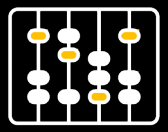
Registration: How to register and other practical information

The specific registration process for the Cybersecurity Essentials and Management training may vary depending on the training provider or institution. However, the general steps are typically straightforward and can be completed online or in person.

1. Online Registration
2. In-Person Registration
3. Additional Practical Information

Memory-corruption Attacks

Memory Safety

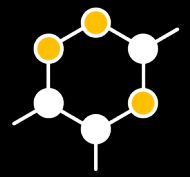


Memory-safe vs memory-unsafe programming systems

Control-flow Attacks

How memory-safety can be violated

Modern Attacks



How modern attacks work?

Course progress

01. Memory-safe Programming Systems

02. Vulnerabilities

03. Control-flow Attacks

04. Code Injection

05. Return-oriented Programming (ROP)

Software

Programs are written in a high-level language

- C, C++, Java, C#, Ruby, Python

They are compiled for execution

- Machine code (unmanaged and unsafe code)
- Virtual-machine code, such as JVM (managed and safe code)

Different architectures exhibit different properties in executing software

- Some generic concepts apply to all

Safe vs Unsafe Systems

Safe programming systems

- Execute managed code in a virtual machine (e.g., Java)
- Perform static analysis and compile time (reject unsafe code) and use run-time checks (e.g., Rust)
- Restricted memory access

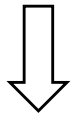
Unsafe programming systems

- Unrestricted memory access (e.g., C/C++)
- Better performance
- Low-level accessing (e.g., drivers)
- Legacy code

Unsafe vs Unsafe Programming Systems

Unsafe (e.g., C/C++)

```
a[i] = j;
```



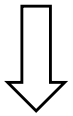
Machine Code

`a` is just a starting address

No way to check if `i` is in the bounds of `a`

Safe (e.g., Java)

```
a[i] = j;
```



Virtual Machine Code

`a` is a well-defined entity

VM knows `a`'s type, bounds, reference counts, etc.

Course progress

01. Memory-safe Programming Systems
02. Vulnerabilities
03. Control-flow Attacks
04. Code Injection
05. Return-oriented Programming (ROP)

Example of a C program

```
#include <stdio.h>

int main(int argc, char *argv[]) {

        int ary[5] = {1, 2, 3, 4, 5};

        fprintf(stderr, "The fifth number of ary is: %d\n", ary[5]);

        return 1;
}
```

Out of bounds access!

```
#include <stdio.h>

int main(int argc, char *argv[]) {

        int ary[5] = {1, 2, 3, 4, 5};

        fprintf(stderr, "The fifth number of ary is: %d\n", ary[5]);

        return 1;
}
```

Vocabulary

Vulnerability

- A software error (also known as bug) that can potentially allow someone to take advantage of the vulnerable program

Exploit

- The process of controlling a program by taking advantage of one or more vulnerabilities
- Not all vulnerabilities can be exploited

