CSCI 4907/6545 Software Security

Fall 2025

Instructor: Jie Zhou

Department of Computer Science
George Washington University

Slides materials are partially credited to Gang Tan of PSU.

Outline of Today's Lecture

- Brief review of last lecture
- Return-oriented Programming (ROP)
- Integer overflows
- Heap overflows

Programming *Correctly* in C is (Extremely) Hard

Simple and primitive language features

- Basic data types (char, integer, boolean, etc.)
- struct
- Pointers
- Basic control flow (conditional branches, loops, etc.)

Pointer: Capability to manipulate memory.

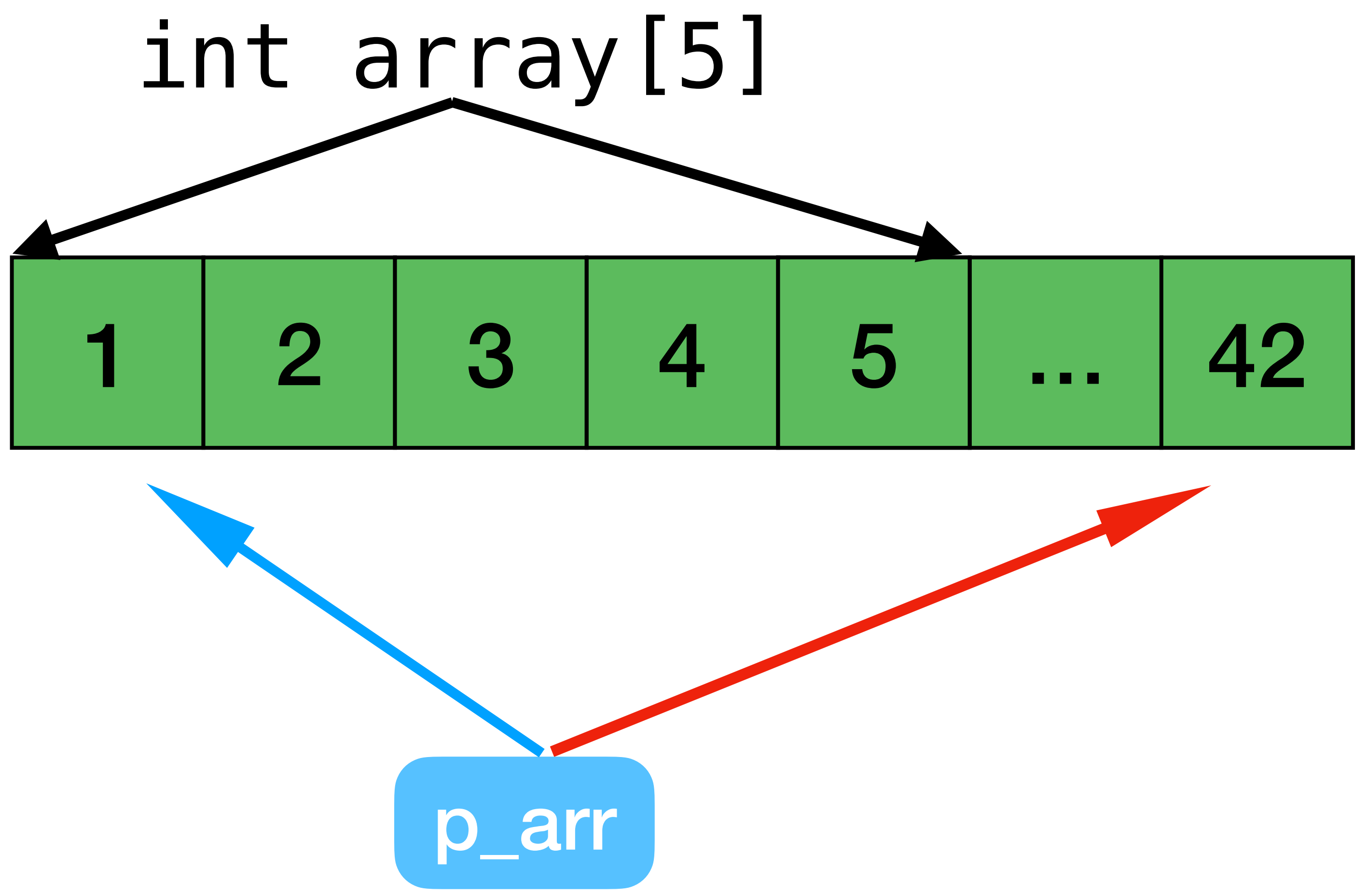
- For C, pointer is usually implemented as a virtual address.
- But this is not the only way to implement pointers.

