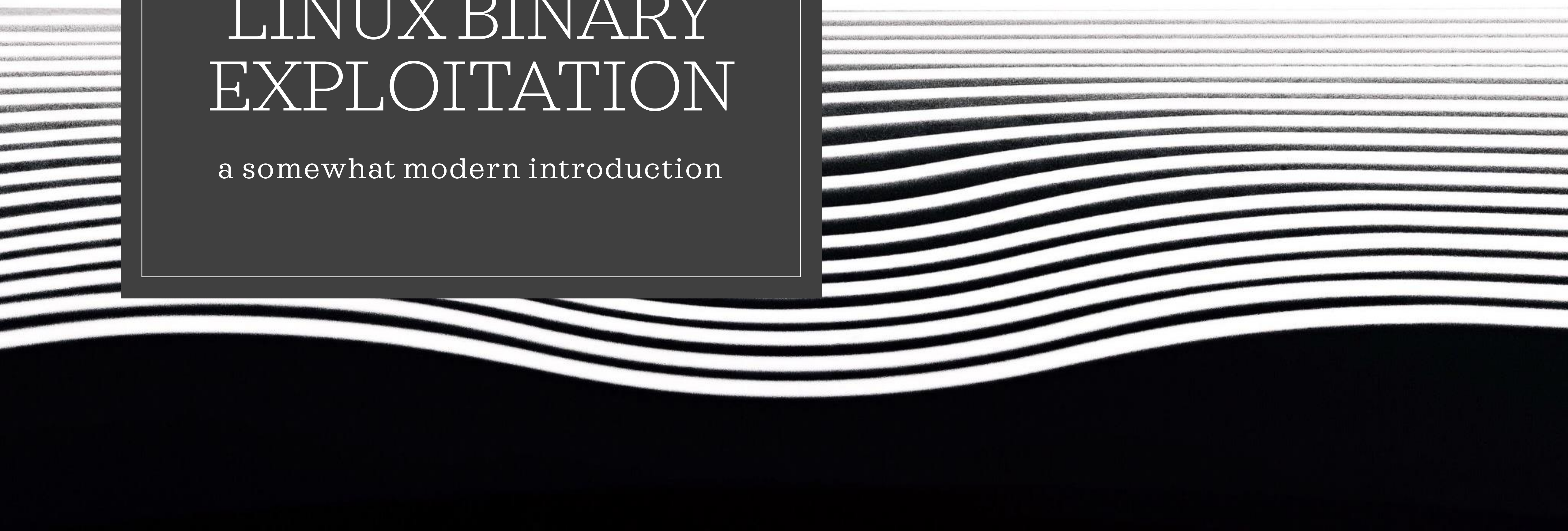
LINUX BINARY EXPLOITATION

a somewhat modern introduction



What to expect?

- This is an introduction to modern 64-bit exploitation on Linux, with all strings attached
- Being an introduction, there is emphasis on the fundamentals
- BinExp is practice-heavy – this is a hands-on class
- After this class you should feel comfortable with low-level stuff and debugger enough to be able to continue your own adventure without extra headaches. However,

you WILL need to practice to git gud ;]

What we are NOT going to cover?

- Windows exploitation and its’ quirks
- Network exploitation
- Reverse engineering*
- Heap abuse
- Race conditions, JOPs, COPs
- Linux-specific quirks (environmental variables, GOT overwrites etc.)
    - I do plan on adding some of these in the future

*there will be emphasis on understanding underlying assembly and how it translates to C, which is definitely an introduction into the topic.

What are we going to cover?

- Assembly primer
- GDB primer
- Memory corruption
- Stack-based exploitation
- Shellcoding primer
- Exploit-mitigation techniques and relevant bypasses
    - Stack Cookies, NX, PIE/ASLR

Before we begin

- Do not be discouraged if you feel like you are falling behind. It takes effort and some time for us to get familiar with low-level thinking. It is going to be awkward until it just “clicks” and becomes easy :]
- Slides and handout will stay shared, don’t waste time making excessive notes – you might miss on things.
- ASK QUESTIONS
- really, don’t be afraid. The worst that can happen is that you’ll learn something new.

10 minutes

SETUP TIME

Download repo, build docker image, make sure everything works correctly

Link: <link>

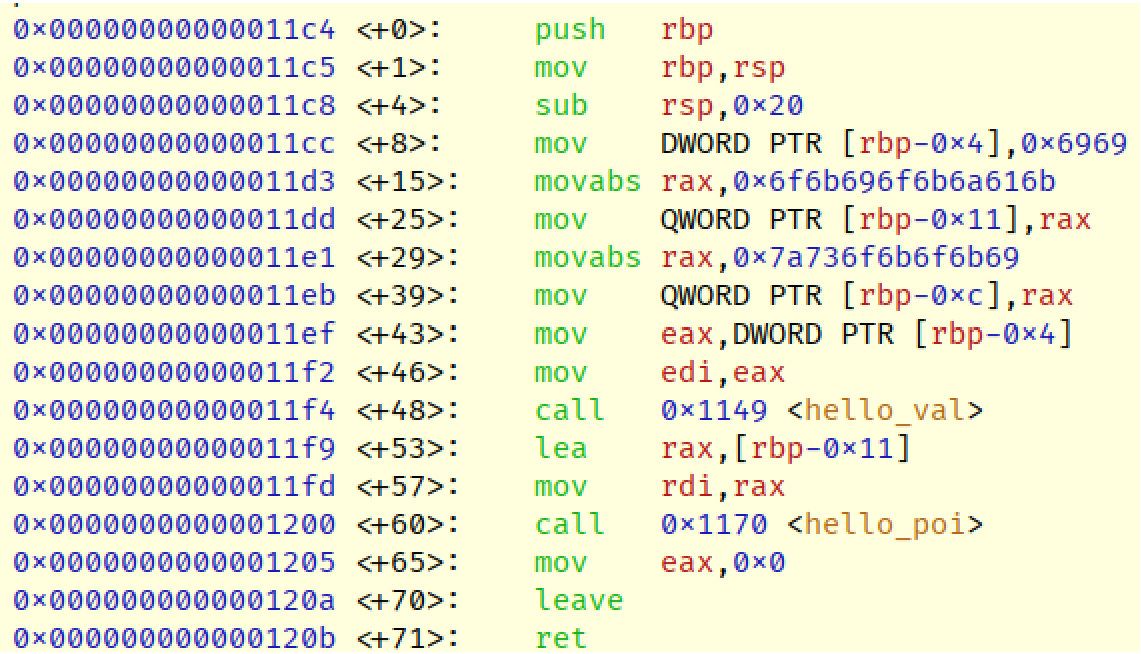
ASSEMBLY PRIMER

What assembly is NOT

- Magic indecipherable language requiring forbidden knowledge to understand

So, assembly

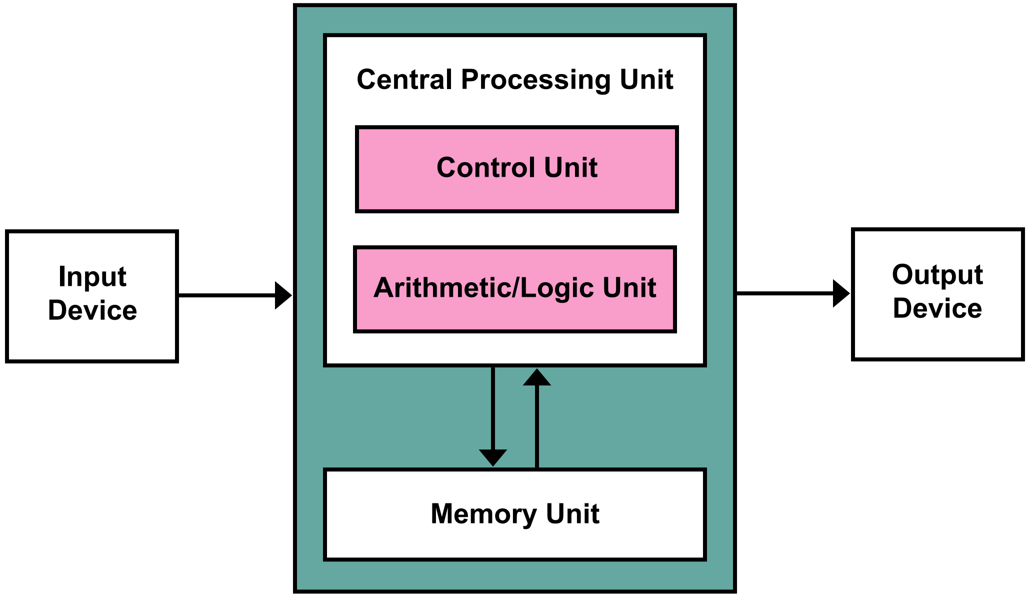
- Is the lowest level human-understandable representation of machine code, it translates directly into instructions getting executed
- Shows what is ACTUALLY being executed by the processor
- There are not many abstractions here
- It might feel different though
  - Architectural decisions (instruction set)
  - OS considerations
  - Calling conventions
  - Compiler optimizations
- We will get to that



```
0x00000000000011c4 <+0>:     push   rbp
0x00000000000011c5 <+1>:     mov    rbp,rsp
0x00000000000011c8 <+4>:     sub    rsp,0x20
0x00000000000011cc <+8>:     mov    DWORD PTR [rbp-0x4],0x6969
0x00000000000011d3 <+15>:    movabs rax,0x6f6b696f6b6a616b
0x00000000000011dd <+25>:    mov    QWORD PTR [rbp-0x11],rax
0x00000000000011e1 <+29>:    movabs rax,0x7a736f6b6f6b69
0x00000000000011eb <+39>:    mov    QWORD PTR [rbp-0xc],rax
0x00000000000011ef <+43>:    mov    eax,DWORD PTR [rbp-0x4]
0x00000000000011f2 <+46>:    mov    edi,eax
0x00000000000011f4 <+48>:    call   0x1149 <hello_val>
0x00000000000011f9 <+53>:    lea    rax,[rbp-0x11]
0x00000000000011fd <+57>:    mov    rdi,rax
0x0000000000001200 <+60>:    call   0x1170 <hello_poi>
0x0000000000001205 <+65>:    mov    eax,0x0
0x000000000000120a <+70>:    leave
0x000000000000120b <+71>:    ret
```

Processor

- Is at the same time more and less complicated than it looks ;]
- For our purposes, however, it is deceptively simple
- A predictable black-box doing our bidding through small, discrete instructions
  - Put instruction and, optionally, some data into it
  - Receive result
  - Rinse and repeat

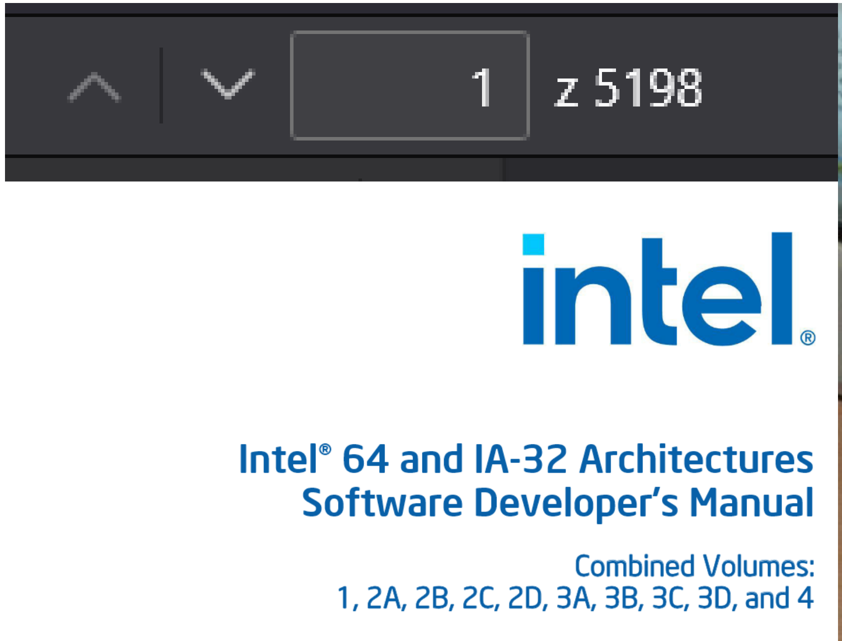


Von Neumann Architecture, courtesy of Wikipedia

By Kapooht - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=25789639

How do we interact with processor?

- We specify *instruction (opcode)*, which tells what operation should be performed
- Data is passed and received through set of *registers* and/or memory *pointers*
- Available instructions, data types, registers, addressing modes, and memory are all specified in ISAs (Instruction Set Architecture)
- Most of the desktop world is running on AMD64 ISA, also known as x86-64, x64, Intel 64
    - We will be working with this one!
- It is an extension of x86 ISA, which was base of most of the 32-bit systems a while ago
- Other ISAs exist, most important are ARM64 and RISC-V
- x64 is a painfully bloated instruction set (think, modern C++). Luckily for our purposes we do not have to nerd over most of them



1 z 5198

intel®

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4

intel
Intel® 64 and IA-32 Architectures Optimization Reference Manual
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 1: Basic Architecture
1
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 3B: System Programming Guide, Part 2
3B
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 3A: System Programming Guide, Part 1
3A
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 2B: Instruction Set Reference, N-Z
2B
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 2B: Instruction Set Reference, A-M
2A

Intel® 64 and IA-32 Architectures
Software Developer's Manual
VOLUME 2B: Instruction Set
Reference, N-Z
Intel® 64 and IA-32 Architectures Software Developer's Manual
2B
1
3B
3A

All this just to be able to use the processor, lmao

Everything is data!

An important distinction to keep in mind as we go through the content, there is nothing inherently different between data and code from the perspective of a processor.

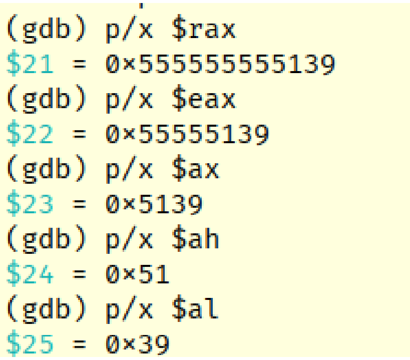
The implications might not be obvious now, just consider that anything passed to the processor can be interpreted as both code to be executed and/or data to be manipulated. They are both just binary streams anyways.

Registers

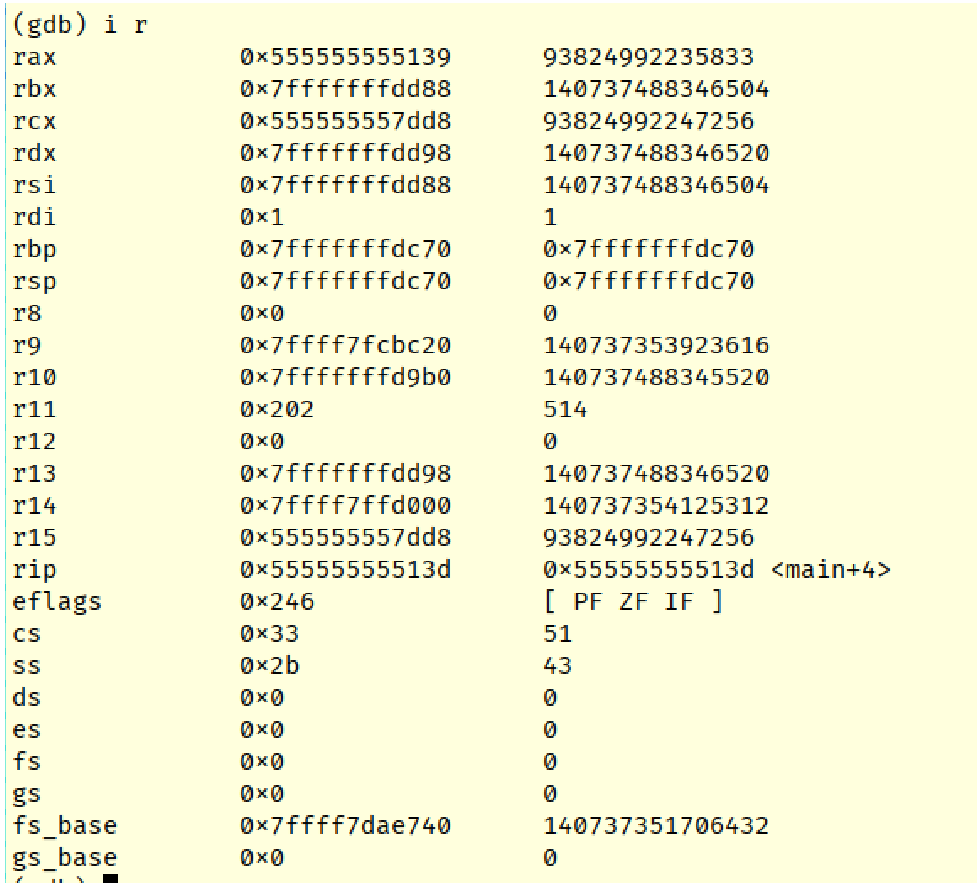
- Represent internal state of the processor, fundamental to understand.
- We are in 64-bit architecture; hence all registers are 8-byte long
- It is possible to use only a subset of available space in each register

Example:

- RAX – 8 bytes
- EAX – 4 bytes
- AX – 2 bytes
- AH – higher byte of AX
- AL – lower byte of AX



```
(gdb) p/x $rax
$21 = 0x555555555139
(gdb) p/x $eax
$22 = 0x55555139
(gdb) p/x $ax
$23 = 0x5139
(gdb) p/x $ah
$24 = 0x51
(gdb) p/x $al
$25 = 0x39
```



```
(gdb) i r
rax            0x555555555139      93824992235833
rbx            0x7fffffffdd88      140737488346504
rcx            0x555555557dd8      93824992247256
rdx            0x7fffffffdd98      140737488346520
rsi            0x7fffffffdd88      140737488346504
rdi            0x1                 1
rbp            0x7fffffffdc70      0x7fffffffdc70
rsp            0x7fffffffdc70      0x7fffffffdc70
r8             0x0                 0
r9             0x7ffff7fcbc20      140737353923616
r10            0x7fffffffd9b0      140737488345520
r11            0x202               514
r12            0x0                 0
r13            0x7fffffffdd98      140737488346520
r14            0x7ffff7ffd000      140737354125312
r15            0x555555557dd8      93824992247256
rip            0x55555555513d      0x55555555513d <main+4>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
fs_base        0x7ffff7dae740      140737351706432
gs_base        0x0                 0
```

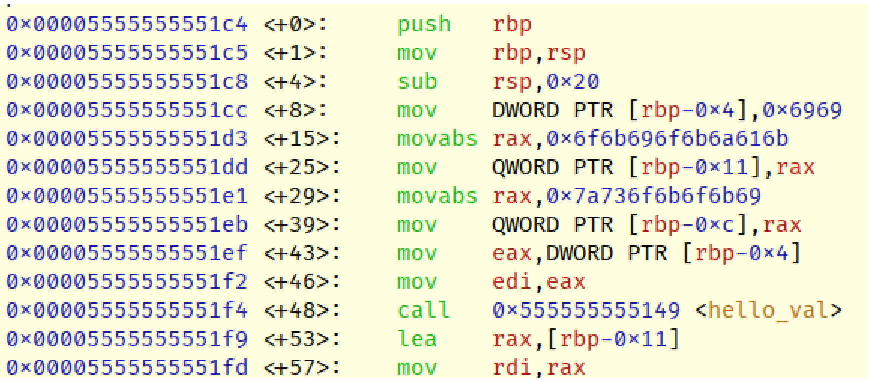
Registers – ones to care about

Like in life, some are more privileged than the others:

- RIP – Instruction Pointer.
  - Points to the next instruction that will be executed. Easily the most important register out there.
- RAX – Accumulator
  - Most frequently used data register. Used for returning data from function calls and to request particular syscall number.
- RSP – Stack Pointer
  - Points to where top of the stack is currently. Extremely important for exploitation and ensuring program works correctly. More on stack later.
- RBP – Base Pointer
  - Less useful than RSP, still important. Used to track base of a stack frame. More on stack later.
- RDI, RSI, RDX, RCX, R8, R9
  - In that particular order. These are used to pass arguments to function calls (Linux convention!)
- All other registers are general-purpose. There are some conventions (like RCX being mainly used for maths), but they have absolutely no extra meaning from our perspective.

Instructions

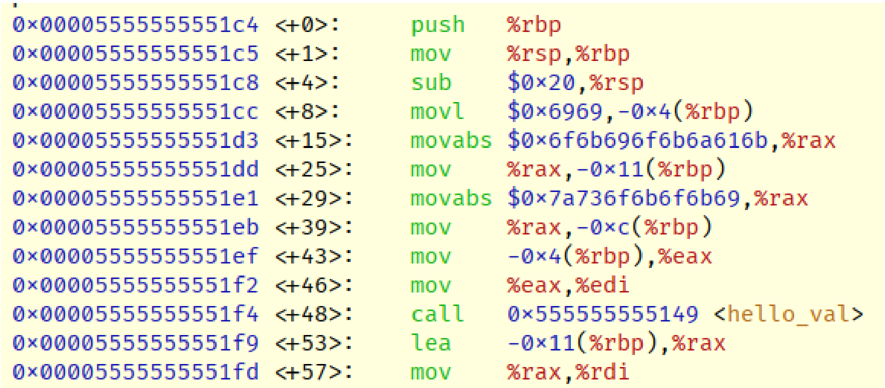
- Instruction consists of an opcode and arguments depending on the opcode used
- Most of the instructions can take arguments of different type or size
- Processor executes one instruction at a time
- x64 has variable instruction length – meaning that different instructions have different lengths.
  - This leads to interesting behaviors. For instance, instruction will be interpreted differently if executed from the middle of its' constituent raw bytes
  - ARM is different, they have fixed-length instruction size.
- There are two major syntax flavors – AT&T and Intel. We are going to use Intel's throughout the course.



```
0x00005555555551c4 <+0>:     push   rbp
0x00005555555551c5 <+1>:     mov    rbp,rsp
0x00005555555551c8 <+4>:     sub    rsp,0x20
0x00005555555551cc <+8>:     mov    DWORD PTR [rbp-0x4],0x6969
0x00005555555551d3 <+15>:    movabs rax,0x6f6b696f6b6a616b
0x00005555555551dd <+25>:    mov    QWORD PTR [rbp-0x11],rax
0x00005555555551e1 <+29>:    movabs rax,0x7a736f6b6f6b69
0x00005555555551eb <+39>:    mov    QWORD PTR [rbp-0xc],rax
0x00005555555551ef <+43>:    mov    eax,DWORD PTR [rbp-0x4]
0x00005555555551f2 <+46>:    mov    edi,eax
0x00005555555551f4 <+48>:    call   0x555555555149 <hello_val>
0x00005555555551f9 <+53>:    lea    rax,[rbp-0x11]
0x00005555555551fd <+57>:    mov    rdi,rax
```

<- Intel

AT&T->



```
0x00005555555551c4 <+0>:     push   %rbp
0x00005555555551c5 <+1>:     mov    %rsp,%rbp
0x00005555555551c8 <+4>:     sub    $0x20,%rsp
0x00005555555551cc <+8>:     movl   $0x6969,-0x4(%rbp)
0x00005555555551d3 <+15>:    movabs $0x6f6b696f6b6a616b,%rax
0x00005555555551dd <+25>:    mov    %rax,-0x11(%rbp)
0x00005555555551e1 <+29>:    movabs $0x7a736f6b6f6b69,%rax
0x00005555555551eb <+39>:    mov    %rax,-0xc(%rbp)
0x00005555555551ef <+43>:    mov    -0x4(%rbp),%eax
0x00005555555551f2 <+46>:    mov    %eax,%edi
0x00005555555551f4 <+48>:    call   0x555555555149 <hello_val>
0x00005555555551f9 <+53>:    lea    -0x11(%rbp),%rax
0x00005555555551fd <+57>:    mov    %rax,%rdi
```

Instructions - basics

Instructions with arguments follow the general structure of:

OPCODE <DESTINATION>, <SOURCE>

Arguments to the instructions can be of three types:

- Immediate values –a raw value that will be processed
- Registers – name of the register, whose value is to be used
- Pointer – a value or register containing address pointing to the actual value

Instructions - examples

Moving immediate value into RAX

MOV RAX, 0x4041424344454647

Adding RDX to RAX and storing result in RAX.

