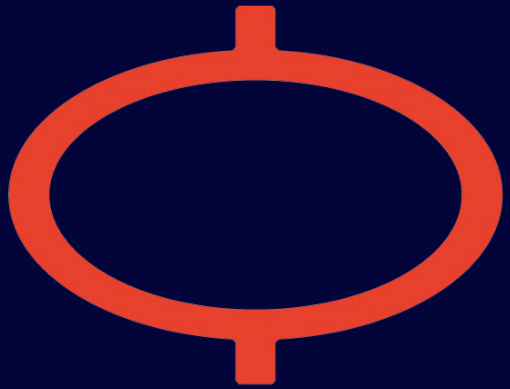


freiheit.com
technologies
hamburg / lisbon

FOUNDED 1999
OVER 120 FULL STACK ENGINEERS
SINCE 2018 NEW OFFICE IN LISBON


R9 0x7ffff7f9ace0 —▸ 0x7ffff7f99e78 —▸ 0x7ffff7ea7440 (__cxxabiv1::__class_type_info::~__class_type_info()) ◂— endbr64
R10 0x7ffff7ab4f40 ◂— stosq qword ptr [rdi], rax /* 0x7ffff7ab4f40 */
R11 0x246
R12 0x401090 (_start) ◂— endbr64
R13 0x7fffffffea30 ◂— 0x1
R14 0x0
R15 0x0
RBP 0x7fffffffe950 ◂— 0x4141414141414141 ('AAAAAAAA')
RSP 0x7fffffffe930 ◂— 0x4141414141414141 ('AAAAAAAA')
RIP 0x4011fb (main+74) ◂— lea rsi, [rip + 0xe2e]
[DISASM]
0x4011e3 <main+50> call 0x401070
0x4011e8 <main+55> lea rax, [rbp - 0x20]
0x4011ec <main+59> mov rsi, rax
0x4011ef <main+62> lea rdi, [rip + 0x2f8a] <0x404180>
0x4011f6 <main+69> call 0x401030
▶ 0x4011fb <main+74> lea rsi, [rip + 0xe2e]
0x401202 <main+81> lea rdi, [rip + 0x2e57] <0x404060>
0x401209 <main+88> call
0x40120e <main+93> mov rdx, rax
0x401211 <main+96> lea rax, [rbp - 0x20]
0x401215 <main+100> mov rsi, rax
[STACK]
00:0000 rsp 0x7fffffffe930 ◂— 0x4141414141414141 ('AAAAAAAA')
... ↓
[BACKTRACE]
▶ f 0 4011fb main+74
Hacking like in the 90s
Binary Exploitation Workshop

$(whoami)

- Vitali Henne (30)
- Software Engineer at freiheit.com
- CTF player with
  - Cyclopropenylidene
  - SauerCloud
  - KITCTF in the past
- I like to break things :>

WTF is a CTF?

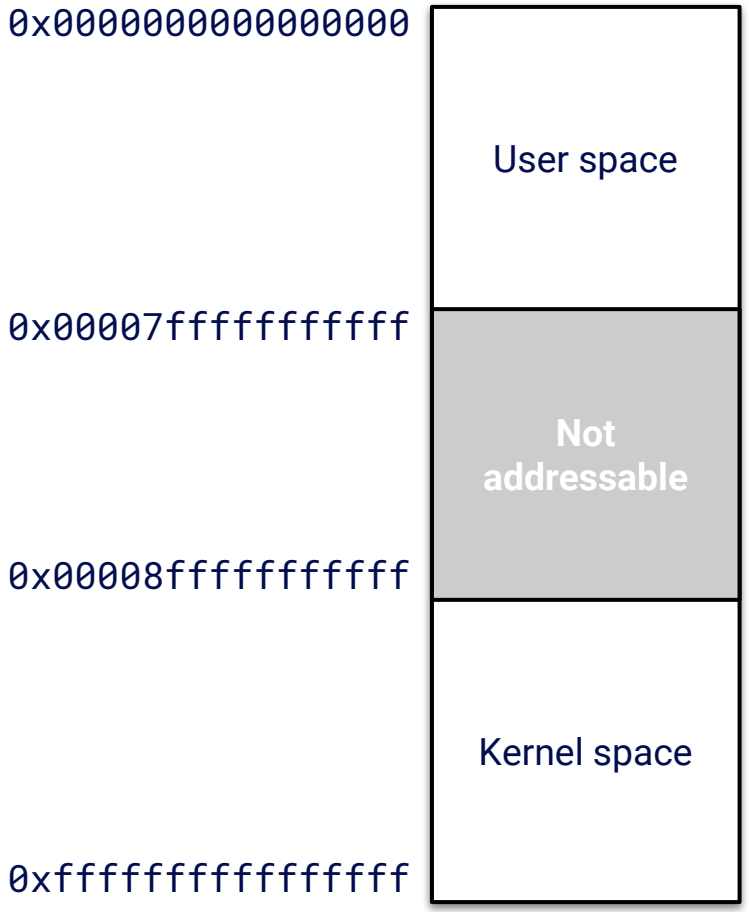
- **Practical infosec competition**
  - Usually organized by teams participating in such competitions for other teams
- **Different categories:**
  - Web
  - Reverse engineering
  - Binary exploitation
  - Cryptography
  - …
- **Duration varies between 12h - 72h**
- **Goal is to retrieve a “flag” - a piece of information hidden in the challenge:**
  - E.g. In binary exploitation the flag is somewhere on the server and you want to exploit the binary to leak this data

WTF is a CTF?

- **Practical infosec competition**
  - Usually organized by teams participating in such competitions for other teams
- **Different categories:**
  - Web
  - Reverse engineering
  - **Binary exploitation**
  - Cryptography
  - …
- **Duration varies between 12h - 72h**
- **Goal is to retrieve a “flag” - a piece of information hidden in the challenge:**
  - E.g. In binary exploitation the flag is somewhere on the server and you want to exploit the binary to leak this data

Memory Layout

- Each process is assigned a distinct virtual address space by the kernel.
- Divided into multiple regions with different access permission:
  - Readable
  - Writeable
  - Executable
- Example: x86_64 address space on Linux

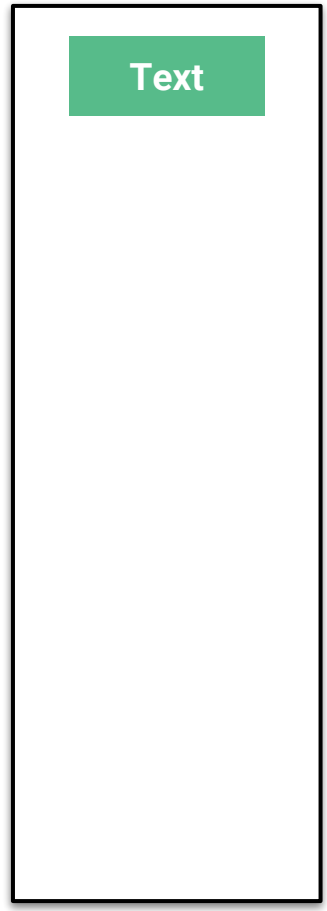


Mem. Layout - Text Segment

- Also called code segment
- Contains machine code of the binary

- Readable | Executable

```
pwndbg> x/8i main
   0x4011c1 <main>:     push   rbp
   0x4011c2 <main+1>:   mov    rbp,rsp
   0x4011c5 <main+4>:   sub    rsp,0x110
   0x4011cc <main+11>:  mov    DWORD PTR [rbp-0x4],0x2a
```



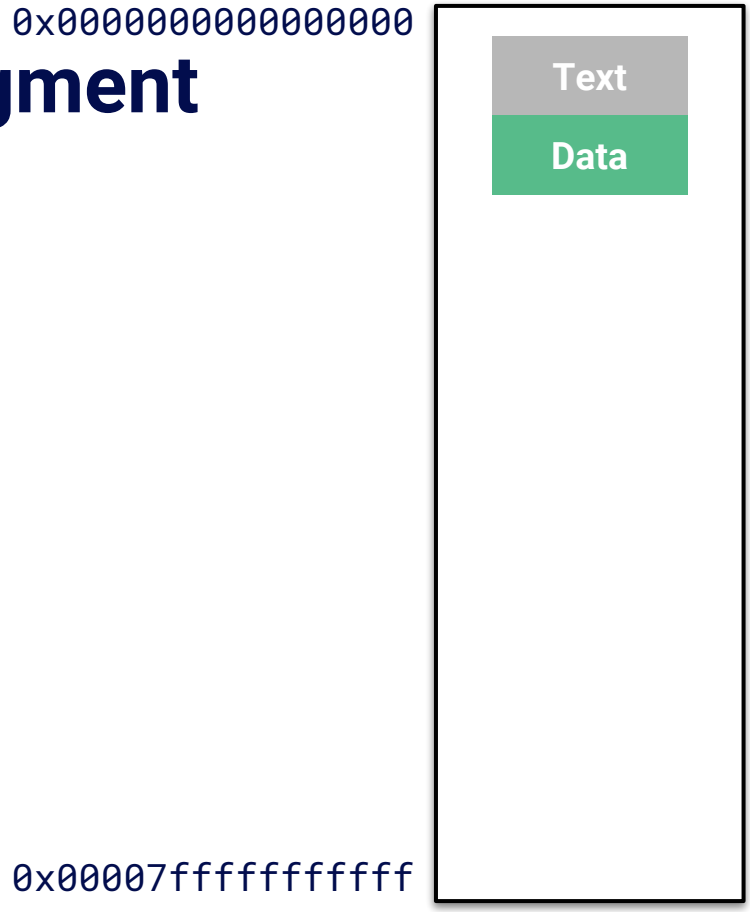
Mem. Layout - Data Segment

- Initialized data of global and static variables
- Fixed size, known on compile time

- Readable
- Readable | Writeable

```
#include <iostream>

char buf[] = "Hello World!";
int main(int argc, char* argv[]) {
    std::cout << buf << std::endl;
}
```



