

## ROP 32bit Linux

- Function arguments are pushed to the stack
- Has to be taken into account when assembling the ROP chain

## Recap Buffer Overflow

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

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

```
main:
;eax holds pointer
;to argv[1]
push eax
call vuln(char*)
...
```

0x00000000

ESP  
(Top of Stack)

0x7FFFFFFF

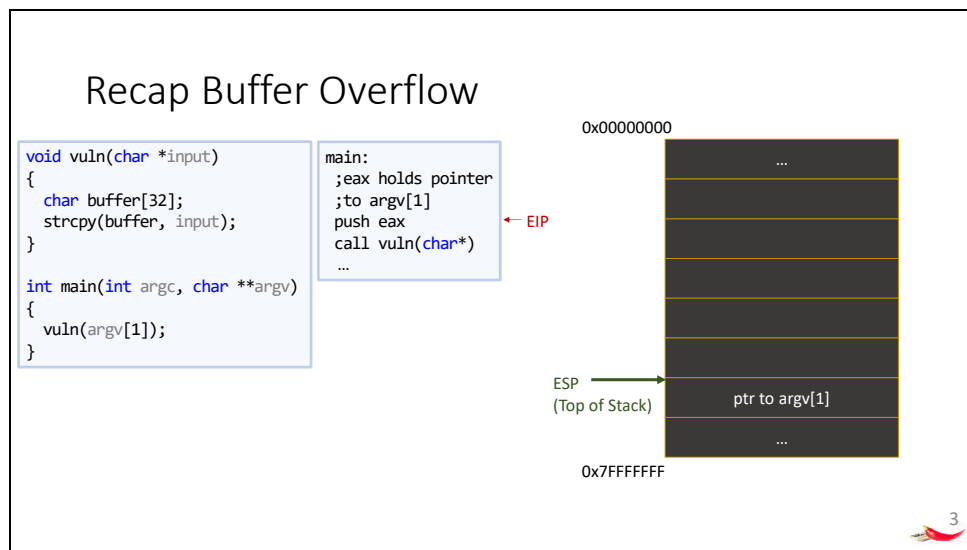
2

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 to a vulnerable function.  
That vuln function copies the input to a local buffer.  
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.

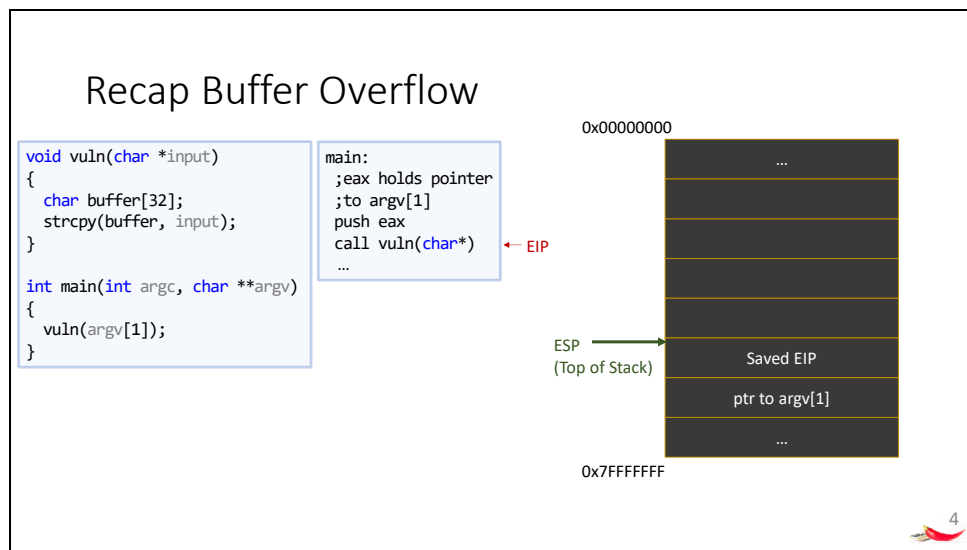
In the middle you can see the compiled assembly code, on the right side there is a sketch of the current stack.



Let's start the execution just before the call to the vuln function.

In my diagram the red instruction pointer, called EIP always points to the instruction that just has been executed.

Here we see that the pointer to argv, that's the input from the user stored in eax, just got pushed to the stack.



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.