C pointers can do almost arbitrary memory manipulation!

- The correctness is at the discretion of programmers.

Buffer Overflows

Reading/writing a buffer out of its bounds.



- It is C/C++ programmers' job to ensure such errors do not happen.
- In contract, most modern languages (e.g., Java, Rust, …) prevent buffer overflows by performing automatic bounds checking.
- The first Internet worm, Morris Worm, and many subsequent ones (CodeRed, Blaster, ...) exploited buffer overflows.
- Buffer overflows are still among the most commonly exploited vulnerabilities.

Better String Library Functions

- Instead of `strcpy()`, use `strncpy()`
- Instead of `strcat()`, use `strncat()`
- Instead of `sprintf()`, use `snprintf()`

Null-termination Errors

```c
int main(int argc, char* argv[]) {
    char a[16], b[16];
    strncpy(a, "0123456789abcdef", sizeof(a));
    printf("%s",a);
    strcpy(b, a);
}
```

What will be printed out?

- a[] not properly terminated.
    - Undefined behaviors, e.g., segmentation fault if printf is executed.

```
$ clang copy.c –o copy
jie@gwsyssec: /tmp
$ ./copy
0123456789abcdef??
                  ??jie@gwsyssec: /tmp
$ ./copy
0123456789abcdef(??jie@gwsyssec: /tmp
$ ./copy
0123456789abcdef?86?jie@gwsyssec: /tmp
```