Vocabulary

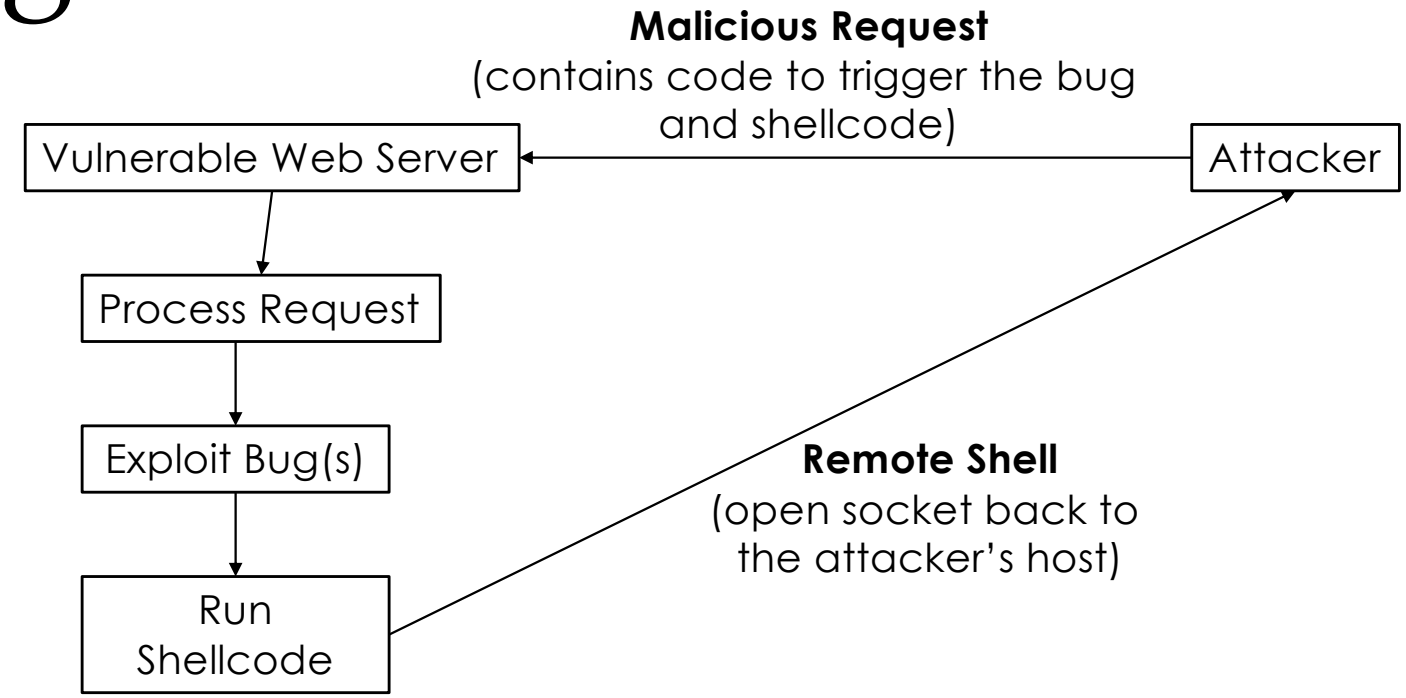
Arbitrary Code Execution

- The state of an exploit where an attacker can execute a program of their choice

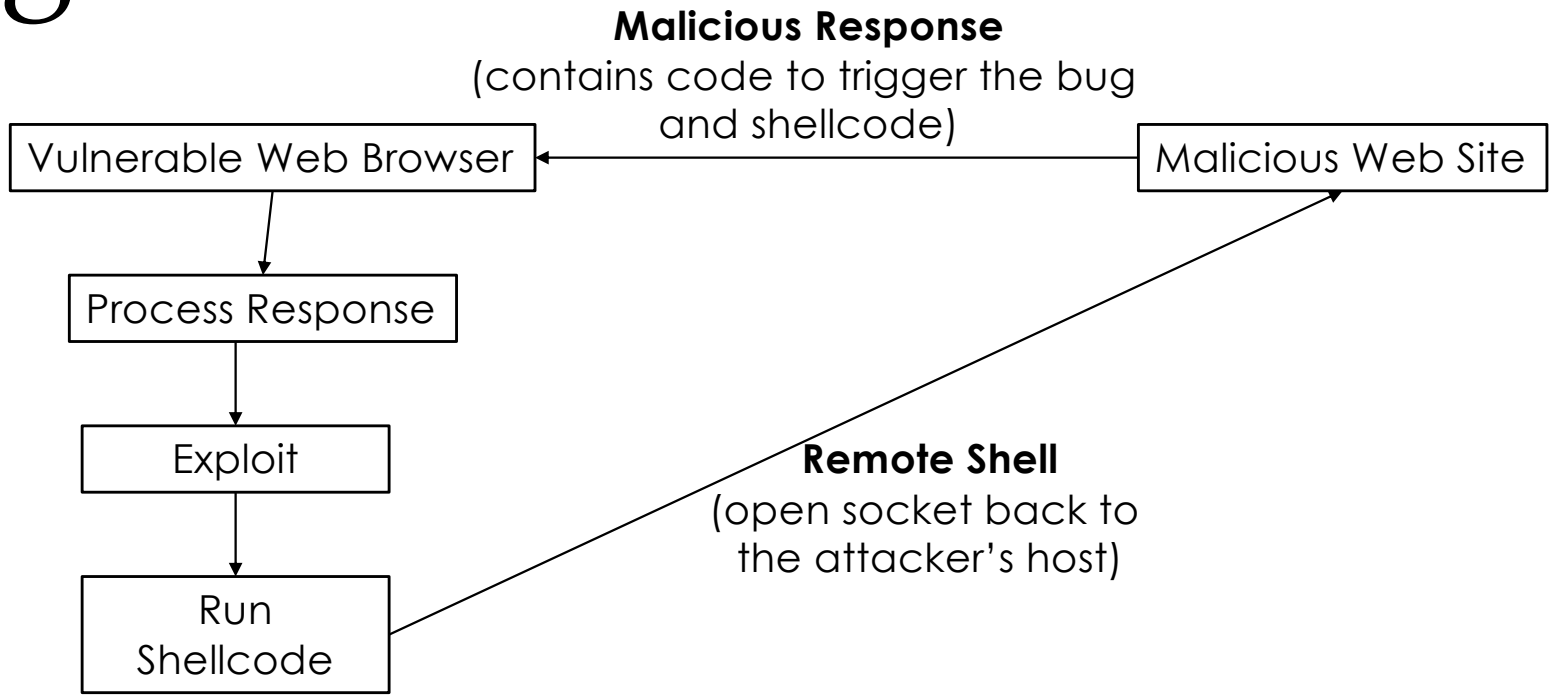
Shellcode

- A machine code that a vulnerable program executes and serves the purposes of the attacker
- Spawn a shell (can be remote), download malware, create a hidden account, manipulate software, etc.
- Heavily architecture dependent

High-level Idea



High-level Idea



Course progress

01. Memory-safe Programming Systems
02. Vulnerabilities
03. Control-flow Attacks
04. Code Injection
05. Return-oriented Programming (ROP)

Functions

Software is composed by several functions

- `main()`, `printf()`, `malloc()`, `create_user()`, etc.