## 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 ebp
mov ebp, esp
sub esp, 32
push[ebp+8]; input
lea eax, [ebp-32]
push eax; buffer
call strcpy
add esp, 8
leave
ret
```

0x00000000

← EIP

ESP → (Top of Stack)

0x7FFFFFFF

...
...
...
Saved EBP
Saved EIP
ptr to argv[1]
...
...

5

We enter the vuln function and the first thing we do is to save the base pointer EBP to the stack.

## 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 ebp
mov ebp, esp
sub esp, 32
push[ebp+8]; input
lea eax, [ebp-32]
push eax; buffer
call strcpy
add esp, 8
leave
ret
```

0x00000000

← EIP ESP (Top of Stack)

← buffer

0x7FFFFFFF

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

## 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 ebp
mov ebp, esp
sub esp, 32
push[ebp+8]; input ← EIP
lea eax, [ebp-32]
push eax; buffer
call strcpy
add esp, 8
leave
ret
```

0x00000000

ESP (Top of Stack) →

ptr to input ← buffer

Saved EBP

Saved EIP

ptr to argv[1]

...

0x7FFFFFFF

7

Next the arguments for the call to `strcpy` get pushed to the stack. Note that the first one that is pushed is the second argument: `input`.

## Recap Buffer Overflow

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

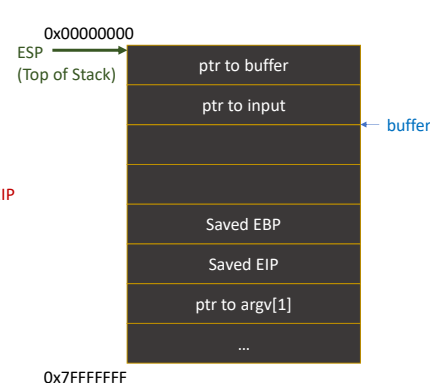
int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

```
vuln(char*) :
push ebp
mov ebp, esp
sub esp, 32
push[ebp+8]; input
lea eax, [ebp-32]
push eax; buffer ← EIP
call strcpy
add esp, 8
leave
ret
```

0x00000000  
ESP  
(Top of Stack)

ptr to buffer
ptr to input ← buffer
Saved EBP
Saved EIP
ptr to argv[1]
...

0x7FFFFFFF



8

Then the first argument of strcpy, the pointer to our buffer, is also pushed to the stack.



## 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 ebp
mov ebp, esp
sub esp, 32
push[ebp+8]; input
lea eax, [ebp-32]
push eax; buffer
call strcpy
add esp, 8
leave
ret

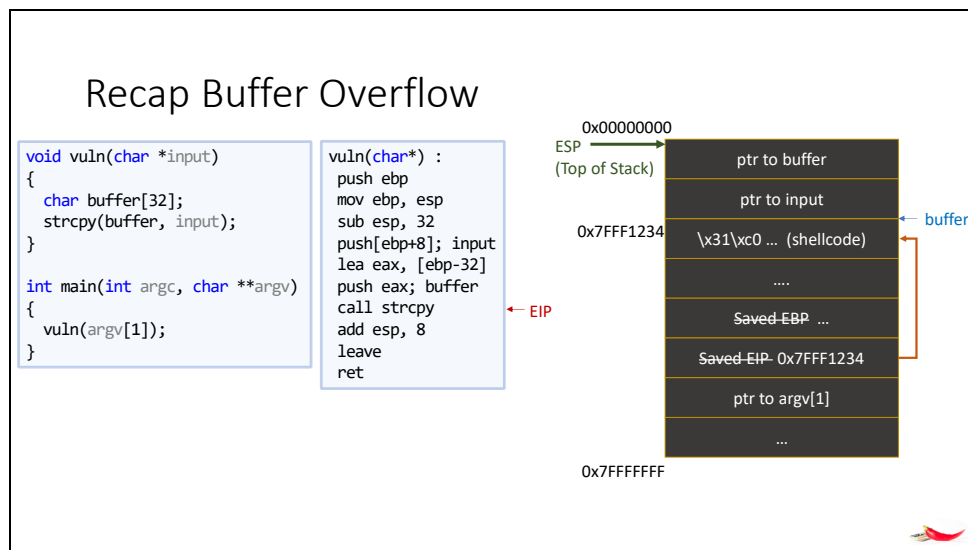
```

0x00000000  
ESP  
(Top of Stack)

0x7FFFFFFF

When strcpy is executed, the input string is copied into the buffer.