Example Illustrating Stack Buffer Overflows

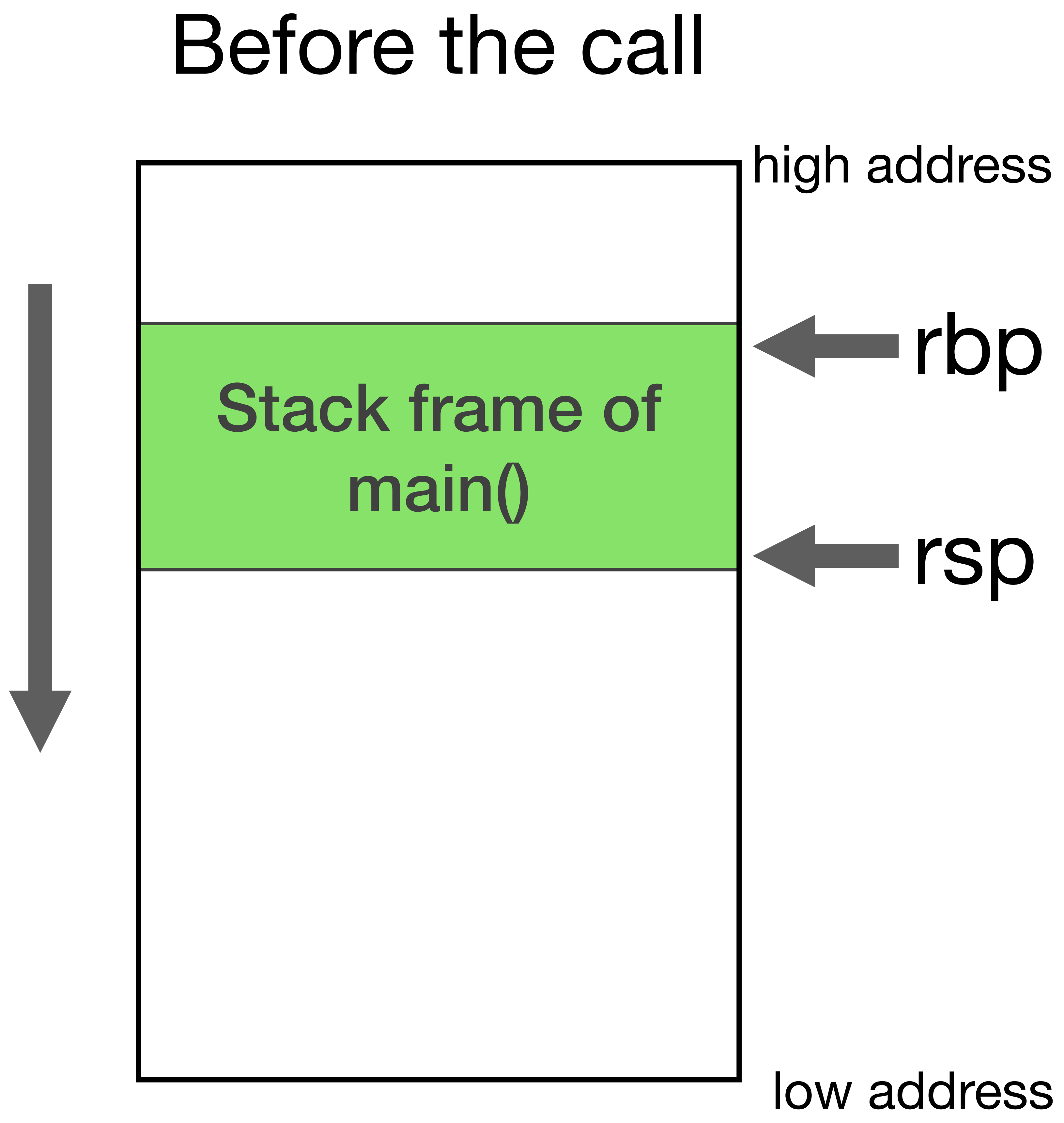
```c
void foo(int a, int b) {
     char buffer[12];
     gets(buffer);
     return;
}

int main() {
     int x;
     x = 0;
     foo(1,2);
     x = 1;
     printf("%d\n",x);
     return 0;
}
```