ADD RAX, RDX

Doubling the value of AX. Remember, AX is much smaller than RAX (2 bytes long).

ADD AX, AX

Moving value stored at address pointed to by RAX into RAX

MOV RAX, [RAX]

=?=

MOV RAX, QWORD PTR [RAX]

A word on words

The last example might feel confusing without additional explanation:

MOV RAX, QWORD PTR [RAX]

QWORD PTR [RAX] -> take a value pointed at by RAX and interpret it as a quad WORD pointer (address to data that is 8 bytes long).

Words are just a way of specifying size of data in computing. In x86/x64 ISA, the values are as follows:

- BYTE – 8 bits
- WORD – 16 bits
- DWORD (double WORD) – 32 bits
- QWORD (quad WORD) – 64 bits

Circling back to our example, it means that value pointed at by RAX is 8 bytes long :]

A word on words

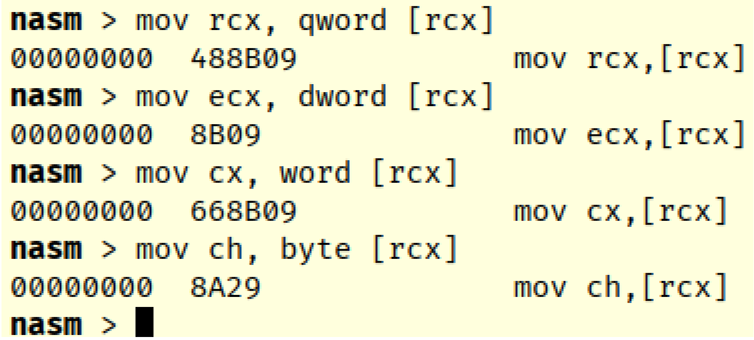
Word sizes directly map to registry sizes, using RCX as an example:

BYTE – CL or CH

WORD – CX

DWORD – ECX

QWORD - RCX



```
nasm > mov rcx, qword [rcx]
00000000  488B09            mov rcx,[rcx]
nasm > mov ecx, dword [rcx]
00000000  8B09              mov ecx,[rcx]
nasm > mov cx, word [rcx]
00000000  668B09            mov cx,[rcx]
nasm > mov ch, byte [rcx]
00000000  8A29              mov ch,[rcx]
nasm > █
```

Fundamental Instructions - MOV

MOVes (actually copies) the value from the SRC to the DST

MOV DST, SRC

MOV accepts all three types of arguments – immediate, registers and pointers

Examples:

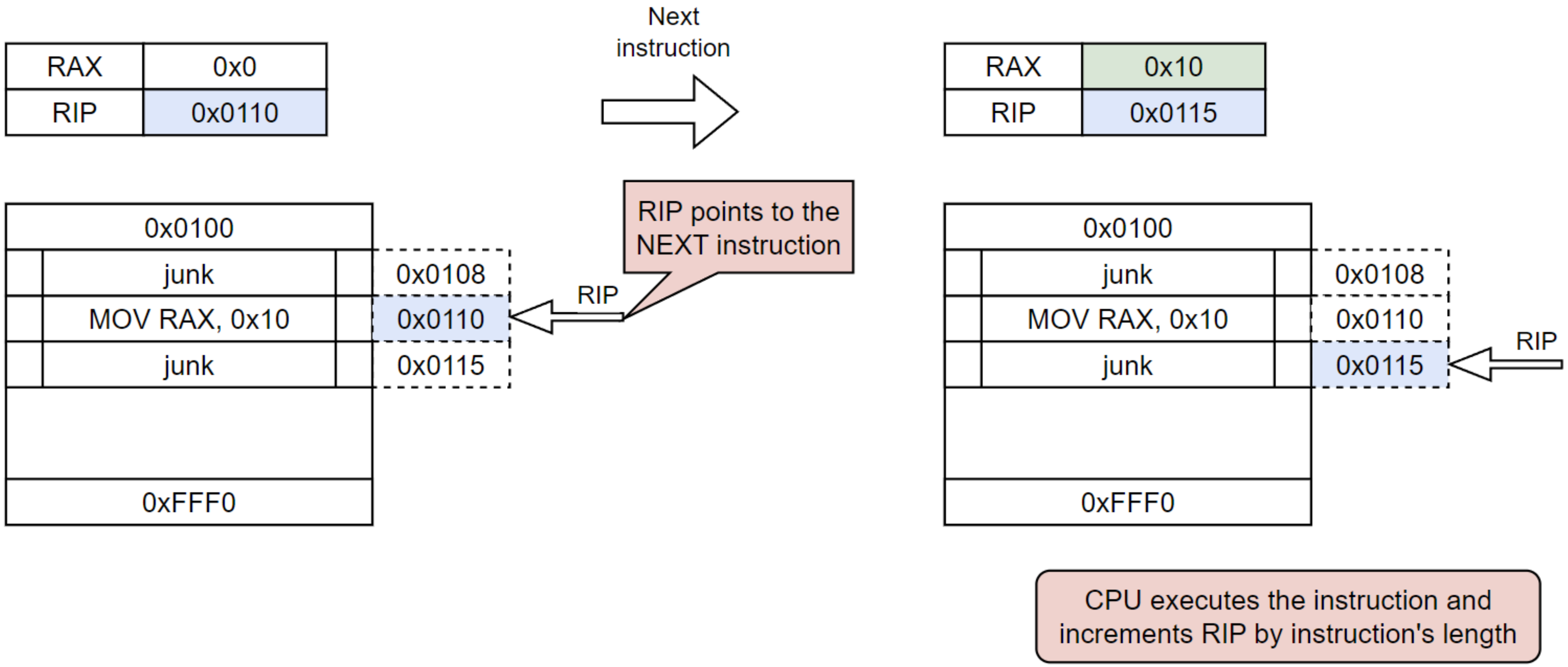
MOV RAX, 0x10

MOV RDI, RAX

MOV RAX, [RBX]

Fundamental Instructions - MOV

Immediate value example



RAX	0x0
RIP	0x0110

Memory	Address
0x0100	
junk	0x0108
MOV RAX, 0x10	0x0110
junk	0x0115
0xFFF0	

RIP points to the NEXT instruction

RAX	0x10
RIP	0x0115

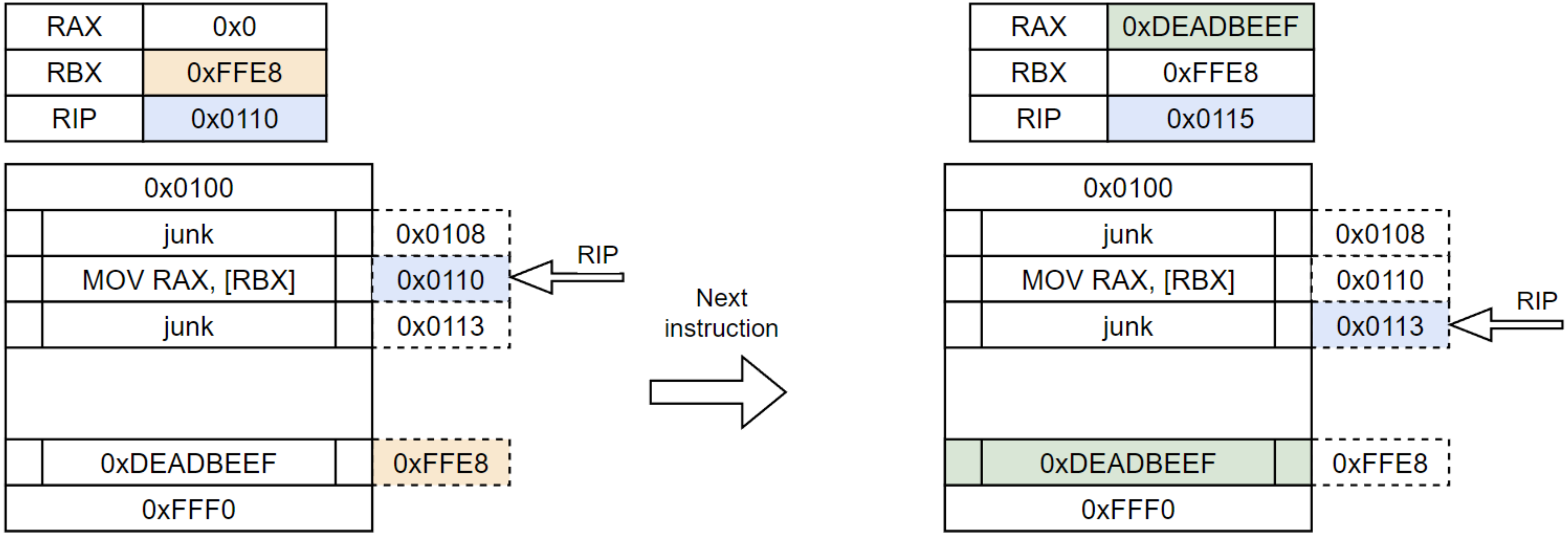
Memory	Address
0x0100	
junk	0x0108
MOV RAX, 0x10	0x0110
junk	0x0115
0xFFF0	

CPU executes the instruction and increments RIP by instruction's length

Fundamental Instructions - MOV

Pointer example

Address in RBX is used to fetch the actual value. Such dereference is denoted with square brackets.



RAX	0x0
RBX	0xFFE8
RIP	0x0110

0x0100	
junk	0x0108
MOV RAX, [RBX]	0x0110
junk	0x0113
0xDEADBEEF	0xFFE8
0xFFF0	

RIP

Next instruction

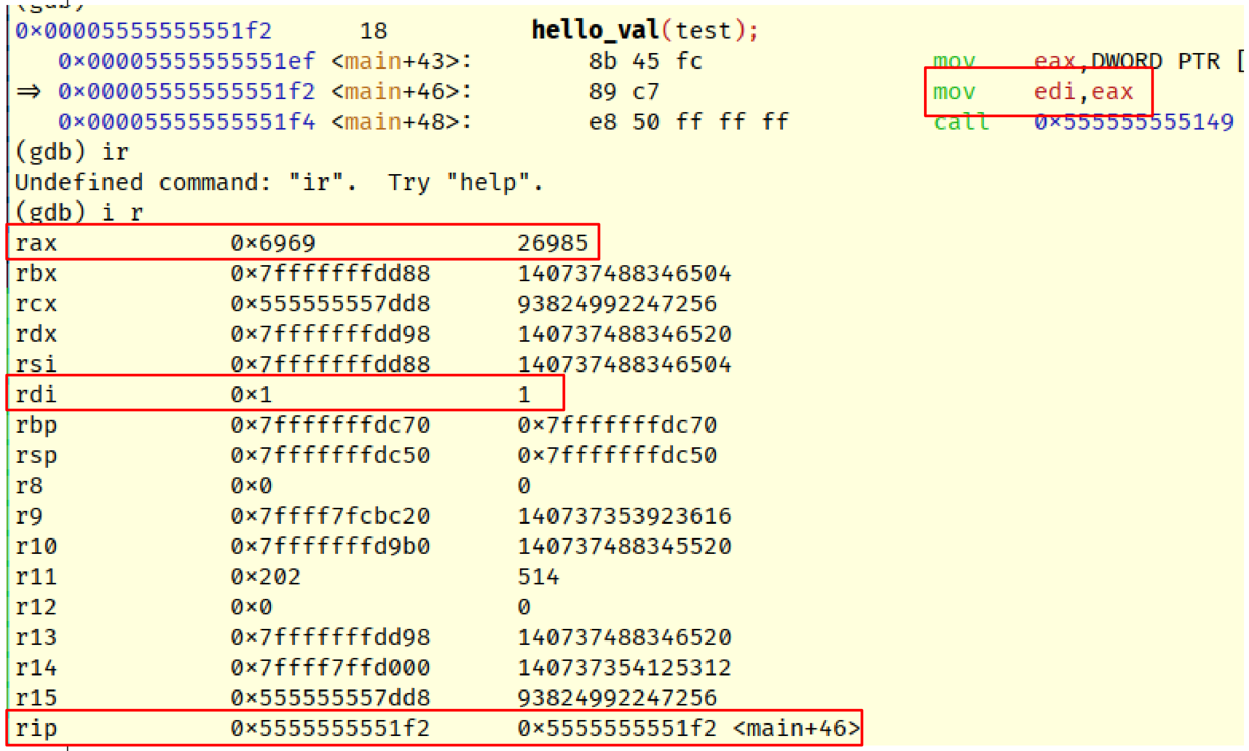
RAX	0xDEADBEEF
RBX	0xFFE8
RIP	0x0115

0x0100	
junk	0x0108
MOV RAX, [RBX]	0x0110
junk	0x0113
0xDEADBEEF	0xFFE8
0xFFF0	

RIP

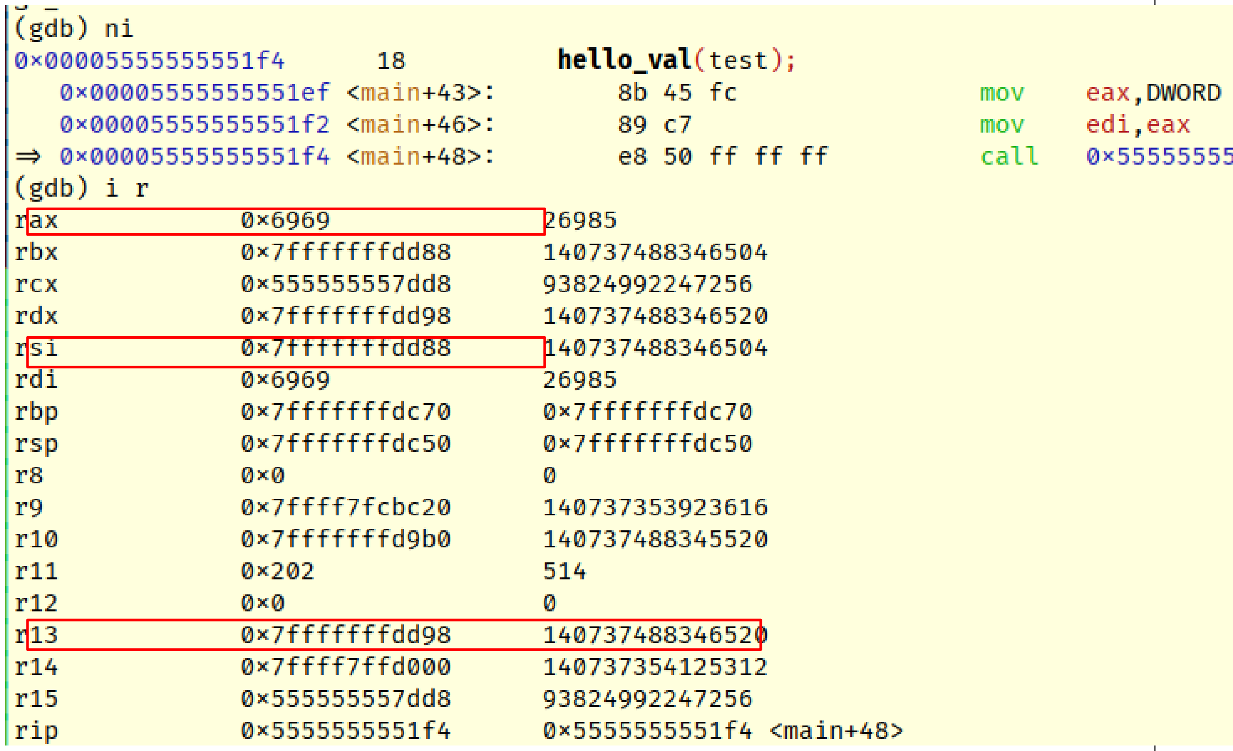
Fundamental Instructions - MOV

Register example from actual debugger



```
0x00005555555551f2      18          hello_val(test);
   0x00005555555551ef <main+43>:       8b 45 fc                    mov     eax,DWORD PTR [
⇒  0x00005555555551f2 <main+46>:       89 c7                       mov     edi,eax
   0x00005555555551f4 <main+48>:       e8 50 ff ff ff              call    0x555555555149
(gdb) ir
Undefined command: "ir".  Try "help".
(gdb) i r
rax            0x6969              26985
rbx            0x7fffffffdd88      140737488346504
rcx            0x555555557dd8      93824992247256
rdx            0x7fffffffdd98      140737488346520
rsi            0x7fffffffdd88      140737488346504
rdi            0x1                 1
rbp            0x7fffffffdc70      0x7fffffffdc70
rsp            0x7fffffffdc50      0x7fffffffdc50
r8             0x0                 0
r9             0x7ffff7fcbc20      140737353923616
r10            0x7fffffffd9b0      140737488345520
r11            0x202               514
r12            0x0                 0
r13            0x7fffffffdd98      140737488346520
r14            0x7ffff7ffd000      140737354125312
r15            0x555555557dd8      93824992247256
rip            0x5555555551f2      0x5555555551f2 <main+46>
```

->



```
(gdb) ni
0x00005555555551f4      18          hello_val(test);
   0x00005555555551ef <main+43>:       8b 45 fc                    mov     eax,DWORD
   0x00005555555551f2 <main+46>:       89 c7                       mov     edi,eax
⇒  0x00005555555551f4 <main+48>:       e8 50 ff ff ff              call    0x5555555
(gdb) i r
rax            0x6969              26985
rbx            0x7fffffffdd88      140737488346504
rcx            0x555555557dd8      93824992247256
rdx            0x7fffffffdd98      140737488346520
rsi            0x7fffffffdd88      140737488346504
rdi            0x6969              26985
rbp            0x7fffffffdc70      0x7fffffffdc70
rsp            0x7fffffffdc50      0x7fffffffdc50
r8             0x0                 0
r9             0x7ffff7fcbc20      140737353923616
r10            0x7fffffffd9b0      140737488345520
r11            0x202               514
r12            0x0                 0
r13            0x7fffffffdd98      140737488346520
r14            0x7ffff7ffd000      140737354125312
r15            0x555555557dd8      93824992247256
rip            0x5555555551f4      0x5555555551f4 <main+48>
```

Fundamental Instructions - LEA

Load Effective Address – used to perform pointers arithmetic and storing results in the DST

LEA DST, [SRC +/- SRC2*x +/- y]

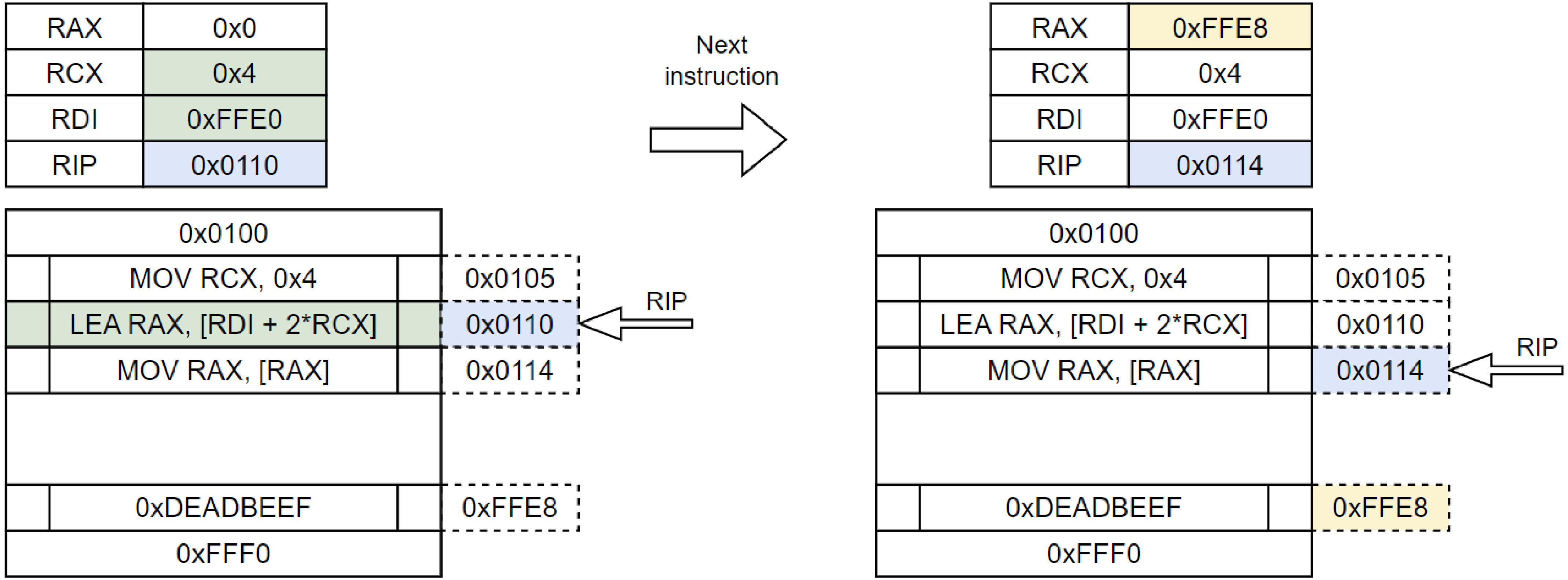
LEA calculates the result of whatever’s in square brackets and store is into DST. It makes calculation much easier by eliminating the need to use intermediate registers.

Examples:

LEA RAX, [RBP - 32]

LEA RAX, [RDI + RSI*4]

Fundamental Instructions – LEA



RAX	0x0
RCX	0x4
RDI	0xFFE0
RIP	0x0110

0x0100	
MOV RCX, 0x4	0x0105
LEA RAX, [RDI + 2*RCX]	0x0110
MOV RAX, [RAX]	0x0114
0xDEADBEEF	0xFFE8
0xFFF0	

RAX	0xFFE8
RCX	0x4
RDI	0xFFE0
RIP	0x0114

0x0100	
MOV RCX, 0x4	0x0105
LEA RAX, [RDI + 2*RCX]	0x0110
MOV RAX, [RAX]	0x0114
0xDEADBEEF	0xFFE8
0xFFF0	

Can you identify what will happen next?

Fundamental Instructions continued

Other important instructions to know:

- ADD/SUB DST, SRC
    - adds/subtracts SRC to/from DST and stores the result in DST
- INC/DEC DST
    - INCrement/DECrement DST by 1
- XCHG DST, SRC
    - eXCHanGe values between DST and SRC
- PUSH/POP
    - put/take data from the stack. We will talk about stack after 1st lab
- CALL/RET
    - call/return from function. We will talk about that when covering stack

Note on endianness

In my experience, one of the most confusing things when coming into low-level exploitation for the first time. Getting it right at the beginning goes a long way.