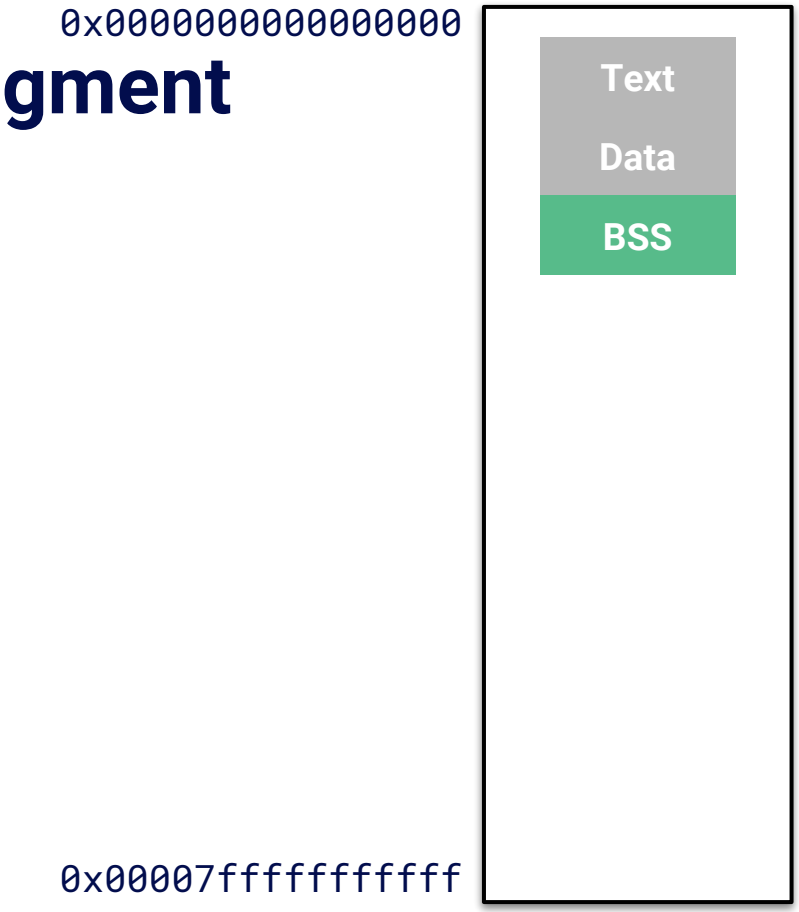
Mem. Layout - BSS Segment

- Uninitialized global and static variables
- Whole segment initialized with 0 by kernel
- Fixed size, known on compile time

- Readable | Writeable

```cpp
#include <iostream>
#include <string.h>

char buf[16];
int main(int argc, char* argv[]) {
    strcpy(buf, "Hello World!");
    std::cout << buf << std::endl;
}
```



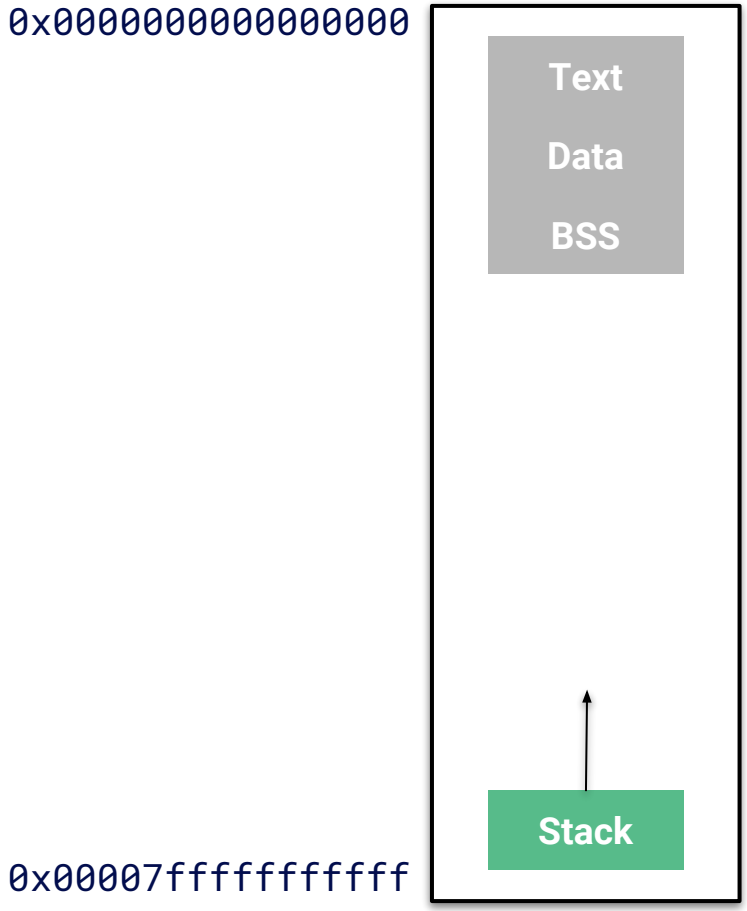
Mem. Layout - Stack

- Located in “high memory” area
- Grows towards lower memory addresses
- Local variables of each function

- Readable | Writeable | ~~Executable~~

```
#include <iostream>

int main(int argc, char* argv[]) {
    char buf[] = "Hello World!";
    std::cout << buf << std::endl;
}
```

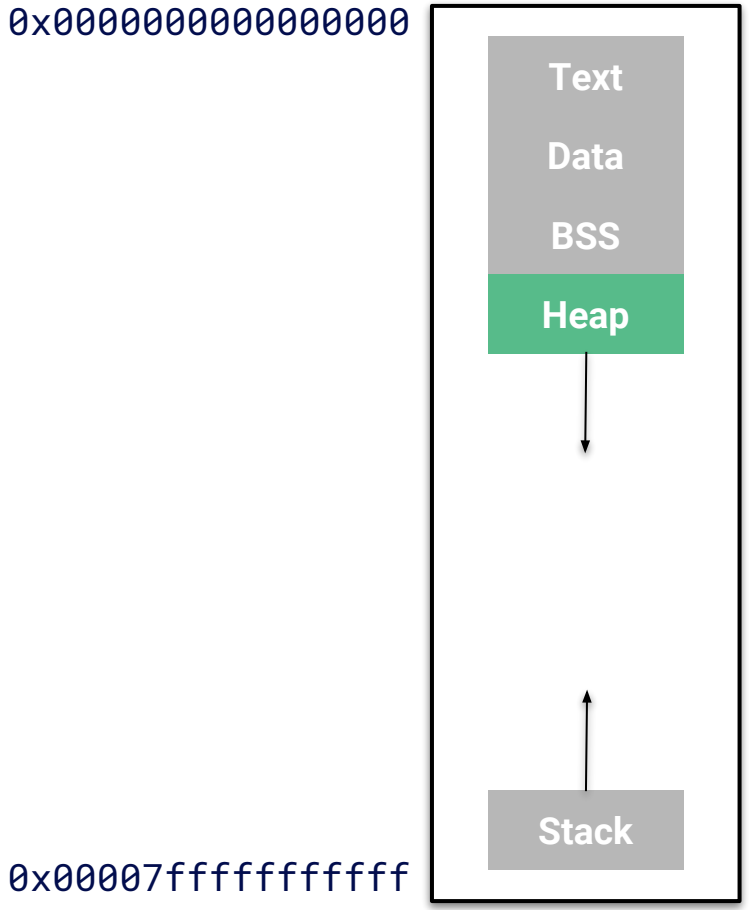


Mem. Layout - Heap

- Located behind the BSS segment
- Grows towards higher memory addresses
- Managed by a heap allocator:
  - Dynamically allocates chunks of memory
  - Chunk size may be defined on runtime

- Readable | Writeable

```
// ...
int main(int argc, char* argv[]) {
    char* buf = malloc(16);
    strcpy(buf, "Hello World!");
    std::cout << buf << std::endl;
    free(buf);
}
```