Functions allow code re-use

- Whenever you want to display a message you simply call `printf`

In a program a function can call a function, and then another function

- All this function chaining is called **control flow**

The life of a function

Whenever a function is called, the control flow of the program is changed

- We need to do this transparently
- Once the function is finished the control flow should be resumed

Functions may take arguments

Functions may return data

Functions may create local data

Vocabulary

When a function `foo` is called

- `foo` is the callee
- The address that called the function is called **call site** (or caller)

The stack

Functions need memory for their work

- This is the stack

This memory is for short lived data

- Once the function is finished we can get rid of the data involved

Architecture dependent

- The main idea does not change

The stack may hold several things

- Function arguments, the return address, the old frame pointer, local arguments

Stack of Intel (32-bit)

The stack grows from higher-memory addresses to lower-memory addresses
- It is like the stack is flipped upside down

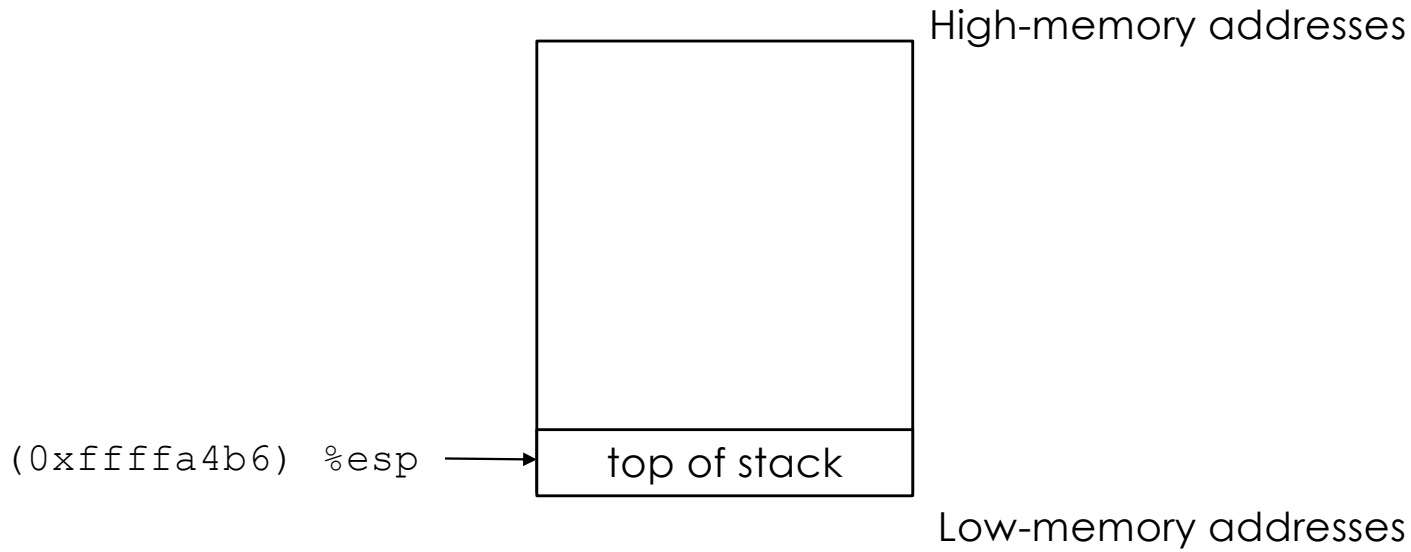
The top of the stack is always kept in a hardware register (`%esp`)

Each function creates a new **stack frame** upon executing
- A virtual portion inside the stack
- The stack frame is destroyed once the functions is finished

The top of the stack frame is kept in a hardware register (`%ebp`)

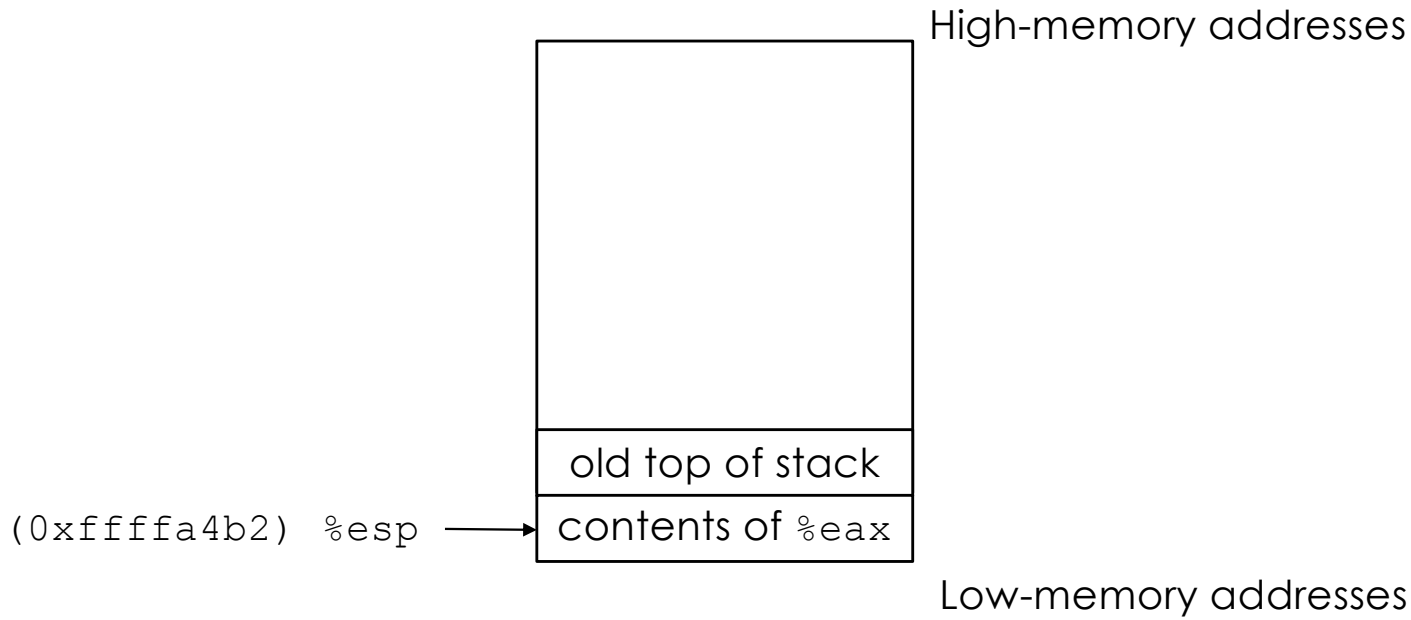
Stack insertion

```
push %eax
   sub 0x4, %esp
   mov %eax, (%esp)
```