> *A big-endian system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest. A little-endian system, in contrast, stores the least-significant byte at the smallest address.*
>
> *Of the two, big-endian is thus closer to the way the digits of numbers are written left-to-right in English, comparing digits to bytes.*
>
> ~Wikipedia

Note on endianness

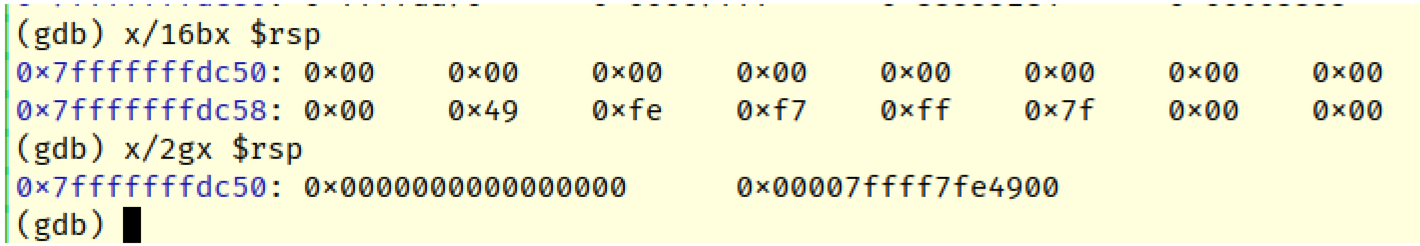
- We, humans, intuitively use big-endian (BE) notation. x86/x64 and many others use little-endian (LE).
- Endianness is important ONLY when it comes down to numbers. However, when inspecting other data types debuggers will also interpret it as number unless specified otherwise. This leads to confusion
  - Numbers are stored “pairwise-reversed” – LE
  - Debuggers by default convert data to BE
  - This is best seen when displaying single bytes vs quad words
  - Just be mindful of this fact when displaying data as numbers

Converting between endiannesses:

0xFFEEDDCC

0xCCDDEEFF

Same piece of data interpreted as single bytes and quad words:



```
(gdb) x/16bx $rsp
0x7fffffffdc50: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdc58: 0x00    0x49    0xfe    0xf7    0xff    0x7f    0x00    0x00
(gdb) x/2gx $rsp
0x7fffffffdc50: 0x0000000000000000      0x00007ffff7fe4900
(gdb) 
```

30 minutes

LAB 1

Observing assembly in GDB

Instruction:

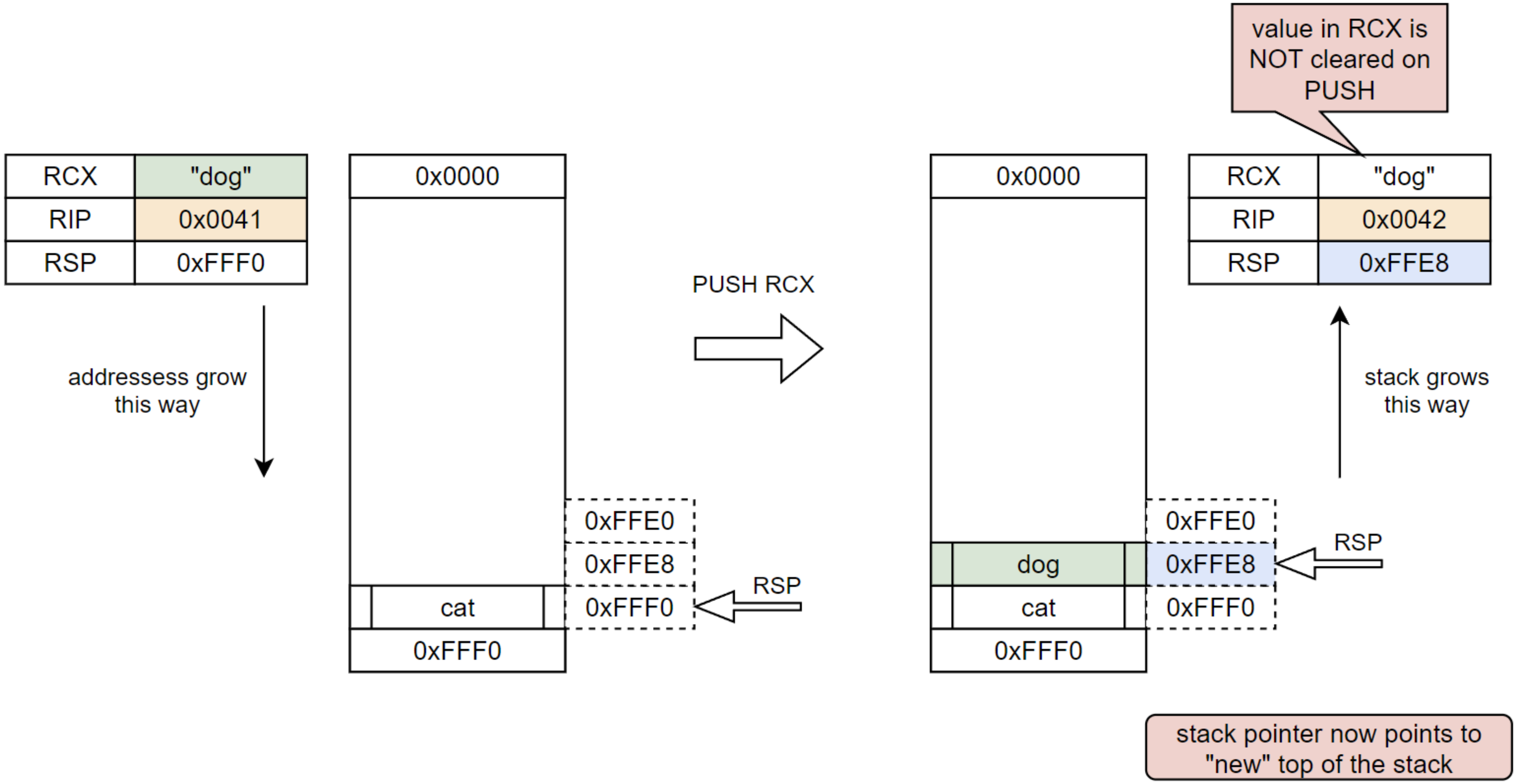
handout/lab1.pdf

Stack

- One of two structures representing program's dynamic memory (other is heap)
- LIFO queue – Last-In-First-Out
- Values are managed by PUSHing them on top of the stack or POPping them from the top
- All elements are 8-byte chunks!
- Keeps track of all function calls, stores local variables and ensures program "knows" where to return from a function.
- Starts at the end of the program's address space and grows upwards (towards lower addresses)

This is can get a little confusing without diagrams…

Stack - PUSHing



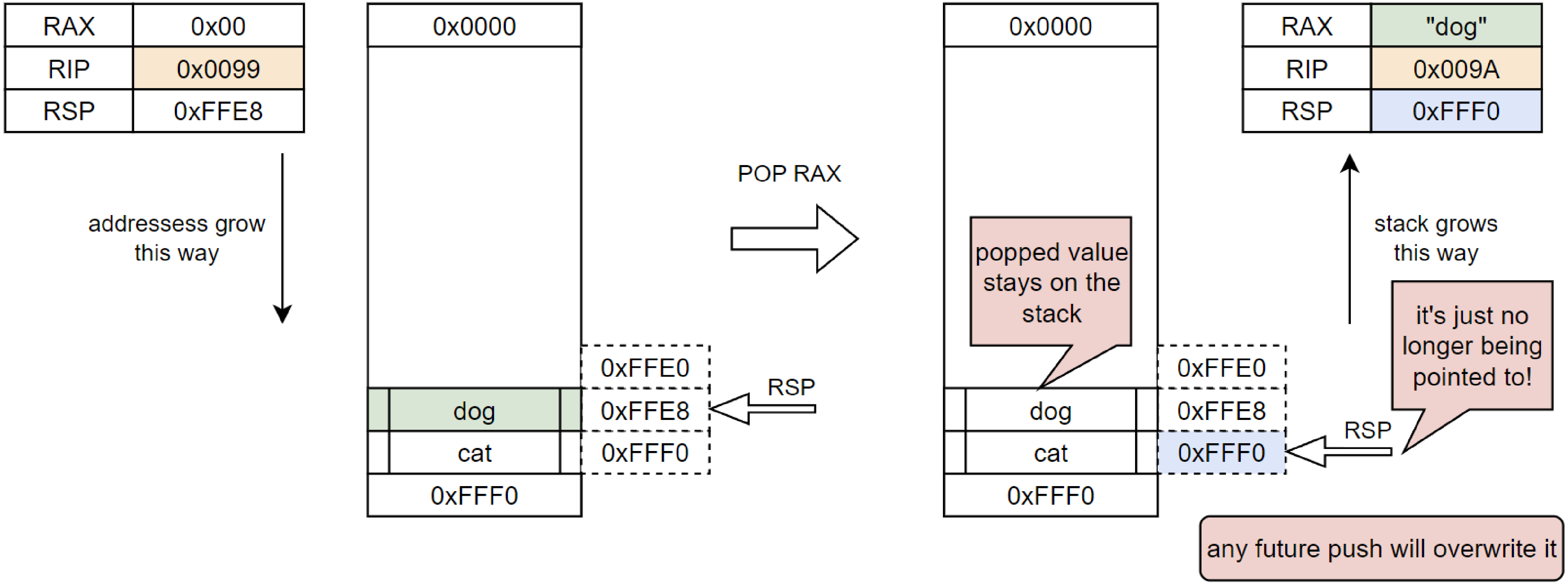
RCX	"dog"
RIP	0x0041
RSP	0xFFF0

RCX	"dog"
RIP	0x0042
RSP	0xFFE8

value in RCX is NOT cleared on PUSH

stack pointer now points to "new" top of the stack

Stack - POPing



RAX	0x00
RIP	0x0099
RSP	0xFFE8

addressess grow this way

0x0000

dog 0xFFE8

cat 0xFFF0

0xFFE0

0xFFF0

RSP

POP RAX

0x0000

popped value stays on the stack

dog 0xFFE8

cat 0xFFF0

0xFFE0

0xFFF0

RSP

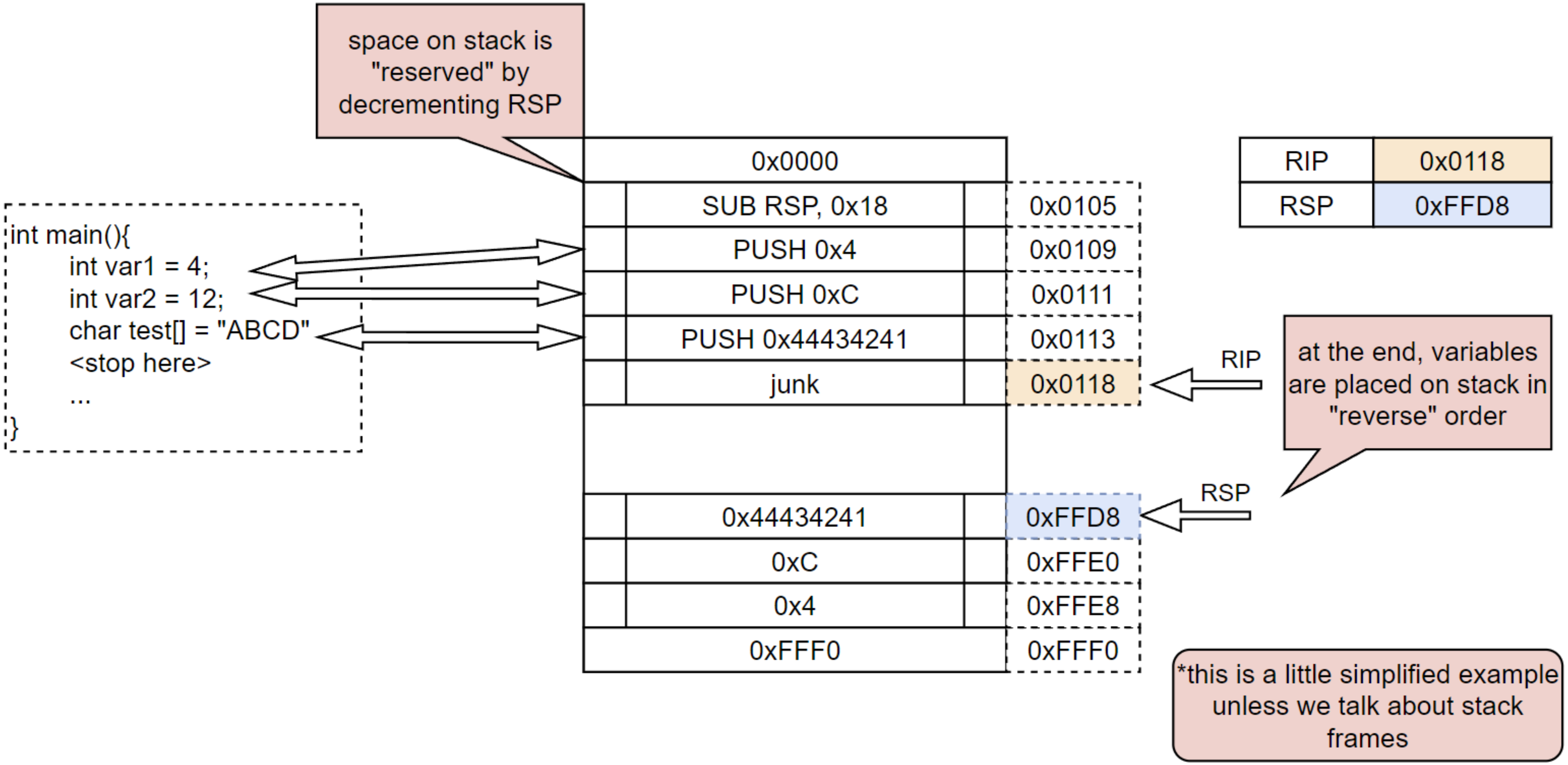
RAX	"dog"
RIP	0x009A
RSP	0xFFF0

stack grows this way

it's just no longer being pointed to!

any future push will overwrite it

Stack – local variables



space on stack is "reserved" by decrementing RSP

```
int main(){
        int var1 = 4;
        int var2 = 12;
        char test[] = "ABCD"
        <stop here>
        ...
}
```

0x0000	
SUB RSP, 0x18	0x0105
PUSH 0x4	0x0109
PUSH 0xC	0x0111
PUSH 0x44434241	0x0113
junk	0x0118
0x44434241	0xFFD8
0xC	0xFFE0
0x4	0xFFE8
0xFFF0	0xFFF0

RIP	0x0118
RSP	0xFFD8

at the end, variables are placed on stack in "reverse" order

*this is a little simplified example unless we talk about stack frames

Stack – important considerations

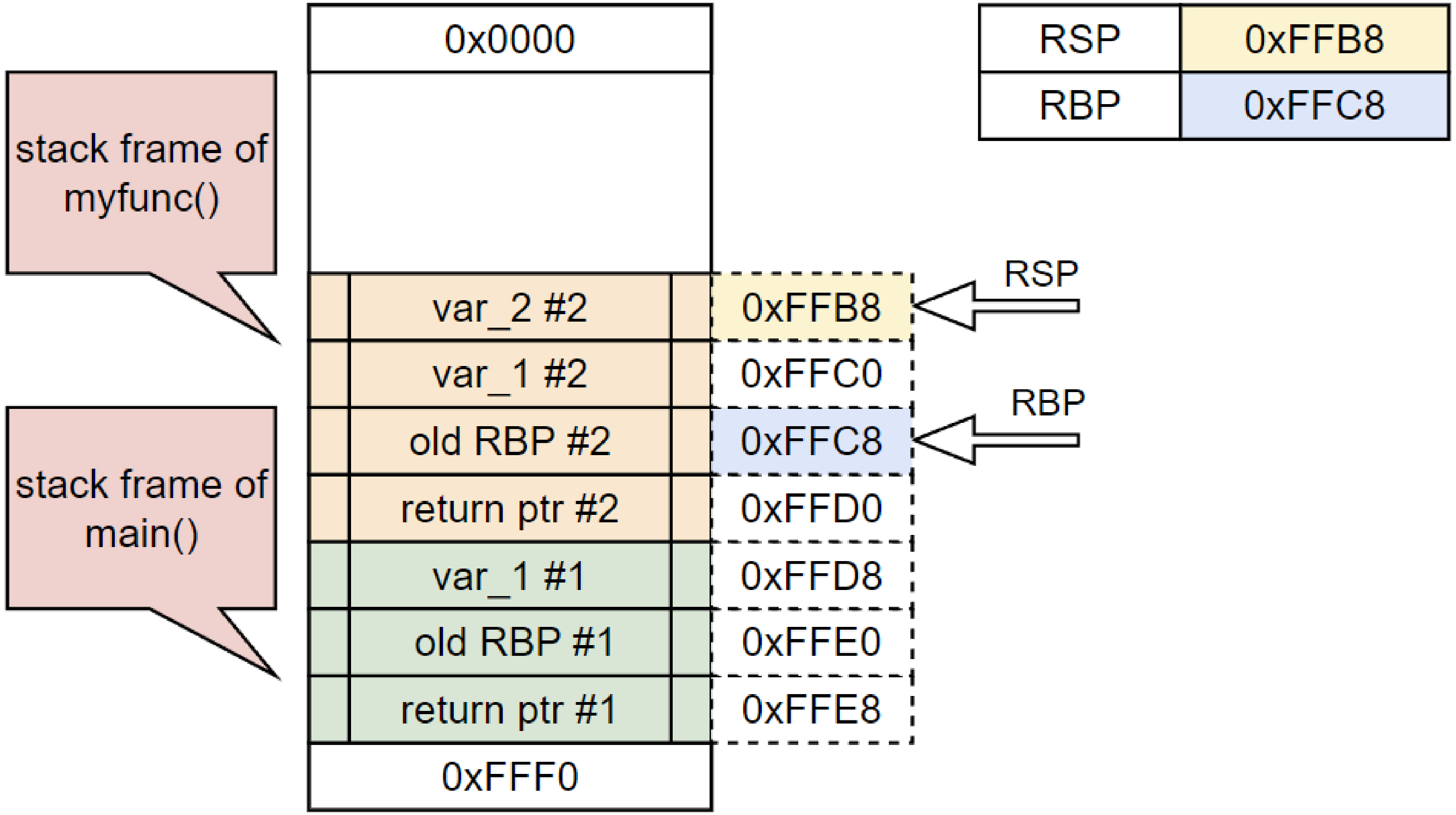
- Stack is where RSP tells it to – in other words, there is absolutely nothing preventing programmer from “moving” stack to any other writable memory region by overwriting RSP.
  - Will it break the program? Most of the time, yes. However, this can be utilized offensively, e.g. with Stack Pivoting technique, which is used to move stack to other memory we have control of.
- While stack grows “downwards” (towards lower addresses), any kind of data access is done by incrementing memory addresses.
  - In some cases this allows attacker to overwrite previously pushed values, leading to memory corruption.

Stack Frames

- A final chapter of assembly primer!
- Stack frames are a way for programs to maintain state between function calls. These are *just* contiguous stack regions, with boundaries tracked through two registers – RBP and RSP
  - RBP – Base Pointer, tracks the base of current stack frame
  - RSP – Stack Pointer, we know it already. Tracks the top of the frame (and top of the stack at the same time).
- On every function invocation (CALL instruction) a stack frame is created, this creates virtual separation between local and other variables.
  - Every function call is an act of redirecting execution; stack frames retain instruction pointer to the instruction following the original CALL, allowing execution to return correctly after handling function’s code.
- On every return (RET instruction) the previous stack boundaries are restored, effectively “destroying” previous frame.
  - In practice, the old data remains on stack. It will be overwritten soon but there is no built-in mechanism of clearing it.

Stack Frames

```
void myfunc(){
        int var_1;
        int var_2;
        <stop here>
        return 0;
}

int main(){
        int var_1;
        myfunc();
}
```



stack frame of myfunc()

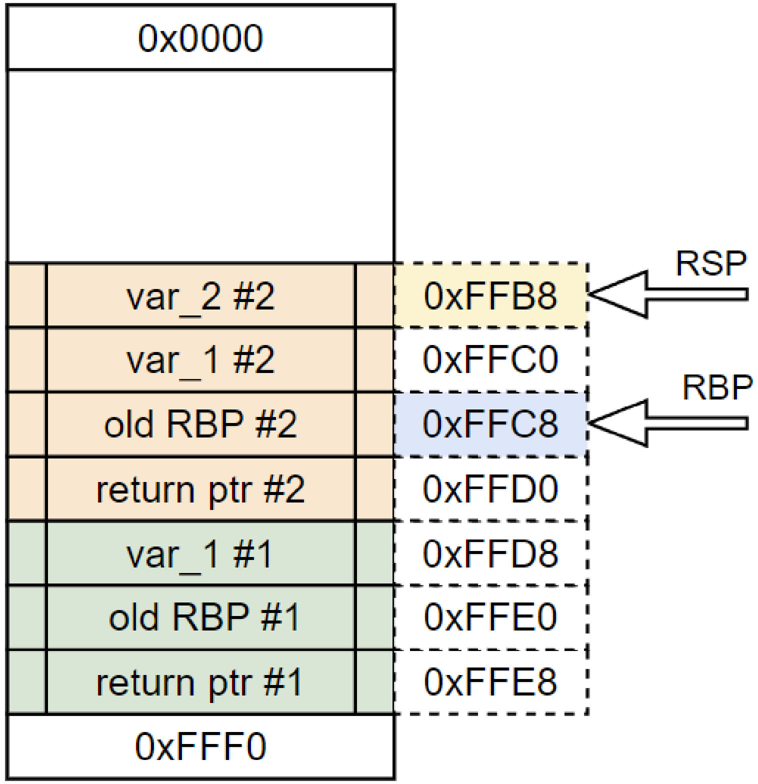
stack frame of main()

0x0000	
var_2 #2	0xFFB8
var_1 #2	0xFFC0
old RBP #2	0xFFC8
return ptr #2	0xFFD0
var_1 #1	0xFFD8
old RBP #1	0xFFE0
return ptr #1	0xFFE8
0xFFF0	

RSP	0xFFB8
RBP	0xFFC8

Stack Frame lifecycle

- CALL instruction is executed, pointing to a target function
  - current RIP is pushed onto stack, forming a return pointer
- Execution is transferred to the address pointed by call
- New function set ups its' stack frame:
  - current RBP is pushed onto stack
  - current RSP is moved to RBP
    - this establishes base of the frame, pointing at old RBP
  - function reserves space on stack for variables by SUBtracting from RSP
- Function runs through
- On function end
  - RBP is MOVed to RSP, "collapsing" stack back to point at old RBP
  - RBP is POPped from the stack
  - return pointer is popped from stack into RIP by issuing RET instruction



Stack Frame lifecycle

These patterns are so common, they have their respective names.

Learning to recognize them is the first step towards efficient skimming through assembly.