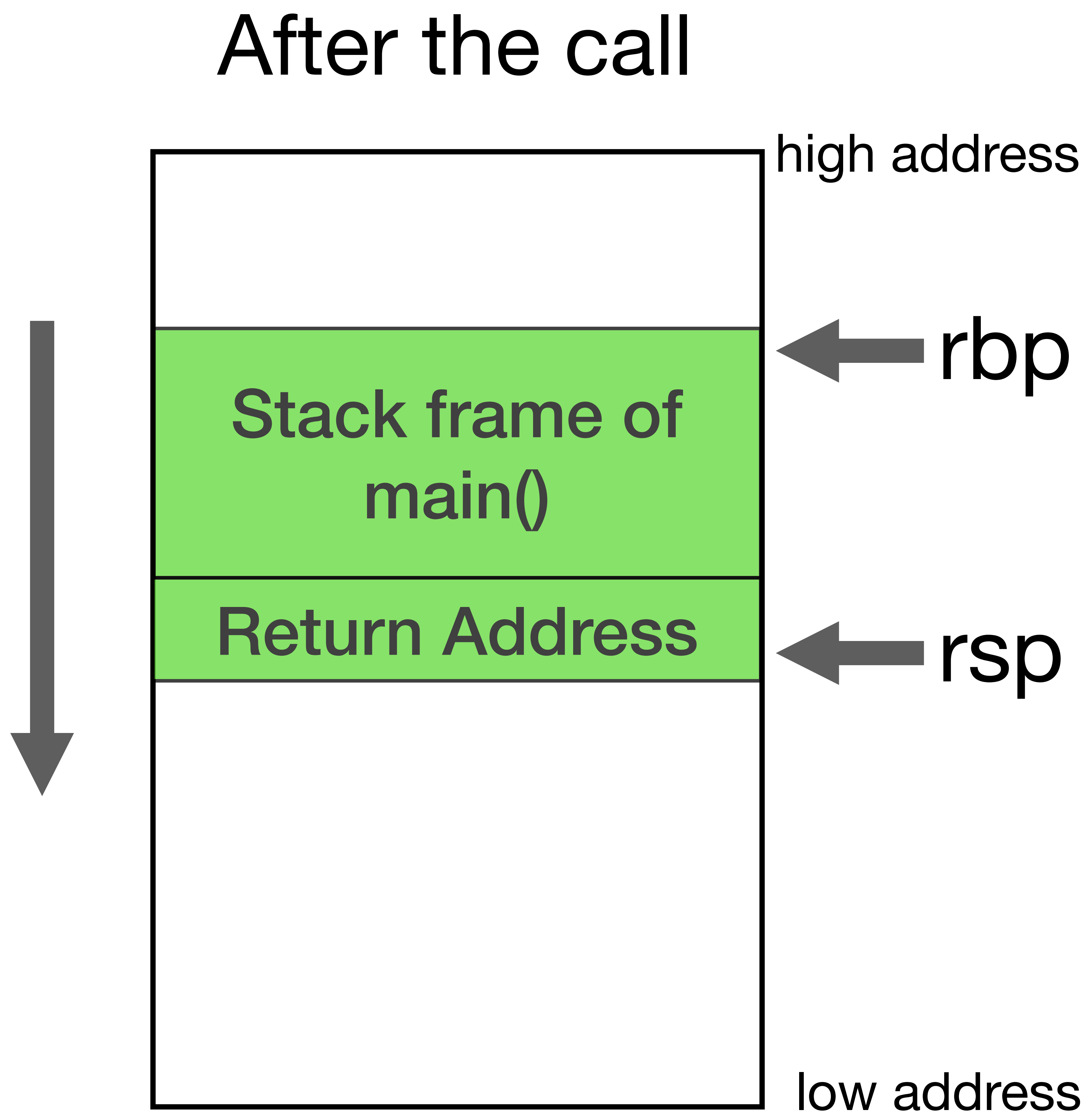
```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000001170 <+0>:     push   %rbp
   0x0000000000001171 <+1>:     mov    %rsp,%rbp
   0x0000000000001174 <+4>:     sub    $0x10,%rsp
   0x0000000000001178 <+8>:     movl   $0x0,-0x4(%rbp)
   0x000000000000117f <+15>:    movl   $0x0,-0x8(%rbp)
   0x0000000000001186 <+22>:    mov    $0x1,%edi
   0x000000000000118b <+27>:    mov    $0x2,%esi
   0x0000000000001190 <+32>:    call   0x1150 <foo>
   0x0000000000001195 <+37>:    movl   $0x1,-0x8(%rbp)
   0x000000000000119c <+44>:    mov    -0x8(%rbp),%esi
   0x000000000000119f <+47>:    lea    0xe5e(%rip),%rdi
   0x00000000000011a6 <+54>:    mov    $0x0,%al
   0x00000000000011a8 <+56>:    call   0x1030 <printf@plt>
   0x00000000000011ad <+61>:    xor    %eax,%eax
   0x00000000000011af <+63>:    add    $0x10,%rsp
   0x00000000000011b3 <+67>:    pop    %rbp
   0x00000000000011b4 <+68>:    ret
```

Compiled by clang-14 on Linux/AMD64

Function Calls: Stack

```
0x0000000000001186 <+22>:    mov     $0x1,%edi
0x000000000000118b <+27>:    mov     $0x2,%esi
0x0000000000001190 <+32>:    call    0x1150 <foo>
```