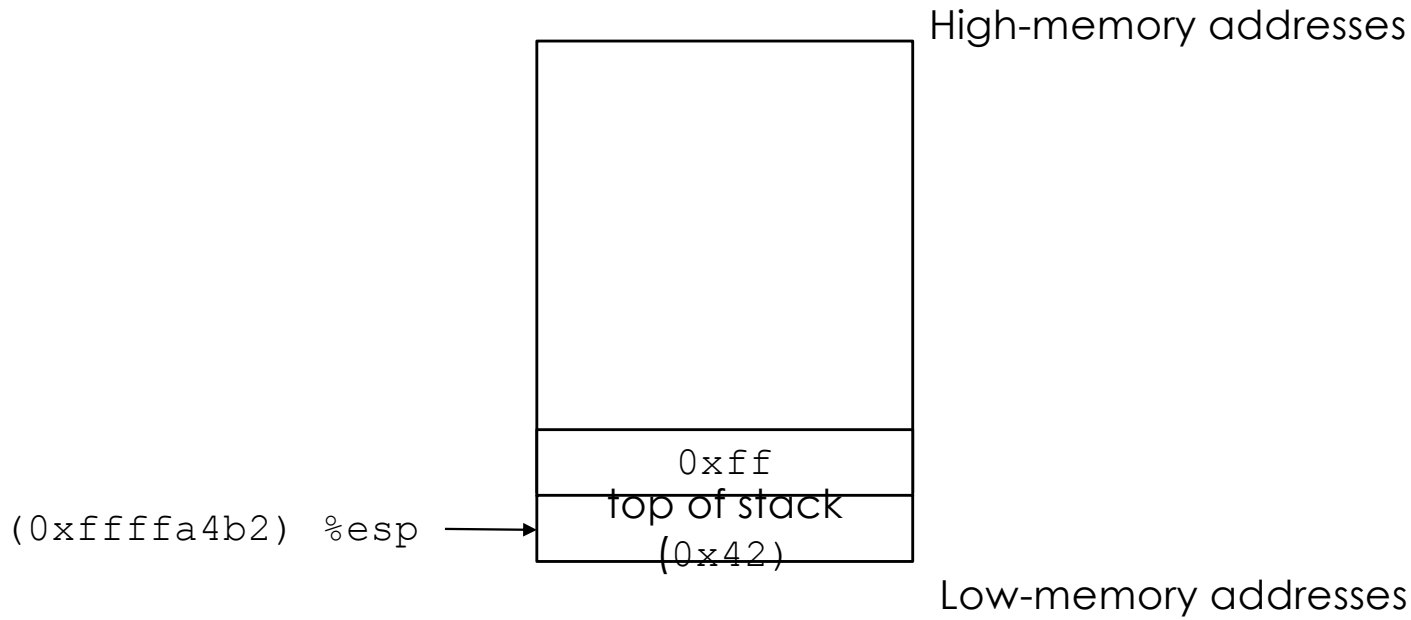
Stack insertion

```
push %eax
   sub 0x4, %esp
   mov %eax, (%esp)
```