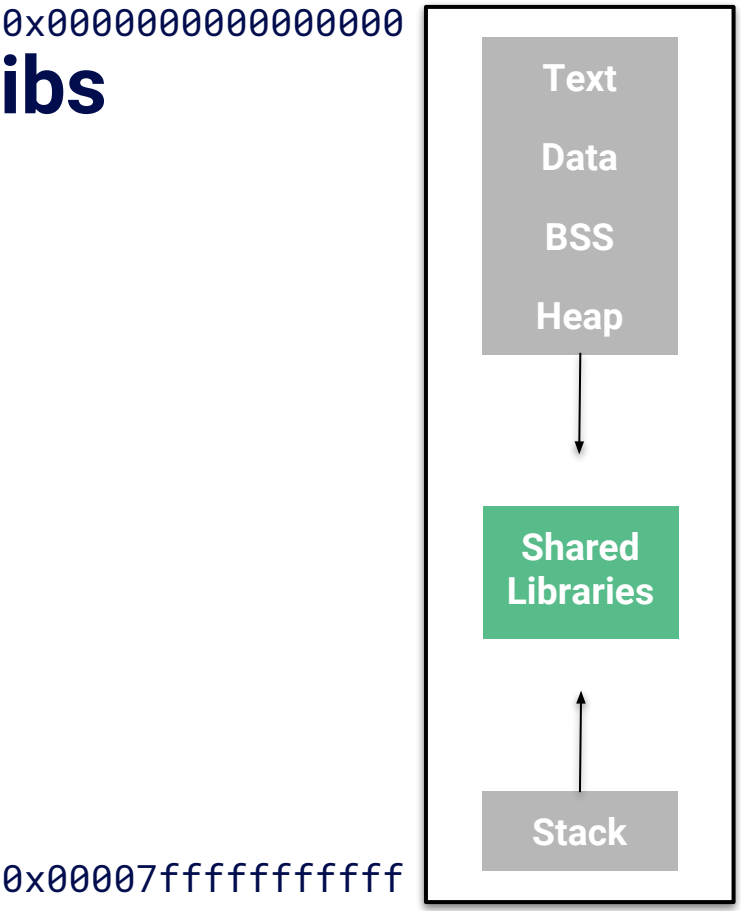


Mem. Layout - Shared Libs

- Memory mapping segment

- Readable
- Readable | Writeable
- Readable | Executable

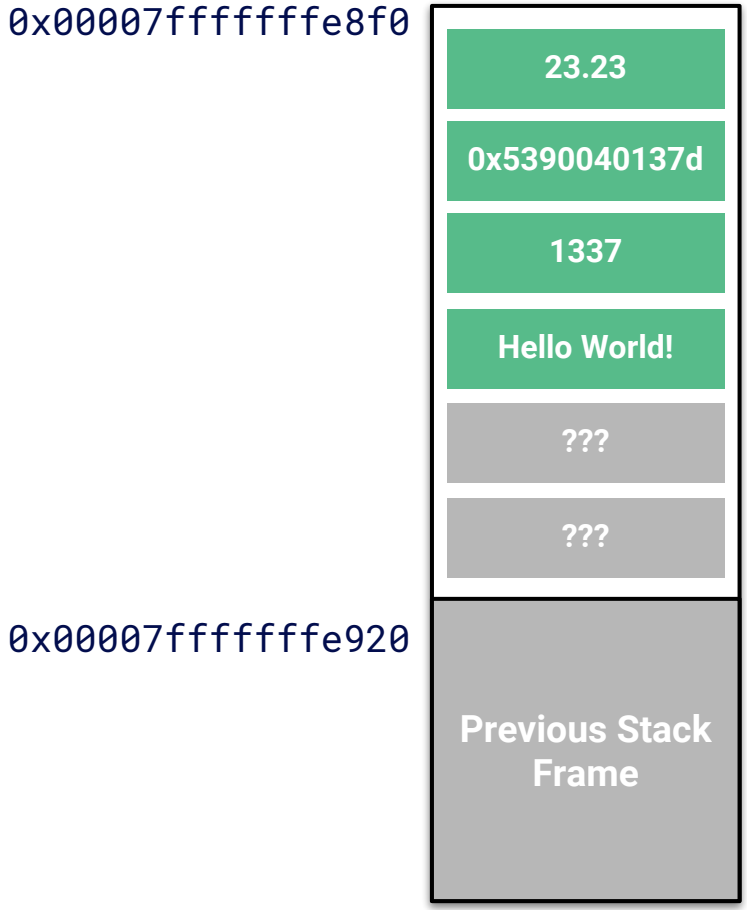
```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x555555554000     0x555555555000 r--p     1000 0      /home/vagrant/my_binary
    0x555555555000     0x555555556000 r-xp     1000 1000   /home/vagrant/my_binary
    0x555555556000     0x555555557000 r--p     1000 2000   /home/vagrant/my_binary
    0x555555557000     0x555555558000 r--p     1000 2000   /home/vagrant/my_binary
    0x555555558000     0x555555559000 rw-p     1000 3000   /home/vagrant/my_binary
    0x7ffff7de4000     0x7ffff7e06000 r--p    22000 0      /usr/lib/libc-2.28.so
    0x7ffff7e06000     0x7ffff7f51000 r-xp   14b000 22000  /usr/lib/libc-2.28.so
    0x7ffff7f51000     0x7ffff7f9d000 r--p    4c000 16d000 /usr/lib/libc-2.28.so
    0x7ffff7f9d000     0x7ffff7f9e000 ---p     1000 1b9000 /usr/lib/libc-2.28.so
    0x7ffff7f9e000     0x7ffff7fa2000 r--p     4000 1b9000 /usr/lib/libc-2.28.so
    0x7ffff7fa2000     0x7ffff7fa4000 rw-p     2000 1bd000 /usr/lib/libc-2.28.so
    0x7ffff7fa4000     0x7ffff7faa000 rw-p     6000 0
    0x7ffff7fce000     0x7ffff7fd1000 r--p     3000 0      [vvar]
    0x7ffff7fd1000     0x7ffff7fd3000 r-xp     2000 0      [vdso]
    0x7ffff7fd3000     0x7ffff7fd5000 r--p     2000 0      /usr/lib/ld-2.28.so
    0x7ffff7fd5000     0x7ffff7ff4000 r-xp    1f000 2000   /usr/lib/ld-2.28.so
    0x7ffff7ff4000     0x7ffff7ffc000 r--p     8000 21000  /usr/lib/ld-2.28.so
    0x7ffff7ffc000     0x7ffff7ffd000 r--p     1000 28000  /usr/lib/ld-2.28.so
    0x7ffff7ffd000     0x7ffff7ffe000 rw-p     1000 29000  /usr/lib/ld-2.28.so
    0x7ffff7ffe000     0x7ffff7fff000 rw-p     1000 0
    0x7ffffffde000     0x7ffffffff000 rw-p    21000 0      [stack]
```



The Stack Frame

- Holds local variables
- Allocate a frame for each function called
  - Required size is known on compile time because all local variables are known

```
void do_stuff() {
    char buf[] = "Hello World!";
    int x = 1337;
    void* ptr = malloc(23);
    float y = 23.23;
}
```



The Call Stack

- ip: points to the next to-be executed instruction
- sp: points to the current end of the stack

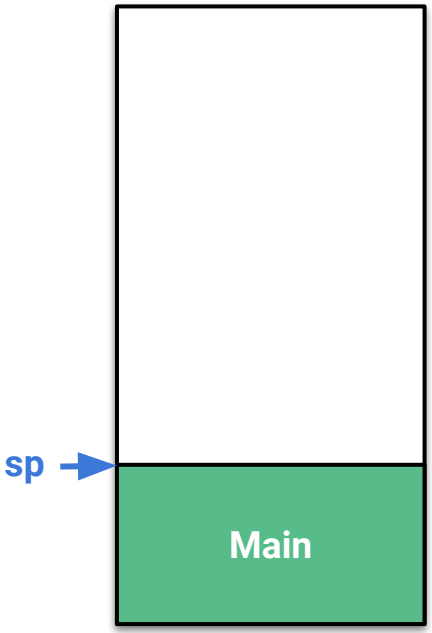
```
void f2() {
    printf("Called f2\n");
    return;
}

void f1() {
    printf("Called f1\n");
    f2();
    return;
}

int main() {
    f1();
    printf("done\n");
}
```

ip →

Main() stack frame is allocated upon invocation



The Call Stack

- ip: points to the next to-be executed instruction
- sp: points to the current end of the stack

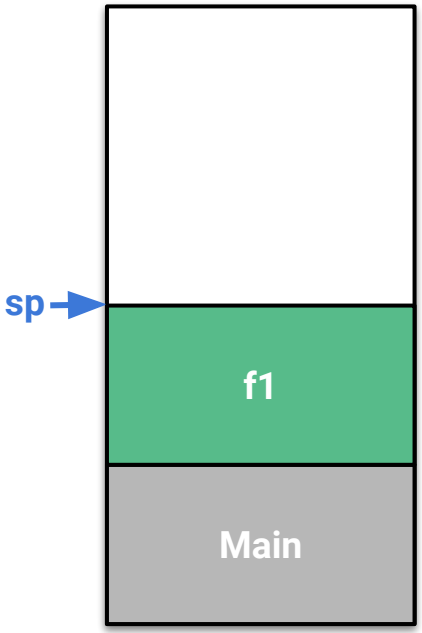
```
void f2() {
    printf("Called f2\n");
    return;
}

void f1() {
    printf("Called f1\n");
    f2();
    return;
}

int main() {
    f1();
    printf("done\n");
}
```

ip →

f1() stack frame is allocated when it's called by main()



The Call Stack

- ip: points to the next to-be executed instruction
- sp: points to the current end of the stack

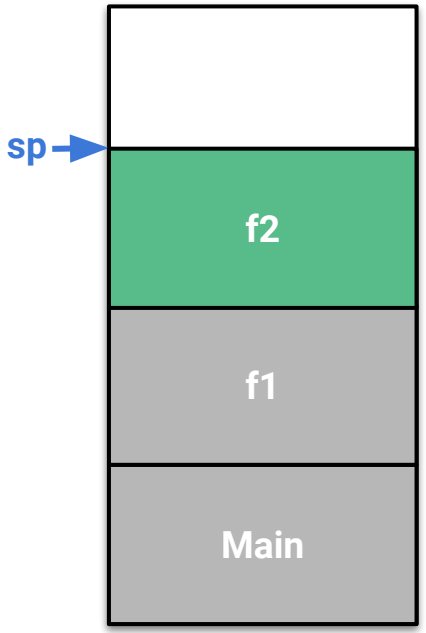
```
void f2() {
    printf("Called f2\n");
    return;
}

void f1() {
    printf("Called f1\n");
    f2();
    return;
}

int main() {
    f1();
    printf("done\n");
}
```

ip →

f2() stack frame is allocated when it's called by f1()



The Call Stack

- ip: points to the next to-be executed instruction
- sp: points to the current end of the stack

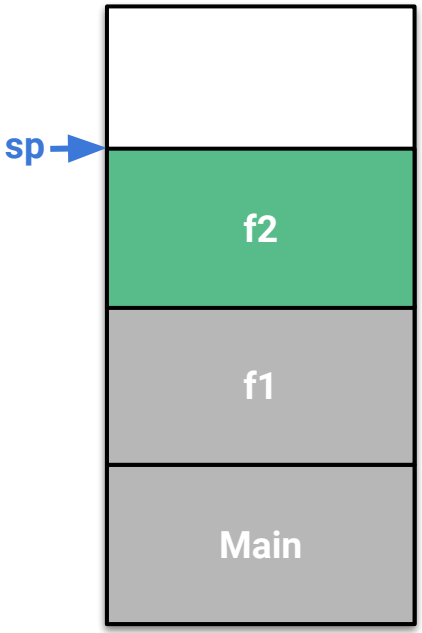
```
void f2() {
    printf("Called f2\n");
    return;
}

void f1() {
    printf("Called f1\n");
    f2();
    return;
}

int main() {
    f1();
    printf("done\n");
}
```

ip →

What happens when f2() returns?



The Call Stack

- **ip**: points to the next to-be executed instruction
- **sp**: points to the current end of the stack

ip

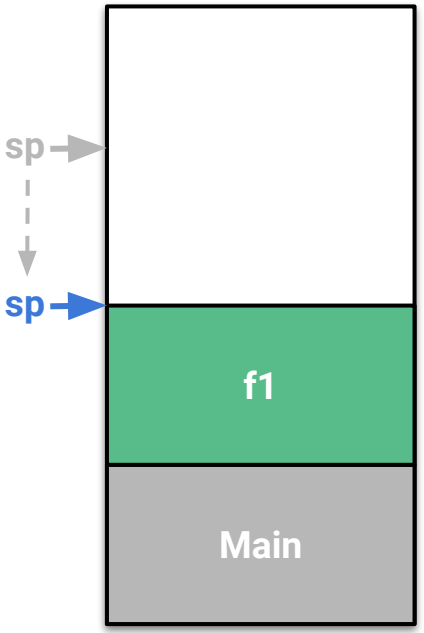
```c
void f2() {
    printf("Called f2\n");
    return;
}

void f1() {
    printf("Called f1\n");
    f2();
    return;
}

int main() {
    f1();
    printf("done\n");
}
```

ip

What happens when f2() returns?
=> Continue execution of f1()
=> Restore f1() stackframe