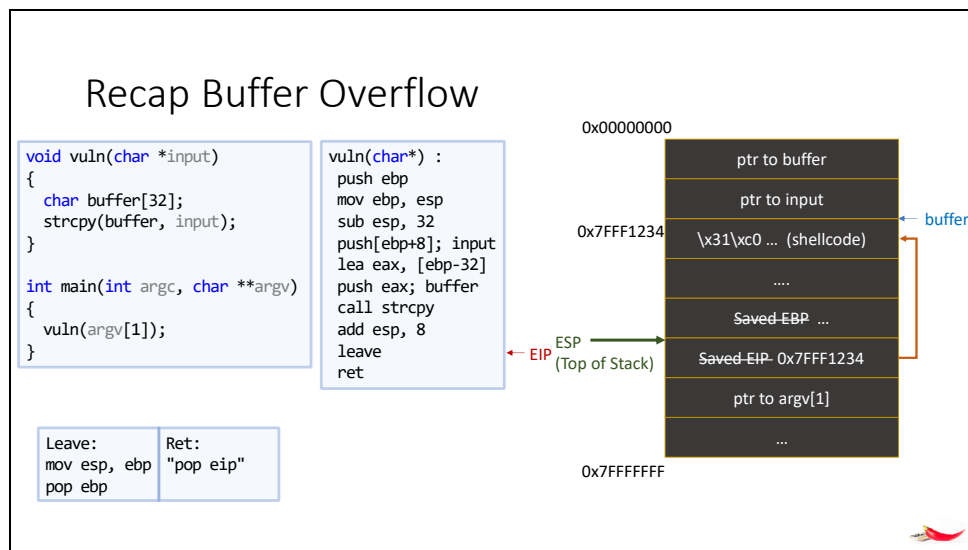
If the input string is longer than the buffer, strcpy will overwrite also the saved base pointer, instruction pointer and maybe more.



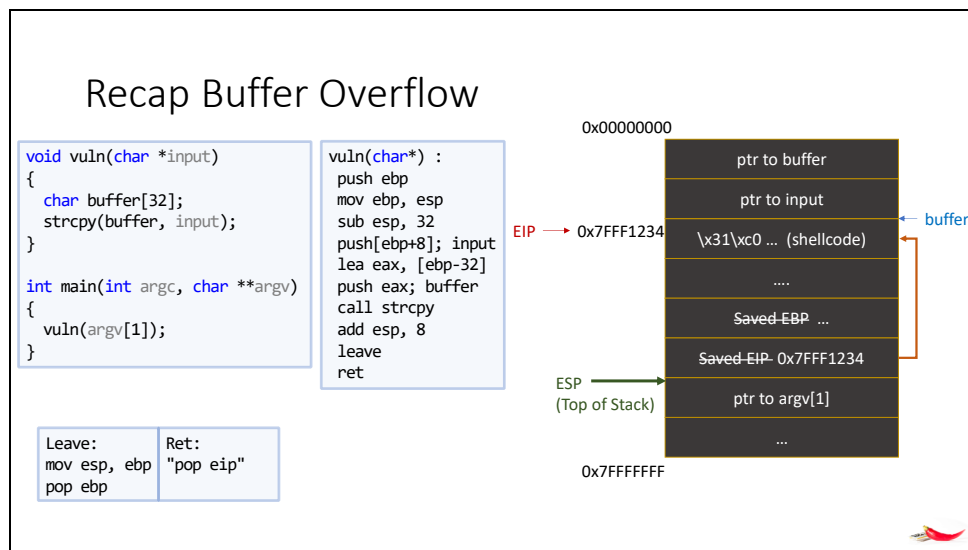
We can abuse this situation by writing shellcode into the buffer instead of As.

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 EIP with the address of our shellcode.



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 EIP so that execution continues at the popped address.



In our case this will cause EIP 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! However this only works if the stack is executable.