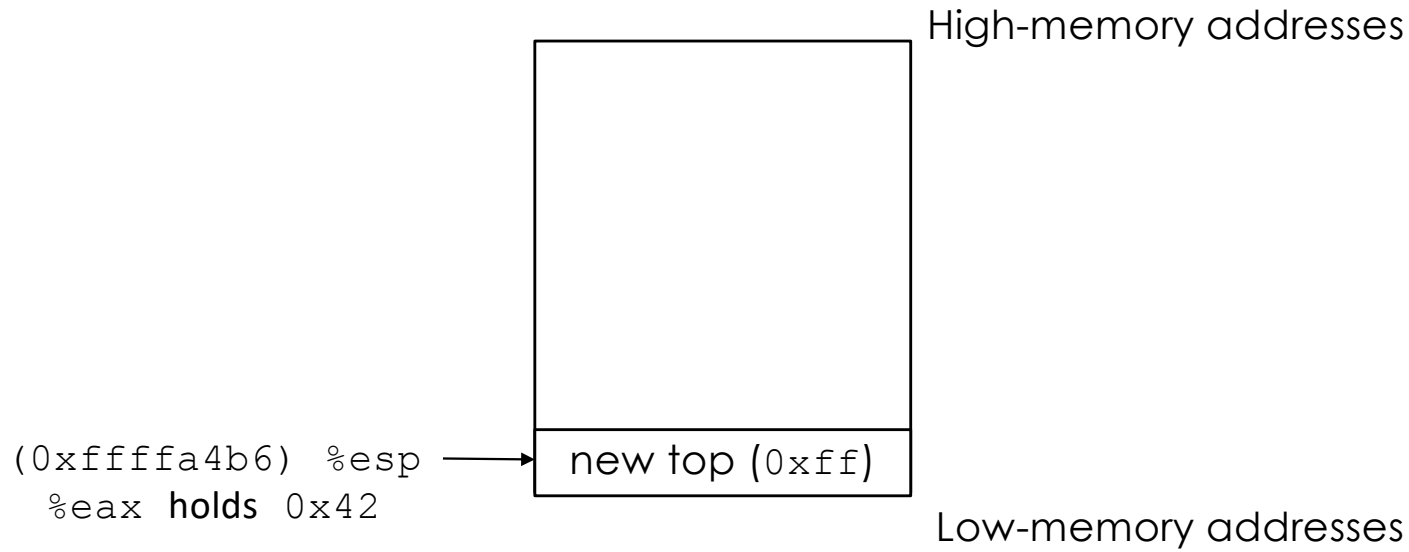
Stack deletion

```
pop %eax
  mov %(esp), %eax
  add 0x4, %esp
```

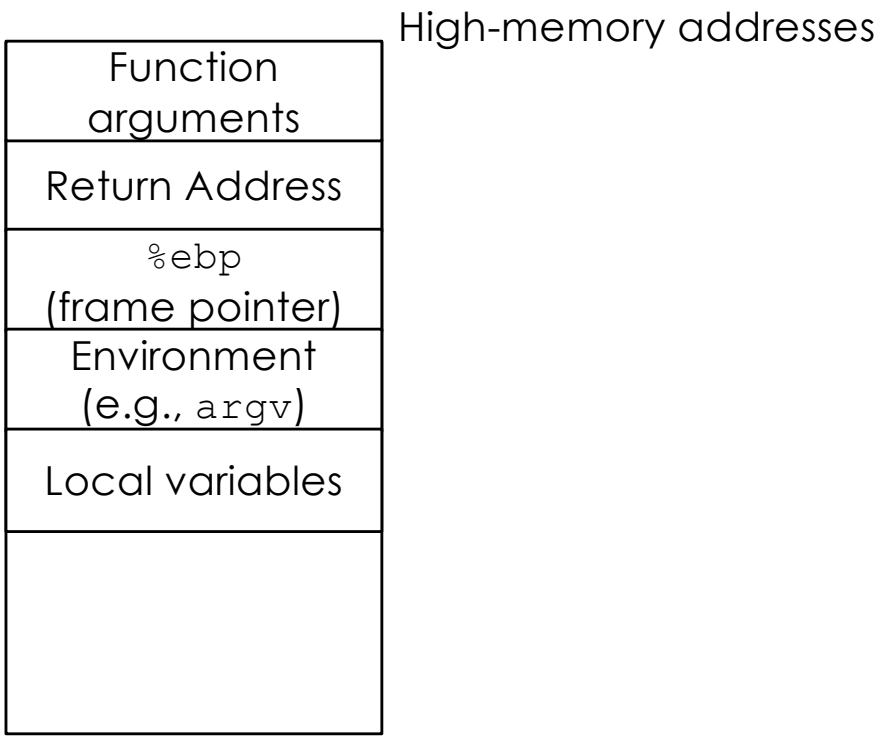


Stack deletion

```
pop %eax
  mov %(esp), %eax
  add 0x4, %esp
```



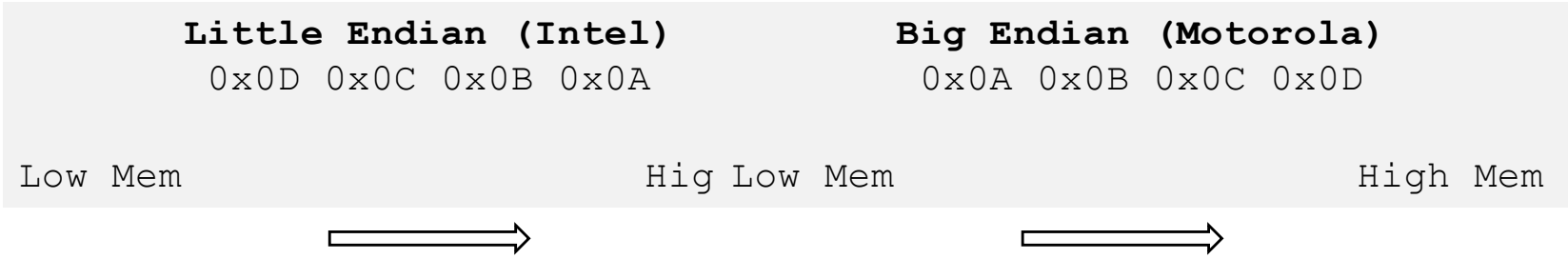
Stack frame in Intel 32-bit



Endianness

Assume the 32-bit word: `0x0A0B0C0D`

Two possible ways to store it in memory



Course progress

01. Memory-safe Programming Systems
02. Vulnerabilities
03. Control-flow Attacks
04. Code Injection
05. Return-oriented Programming (ROP)

Shellcode

A program that is *stuffed* in a user input

- spawns a (remote) shell, downloads malware, creates a user, elevates (becomes root), installs backdoor, etc.