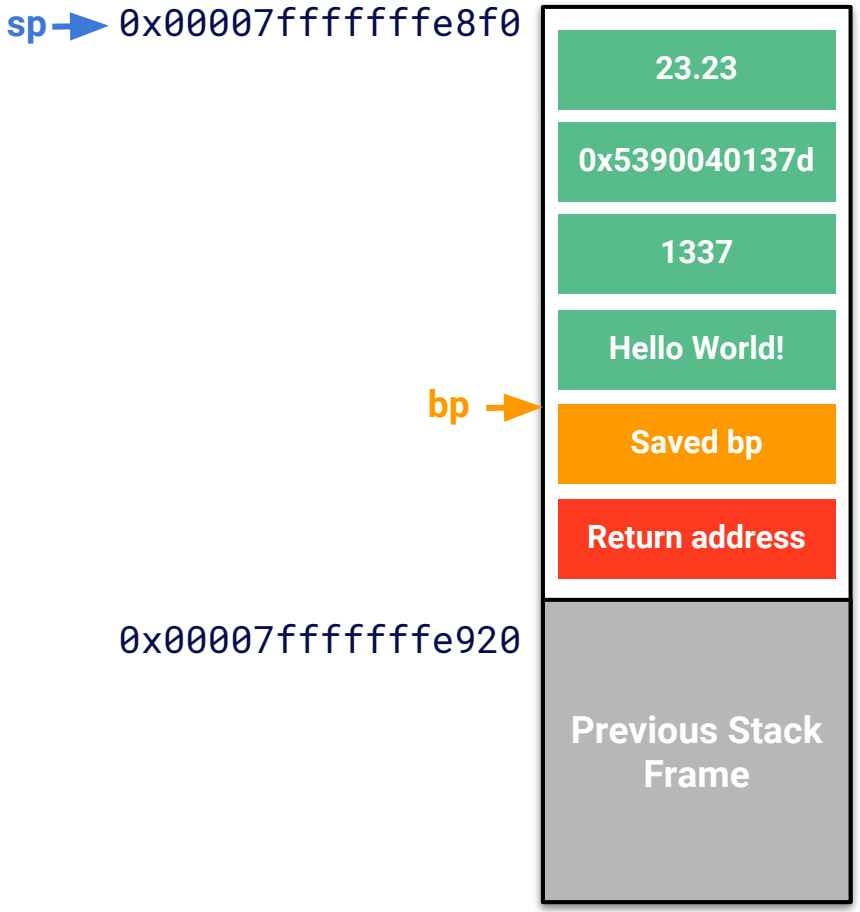


The Call Stack

Restoration of previous state

a) Return address
b) Previous stack pointer
   i) Use a “stack base pointer” **bp** pointing to the start of the stack frame

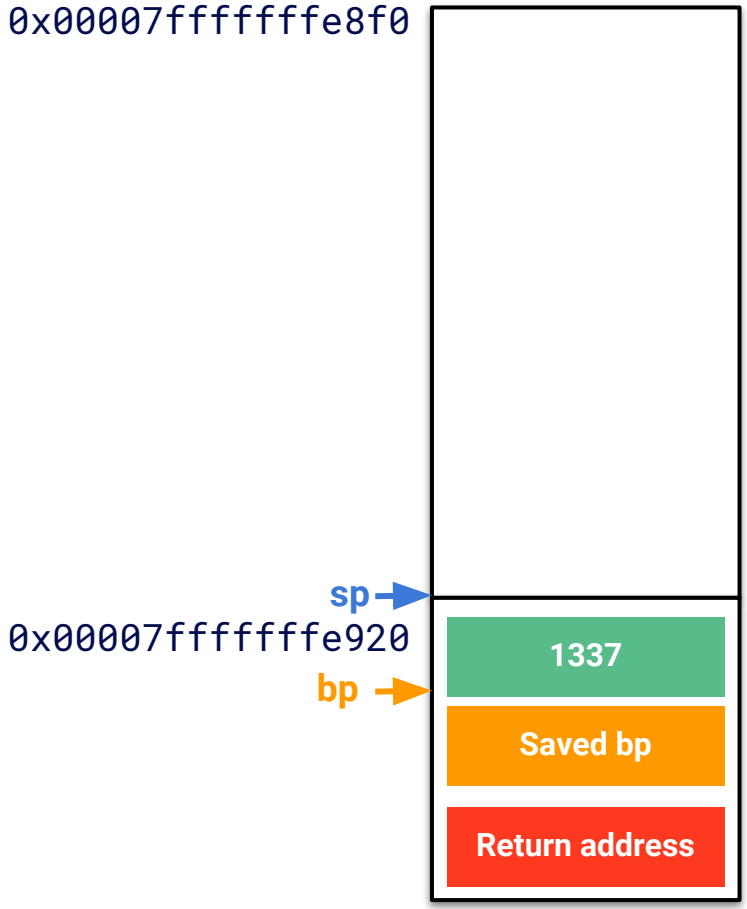
```
void do_stuff() {
    char buf[] = "Helo World!";
    int x = 1337;
    void* ptr = malloc(23);
    float y = 23.23;
}
```



Function Prologue

```
(1) call foo (push rip, jmp foo)

(2) push rbp
(3) mov rbp, rsp
(4) sub rsp 32
```

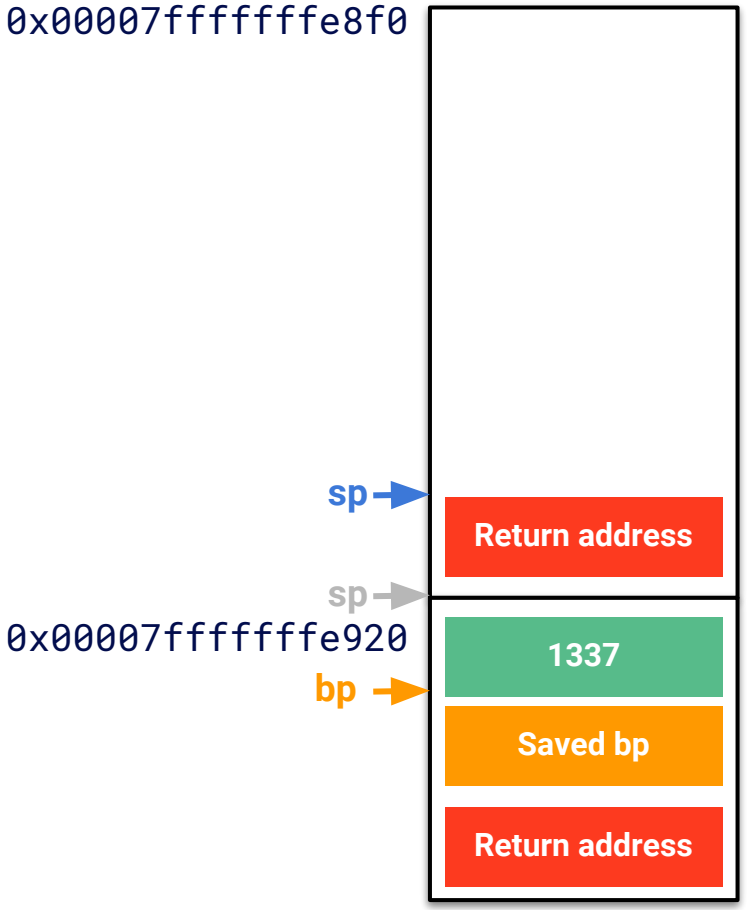


Function Prologue

- The call instruction pushes **ip** on the stack

```
(1) call foo (push rip, jmp foo)

(2) push rbp
(3) mov rbp, rsp
(4) sub rsp 32
```

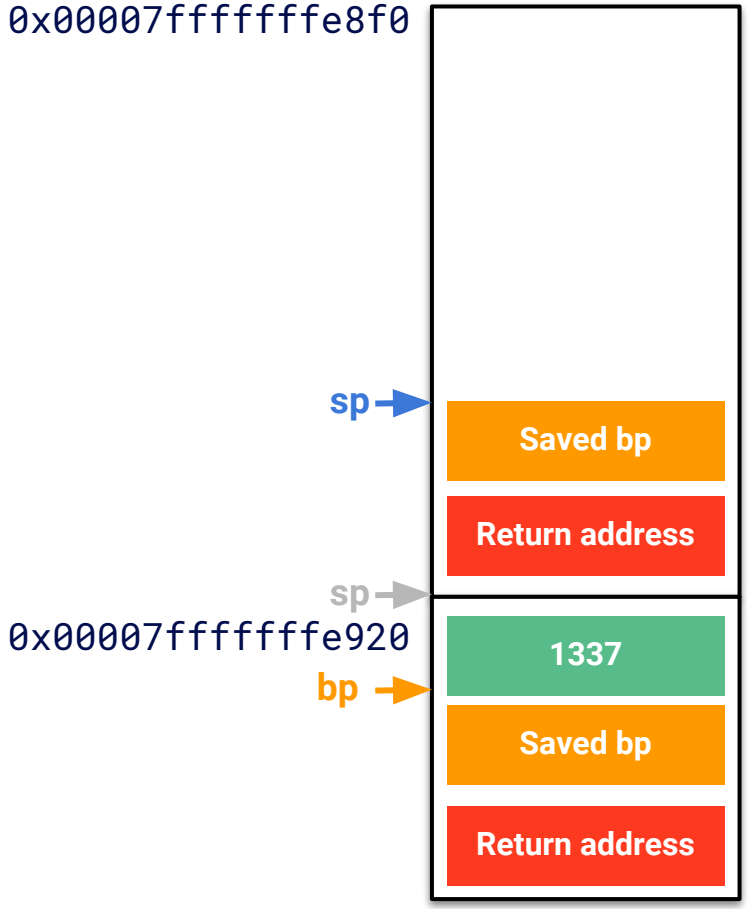


Function Prologue

- The call instruction pushes **ip** on the stack
- Save the previous base pointer

```
(1) call foo (push rip)

(2) push rbp
(3) mov rbp, rsp
(4) sub rsp 32
```

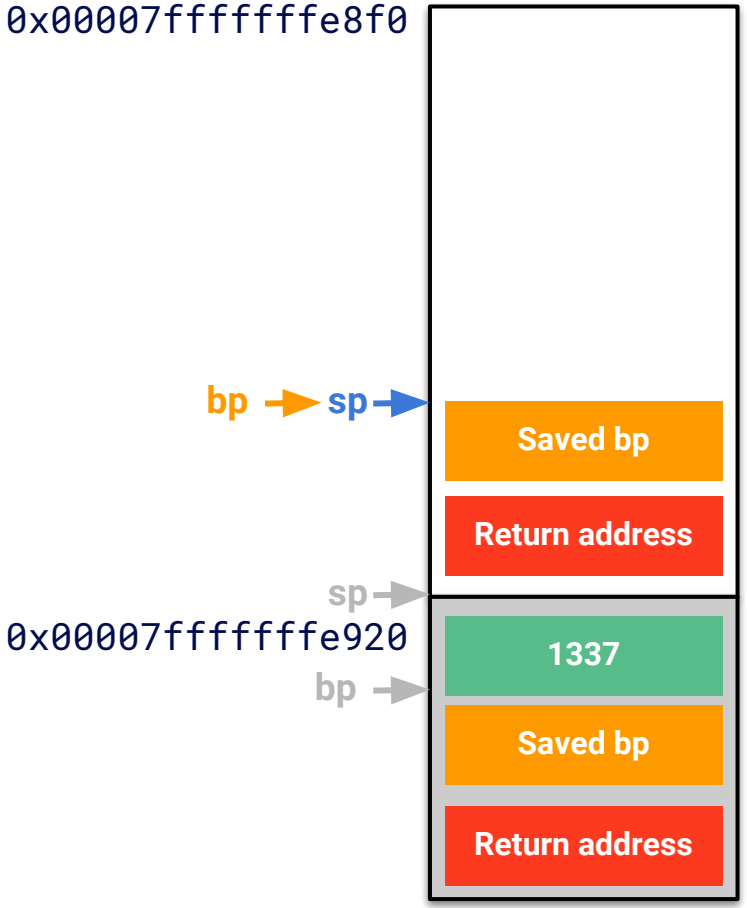


Function Prologue

- The call instruction pushes **ip** on the stack
- Save the previous base pointer
- Set new base pointer

```
(1) call foo (push rip, jmp foo)

(2) push rbp
(3) mov rbp, rsp
(4) sub rsp 32
```