## 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)



## 32 Bit Calling Convention Linux – Ret2libc

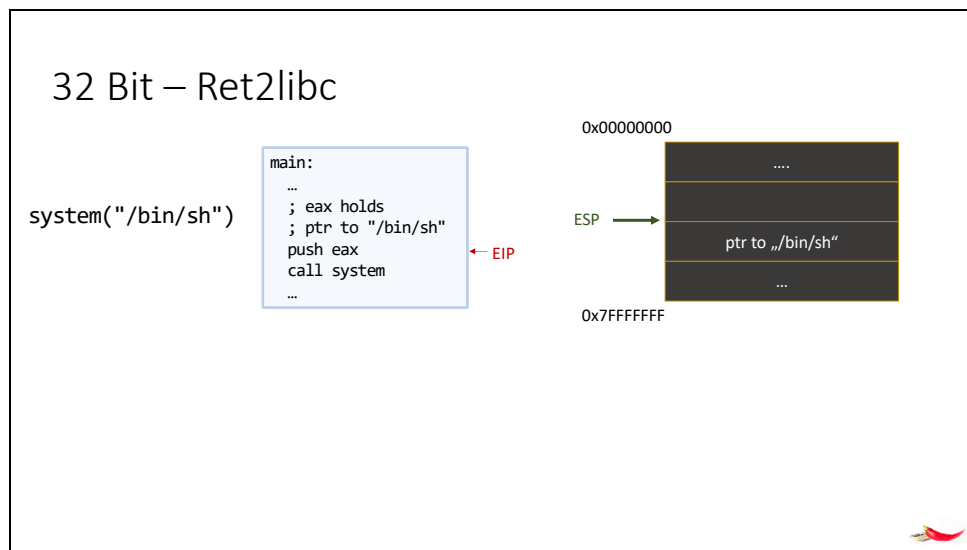
- Function arguments are pushed to the stack



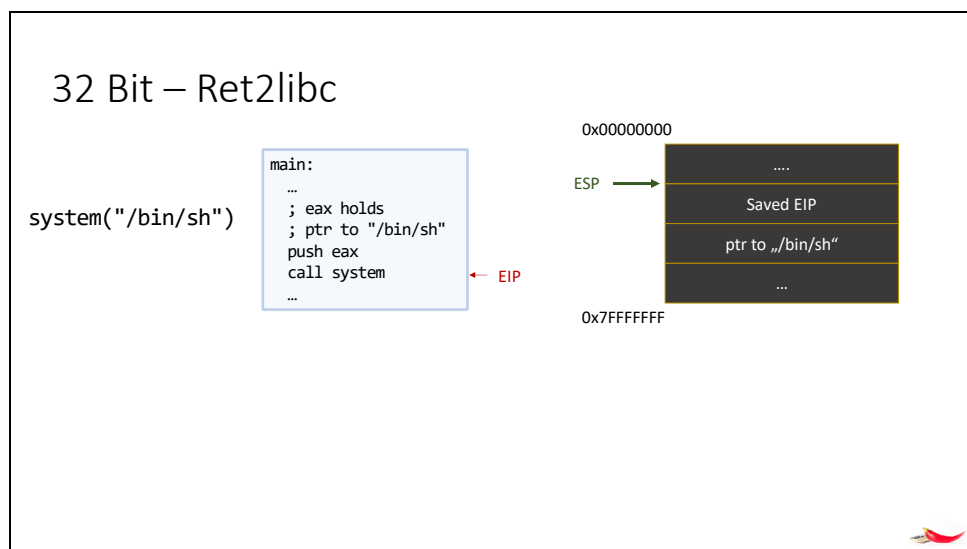
A quick note: Under 32 Bit in Linux the Calling Convention specifies that Function arguments are pushed to the stack

With our payload we want to achieve a call to system.

Lets look how system would be called normally first to see how our payload needs to be structured.