The program is usually small and to the point

- Especially in buffer overflows, the vulnerable buffers may have limited size
- No \0s allowed, otherwise string copies can destroy the shellcode

Highly architectural dependent

Highly unorthodox programming involved

- Custom assembly

Spawning a shell

```
int execve(const char *filename,
           char *const argv[],
           char *const envp[]);
```

Example:

- `execve("/bin/sh", NULL, NULL);`
- Remember, we don't do proper programming here!

System calls in Linux

Can be invoked using a software interrupt

- assembly instruction: `int`
- E.g., for `execve` the interrupt number is `0x0b` (or 11 in decimal)
- Parameters are passed in registers

The OS has a system call table

- Each system call number invokes the appropriate code
- `/usr/include/asm/unistd_32.h`:

```
#define __NR_execve 11
```

`execve` in Linux/IA32

- `execve("/bin/sh", NULL, NULL);`

`%eax`: return value

`%ebx`

- first argument, address of memory that "/bin/sh" is stored

`%edx` and **`%ecx`**

- 2nd and 3rd argument
- We can simply zero these

Hacks

"/bin/sh" is 7 bytes, would be nice if it was 8 bytes

Easy dirty fix

- /bin//sh

No 0s

- `strcpy` can split the shellcode if 0s are contained
- If we need to zero one register we use **`xor`**

Push **/bin//sh**

char	**h**	**s**	**/**	**/**	**n**	**i**	**b**	**/**
ASCII (hex)	0x68	0x73	0x2f	0x2f	0x6e	0x69	0x62	0x2f
value	**0x68732f2f**				**0x6e69622f**			

Shellcode for `execve("/bin/sh");`

```
.section .data

.section .text

.globl _start


_start:

                    xor                 %eax,%eax

                    push                %eax                                                    # \0

                    push                $0x68732f2f         # hs//

                    push                $0x6e69622f         # nib/

                    mov                 %esp,%ebx

                    xor                                                     %ecx,%ecx

                    xor                                     %edx,%edx

                    mov                 $0xb,%al

                    int                 $0x80
```

Course progress

01. Memory-safe Programming Systems
02. Vulnerabilities
03. Control-flow Attacks
04. Code Injection
05. Return-oriented Programming (ROP)

Non executable memory

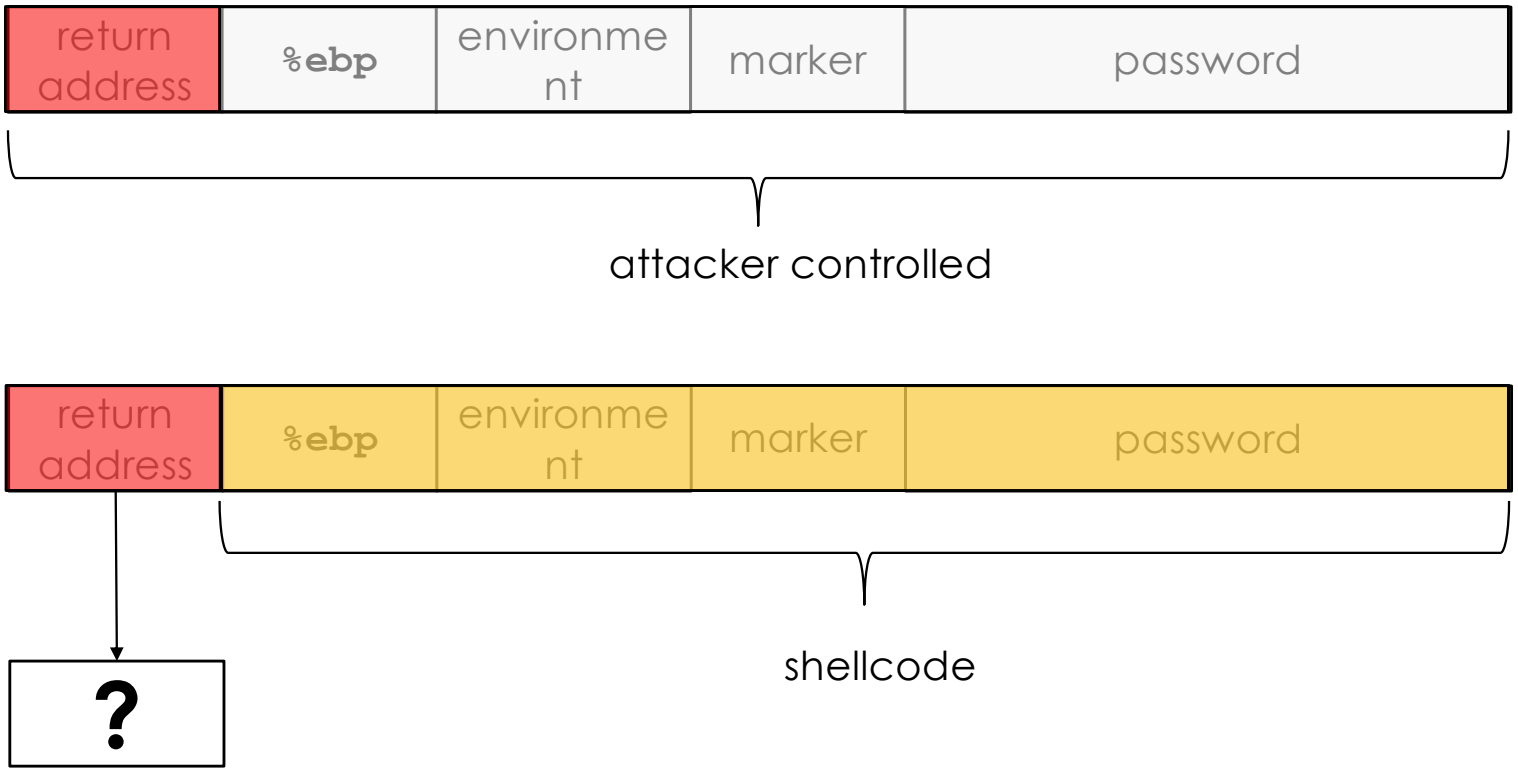
Several names
- NX-bit (Non-Executable Bit)
- DEP (Data Execution Prevention)
- W^X (Write XOR Execute)