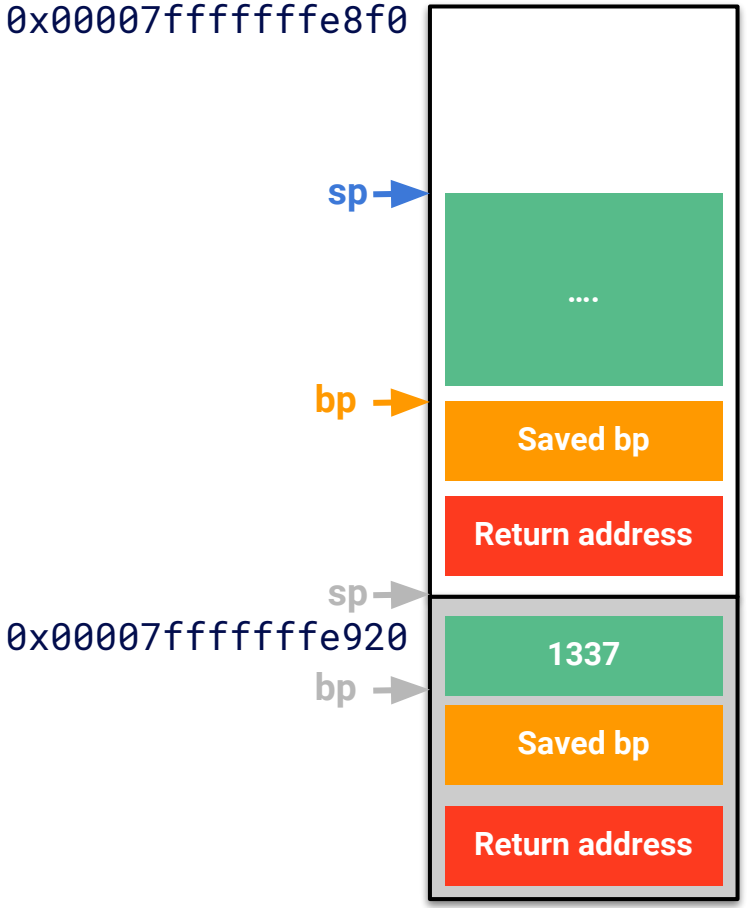


Function Prologue

- The call instruction pushes **ip** on the stack
- Save the previous base pointer
- Set new base pointer
- Make room on the stack for local args

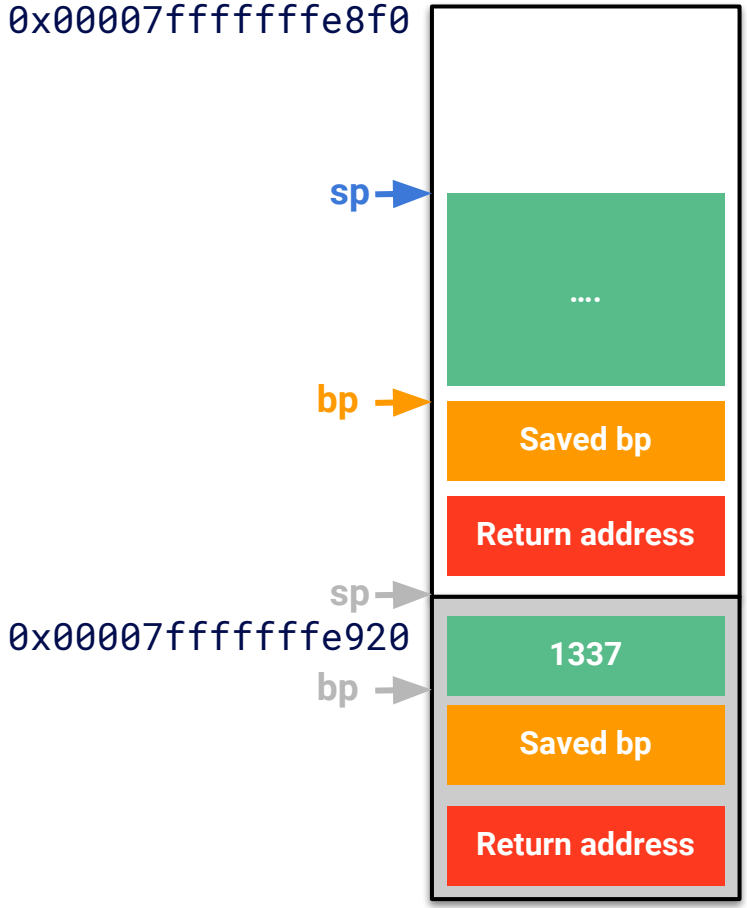
```
(1) call foo (push rip, jmp foo)

(2) push rbp
(3) mov rbp, rsp
(4) sub rsp 32
```



Function Epilogue

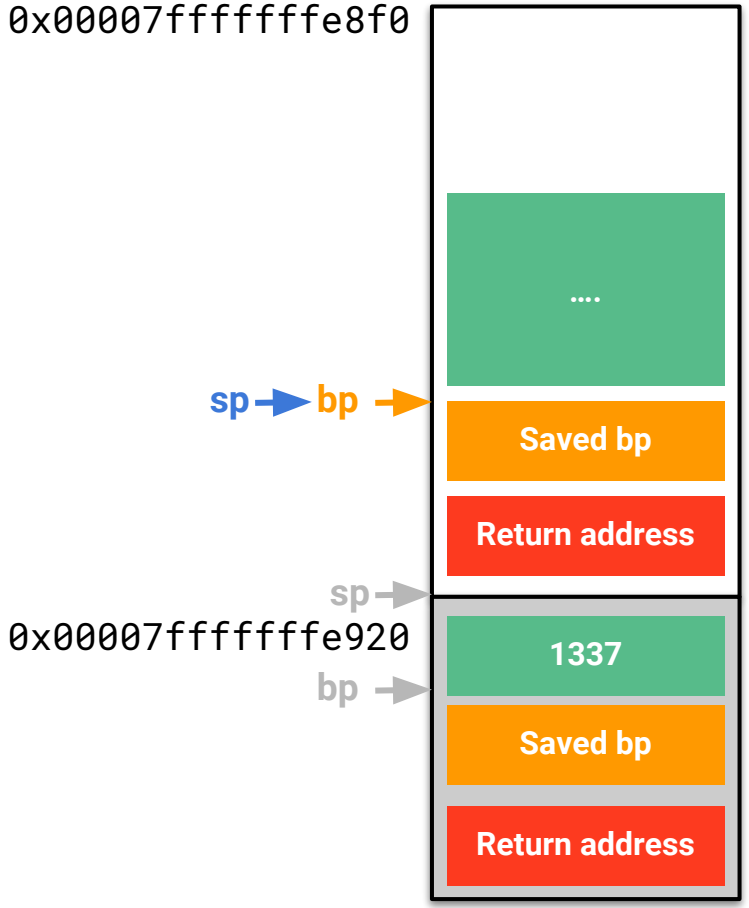
```
(1) mov rsp, rbp
(2) pop rbp
(3) ret (pop rip)
```



Function Epilogue

- “Cleanup” local arguments

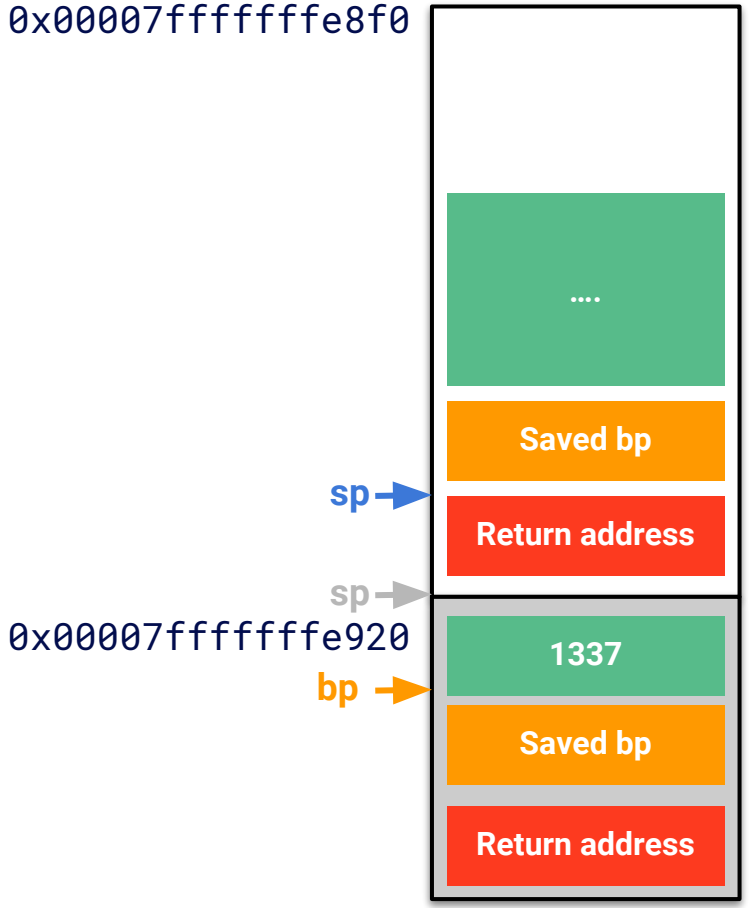
```
(1) mov rsp, rbp
(2) pop rbp
(3) ret (pop rip)
```