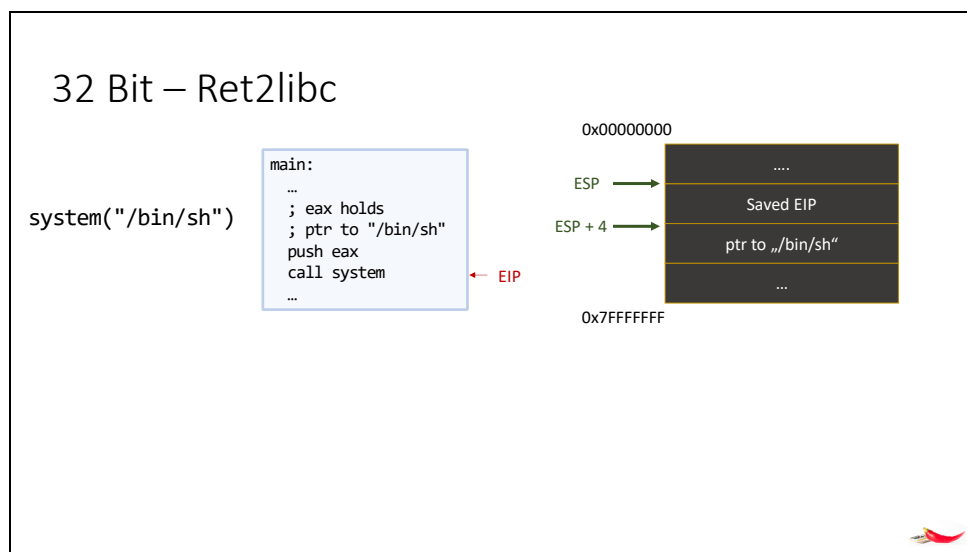


In the following you see how a call to system(„bin/sh“) would be normally executed.  
According to the 32 Bit calling convention, the function argument, in our case a 'pointer to the string („bin/sh“)' gets pushed to the stack.  
That just got executed here.  
After that, system is called.



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

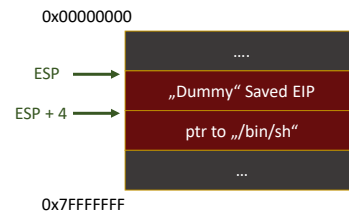




Notice that the argument for the function is expected at ESP+4 at the time of the function call.

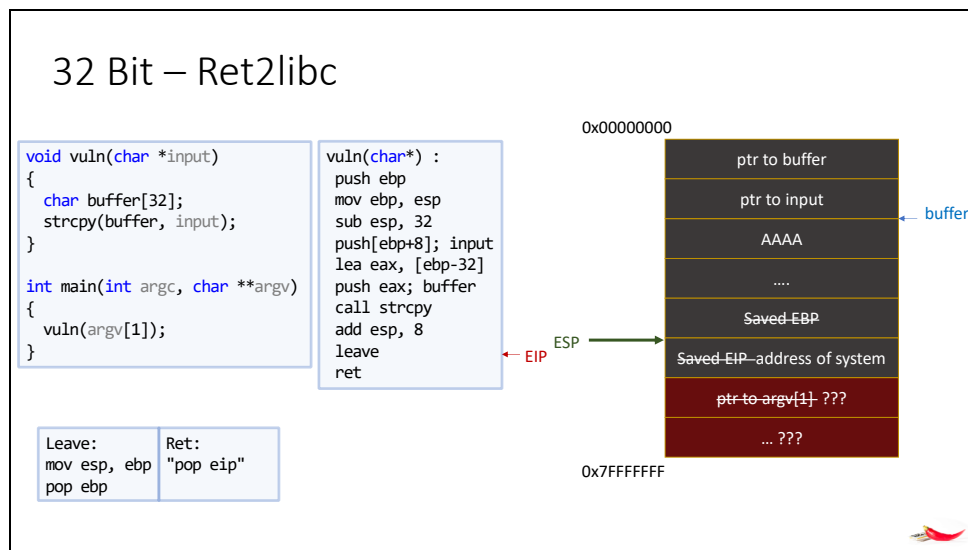
## 32 Bit – Ret2libc

- We want to call:  
`system("/bin/sh")`
- system expects the argument after the saved EIP, which means `esp+4` at the time of the call.



So this is the state of the stack we want to achieve:

We want to call `system(bin/sh)`, and `system` expects the argument after the saved EIP, that means at `ESP+4` at the time of the call.