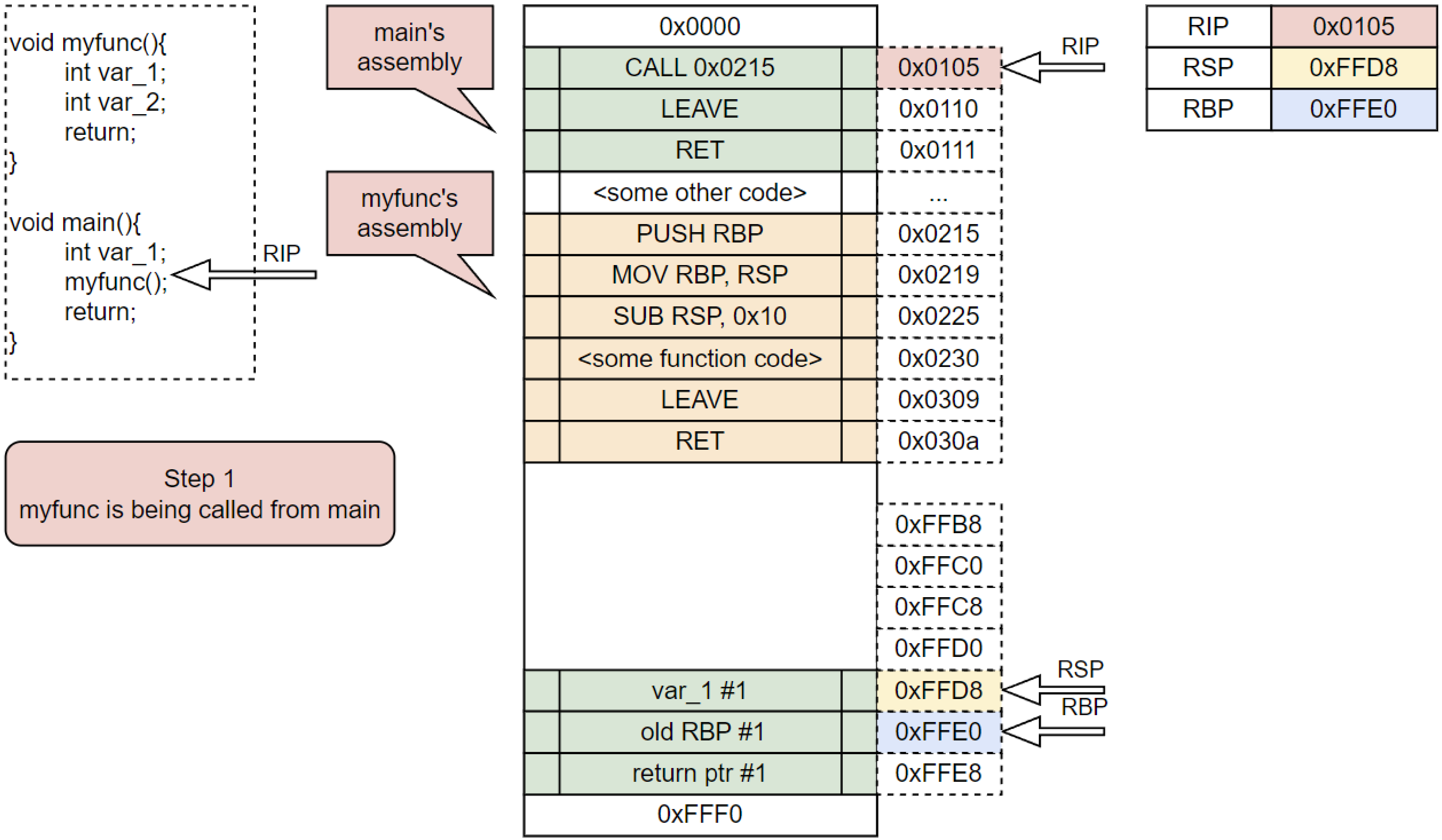
- CALL instruction is executed, pointing to a target function
    - current RIP is pushed onto stack, forming a return pointer
- Execution is transferred to the address pointed by call
- New function set ups its' stack frame:
    - current RBP is pushed onto stack
    - current RSP is moved to RBP
        - this establishes base of the frame, pointing at old RBP
    - function reserves space on stack for variables by SUBtracting from RSP

Function Prologue

- Function runs through
- On function end
    - RBP is MOVed to RSP, "collapsing" stack back to point at old RBP
    - RBP is POPped from the stack
    - return pointer is popped from stack into RIP by issuing RET instruction

Function Epilogue

Stack Frame lifecycle example



```
void myfunc(){
        int var_1;
        int var_2;
        return;
}

void main(){
        int var_1;
        myfunc();
        return;
}
```

0x0000		
CALL 0x0215	0x0105	RIP
LEAVE	0x0110	
RET	0x0111	
<some other code>	...	
PUSH RBP	0x0215	
MOV RBP, RSP	0x0219	
SUB RSP, 0x10	0x0225	
<some function code>	0x0230	
LEAVE	0x0309	
RET	0x030a	
	0xFFB8	
	0xFFC0	
	0xFFC8	
	0xFFD0	
var_1 #1	0xFFD8	RSP
old RBP #1	0xFFE0	RBP
return ptr #1	0xFFE8	
0xFFF0		

RIP	0x0105
RSP	0xFFD8
RBP	0xFFE0

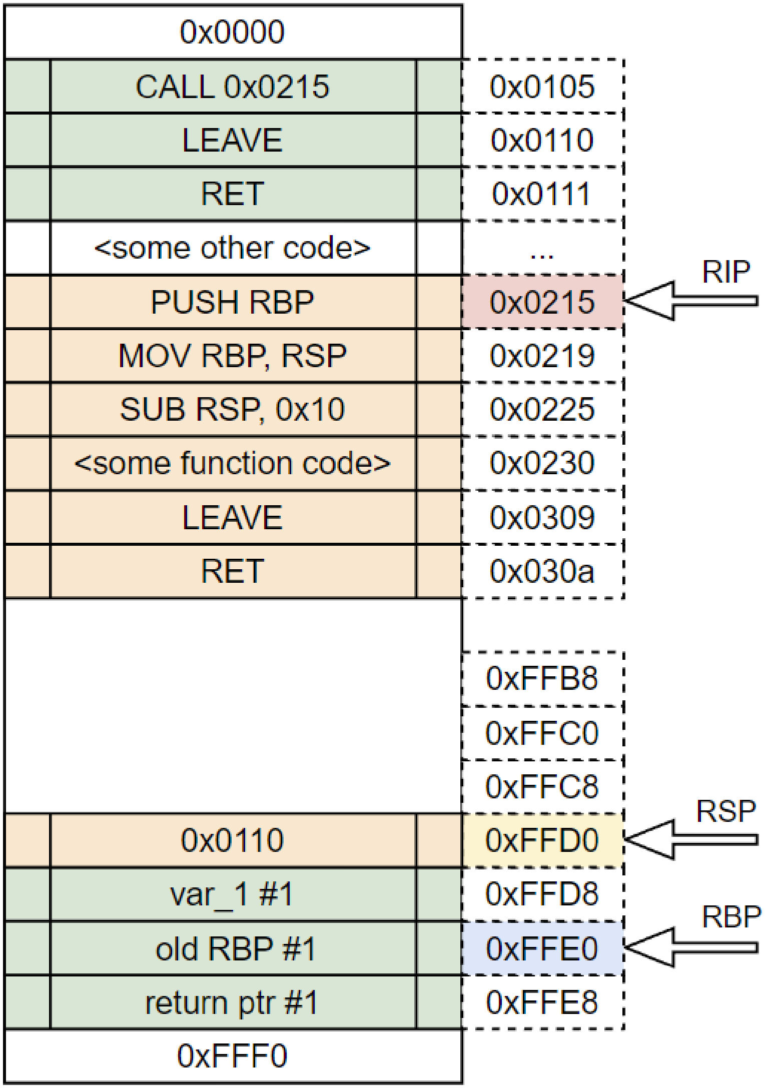
Step 1
myfunc is being called from main

Stack Frame lifecycle example

```
void myfunc(){      <- RIP
        int var_1;
        int var_2;
        return;
}

void main(){
        int var_1;
        myfunc();
        return;
}
```

return pointer pushed on the stack
-
execution transferred to myfunc

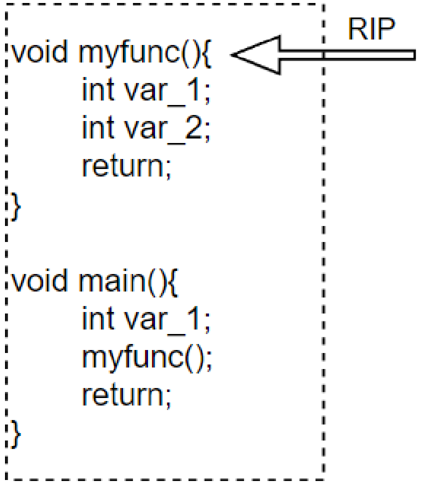
address of instruction following CALL



0x0000	
CALL 0x0215	0x0105
LEAVE	0x0110
RET	0x0111
<some other code>	...
PUSH RBP	0x0215
MOV RBP, RSP	0x0219
SUB RSP, 0x10	0x0225
<some function code>	0x0230
LEAVE	0x0309
RET	0x030a
	0xFFB8
	0xFFC0
	0xFFC8
0x0110	0xFFD0
var_1 #1	0xFFD8
old RBP #1	0xFFE0
return ptr #1	0xFFE8
0xFFF0	

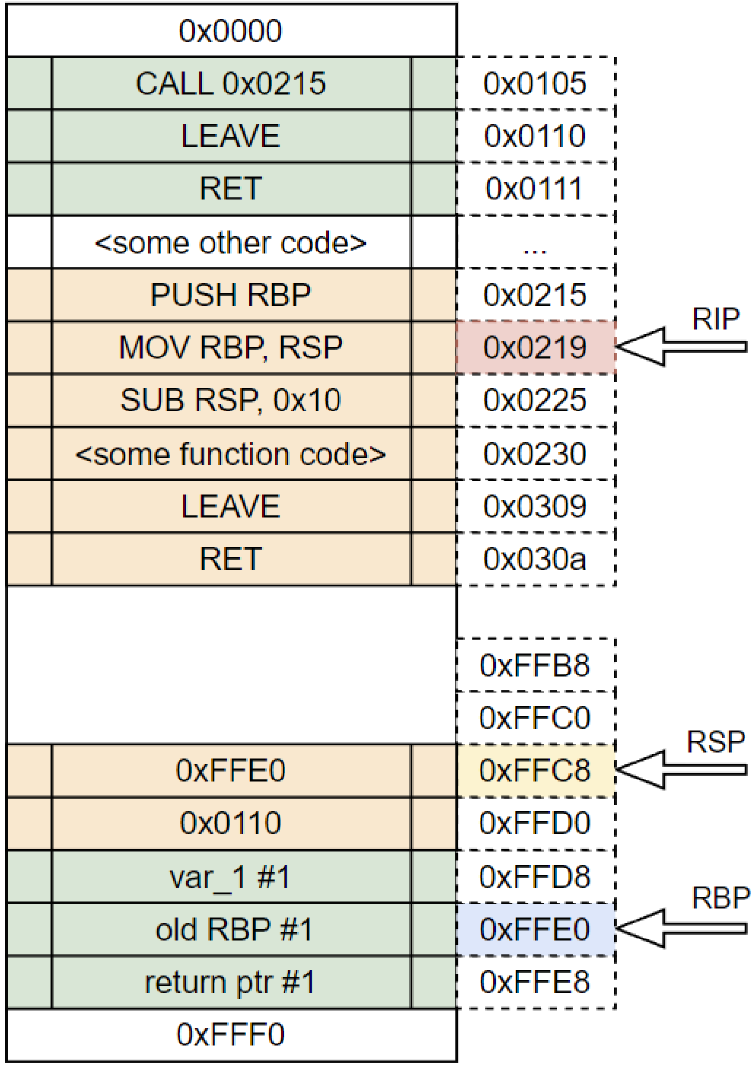
RIP	0x0215
RSP	0xFFD0
RBP	0xFFE0

Stack Frame lifecycle example



```
void myfunc(){
        int var_1;
        int var_2;
        return;
}

void main(){
        int var_1;
        myfunc();
        return;
}
```

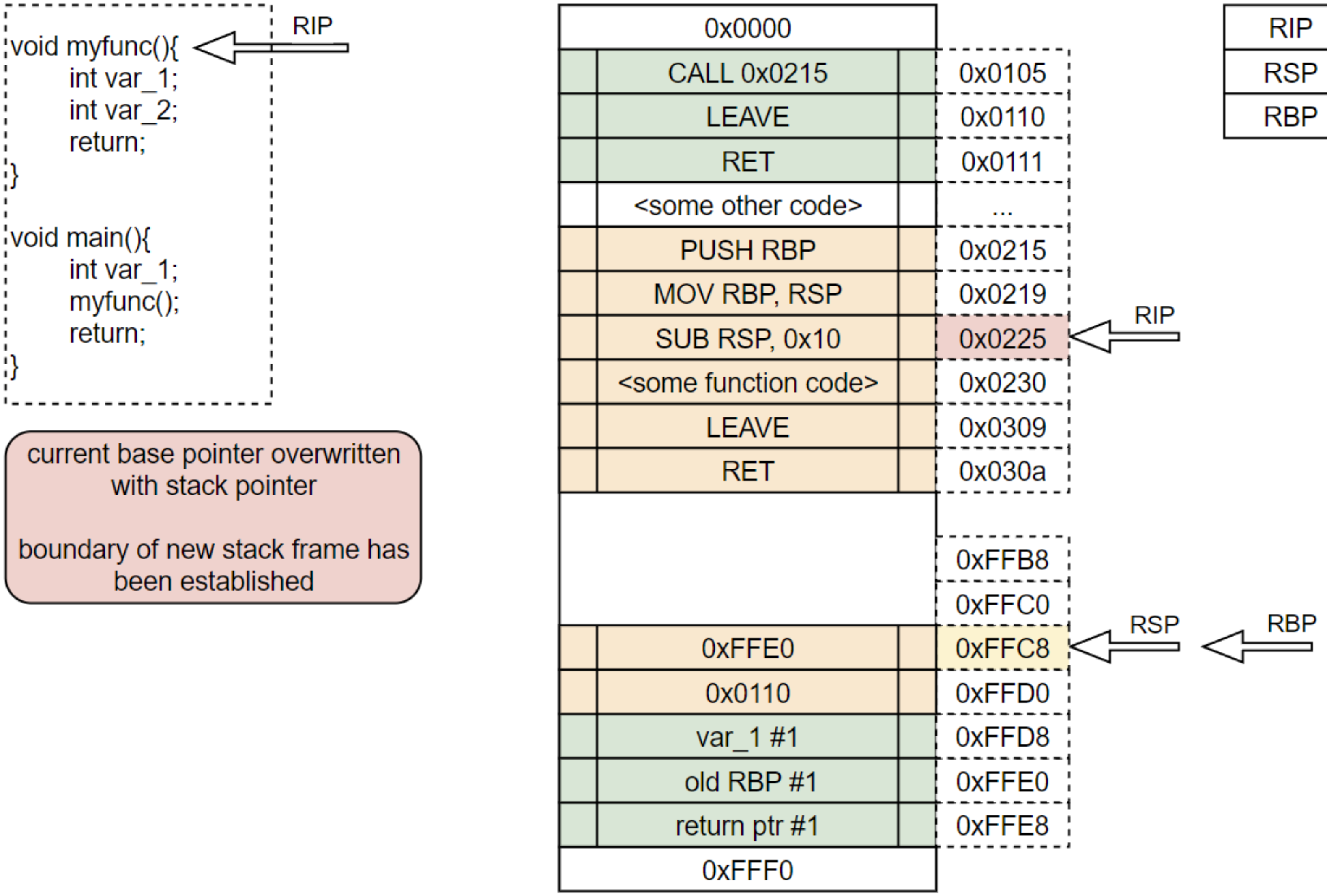
base pointer saved on stack



0x0000		
CALL 0x0215	0x0105	
LEAVE	0x0110	
RET	0x0111	
<some other code>	...	
PUSH RBP	0x0215	
MOV RBP, RSP	0x0219	RIP
SUB RSP, 0x10	0x0225	
<some function code>	0x0230	
LEAVE	0x0309	
RET	0x030a	
	0xFFB8	
	0xFFC0	
0xFFE0	0xFFC8	RSP
0x0110	0xFFD0	
var_1 #1	0xFFD8	
old RBP #1	0xFFE0	RBP
return ptr #1	0xFFE8	
0xFFF0		

RIP	0x0219
RSP	0xFFC8
RBP	0xFFE0

Stack Frame lifecycle example



```
void myfunc(){
        int var_1;
        int var_2;
        return;
}

void main(){
        int var_1;
        myfunc();
        return;
}
```

current base pointer overwritten with stack pointer

boundary of new stack frame has been established

0x0000	
CALL 0x0215	0x0105
LEAVE	0x0110
RET	0x0111
<some other code>	...
PUSH RBP	0x0215
MOV RBP, RSP	0x0219
SUB RSP, 0x10	0x0225
<some function code>	0x0230
LEAVE	0x0309
RET	0x030a
	0xFFB8
	0xFFC0
0xFFE0	0xFFC8
0x0110	0xFFD0
var_1 #1	0xFFD8
old RBP #1	0xFFE0
return ptr #1	0xFFE8
0xFFF0	

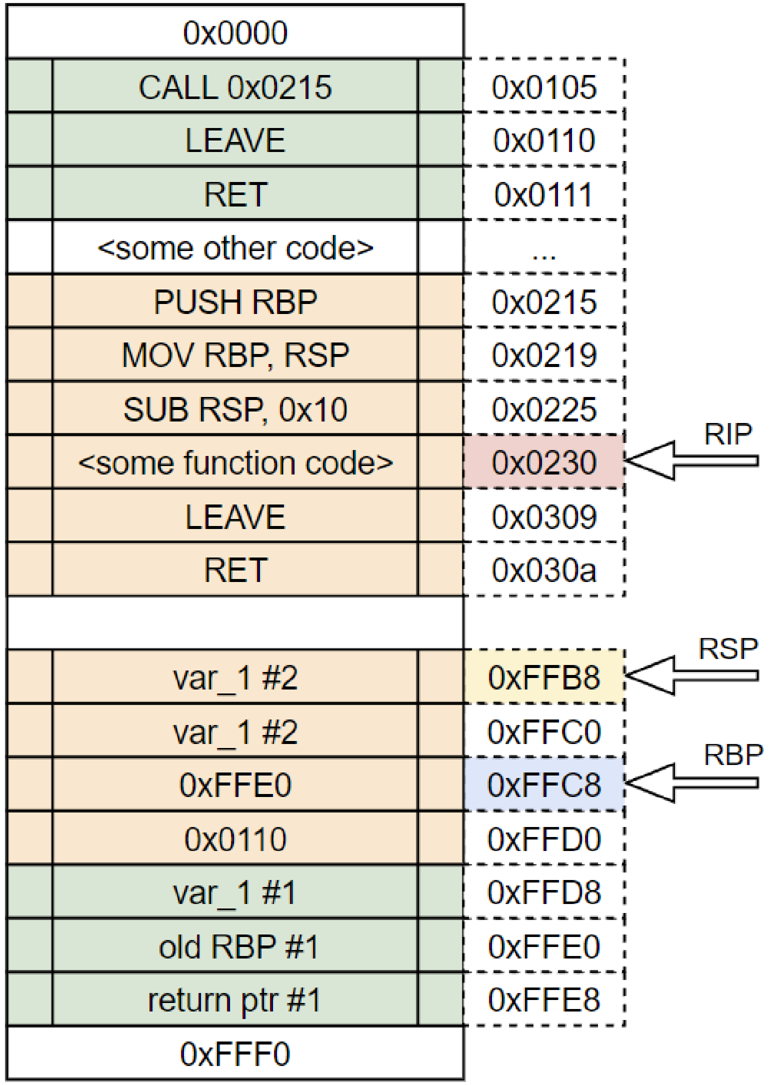
RIP	0x0225
RSP	0xFFC8
RBP	0xFFC8

Stack Frame lifecycle example

```
void myfunc(){
        int var_1;
        int var_2;
        return;
}

void main(){
        int var_1;
        myfunc();
        return;
}
```

RIP

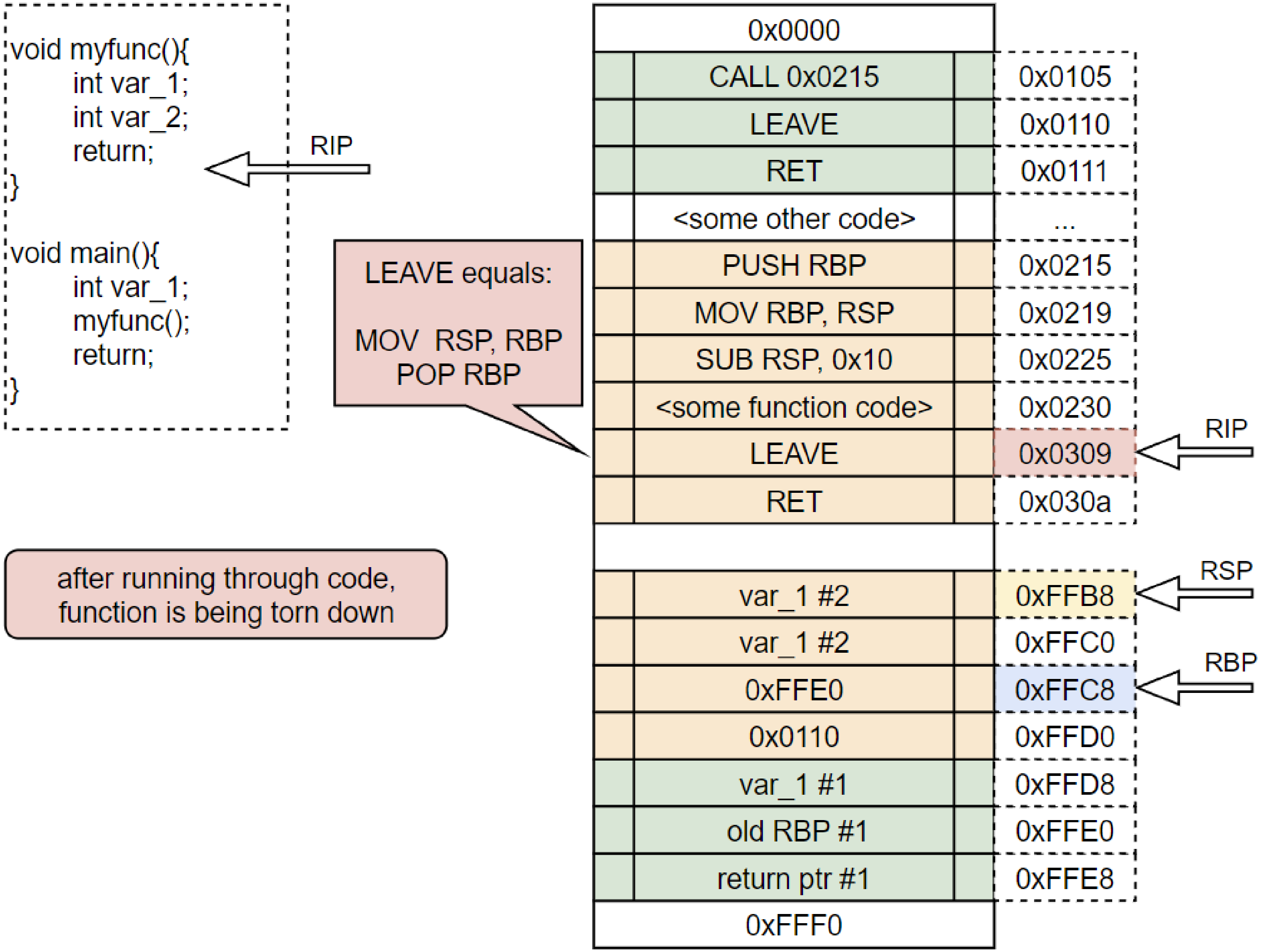
reserving space for variables



0x0000	
CALL 0x0215	0x0105
LEAVE	0x0110
RET	0x0111
<some other code>	...
PUSH RBP	0x0215
MOV RBP, RSP	0x0219
SUB RSP, 0x10	0x0225
<some function code>	0x0230
LEAVE	0x0309
RET	0x030a
var_1 #2	0xFFB8
var_1 #2	0xFFC0
0xFFE0	0xFFC8
0x0110	0xFFD0
var_1 #1	0xFFD8
old RBP #1	0xFFE0
return ptr #1	0xFFE8
0xFFF0	