Function Epilogue

- “Cleanup” local arguments
- Restore previous base pointer

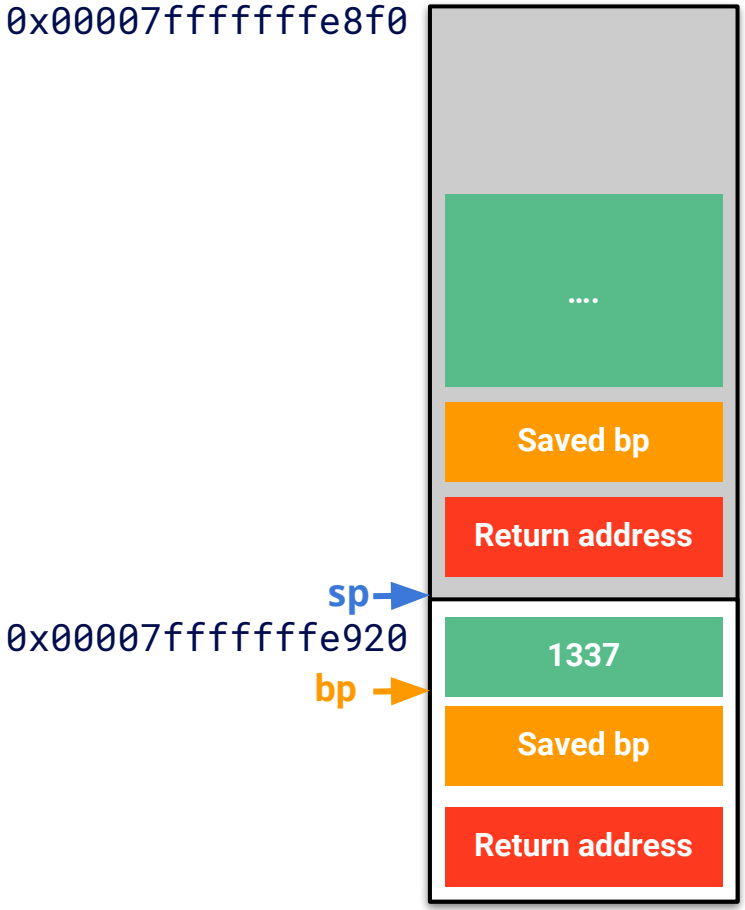
```
(1) mov rsp, rbp
(2) pop rbp
(3) ret (pop rip)
```



Function Epilogue

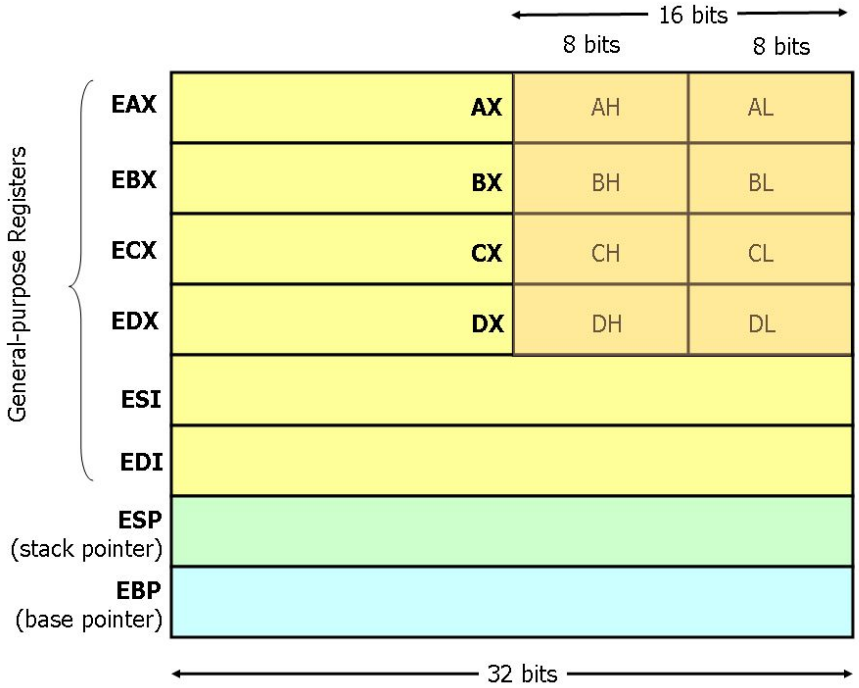
- “Cleanup” local arguments
- Restore previous base pointer
- The ret instruction pops **ip** from the stack

```
(1) mov rsp, rbp
(2) pop rbp
(3) ret (pop rip)
```



x86 Assembly 101 - Registers

- Lower parts of general purpose registers can be accessed individually
- x64 extends registers to 64 bit



Current issue : #49 | Release date : 1996-11-08 | Editor : daemon9

Author : Aleph1

.oO Phrack 49 Oo.

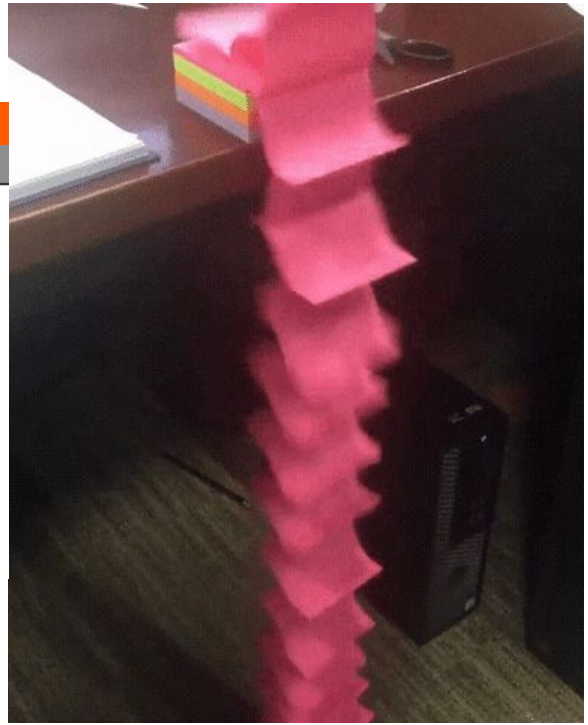
Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

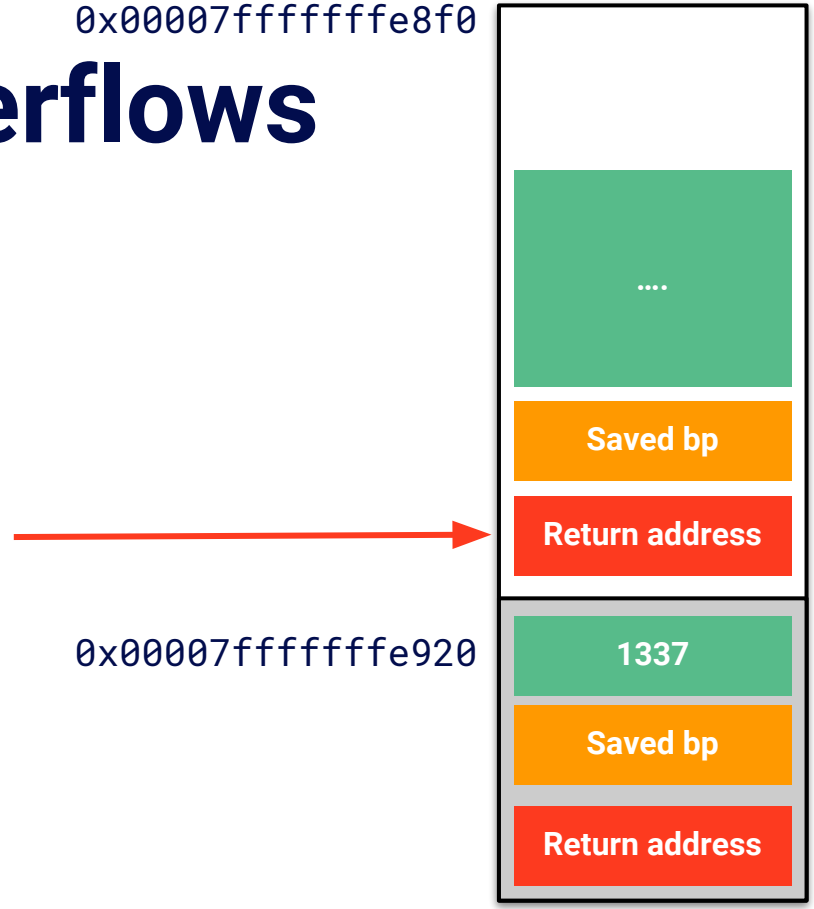


Exploiting Stack Overflows

Goal:

Change control flow to achieve arbitrary code execution

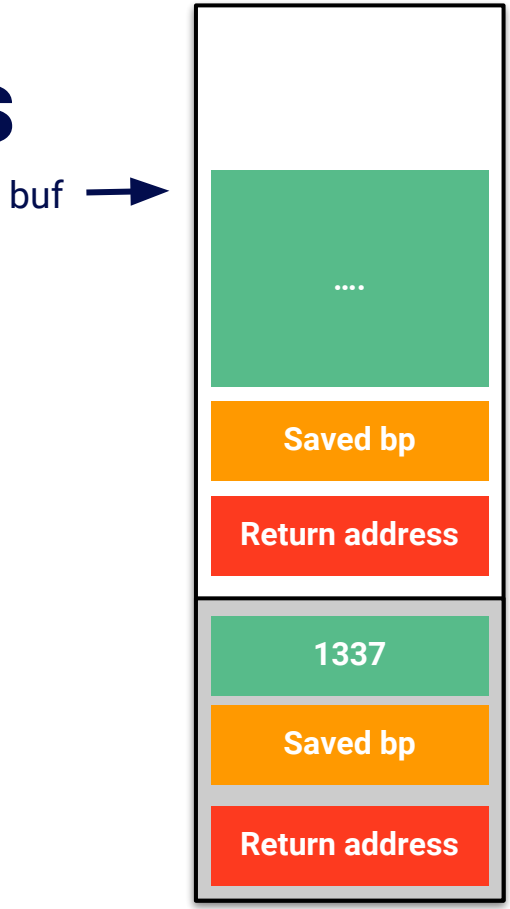
Execution will continue at the address stored here!



Exploiting Stack Overflows

What happens if we were to overflow a buffer on the stack?

```
void fn(char *input) {
    char buf[16];
    strcopy(buf, input);
}
```



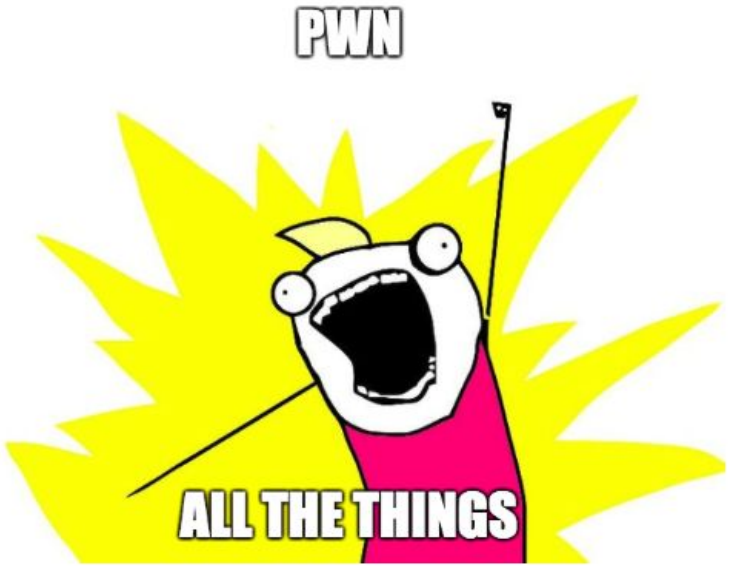
Exploiting Stack Overflows

What happens if we were to overflow a buffer on the stack? “strcopy” has no check for length!