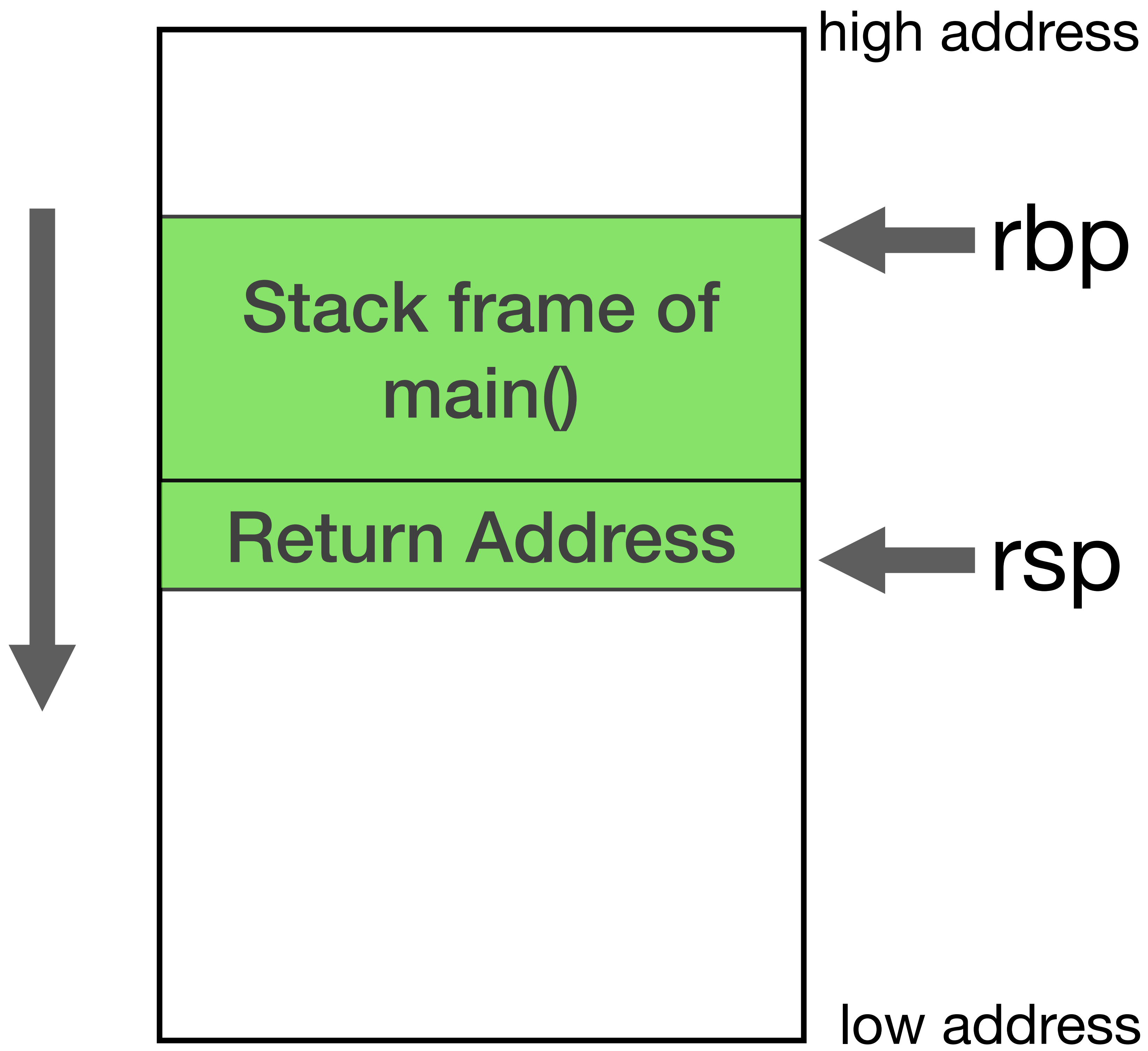


Function Initialization: Stack

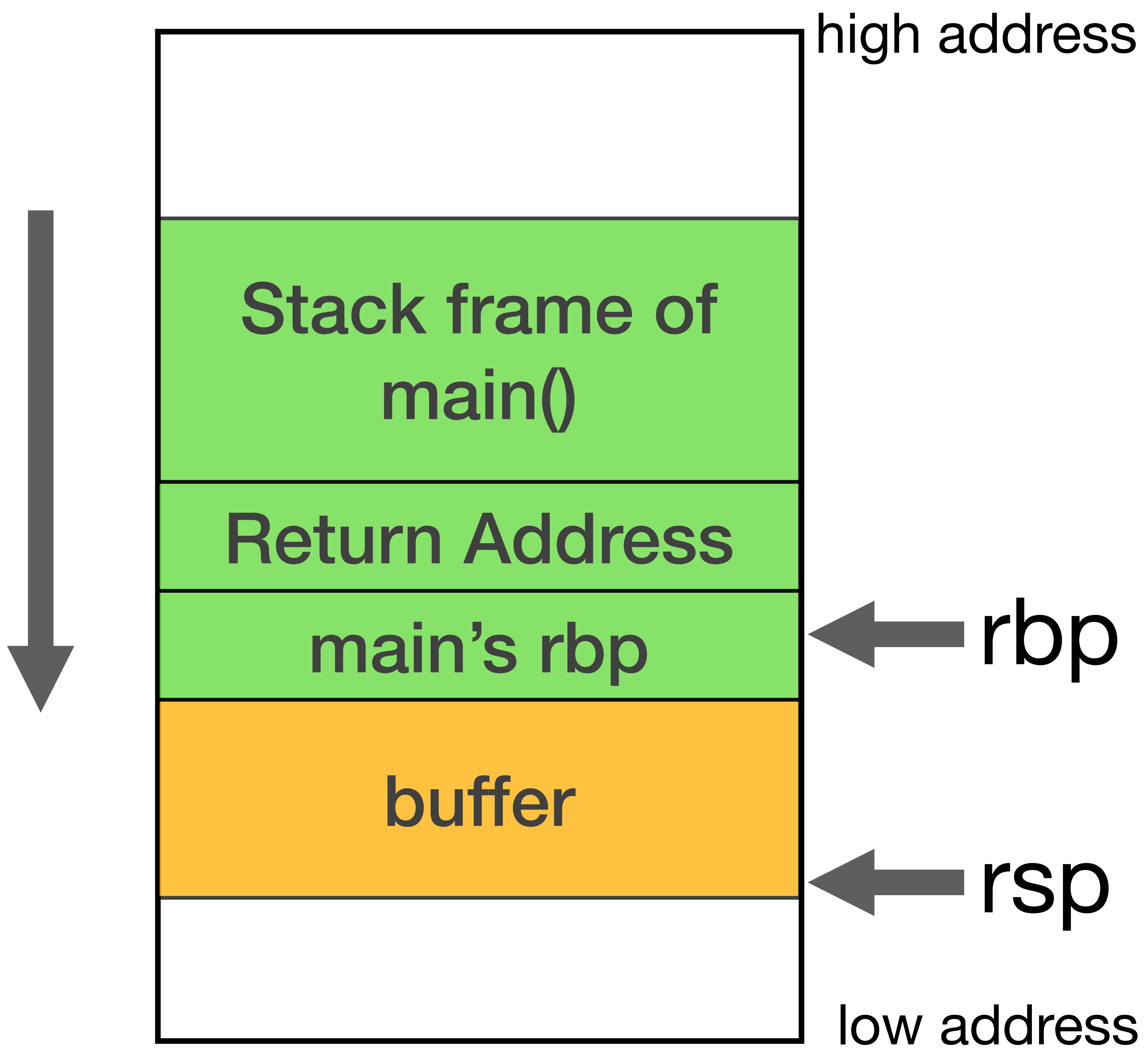
```
void foo(int a, int b) {
     char buffer[12];
     gets(buffer);
     return;
}
```

```
0x0000000000001150 <+0>:     push   %rbp
0x0000000000001151 <+1>:     mov    %rsp,%rbp
0x0000000000001154 <+4>:     sub    $0x20,%rsp
```