RIP	0x0230
RSP	0xFFB8
RBP	0xFFC8

Stack Frame lifecycle example



```
void myfunc(){
        int var_1;
        int var_2;
        return;
}

void main(){
        int var_1;
        myfunc();
        return;
}
```

LEAVE equals:

MOV RSP, RBP
POP RBP

after running through code,
function is being torn down

0x0000		
CALL 0x0215	0x0105	
LEAVE	0x0110	
RET	0x0111	
<some other code>	...	
PUSH RBP	0x0215	
MOV RBP, RSP	0x0219	
SUB RSP, 0x10	0x0225	
<some function code>	0x0230	
LEAVE	0x0309	RIP
RET	0x030a	
var_1 #2	0xFFB8	RSP
var_1 #2	0xFFC0	
0xFFE0	0xFFC8	RBP
0x0110	0xFFD0	
var_1 #1	0xFFD8	
old RBP #1	0xFFE0	
return ptr #1	0xFFE8	
0xFFF0		

RIP	0x0309
RSP	0xFFB8
RBP	0xFFC8

Stack Frame lifecycle example

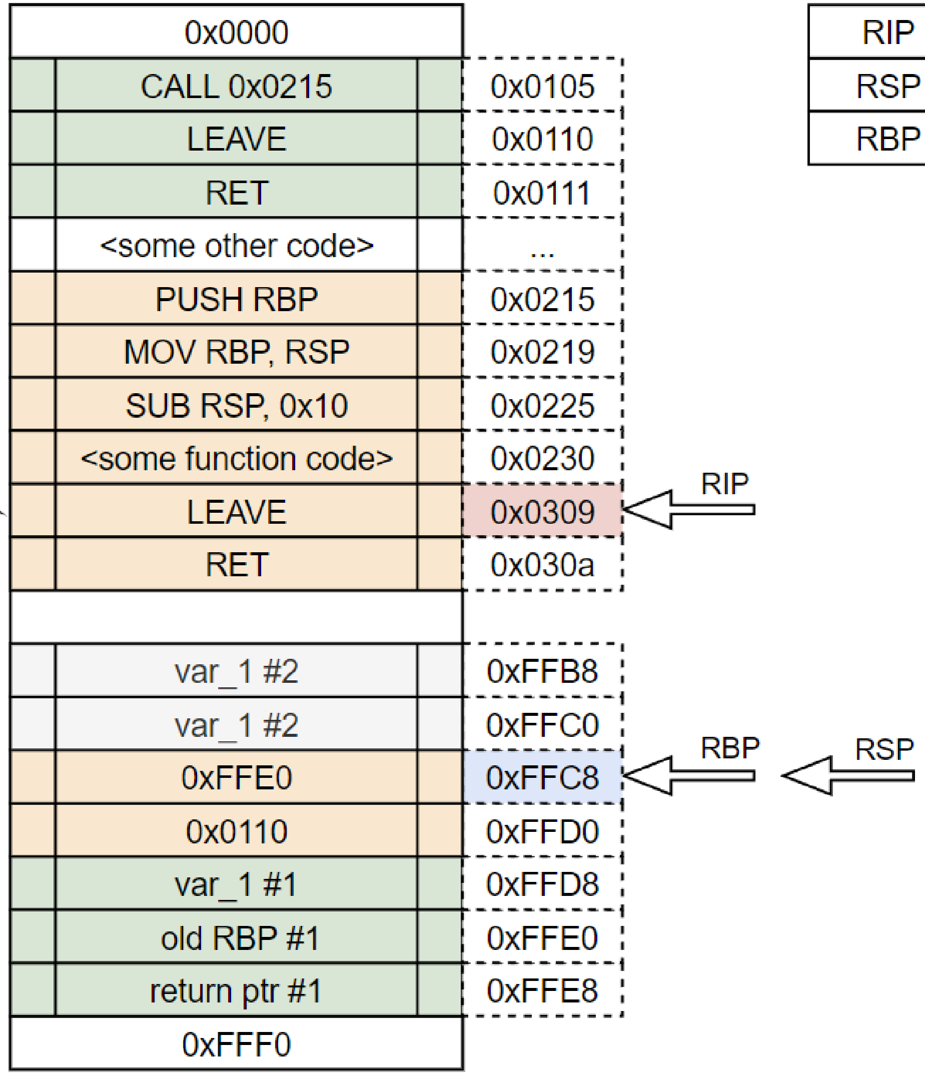
```
void myfunc(){
	int var_1;
	int var_2;
	return;
}

void main(){
	int var_1;
	myfunc();
	return;
}
```

RIP

LEAVE equals:

MOV RSP, RBP
POP RBP

MOV RSP, RBP

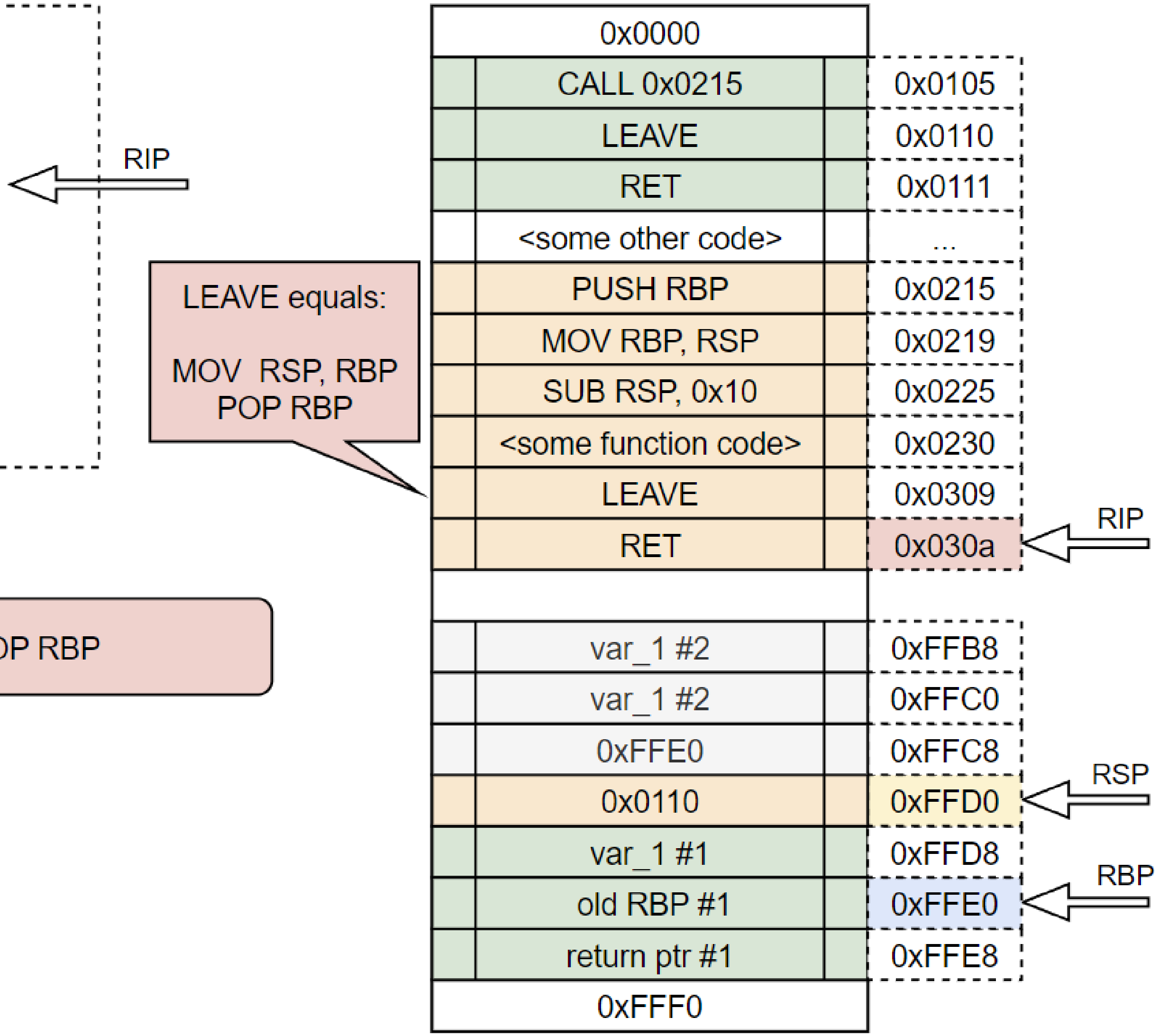


0x0000	
CALL 0x0215	0x0105
LEAVE	0x0110
RET	0x0111
<some other code>	...
PUSH RBP	0x0215
MOV RBP, RSP	0x0219
SUB RSP, 0x10	0x0225
<some function code>	0x0230
LEAVE	0x0309
RET	0x030a
var_1 #2	0xFFB8
var_1 #2	0xFFC0
0xFFE0	0xFFC8
0x0110	0xFFD0
var_1 #1	0xFFD8
old RBP #1	0xFFE0
return ptr #1	0xFFE8
0xFFF0	

RIP	0x0309
RSP	0xFFC8
RBP	0xFFC8

Stack Frame lifecycle example

```
void myfunc(){
        int var_1;
        int var_2;
        return;
}

void main(){
        int var_1;
        myfunc();
        return;
}
```



LEAVE equals:

MOV RSP, RBP
POP RBP

0x0000		
CALL 0x0215	0x0105	
LEAVE	0x0110	
RET	0x0111	
<some other code>	...	
PUSH RBP	0x0215	
MOV RBP, RSP	0x0219	
SUB RSP, 0x10	0x0225	
<some function code>	0x0230	
LEAVE	0x0309	
RET	0x030a	RIP
var_1 #2	0xFFB8	
var_1 #2	0xFFC0	
0xFFE0	0xFFC8	
0x0110	0xFFD0	RSP
var_1 #1	0xFFD8	
old RBP #1	0xFFE0	RBP
return ptr #1	0xFFE8	
0xFFF0		

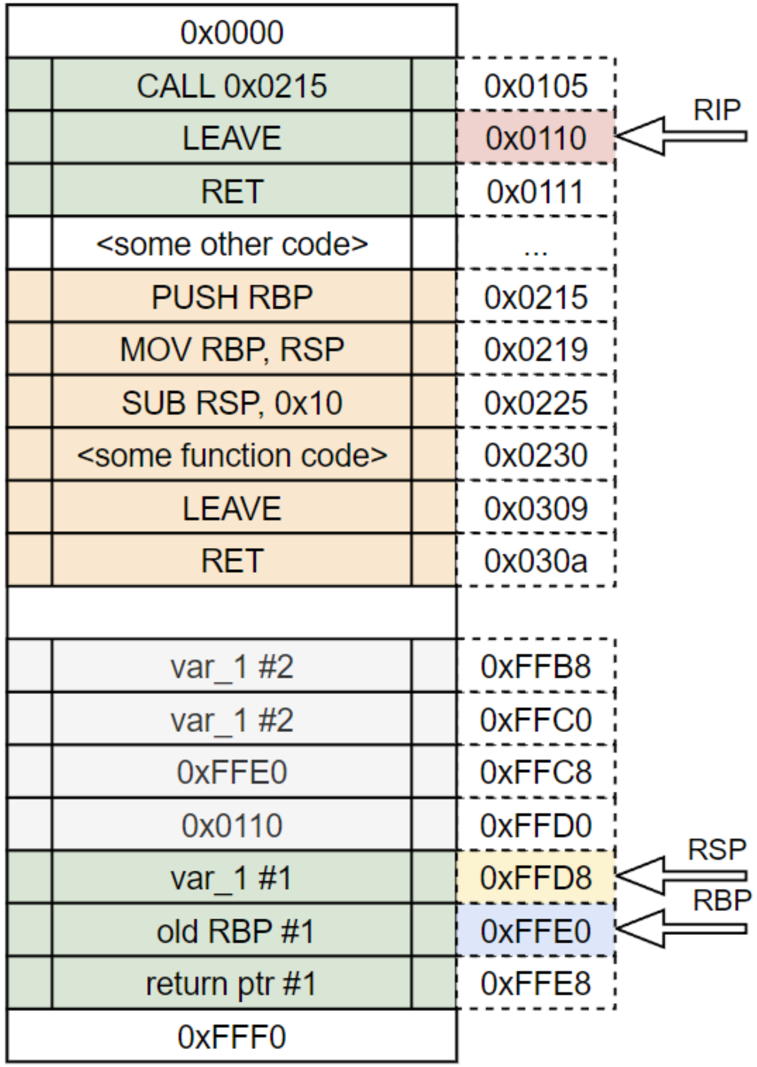
RIP	0x030A
RSP	0xFFD0
RBP	0xFFE0

POP RBP

Stack Frame lifecycle example

```
void myfunc(){
        int var_1;
        int var_2;
        return;
}

void main(){
        int var_1;
        myfunc();
        return;           <- RIP
}
```

finally, program pops the return address straight into RIP using RET



0x0000		
CALL 0x0215	0x0105	
LEAVE	0x0110	RIP
RET	0x0111	
<some other code>	...	
PUSH RBP	0x0215	
MOV RBP, RSP	0x0219	
SUB RSP, 0x10	0x0225	
<some function code>	0x0230	
LEAVE	0x0309	
RET	0x030a	
var_1 #2	0xFFB8	
var_1 #2	0xFFC0	
0xFFE0	0xFFC8	
0x0110	0xFFD0	
var_1 #1	0xFFD8	RSP
old RBP #1	0xFFE0	RBP
return ptr #1	0xFFE8	
0xFFF0		

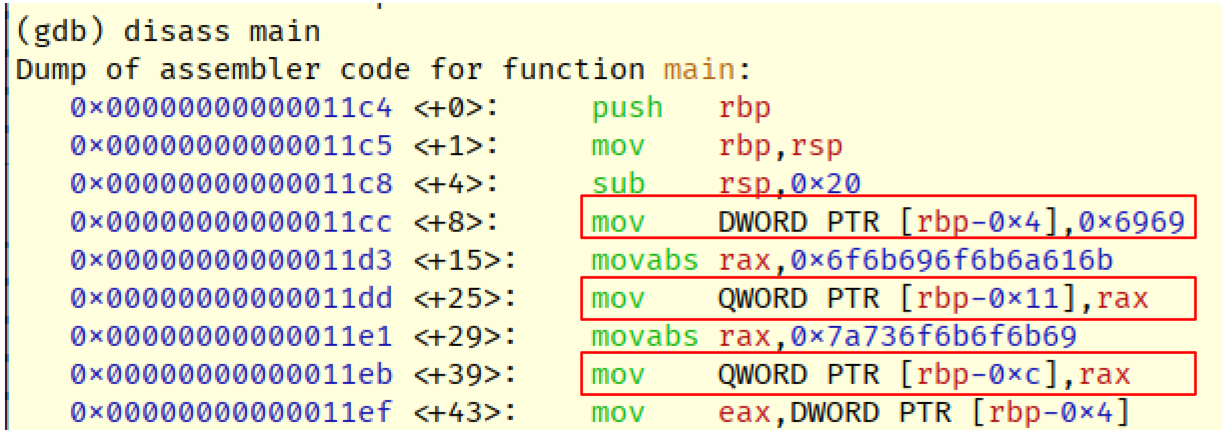
RIP	0x0110
RSP	0xFFD8
RBP	0xFFE0

Purpose of Base Pointer (RBP)

- Up until now we've talked about RBP being used only to delimit frames
- By being stable point of reference, it is also used to access local variables on stack
  - Recall that stack can be allocated only in multiplies of 8 while some data types can be shorter or longer.
  - By MOVing memory on allocated stack space instead of pushing it, programs have more flexibility with memory management
  - e.g. storing two ints (2 x 4bytes) in one 8-byte stack segment is impossible through PUSHes alone

example of addressing local variable through RBP

MOV [RBP - 0x8], RDI



```
(gdb) disass main
Dump of assembler code for function main:
   0x00000000000011c4 <+0>:     push   rbp
   0x00000000000011c5 <+1>:     mov    rbp,rsp
   0x00000000000011c8 <+4>:     sub    rsp,0x20
   0x00000000000011cc <+8>:     mov    DWORD PTR [rbp-0x4],0x6969
   0x00000000000011d3 <+15>:    movabs rax,0x6f6b696f6b6a616b
   0x00000000000011dd <+25>:    mov    QWORD PTR [rbp-0x11],rax
   0x00000000000011e1 <+29>:    movabs rax,0x7a736f6b6f6b69
   0x00000000000011eb <+39>:    mov    QWORD PTR [rbp-0xc],rax
   0x00000000000011ef <+43>:    mov    eax,DWORD PTR [rbp-0x4]
```

⧗ 30 minutes

LAB 2

Observing assembly in GDB p.2

Instruction:

handout/lab2.pdf

BUFFER OVERFLOWS

Buffer overflows

- a very classic vulnerability, which spawned entire field of memory corruption bugs and remains potent to this day
- occurs when too much data is copied to a buffer, most of the times due to lack of input bounds checking or wrong API usage
- when overflow occurs, memory following the initial buffer gets corrupted with input data
- as we've observed in previous chapter local variables and saved pointers coexist in the same stack frames. Thanks to that we can:
  - overwrite some other variable on stack and abuse program's logic
  - overwrite return pointer to take control over what is being executed
  - just smash all the way down through the stack until we find something useful

Stack smashing

- Most of the time we will want to overflow buffer so that we can take control of saved return address. When RETurning from function, this value will be placed in RIP, effectively allowing us to take control of the program.
- Few techniques here:
  - ret2win – CTF classic, return to some function unavailable through the normal code flow which results in system shell or yields flag. One can think of it as returning to administrative or privileged function in normal software.
  - ret2shellcode* – we inject data that can be interpreted as valid instructions somewhere in the program's memory and then point the RIP to its beginning

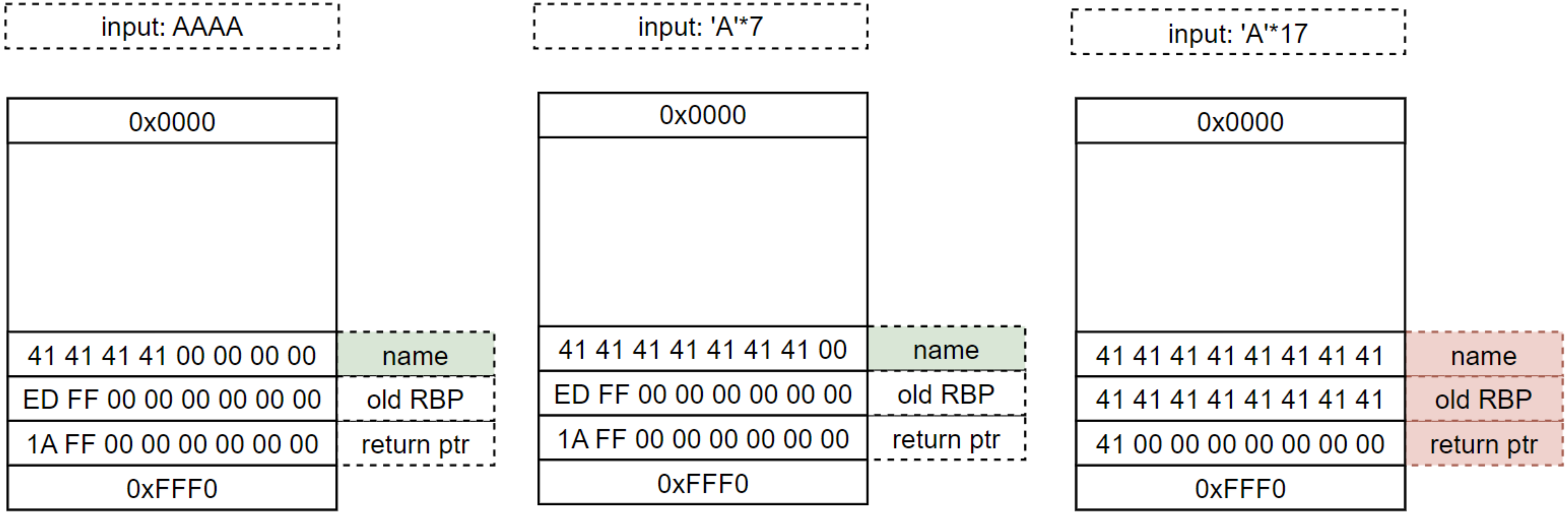
Remember?

Everything is data!

*this won't work with modern mitigations enabled, we do need to begin somewhere though

Stack smashing – overflow example

```
int main(){
        char name[8];
        printf('Enter name: ');
        gets(name);
        return;
}
```



input: AAAA

0x0000	
41 41 41 41 00 00 00 00	name
ED FF 00 00 00 00 00 00	old RBP
1A FF 00 00 00 00 00 00	return ptr
0xFFF0	

input: 'A'*7

0x0000	
41 41 41 41 41 41 41 00	name
ED FF 00 00 00 00 00 00	old RBP
1A FF 00 00 00 00 00 00	return ptr
0xFFF0	

input: 'A'*17

0x0000	
41 41 41 41 41 41 41 41	name
41 41 41 41 41 41 41 41	old RBP
41 00 00 00 00 00 00 00	return ptr
0xFFF0	

⌛ 40 minutes

LAB 3

Exploiting buffer overflows with no mitigations enabled

Instruction:

handout/lab3.pdf

SHELLCODING PRIMER

Shellcode

- Shellcode is just a machine code ready to be executed on processor
- Since it translates directly to assembly, we can leverage our knowledge to understand or build one
- Shellcoding is an art in itself and its' tricks are definitely beyond scope of our workshop
- However, I want to lay some groundwork that will allow you to continue diving deeper in these concepts later without feeling overwhelmed.
- Writing assembly is really important to learn, we WILL need that understanding for building ROP chains
    - There is also much added value in the fact, that writing shellcode makes understanding assembly so much easier through practice

Shellcode

- During the labs we'll be using Keystone Engine, a multi-architecture assembler framework
- There's a bit of boilerplate involved, relevant templates will be provided so we don't waste time learning the framework's overhead now.
- The important takeaway is this – we will focus on writing plain assembly, framework will convert it to machine code for us, which we can use in our exploits later.

OS considerations

- How does a processor really work with an operating system – each is so different, there seems to be a disconnection between the OS and assembly itself
  - Answer to this question has been an a-ha moment for me, everything just clicked from there.
- When programming in a higher-level language we just take it for granted – import some library and call functions that are provided there.
- But these libraries and functions at some point have to actually execute some code interacting with the OS. How do they do that?
- Enter, syscalls

Syscalls

- Syscalls or system calls are a suite of primitive instructions defined by the OS itself that can be called through assembly
  - Again, these are dictated by the OS only, therefore each OS will have its' own syscall set and convention
- In general, invoking a system call is performed by filling relevant registers with arguments that will be passed to the OS and executing a SYSCALL opcode.
- Again, registers and syscall numbers differ between systems, luckily Linux's system calls are well documented and understood. Their numbers are fixed and known
  - On the contrary, Windows' syscall are a major pain to work with, their numbers may or may not change between builds, documentation is scarce and every behaviour is subject to change at any given moment. Truly a disaster (it is fascinating though).