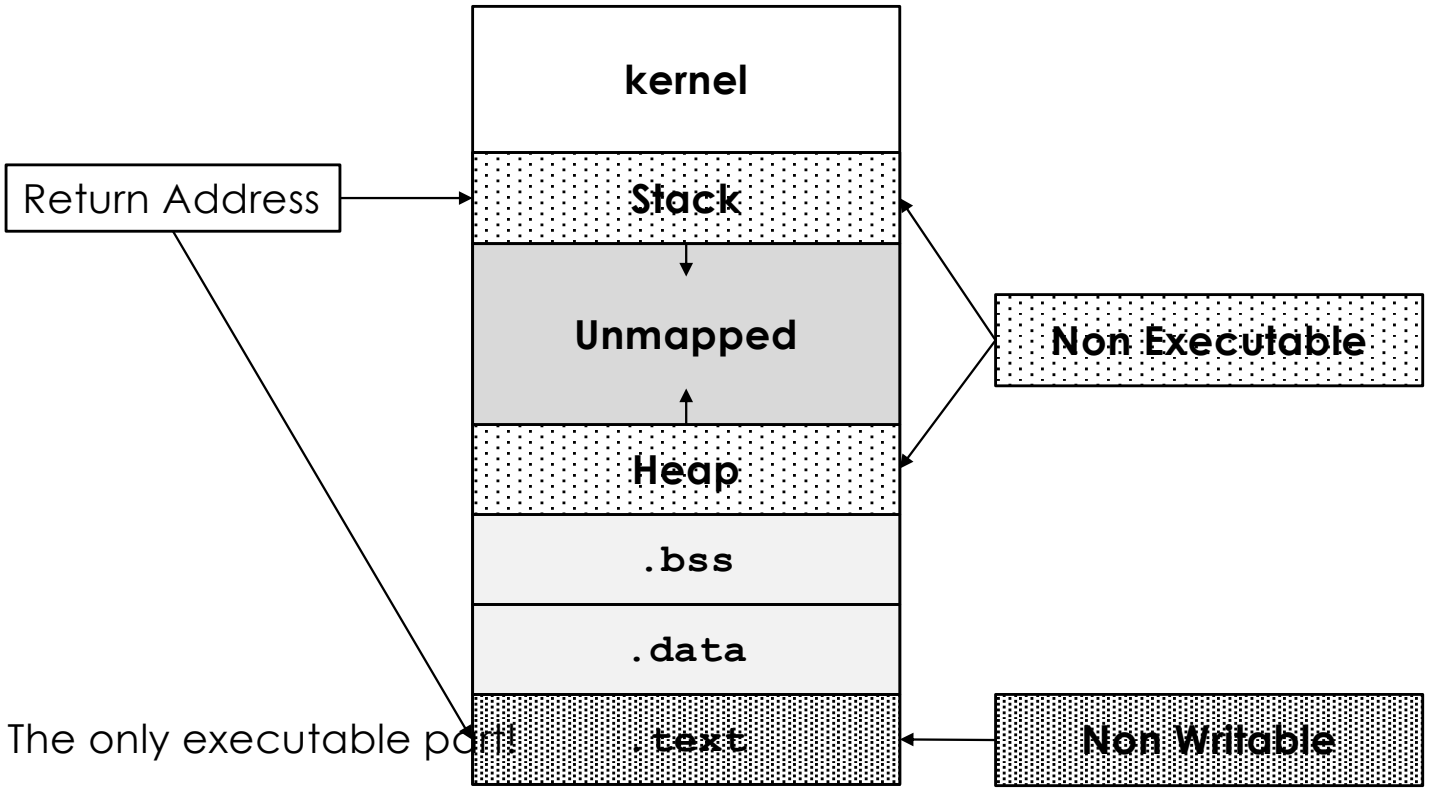
Enforced in hardware (MMU)

Memory pages cannot be executable and writable
- Stack and heap are writable but not executable
- Code is executable but not writable

Permissions can change using system calls
- `mprotect()` for Linux
- `VirtualProtect()` for Windows