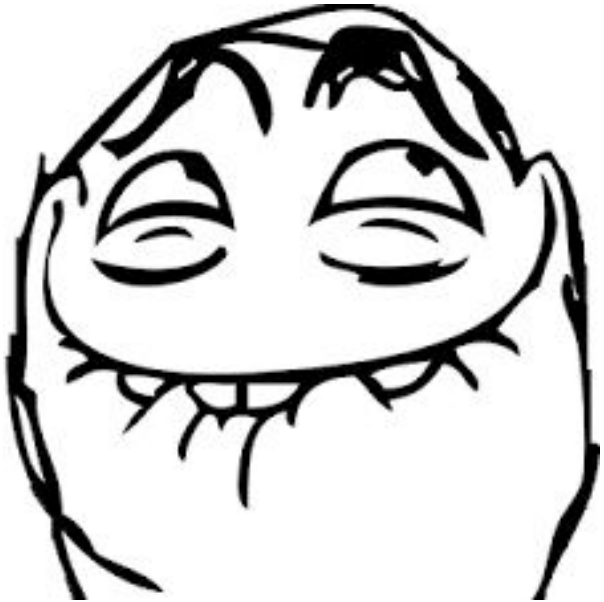
```
void fn(char *input) {
    char buf[16];
    strcopy(buf, input);
}
```

```
fn("AAAAAAAAAAAAAAAAAAAAAAAA...")
```

1. Get control flow
2. Redirect to your code
3. ???
4. Profit

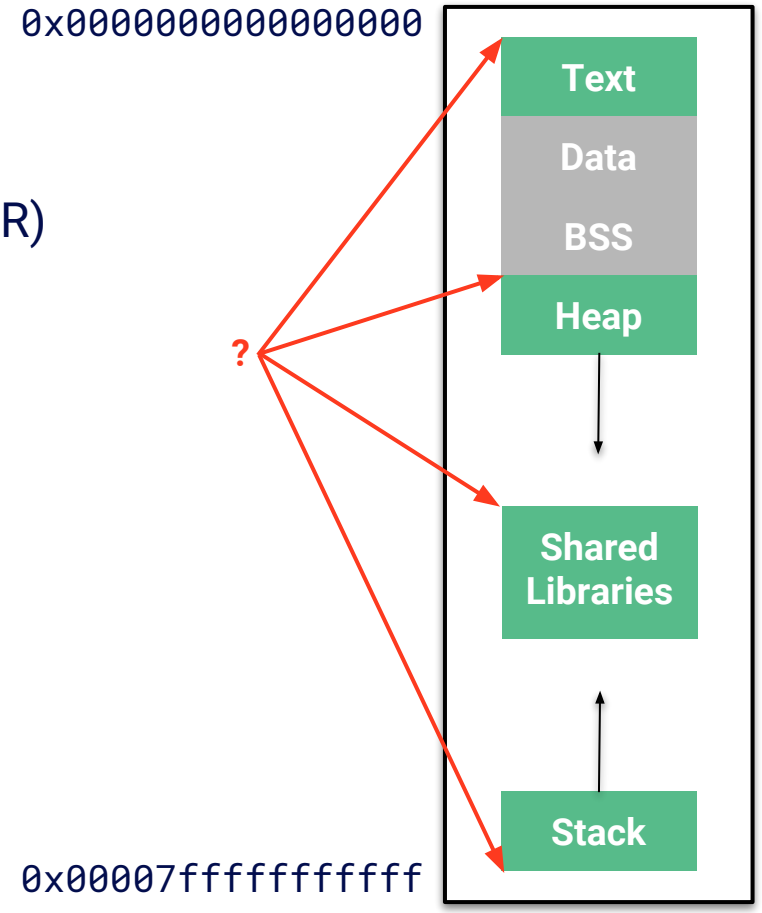


Demo time!!!



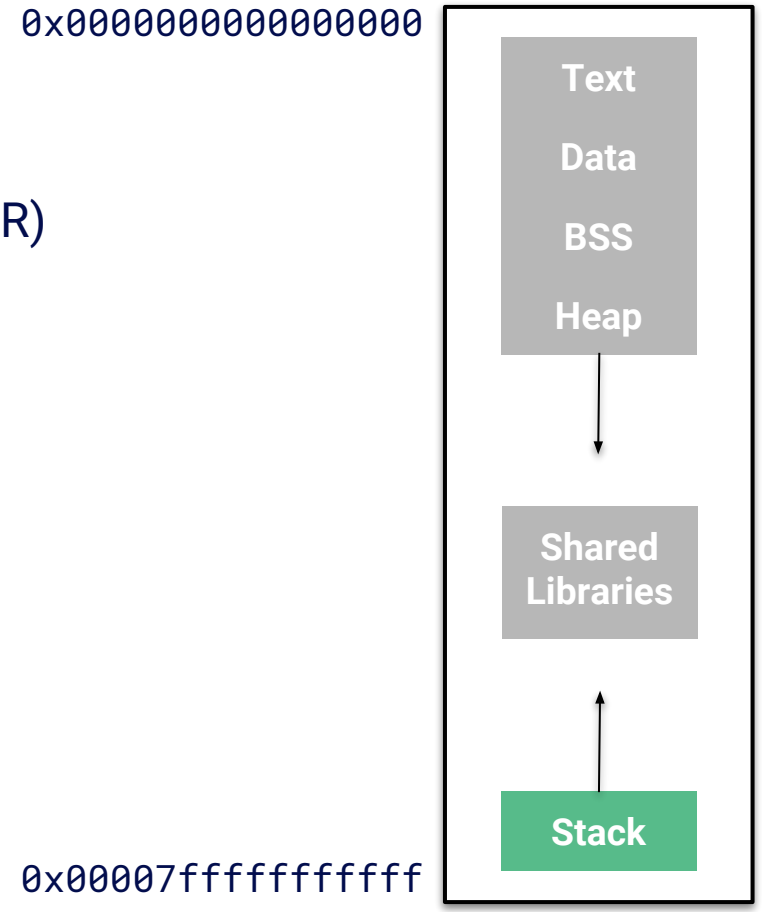
Mitigations

- Address Space Layout Randomization (ASLR)
  - Randomize locations of different memory regions
  - => Need an information leak to bypass



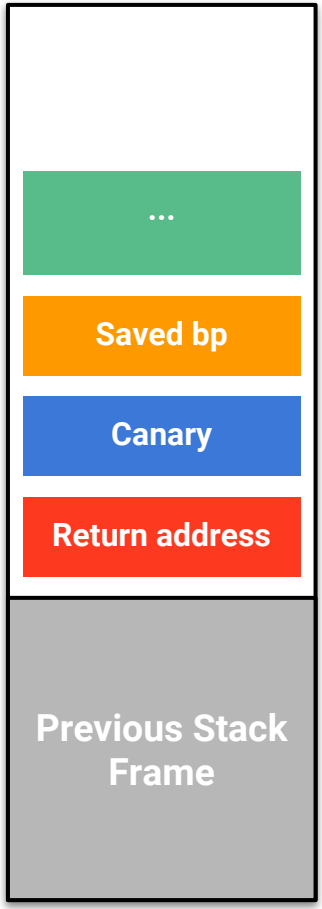
Mitigations

- Address Space Layout Randomization (ASLR)
  - Randomize locations of different memory regions
  - => Need an information leak to bypass
- Non-Executable Stack (NX)
  - Readable | Writeable | ~~Executable~~
  - => Use return-oriented programming (ROP)



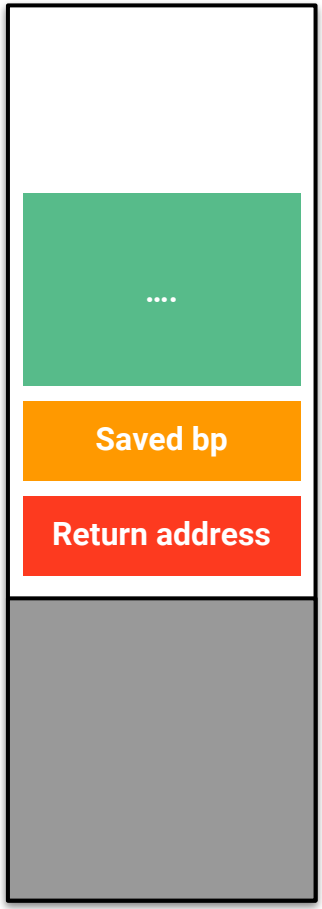
Mitigations

- Address Space Layout Randomization (ASLR)
  - Randomize locations of different memory regions
  - => Need an information leak to bypass
- Non-Executable Stack (NX)
  - Readable | Writeable | ~~Executable~~
  - => Use return-oriented programming (ROP)
- Stack Canaries
  - Place a random value right before the return address
  - Compare to its initial value before returning
  - Terminate program if they don't match
  - => Either need to leak the canary or have non-linear overflow



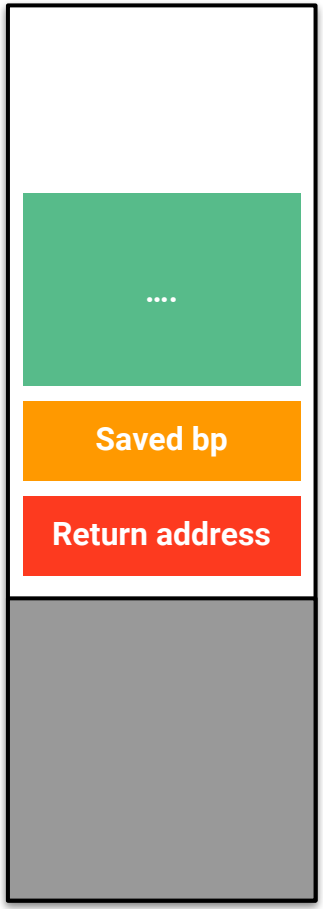
Return-Oriented Programming

- If the nx bit is set, we can’t execute code we write :’(



Return-Oriented Programming

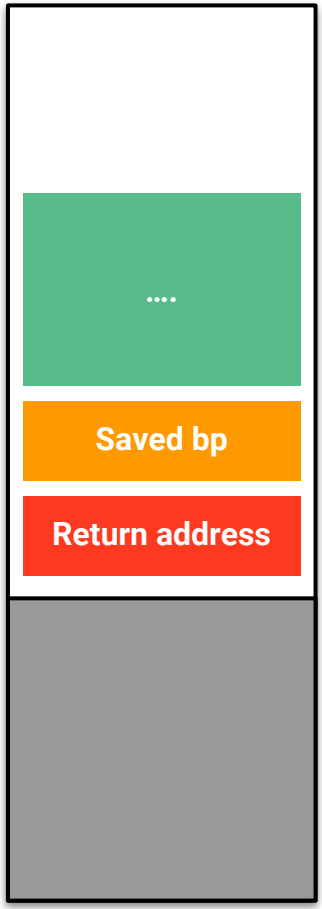
- If the nx bit is set, we can’t execute code we write :’(
- => Reuse existing code