Linux syscalls

Recipe for Linux is very simple

- Fill RAX with syscall number we want to call
    - during this workshop we'll use execve() exclusively, its' number is 0x3B
- Fill any other registers as required by documentation
- Execute SYSCALL instruction
- Result will be returned through RAX

https://filippo.io/linux-syscall-table/

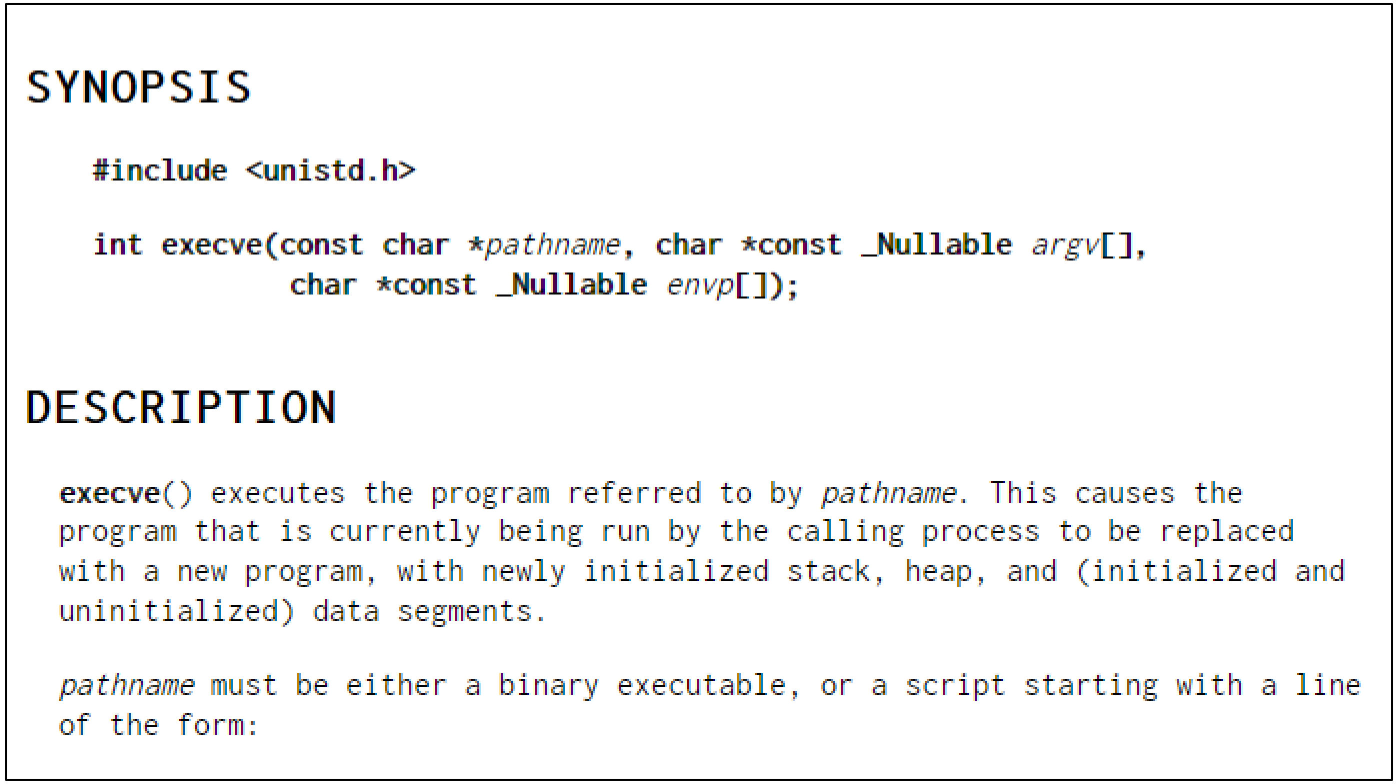
Linux calling convention

- This is a perfect moment to touch on Linux's calling convention
- Have you ever wondered how are arguments passed to the functions? This is defined by calling convention*
- In particular it goes like this:
  - RDI – 1st arg
  - RSI – 2nd arg
  - RDX – 3rd arg (and so on)
  - RCX
  - R8
  - R9
  - Any other arguments go on stack

*calling convention also specifies other things such as callee- and caller-saved registers. We intentionally skip over that as it does not add any substance for our current purposes

Calling execve

- To finally tie all these concepts together, consider the following excerpt from manual of execve:



SYNOPSIS

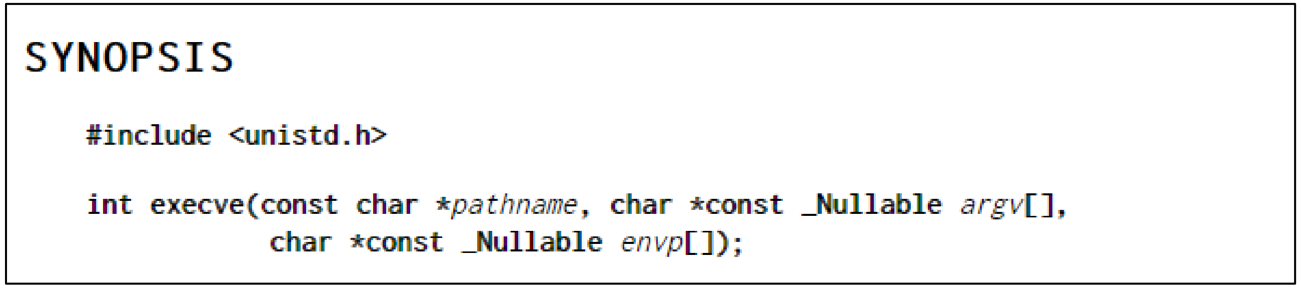
```
#include <unistd.h>

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

DESCRIPTION

execve() executes the program referred to by *pathname*. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments.

pathname must be either a binary executable, or a script starting with a line of the form:

Calling execve



SYNOPSIS

```
    #include <unistd.h>

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

- It can be summed as: “start a program given in *pathname argument”
- To keep things manageable, lets consider starting a local shell
- To achieve that we want to set:
  - 1st argument: a pointer pointing towards address where “/bin/sh” string is present
    - Conventionally, C-style strings are null-delimited. That means, 0x00 byte is the last character of every string. **Very** conveniently, “/bin/sh\x00” is exactly 8-bytes long.
  - 2nd argument: 0x00
  - 3rd argument: 0x00
- Why the zeroes? Well these arguments have to be provided but they also are nullable (as seen in docs)
  - argv would contain arguments for our program
  - envp would be the environmental variables passed to the program
  - these *are* useful, but safe to ignore if we just want to execute a specified binary as in our case

Writing our shellcode

Now that we've defined our needs, let's convert them into required processor's state to achieve that:

- RAX – 0x3B
- RDI – pointer to /bin/sh (1st arg)
- RSI – 0x00 (2nd arg)
- RDX – 0x00 (3rd arg)
- and that's about it

Writing our shellcode

Now that we've defined our needs, let's convert them into required processor's state to achieve that:

- RAX - 0x3B
- RDI - pointer to /bin/sh (1st arg)
- RSI - 0x00 (2nd arg)
- RDX - 0x00 (3rd arg)
- and that's about it

```
PUSH 0x3B
POP RAX
MOV RBX, 0x0068732f6e69622f
PUSH RBX
MOV RDI, RSP
XOR RSI, RSI
XOR RDX, RDX
SYSCALL
```

Writing our shellcode

While our shellcode should be mostly understood by now, this snippet calls for an explanation:

```
MOV RBX, 0x0068732f6e69622f
PUSH RBX
MOV RDI, RSP
```

- Remember the endianness discussion – we are pushing a “/bin/sh\x00” string in the form of a number. Therefore, we have to convert each character to its numeric value and input bytes-reversed
  - If you’d put it through some hex-to-string tool without specifying endianness, you’d see “\x00hs/nib/”
- Second thing is – execve asks for an address of string
  - not a string itself
  - this is achieved by pushing string on the stack (which now points to it)
  - PUSH instruction does not support pushing immediate value bigger than 4 bytes. We have to use intermediate register

Badchars

- Often, programs will process some of the characters from our shellcode, effectively breaking it. Such characters are called bad characters, or badchars.
- Most of the time nullbyte will be a badchar due to being string delimiter.
- There are two tactics of bypassing badchars:
  - rewriting shellcode so that it does not contain them
  - encoding shellcode and using a dynamic decode routine during runtime (this is what payload encoders do)
- In general, rewriting shorter shellcodes is feasible and relatively simple, the only thing needed is patience and creativity.
- Encoders are a mile deep, explore them at your own peril
  - https://danielsauder.com/2015/08/26/an-analysis-of-shikata-ga-nai/

Brainstorming nullbyte evasion ideas

- MOV RAX, 0 -> XOR RAX, RAX
- MOV AX, 0x3B -> PUSH 0x3B ; POP RAX
- MOV EAX, 0x002F2F2F - > MOV EAX, 0xFF1F2F2F ; INC EAX
- MOV EAX, 0x002F2F2F - > MOV EAX, 0x012F2F2F ; DEC EAX

I highly recommend tinkering with msf-nasm_shell to debug what instructions have nullbytes

30 minutes

LAB 4

Generating own shellcode, nullbytes challenge

Instruction:

handout/lab4.pdf

MITIGATIONS

Modern mitigations (Linux)

- Prevalence of memory corruption bugs and their impact forced industry to invent some mitigations to make exploitation of such bugs much more difficult.
- As we will see, they are at the same time highly effective and bypassable.
- With all mitigations enabled, at the very least we are going to need two bugs to exploit the application
- What we are going to cover
  - Stack canaries
  - NX/DEP
  - PIE/ASLR

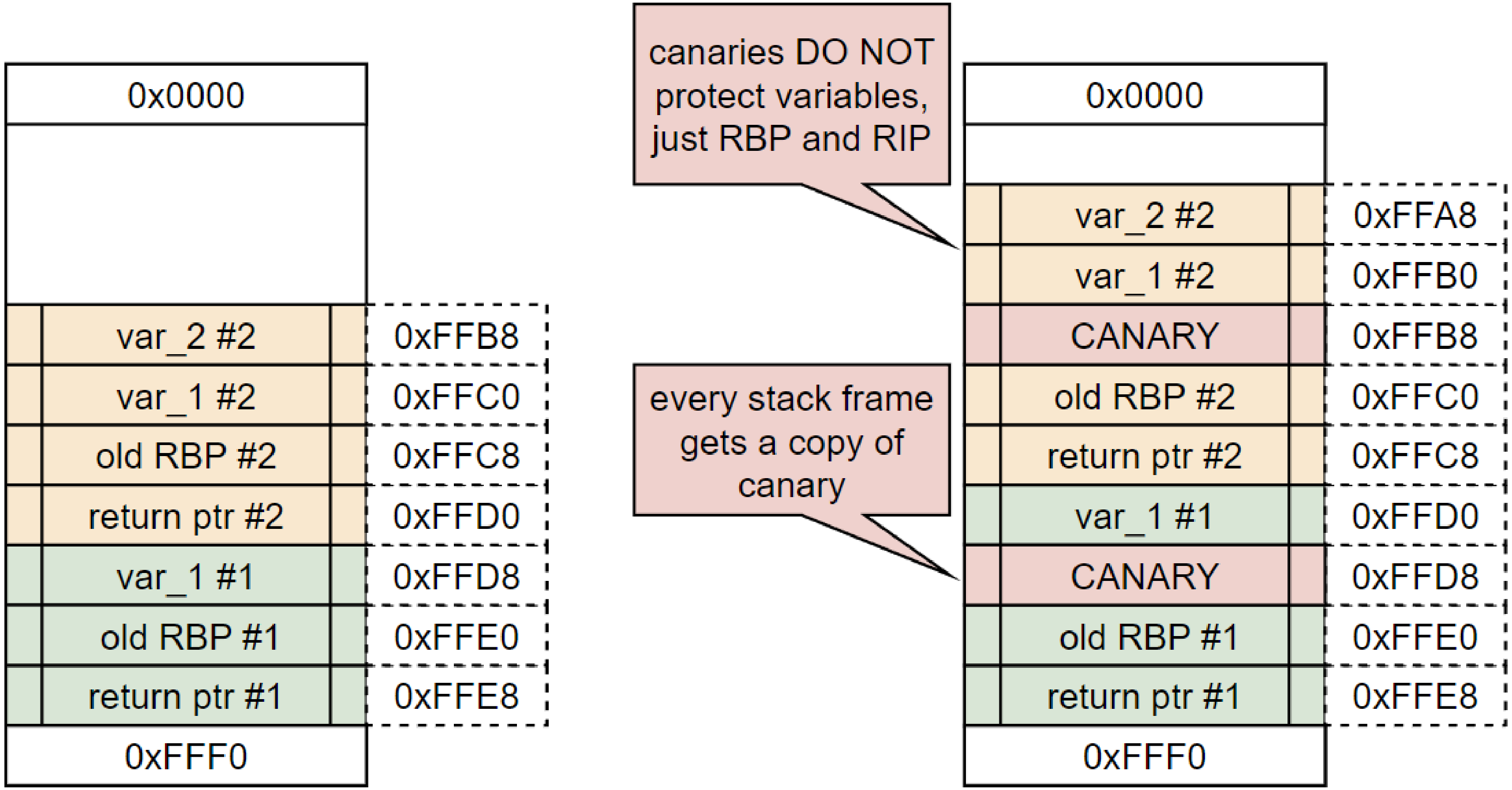
- For comparison, Windows offers (at least) the following mitigations: NX/DEP, ASLR/kASLR, CFG/kCFG, SMEP, ACG, CIG, CET, XFG, VBS, HVCI
  - Yup, that’s why starting your binexp adventure with Linux makes it a more streamlined experience ;]

Stack canaries

- Very simple, yet powerful
- Canaries are randomly generated values that are pushed in every stack frame on top of saved RBP
  - canary value is generated once per binary execution and stays the same across function calls. Its' value is stored in qword [fs:0x28]
- When RETurning from function calls, value on stack is checked against reference value
- If there is no match (value is corrupted by overflow for example), program crashes
- Two major ways of bypassing:
  - leaking cookie value either from stack or [fs:0x28] and aligning it correctly in overflow payload
  - precise overwrite bugs that do not affect the cookie

Stack canaries

```
void myfunc(){
        int var_1;
        int var_2;
        return;
}

int main(){
        int var_1;
        myfunc();
}
```



canaries DO NOT protect variables, just RBP and RIP

every stack frame gets a copy of canary

⌛ 60 minutes

LAB 5

Bypassing stack canaries

Instruction:

handout/lab5.pdf

NX/DEP

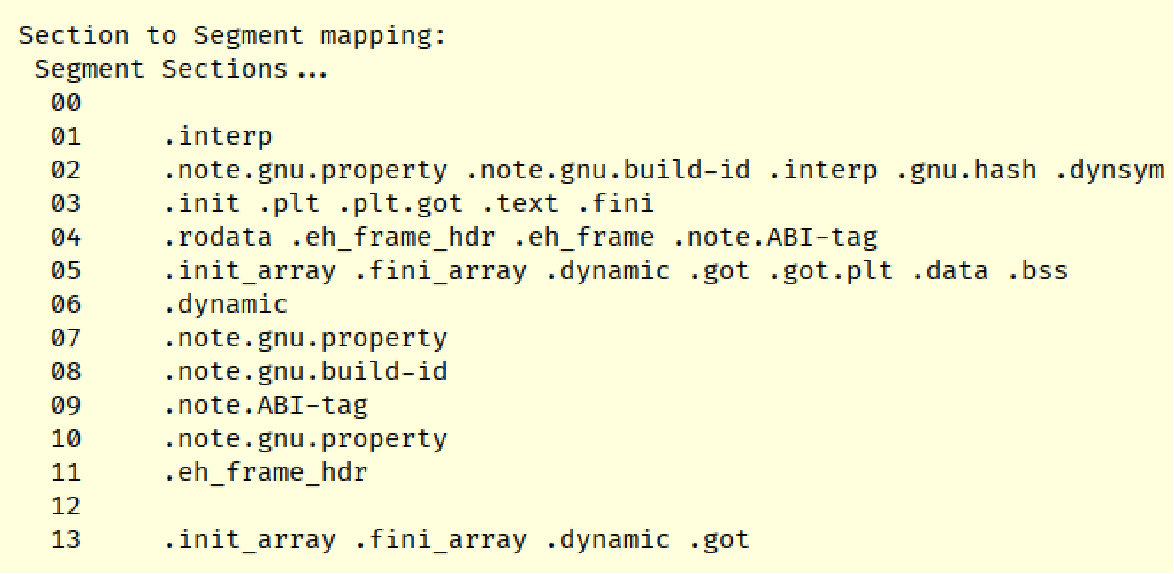
- No-Execute or Data Execution Prevention are two names to describe the same security mechanism – make the memory regions that contain data non-executable
- Most importantly, this affects stack
- If execution is attempted from memory region marked as NX, program crashes
- This effectively prohibits us from placing our own code in memory and returning to it
- Bypassing this mitigation requires an entirely new school of thought, enter ROP – Return Oriented Programming

Sections and program memory

- To continue further we have to briefly touch on two topics
  - Binary sections
  - How programs are mapped into memory
- We will just scratch the surface here. There is quite a bit of depth in both topics that is way beyond scope of this workshop

Binary sections

- Compiled binaries are not just a blobs of code/data, there is underlying structure dependent on executable file format
  - Linux uses ELF, windows uses PE. Both are loosely based on COFF (trivia for curious minds)
- Sections form the backbone of executables. These are named parts of file that contain specific types of data. Few examples:
  - .text – contains the actual code that gets executed
  - .rodata – read-only data, contains data that will not change throughout execution, for example static strings
  - .data – initialized global variables
  - .bss – uninitialized global variables



```
Section to Segment mapping:
 Segment Sections...
  00
  01     .interp
  02     .note.gnu.property .note.gnu.build-id .interp .gnu.hash .dynsym
  03     .init .plt .plt.got .text .fini
  04     .rodata .eh_frame_hdr .eh_frame .note.ABI-tag
  05     .init_array .fini_array .dynamic .got .got.plt .data .bss
  06     .dynamic
  07     .note.gnu.property
  08     .note.gnu.build-id
  09     .note.ABI-tag
  10     .note.gnu.property
  11     .eh_frame_hdr
  12
  13     .init_array .fini_array .dynamic .got
```

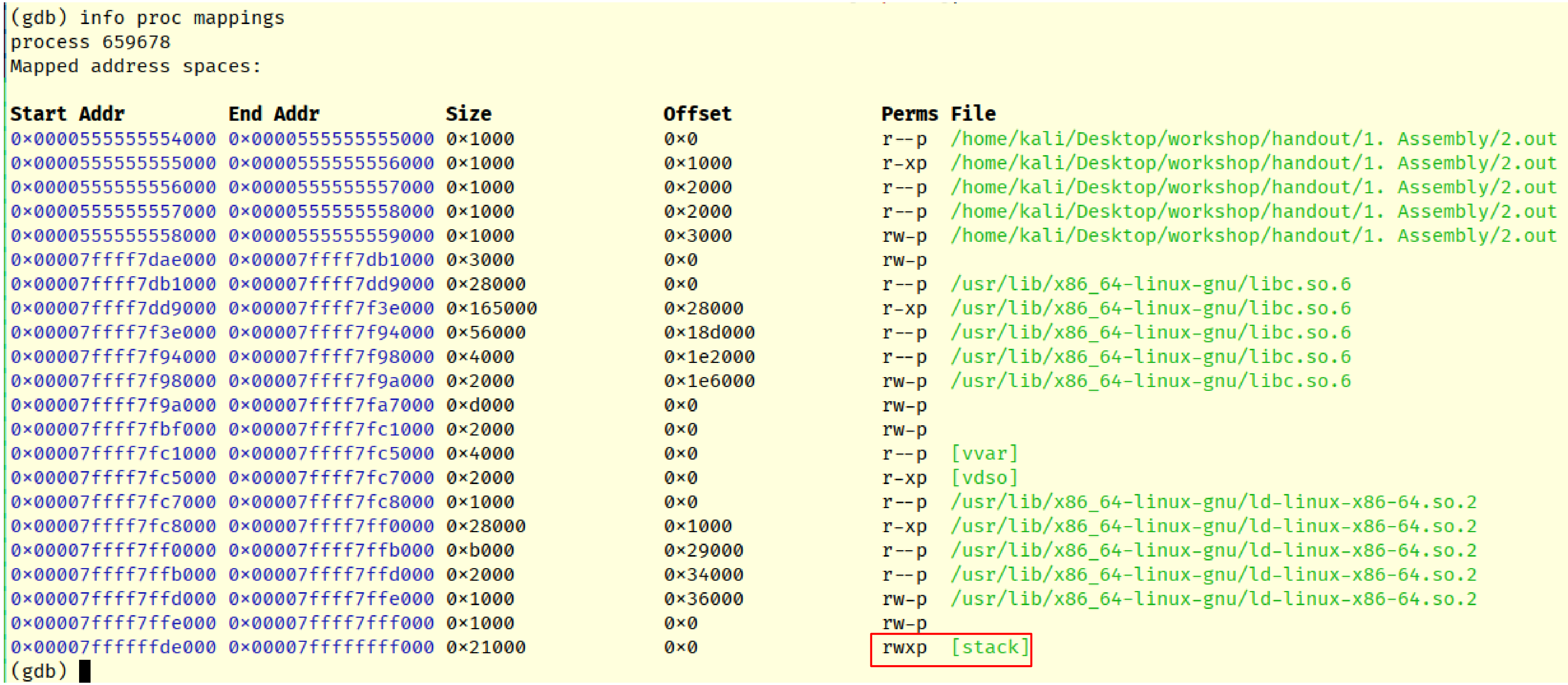
Exemplary dumped sections from ELF

Mapping programs to memory

- Binary cannot be run in the void. It first has to be loaded in the memory, together with any libraries it depends on.
- Every program gets its own virtual address space provided by OS.
- This is what sections are used for - loader maps them in the memory of the program and assigns necessary permissions (r/w/x)
  - For example, .text section becomes a memory region with read/execute privileges.
- Any necessary dependencies are mapped into the program's address space too
  - Linux uses .so (shared object) library files, Windows users are probably familiar with its counterpart – DLLs
- After loading binaries into memory, other vital regions are mapped (such as stack or heap)
- Finally, code starts executing at entry point