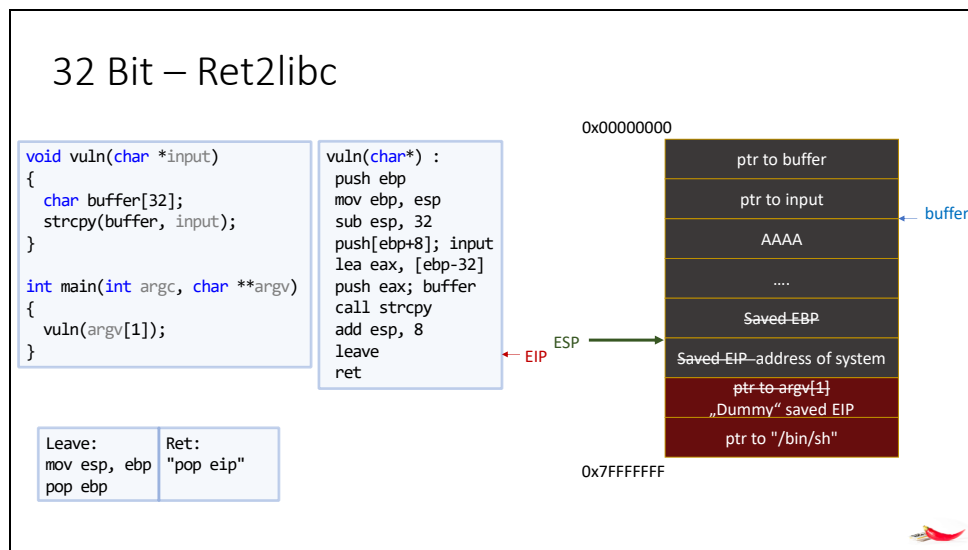


Lets include this in our payload.

Imagine the example from earlier.

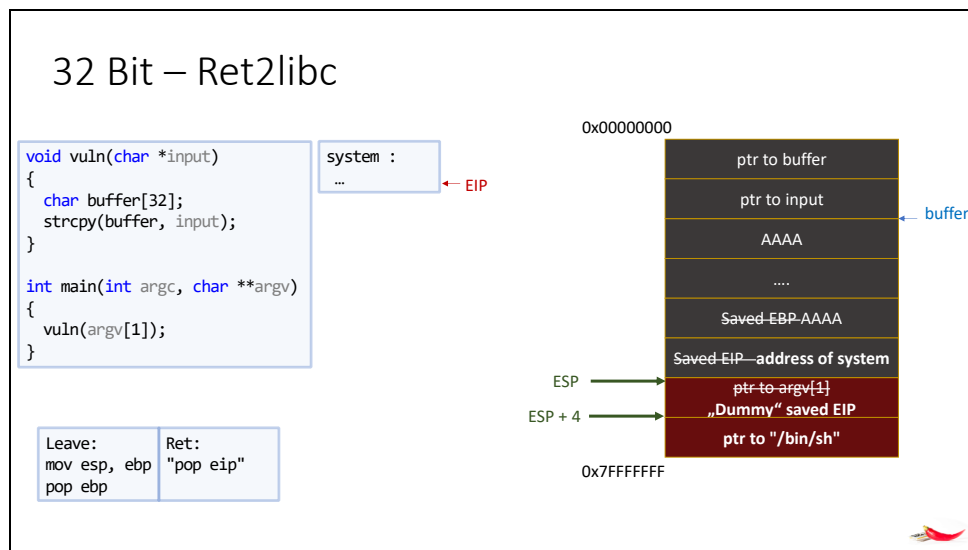
instead of overwriting the saved instruction pointer with the address of shellcode, we overwrite it with the address of system.

But we want to call system with an argument, with the address of (bin/sh).



System expects a saved instruction pointer on the stack at the time of the call, so we have to write a ‚dummy‘ EIP, before we can write the argument, which is a pointer to the string ‚/bin/sh‘).

Lets look what happens if we hit the return now, with this state of the stack:  
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 system.

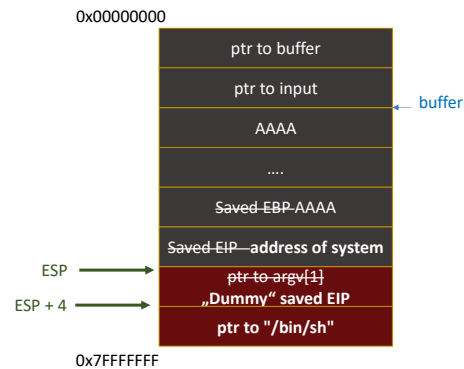


So execution continues with system, and system expects a return address at ESP. We store the dummy there because we will never reach this . System also expects its argument at ESP+4, which is with our payload, now „bin/sh“. Now this will get executed and give us a shell.

## 32 Bit – Ret2libc

Payload = "A" \* 32

- + "AAAA" (saved EBP)
- + address of system (saved EIP)
- + "BBBB" (Dummy saved EIP)
- + address of "/bin/sh"



So to summarize: the payload is :

The number of A's until we reach the saved EIP,

Then we overwrite the saved EIP with the address of system,

Then we give system a Dummy saved EIP, this can be anything, for example four Bs, or the address to exit to end the process properly after our shell closes.

And at last we give system the argument which we want to get executed, in our case, bin/sh.