Return-Oriented Programming

- If the nx bit is set, we can't execute code we write :'(
- => Reuse existing code
- To call system("/bin/sh") on x64
  - Have a pointer to "/bin/sh" in RDI
  - Call system function
- Reminder: Function epilogue:

```
(1) mov rsp, rbp
(2) pop rbp
(3) ret (pop rip)
```

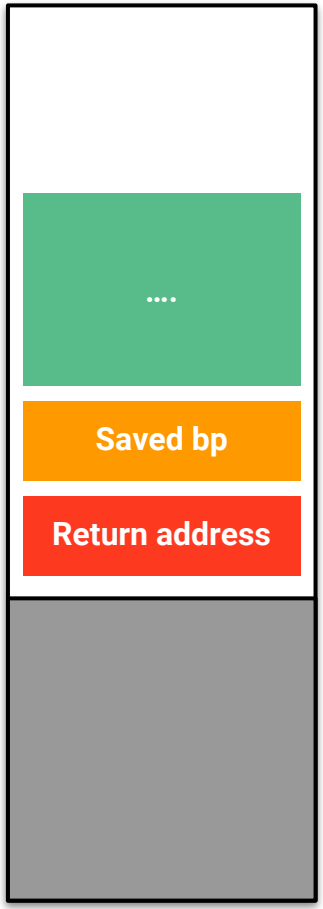


Return-Oriented Programming

Goal:

- Have a pointer to “/bin/sh” in RDI
- Call system function

```
(1) mov rsp, rbp
(2) pop rbp
(3) ret (pop rip)
```



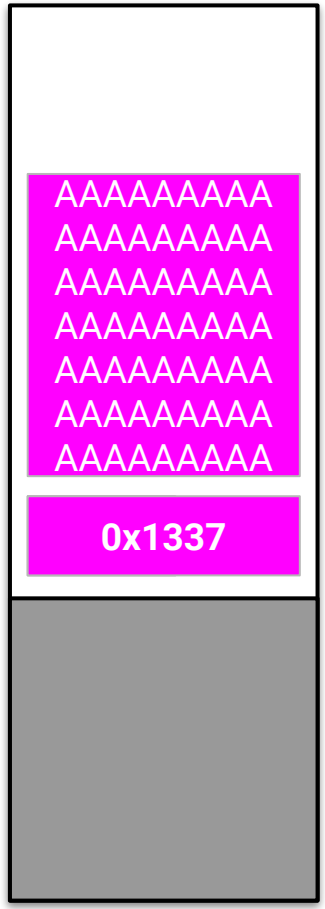
Return-Oriented Programming

Goal:

- Have a pointer to “/bin/sh” in RDI
- Call system function

=> Find a code block with **pop rdi; ret** (assume its address is 0x1337)

```
(1) mov rsp, rbp
(2) pop rbp
(3) ret (pop rip)
```



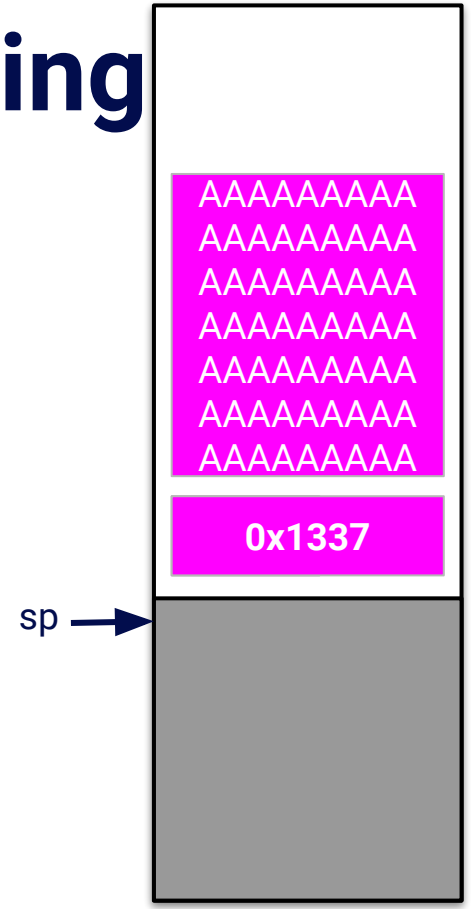
Return-Oriented Programming

Goal:

- Have a pointer to “/bin/sh” in RDI
- Call system function

=> Find a code block with **pop rdi; ret** (assume its address is 0x1337)

```
(1) mov rsp, rbp
(2) pop rbp
(3) ret (pop rip)
```



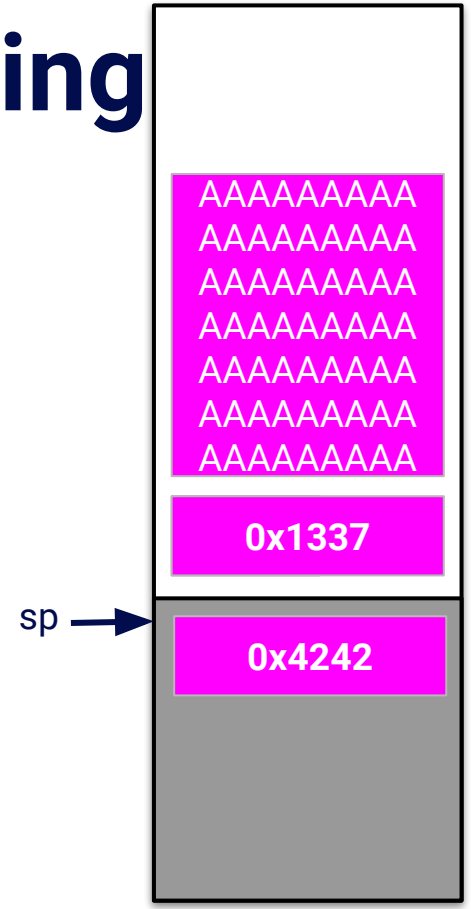
Return-Oriented Programming

Goal:

- Have a pointer to “/bin/sh” in RDI
- Call system function

=> Find a code block with **pop rdi; ret** (assume its address is 0x1337)
=> Write address of **“/bin/sh”** after it (assume its address is 0x4242)

```
(1) mov rsp, rbp
(2) pop rbp
(3) ret (pop rip)
```



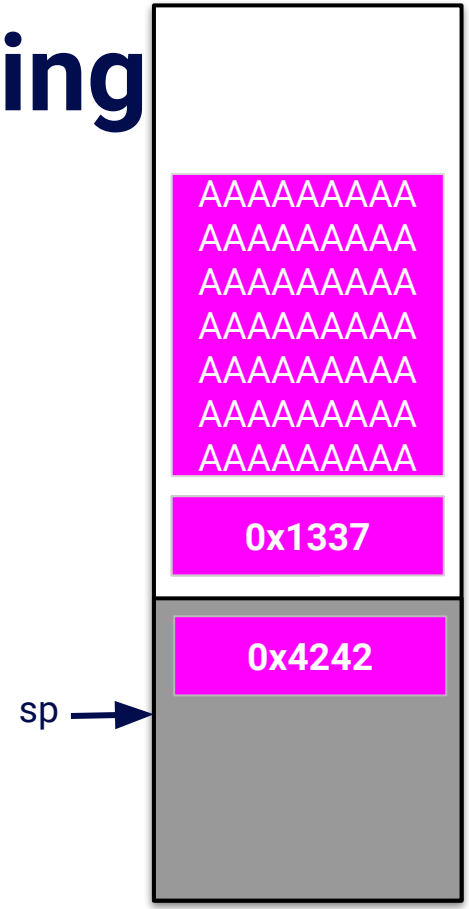
Return-Oriented Programming

Goal:

- Have a pointer to “/bin/sh” in RDI
- Call system function

=> Find a code block with **pop rdi; ret** (assume its address is 0x1337)
=> Write address of **“/bin/sh”** after it (assume its address is 0x4242)

```
(1) mov rsp, rbp
(2) pop rbp
(3) ret (pop rip)
```



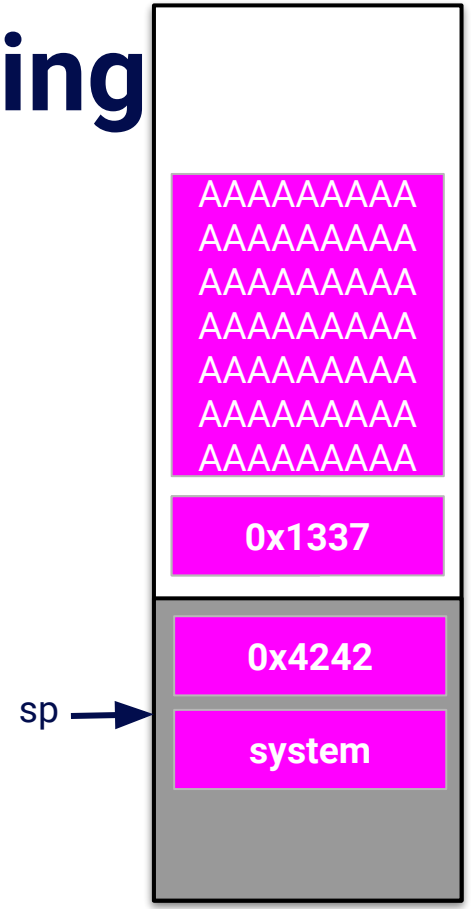
Return-Oriented Programming

Goal:

- Have a pointer to “/bin/sh” in RDI
- Call system function

=> Find a code block with **pop rdi; ret** (assume its address is 0x1337)
=> Write address of **“/bin/sh”** after it (assume its address is 0x4242)
=> Write address of system after it

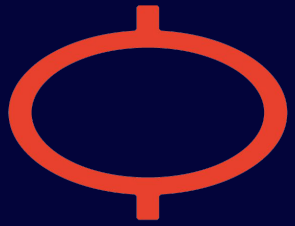
```
(1) mov rsp, rbp
(2) pop rbp
(3) ret (pop rip)
```




@LateNightSeth
BOOM. EASY AS THAT.

Resources

- https://ctftime.org <= All you need to know about upcoming CTFs
- http://overthewire.org <= Lots of wargames
- https://picoctf.com/ <= Very beginner friendly CTF, running all year long



HACKERS WANTED.

jobs@freiheit.com