Mapping example

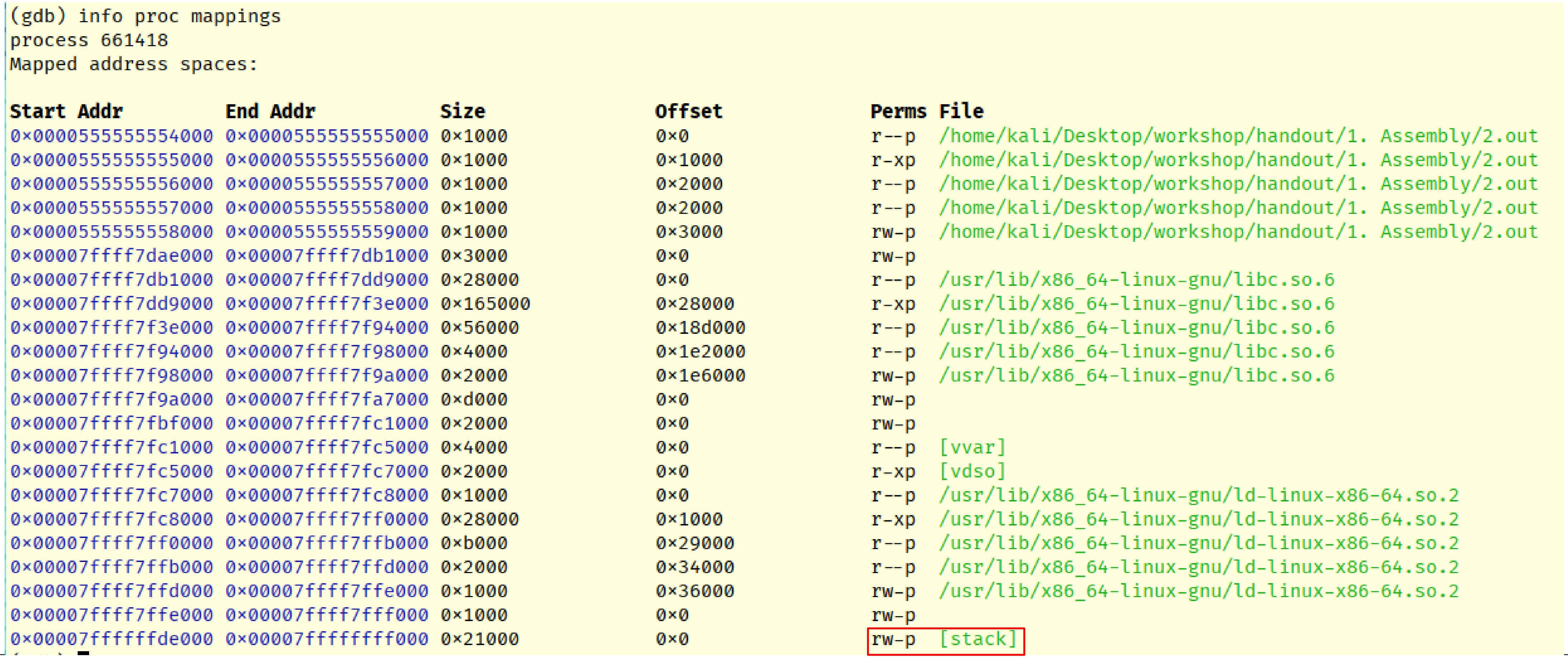
Here’s exemplary mapping with no NX enabled.
Notice stack’s permissions



```
(gdb) info proc mappings
process 659678
Mapped address spaces:

Start Addr         End Addr           Size               Offset             Perms File
0x0000555555554000 0x0000555555555000 0x1000             0x0                r--p  /home/kali/Desktop/workshop/handout/1. Assembly/2.out
0x0000555555555000 0x0000555555556000 0x1000             0x1000             r-xp  /home/kali/Desktop/workshop/handout/1. Assembly/2.out
0x0000555555556000 0x0000555555557000 0x1000             0x2000             r--p  /home/kali/Desktop/workshop/handout/1. Assembly/2.out
0x0000555555557000 0x0000555555558000 0x1000             0x2000             r--p  /home/kali/Desktop/workshop/handout/1. Assembly/2.out
0x0000555555558000 0x0000555555559000 0x1000             0x3000             rw-p  /home/kali/Desktop/workshop/handout/1. Assembly/2.out
0x00007ffff7dae000 0x00007ffff7db1000 0x3000             0x0                rw-p
0x00007ffff7db1000 0x00007ffff7dd9000 0x28000            0x0                r--p  /usr/lib/x86_64-linux-gnu/libc.so.6
0x00007ffff7dd9000 0x00007ffff7f3e000 0x165000           0x28000            r-xp  /usr/lib/x86_64-linux-gnu/libc.so.6
0x00007ffff7f3e000 0x00007ffff7f94000 0x56000            0x18d000           r--p  /usr/lib/x86_64-linux-gnu/libc.so.6
0x00007ffff7f94000 0x00007ffff7f98000 0x4000             0x1e2000           r--p  /usr/lib/x86_64-linux-gnu/libc.so.6
0x00007ffff7f98000 0x00007ffff7f9a000 0x2000             0x1e6000           rw-p  /usr/lib/x86_64-linux-gnu/libc.so.6
0x00007ffff7f9a000 0x00007ffff7fa7000 0xd000             0x0                rw-p
0x00007ffff7fbf000 0x00007ffff7fc1000 0x2000             0x0                rw-p
0x00007ffff7fc1000 0x00007ffff7fc5000 0x4000             0x0                r--p  [vvar]
0x00007ffff7fc5000 0x00007ffff7fc7000 0x2000             0x0                r-xp  [vdso]
0x00007ffff7fc7000 0x00007ffff7fc8000 0x1000             0x0                r--p  /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7fc8000 0x00007ffff7ff0000 0x28000            0x1000             r-xp  /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7ff0000 0x00007ffff7ffb000 0xb000             0x29000            r--p  /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7ffb000 0x00007ffff7ffd000 0x2000             0x34000            r--p  /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7ffd000 0x00007ffff7ffe000 0x1000             0x36000            rw-p  /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7ffe000 0x00007ffff7fff000 0x1000             0x0                rw-p
0x00007ffffffde000 0x00007ffffffff000 0x21000            0x0                rwxp  [stack]
(gdb) 
```

Mapping example

Here’s the same binary with NX enabled.
Notice stack’s permissions



```
(gdb) info proc mappings
process 661418
Mapped address spaces:

Start Addr         End Addr           Size               Offset             Perms  File
0x0000555555554000 0x0000555555555000 0x1000             0x0                r--p   /home/kali/Desktop/workshop/handout/1. Assembly/2.out
0x0000555555555000 0x0000555555556000 0x1000             0x1000             r-xp   /home/kali/Desktop/workshop/handout/1. Assembly/2.out
0x0000555555556000 0x0000555555557000 0x1000             0x2000             r--p   /home/kali/Desktop/workshop/handout/1. Assembly/2.out
0x0000555555557000 0x0000555555558000 0x1000             0x2000             r--p   /home/kali/Desktop/workshop/handout/1. Assembly/2.out
0x0000555555558000 0x0000555555559000 0x1000             0x3000             rw-p   /home/kali/Desktop/workshop/handout/1. Assembly/2.out
0x00007ffff7dae000 0x00007ffff7db1000 0x3000             0x0                rw-p
0x00007ffff7db1000 0x00007ffff7dd9000 0x28000            0x0                r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
0x00007ffff7dd9000 0x00007ffff7f3e000 0x165000           0x28000            r-xp   /usr/lib/x86_64-linux-gnu/libc.so.6
0x00007ffff7f3e000 0x00007ffff7f94000 0x56000            0x18d000           r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
0x00007ffff7f94000 0x00007ffff7f98000 0x4000             0x1e2000           r--p   /usr/lib/x86_64-linux-gnu/libc.so.6
0x00007ffff7f98000 0x00007ffff7f9a000 0x2000             0x1e6000           rw-p   /usr/lib/x86_64-linux-gnu/libc.so.6
0x00007ffff7f9a000 0x00007ffff7fa7000 0xd000             0x0                rw-p
0x00007ffff7fbf000 0x00007ffff7fc1000 0x2000             0x0                rw-p
0x00007ffff7fc1000 0x00007ffff7fc5000 0x4000             0x0                r--p   [vvar]
0x00007ffff7fc5000 0x00007ffff7fc7000 0x2000             0x0                r-xp   [vdso]
0x00007ffff7fc7000 0x00007ffff7fc8000 0x1000             0x0                r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7fc8000 0x00007ffff7ff0000 0x28000            0x1000             r-xp   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7ff0000 0x00007ffff7ffb000 0xb000             0x29000            r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7ffb000 0x00007ffff7ffd000 0x2000             0x34000            r--p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7ffd000 0x00007ffff7ffe000 0x1000             0x36000            rw-p   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7ffe000 0x00007ffff7fff000 0x1000             0x0                rw-p
0x00007ffffffde000 0x00007ffffffff000 0x21000            0x0                rw-p   [stack]
```

ROP

- Now that we know how NX/DEP is achieved (by removing executable memory permissions from data regions), lets introduce ROP
- ROP or Return-Oriented Programming is a brilliant technique basing on the idea of code reuse.
- By reusing code from the executables that are already mapped into memory (program's binary and libraries) it is most of the time possible to craft a functional shellcode
- To achieve this, exploit developer :
  - looks for assembly instruction chains (called gadgets) that end with RET in the target binary
  - drafts a functional shellcode using these instruction chains (called ROP chain)
  - places addresses pointing to gadgets on the stack, one after another
  - using memory corruption bug, overwrites RIP to point to the first gadget
  - program starts executing these small assembly instruction sets. Since each is ending with RET, they will pop address of next one from the stack and continue execution.

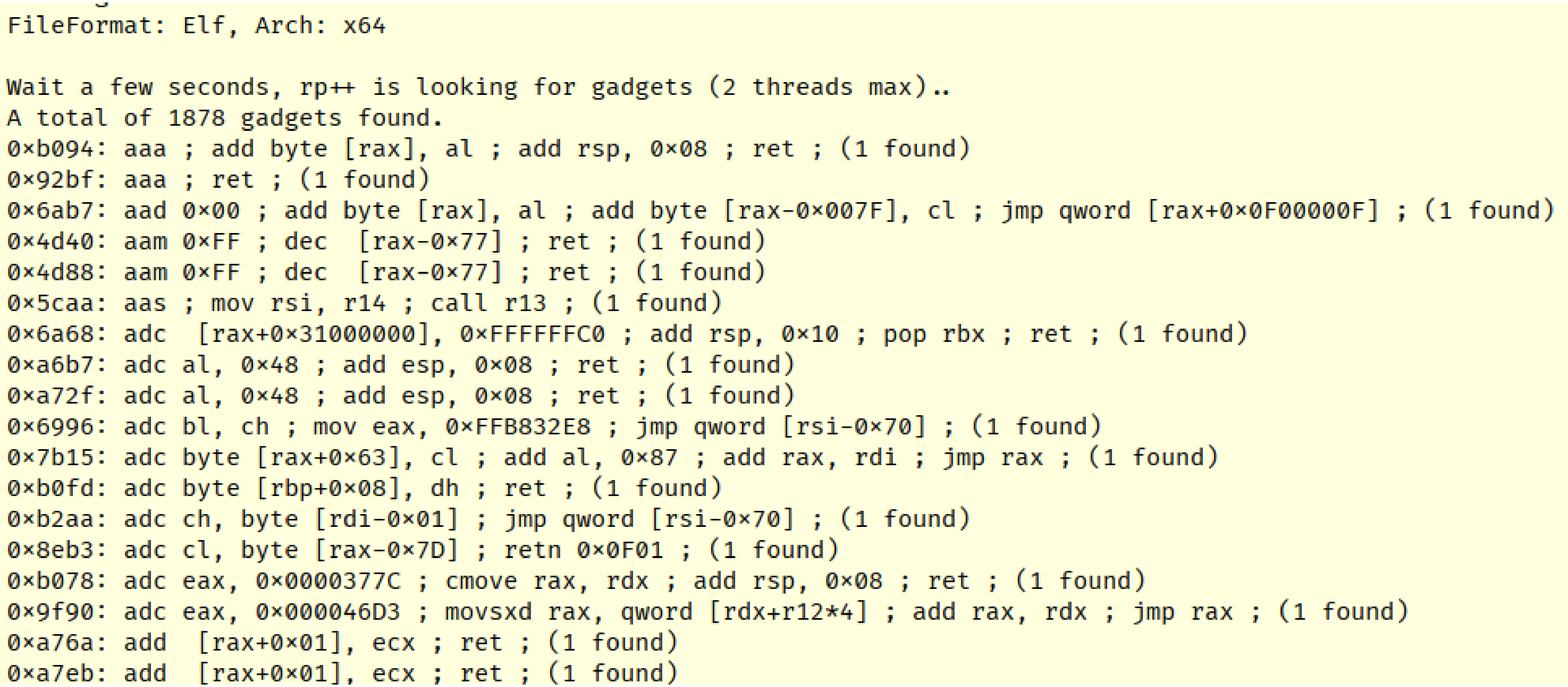
The important part is that stack is not containing code anymore, just pointers to pieces of legitimate code that are fetched through RETs into RIP. This does not violate NX/DEP since it's not stack memory that is being executed. Through creative combining these pieces, malicious effect is achieved.

We are definitely in dire need of visuals for this one!

ROP

The following is how an exploit developer would look for instruction chains with rp++.

You can see offset in a binary where each chain begins and its constituent instructions. The idea is to combine such chains into something useful. Essentially, it’s puzzle-like shellcoding with constraints.



```
FileFormat: Elf, Arch: x64

Wait a few seconds, rp++ is looking for gadgets (2 threads max)..
A total of 1878 gadgets found.
0xb094: aaa ; add byte [rax], al ; add rsp, 0x08 ; ret ; (1 found)
0x92bf: aaa ; ret ; (1 found)
0x6ab7: aad 0x00 ; add byte [rax], al ; add byte [rax-0x007F], cl ; jmp qword [rax+0x0F00000F] ; (1 found)
0x4d40: aam 0xFF ; dec  [rax-0x77] ; ret ; (1 found)
0x4d88: aam 0xFF ; dec  [rax-0x77] ; ret ; (1 found)
0x5caa: aas ; mov rsi, r14 ; call r13 ; (1 found)
0x6a68: adc  [rax+0x31000000], 0xFFFFFFC0 ; add rsp, 0x10 ; pop rbx ; ret ; (1 found)
0xa6b7: adc al, 0x48 ; add esp, 0x08 ; ret ; (1 found)
0xa72f: adc al, 0x48 ; add esp, 0x08 ; ret ; (1 found)
0x6996: adc bl, ch ; mov eax, 0xFFB832E8 ; jmp qword [rsi-0x70] ; (1 found)
0x7b15: adc byte [rax+0x63], cl ; add al, 0x87 ; add rax, rdi ; jmp rax ; (1 found)
0xb0fd: adc byte [rbp+0x08], dh ; ret ; (1 found)
0xb2aa: adc ch, byte [rdi-0x01] ; jmp qword [rsi-0x70] ; (1 found)
0x8eb3: adc cl, byte [rax-0x7D] ; retn 0x0F01 ; (1 found)
0xb078: adc eax, 0x0000377C ; cmove rax, rdx ; add rsp, 0x08 ; ret ; (1 found)
0x9f90: adc eax, 0x000046D3 ; movsxd rax, qword [rdx+r12*4] ; add rax, rdx ; jmp rax ; (1 found)
0xa76a: add  [rax+0x01], ecx ; ret ; (1 found)
0xa7eb: add  [rax+0x01], ecx ; ret ; (1 found)
```

Libc ROP chain

- Below is a draft of ROP chain I've prepared from the gadgets pulled from libc.
- Imagine them being executed sequentially, it is essentially our shell-spawning shellcode from earlier chapter done with different instructions.

```
0x7d1b2:         xor edx, edx ; mov eax, edx ; ret
0x3f80b:         pop rax ; ret
0x3b – execve syscall no.
0x28bb2:         pop rdi ; ret
<libc /bin/sh address*>
0xfc77e:         xor esi, esi ; syscall
```

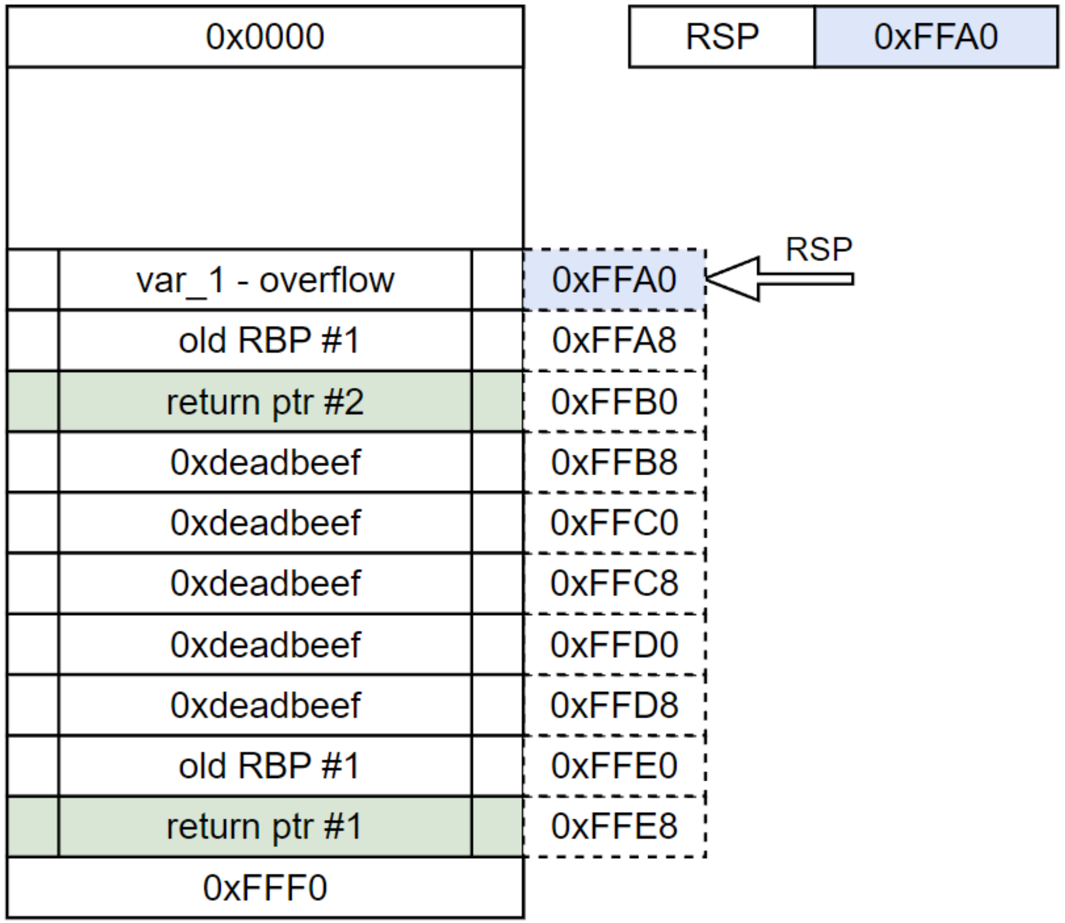
*what's with the libc address? Turns out libc library contains hardcoded string "/bin/sh", which is extremely useful for us as we can just point to its' address in execve syscall.

btw – libc is always mapped into program's memory if dynamically linked (default behaviour)

ROP visualized

Imagine the following stack setup in some function call with 40 bytes reserved for variables in previous frame

we overflow in here



RSP	0xFFA0

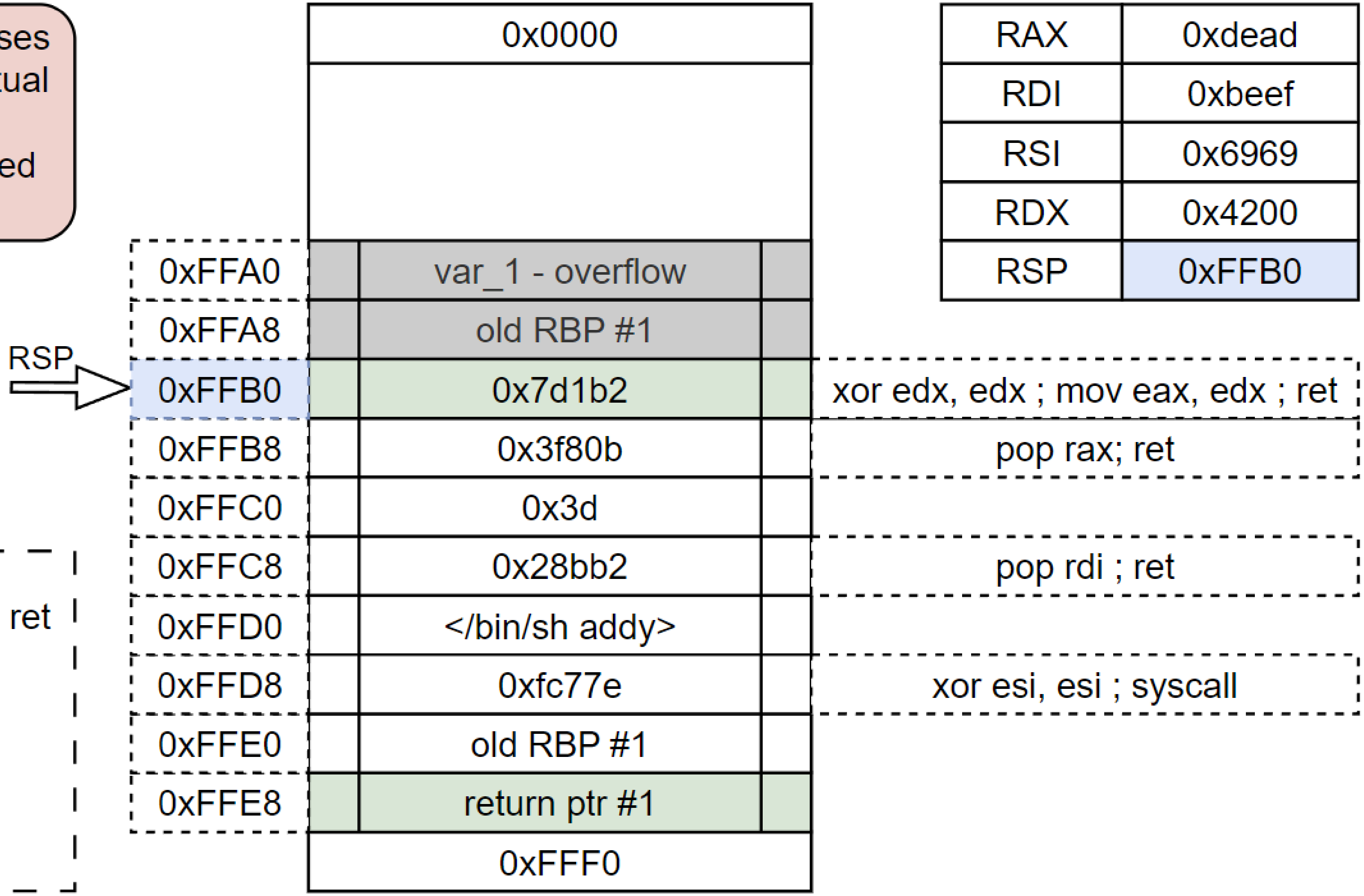
RSP

0x0000	
var_1 - overflow	0xFFA0
old RBP #1	0xFFA8
return ptr #2	0xFFB0
0xdeadbeef	0xFFB8
0xdeadbeef	0xFFC0
0xdeadbeef	0xFFC8
0xdeadbeef	0xFFD0
0xdeadbeef	0xFFD8
old RBP #1	0xFFE0
return ptr #1	0xFFE8
0xFFF0	

```
0x7d1b2: xor edx, edx ; mov eax, edx ; ret
0x3f80b: pop rax ; ret
0x3d
0x28bb2: pop rdi ; ret
<libc /bin/sh address>
0xfc77e: xor esi, esi ; syscall
```

ROP visualized

We overflow so that gadgets' addresses are placed on stack together with actual data we want to use (0x3d).
Function is now exiting, next executed instruction will be RET.