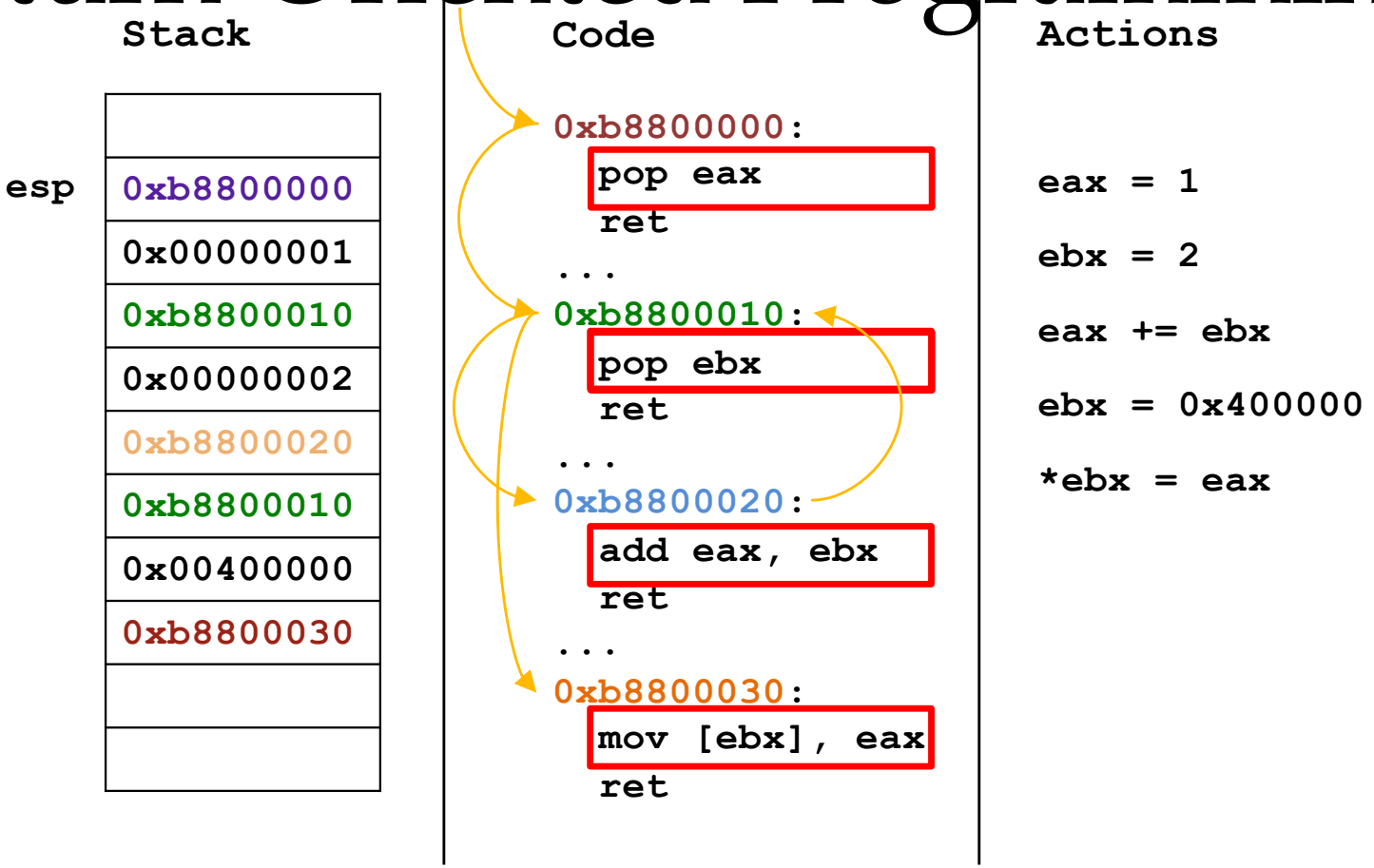
Where should we jump?





The only executable part!

Return-Oriented Programming



Slide taken from http://www.ieee-security.org/TC/SP2012/slides/Smashing%20the%20Gadgets.pptx

Code Reuse

Programs include large parts of existing code
- Available during exploitation

Small sequences of instructions ending with a `ret`
- They called gadgets
- They execute some code and *return* (i.e., read the stack for the next gadget)
- Stack is attacker-controlled (use `esp` as the program counter)

Chaining several gadgets
- Return-Oriented Programming (ROP)
- Turing complete
- In practice used to make a region executable

Gadgets

We saw gadgets like

- `add eax, ebx; ret`
- `mov [ebx], eax; ret`
- `pop eax; ret`
- `...`

Is this code realistic?

Intel and the CISC architecture

Dense instruction set

- All values map to a valid instruction

Non-aligned instructions

Variable-length instructions

- Jumping in the middle of an instruction generates new instructions

Defending ROP

ROP is based on exact knowledge of the code layout

- Code addresses are the beginning of the ROP gadgets

Randomization

- Address Space Layout Randomization (ASLR) (`/proc/sys/kernel/randomize_va_space`)
- Fine-grained

Position Independent Code

Information Leaks

Bugs that let you read the process layout

- So far, overflows were used to overwrite control data

ASLR

- Revealing the address where a shared library is mapped is enough
- All gadgets are just moved to a new offset

Stack canaries

- Revealing the contents of the stack is enough
- The canary is stored in the stack

Thank you

Please send all questions to:
athanasopoulos.elias@ucy.ac.cy

CyberSecPro