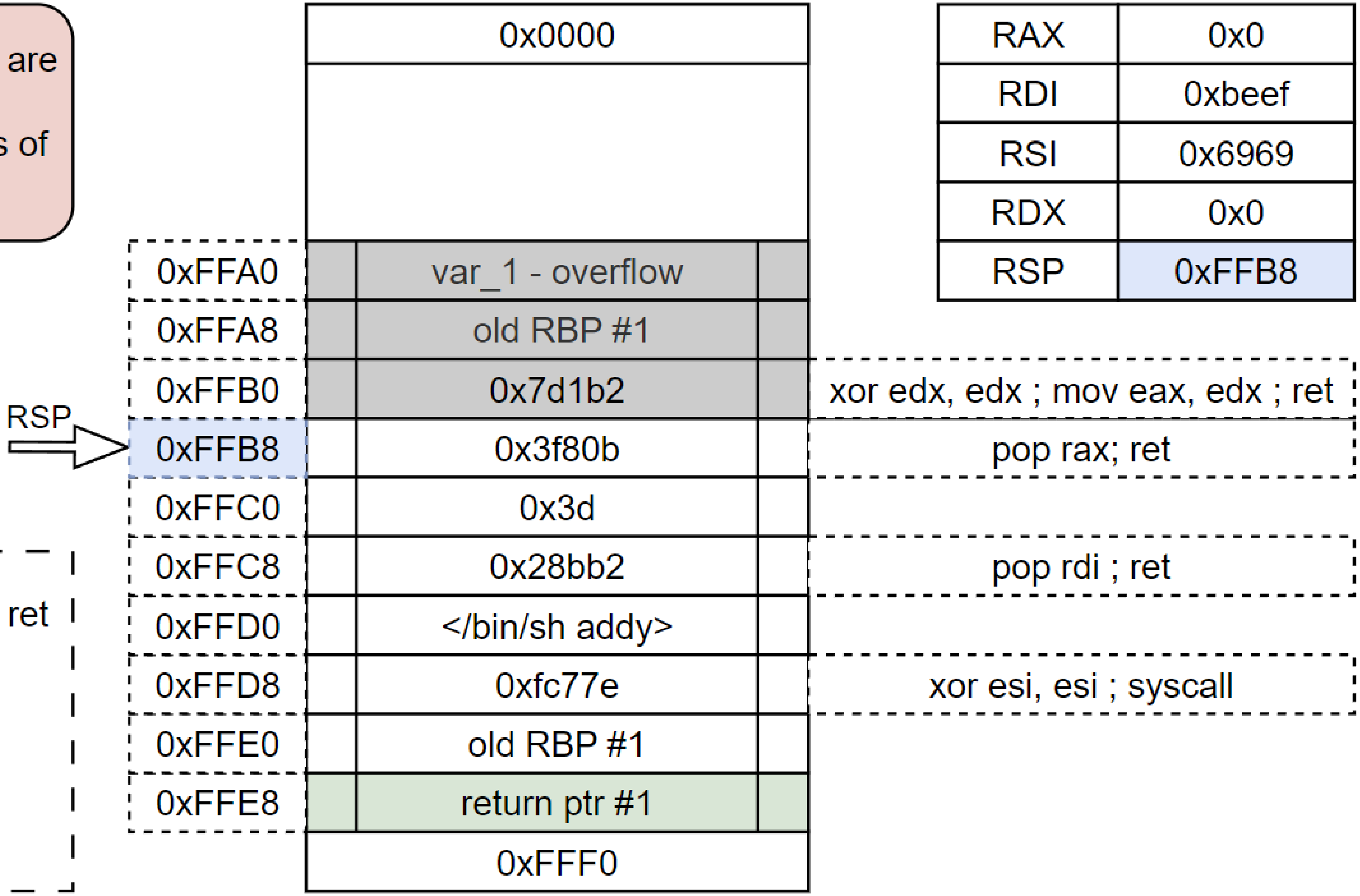
	0x0000	
0xFFA0	var_1 - overflow	
0xFFA8	old RBP #1	
RSP → 0xFFB0	0x7d1b2	xor edx, edx ; mov eax, edx ; ret
0xFFB8	0x3f80b	pop rax; ret
0xFFC0	0x3d	
0xFFC8	0x28bb2	pop rdi ; ret
0xFFD0	</bin/sh addy>	
0xFFD8	0xfc77e	xor esi, esi ; syscall
0xFFE0	old RBP #1	
0xFFE8	return ptr #1	
	0xFFF0	

RAX	0xdead
RDI	0xbeef
RSI	0x6969
RDX	0x4200
RSP	0xFFB0

0x7d1b2: xor edx, edx ; mov eax, edx ; ret
0x3f80b: pop rax ; ret
0x3d
0x28bb2: pop rdi ; ret
<libc /bin/sh address>
0xfc77e: xor esi, esi ; syscall

ROP visualized

gadget gets executed, RDX and RAX are zeroed.
RET is called again, popping address of next gadget into RIP



	0x0000	
0xFFA0	var_1 - overflow	
0xFFA8	old RBP #1	
0xFFB0	0x7d1b2	xor edx, edx ; mov eax, edx ; ret
0xFFB8	0x3f80b	pop rax; ret
0xFFC0	0x3d	
0xFFC8	0x28bb2	pop rdi ; ret
0xFFD0	</bin/sh addy>	
0xFFD8	0xfc77e	xor esi, esi ; syscall
0xFFE0	old RBP #1	
0xFFE8	return ptr #1	
	0xFFF0	

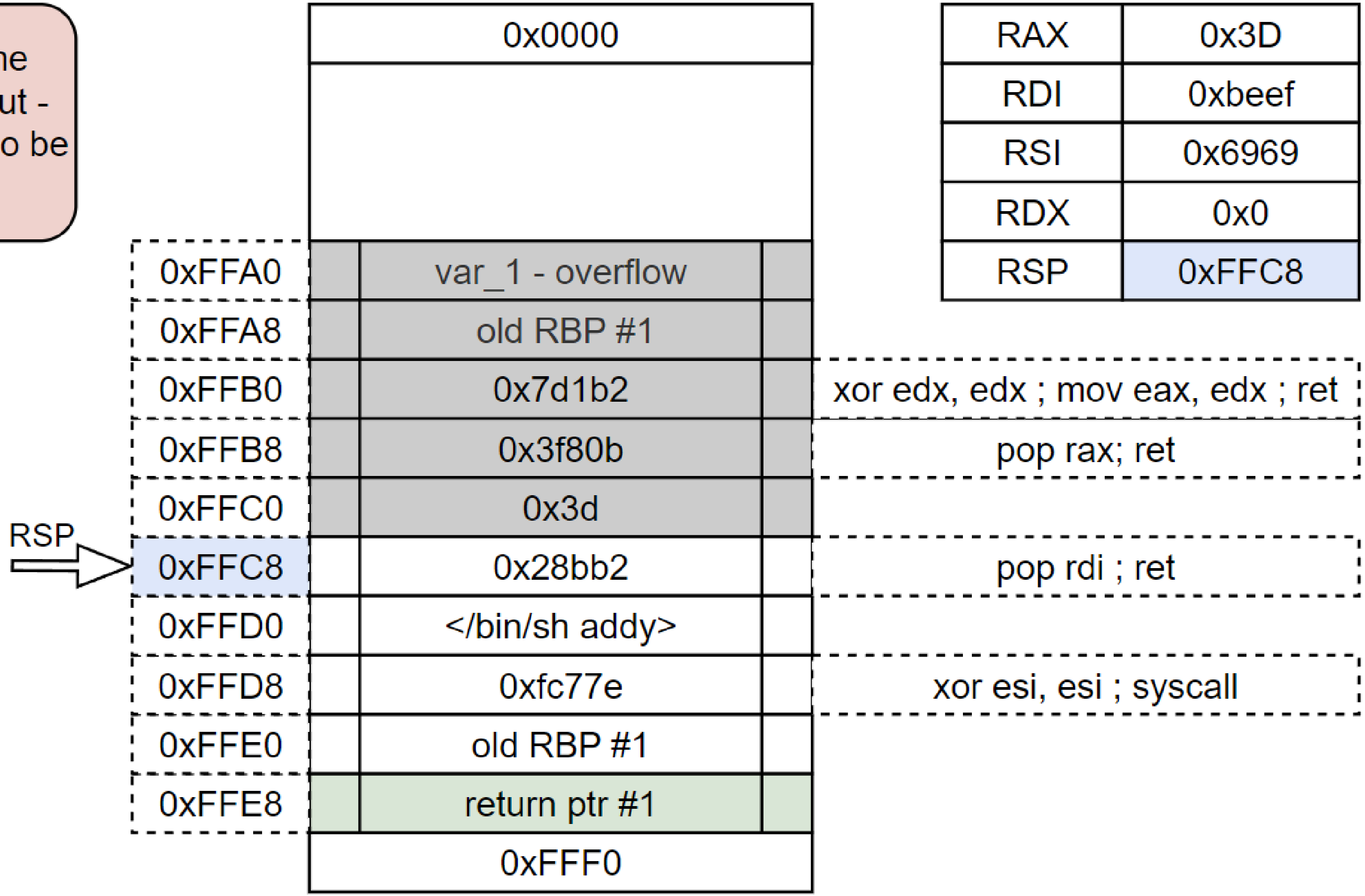
RAX	0x0
RDI	0xbeef
RSI	0x6969
RDX	0x0
RSP	0xFFB8

0x7d1b2: xor edx, edx ; mov eax, edx ; ret
0x3f80b: pop rax ; ret
0x3d
0x28bb2: pop rdi ; ret
<libc /bin/sh address>
0xfc77e: xor esi, esi ; syscall

ROP visualized

syscall number gets popped from the stack notice how it affects stack layout - arguments to our gadgets are bound to be placed after them.

0x7d1b2: xor edx, edx ; mov eax, edx ; ret
0x3f80b: pop rax ; ret
0x3d
0x28bb2: pop rdi ; ret
<libc /bin/sh address>
0xfc77e: xor esi, esi ; syscall



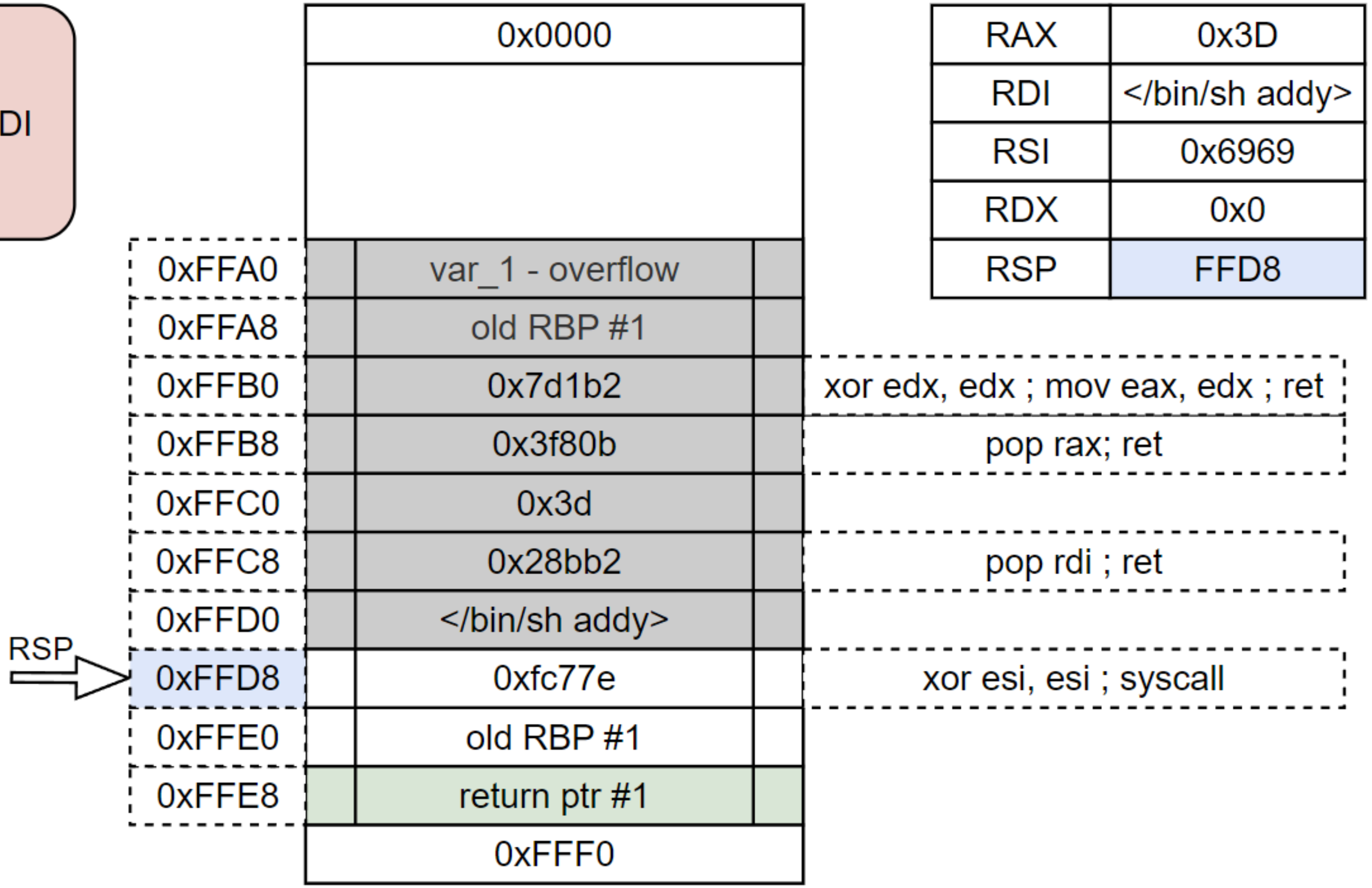
	0x0000	
0xFFA0	var_1 - overflow	
0xFFA8	old RBP #1	
0xFFB0	0x7d1b2	xor edx, edx ; mov eax, edx ; ret
0xFFB8	0x3f80b	pop rax; ret
0xFFC0	0x3d	
0xFFC8	0x28bb2	pop rdi ; ret
0xFFD0	</bin/sh addy>	
0xFFD8	0xfc77e	xor esi, esi ; syscall
0xFFE0	old RBP #1	
0xFFE8	return ptr #1	
	0xFFF0	

RAX	0x3D
RDI	0xbeef
RSI	0x6969
RDX	0x0
RSP	0xFFC8

ROP visualized

address of /bin/sh string gets into RDI

0x7d1b2: xor edx, edx ; mov eax, edx ; ret
0x3f80b: pop rax ; ret
0x3d
0x28bb2: pop rdi ; ret
<libc /bin/sh address>
0xfc77e: xor esi, esi ; syscall



	0x0000	
0xFFA0	var_1 - overflow	
0xFFA8	old RBP #1	
0xFFB0	0x7d1b2	xor edx, edx ; mov eax, edx ; ret
0xFFB8	0x3f80b	pop rax; ret
0xFFC0	0x3d	
0xFFC8	0x28bb2	pop rdi ; ret
0xFFD0	</bin/sh addy>	
0xFFD8	0xfc77e	xor esi, esi ; syscall
0xFFE0	old RBP #1	
0xFFE8	return ptr #1	
	0xFFF0	

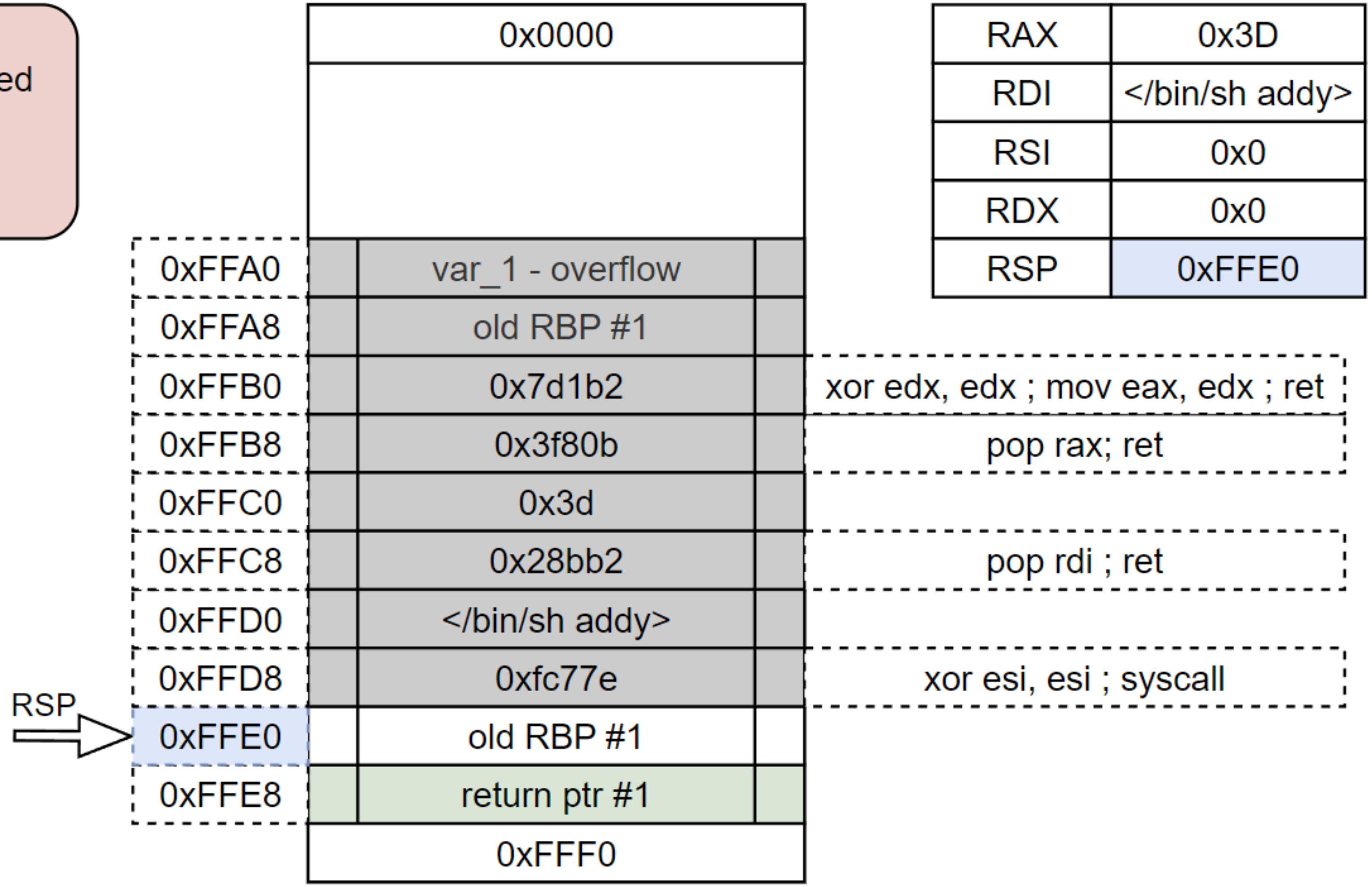
RAX	0x3D
RDI	</bin/sh addy>
RSI	0x6969
RDX	0x0
RSP	FFD8

ROP visualized

RSI gets zeroed and syscall is issued

at this point, shell is spawned

```
0x7d1b2: xor edx, edx ; mov eax, edx ; ret
0x3f80b: pop rax ; ret
0x3d
0x28bb2: pop rdi ; ret
<libc /bin/sh address>
0xfc77e: xor esi, esi ; syscall
```



	0x0000	
0xFFA0	var_1 - overflow	
0xFFA8	old RBP #1	
0xFFB0	0x7d1b2	xor edx, edx ; mov eax, edx ; ret
0xFFB8	0x3f80b	pop rax; ret
0xFFC0	0x3d	
0xFFC8	0x28bb2	pop rdi ; ret
0xFFD0	</bin/sh addy>	
0xFFD8	0xfc77e	xor esi, esi ; syscall
0xFFE0	old RBP #1	
0xFFE8	return ptr #1	
	0xFFF0	

RAX	0x3D
RDI	</bin/sh addy>
RSI	0x0
RDX	0x0
RSP	0xFFE0

⌛ 40 minutes

LAB 6

Bypassing DEP with ROPs

Instruction:

handout/lab6.pdf

ASLR

- Up to this point, all addresses we've used for exploitation were hardcoded – we knew them ahead of time, ASLR changes that.
- ASLR or Address Space Layout Randomization is a mechanism of randomizing base addresses of all modules loaded into the program's memory.
- Addresses are randomized with every binary launch.
- This seemingly minor change forces us to leak addresses from application during runtime. Effectively this requires us finding at least two bugs – infoleak and memory corruption
- ASLR is enforced by kernel; not a compilation flag.

ASLR

- To reiterate - ASLR is a mechanism of randomizing **base addresses** of all modules loaded into the program's memory.
- Despite being powerful, ASLR can't just take any binary and shuffle its' internal addresses around
- Instead, it randomizes the **base address** only, that is, the start address of where the binary will be mapped into memory.
  - This means that all offsets, or "distances" between pieces of code inside binary remain the same
  - This process is applied separately for every library loaded into memory
- To bypass ASLR, we essentially need to leak address of anything stored in the target binary. We can then calculate its' offset from the base of binary and obtain the base address

ASLR bypass example

Assume we want to execute system() from libc.

- We obtain its’ offset from base of libc.so.6; in this case it is equal to 0x53110
- We find and utilize an infoleak in the app.
  - Let’s assume we’ve leaked an address of printf() from libc - 0x7ffff7e0a900
- In the same manner as with system(), we obtain printf’s offset from libc base - 0x59900
- Now we can calculate the base address, where libc has been mapped
  - libc_base = 0x7ffff7e0a900 - 0x59900 = 0x00007ffff7db1000
- To obtain system()’s addy, all we need to do is to apply its’ offset to base:
  - system_addy = libc_base + 0x53110 = 0x7ffff7e04110
- Finally, we plug this address into our payload and execute
- That’s why automating our payloads goes a long way, we can do all that in few lines of code

PIE

- One important consideration, albeit rarely existing today, is PIE
- PIE stands for Position-Independent Executable
- It’s a way how programs can be compiled and allows the code to be placed at artificial memory address and get executed
  - This mainly has to do with how historically a lot of addresses were hardcoded, forcing binaries to be mapped consistently (at the same base addresses) over time.
  - PIE requires everything work with offsets, no hardcoded pointers
- A binary has to be compiled with PIE support to make ASLR work. It will break otherwise.
- Almost everything is now compiled with PIE enabled, however if you ever find a binary without one, its’ addresses can be used for ASLR bypass.

How a leak can look like

```
struct user{
            char username[32];
            void (*printf_add)();
} g_user;
```

- Leaks can be different and subtle but in general to leak a pointer you'd look to abuse some printing functionality and/or custom structures placed by the programmer
- There is no blueprint for leaking things as these are always application-dependent
- A good example would be a custom structure, containing a user-supplied input and some address, such as the one on the top of this slide
  - Remember that C-style strings are null-delimited and most of the printing functions will read UNTIL nullbyte
  - Imagine you were able to overwrite the username in the example struct with exactly 32 non-zero characters
  - What will happen if puts(username) is called?
  - We'll explore this in the final lab

⧗ 40 minutes

LAB 7

Bypassing ASLR

Instruction:

handout/lab7.pdf

Conclusion

- It has been a long day, congratulations for going through
- Are you confused, tired, overwhelmed? That's perfectly right
    - This has been a LOAD of knowledge and if you have no familiarity with the topic, don't expect to remember everything
    - The point is, if you want any of this to stick, you got to practice. There are some challenges to go through and I will probably add more over time.
- Some of really great resources
    - https://www.corelan.be/index.php/articles/ - a classic with focus on Windows, most of the time outdated but the fundamentals do not change
    - https://wargames.ret2.systems/ - paid but oh-so-wonderful
    - https://guyinatuxedo.github.io/ - it has not worked for me but it just might for you
    - Books – some love them some hate them, I love a good technical book, especially a little older
        - Hacking – the art of exploitation
        - The Shellcoder's Handbook
        - PoC || GTFO