Before the call



After the call

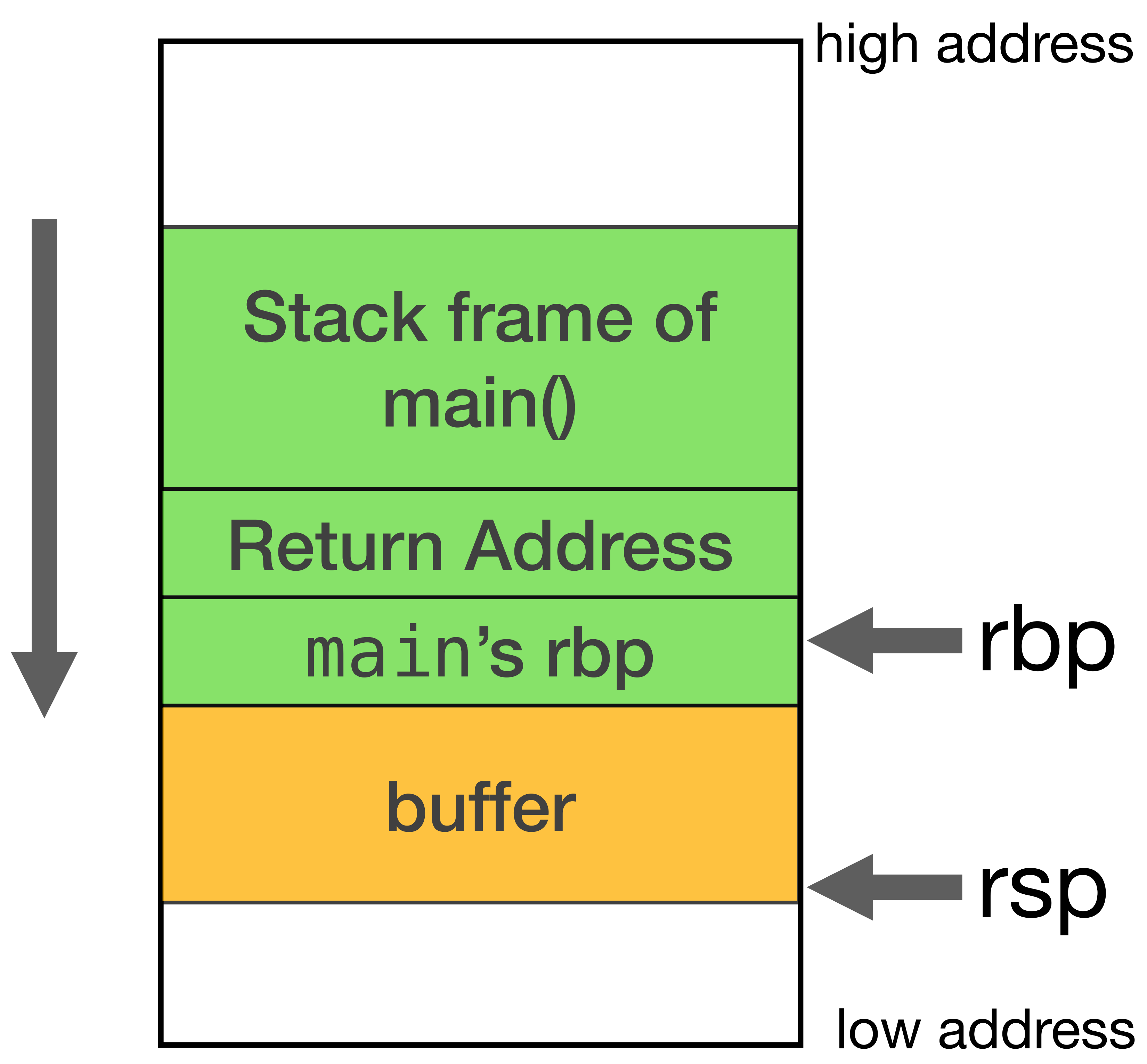


Function Return: Stack

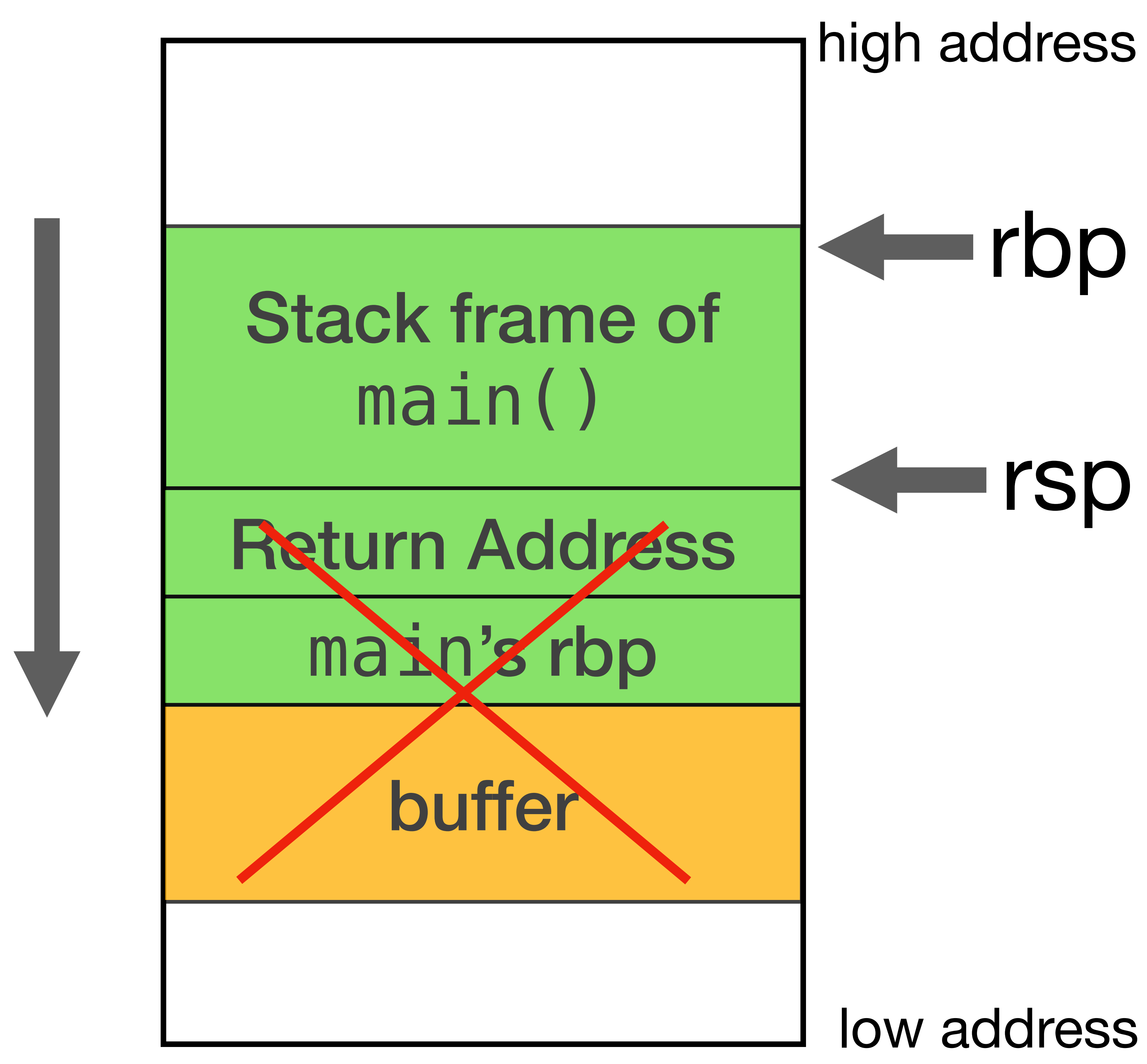
```c
void foo(int a, int b) {
     char buffer[12];
     gets(buffer);
     return;
}
```

```
0x0000000000001169 <+25>:    add    $0x20,%rsp
0x000000000000116d <+29>:    pop    %rbp
0x000000000000116e <+30>:    ret
```

Before the call



After the call

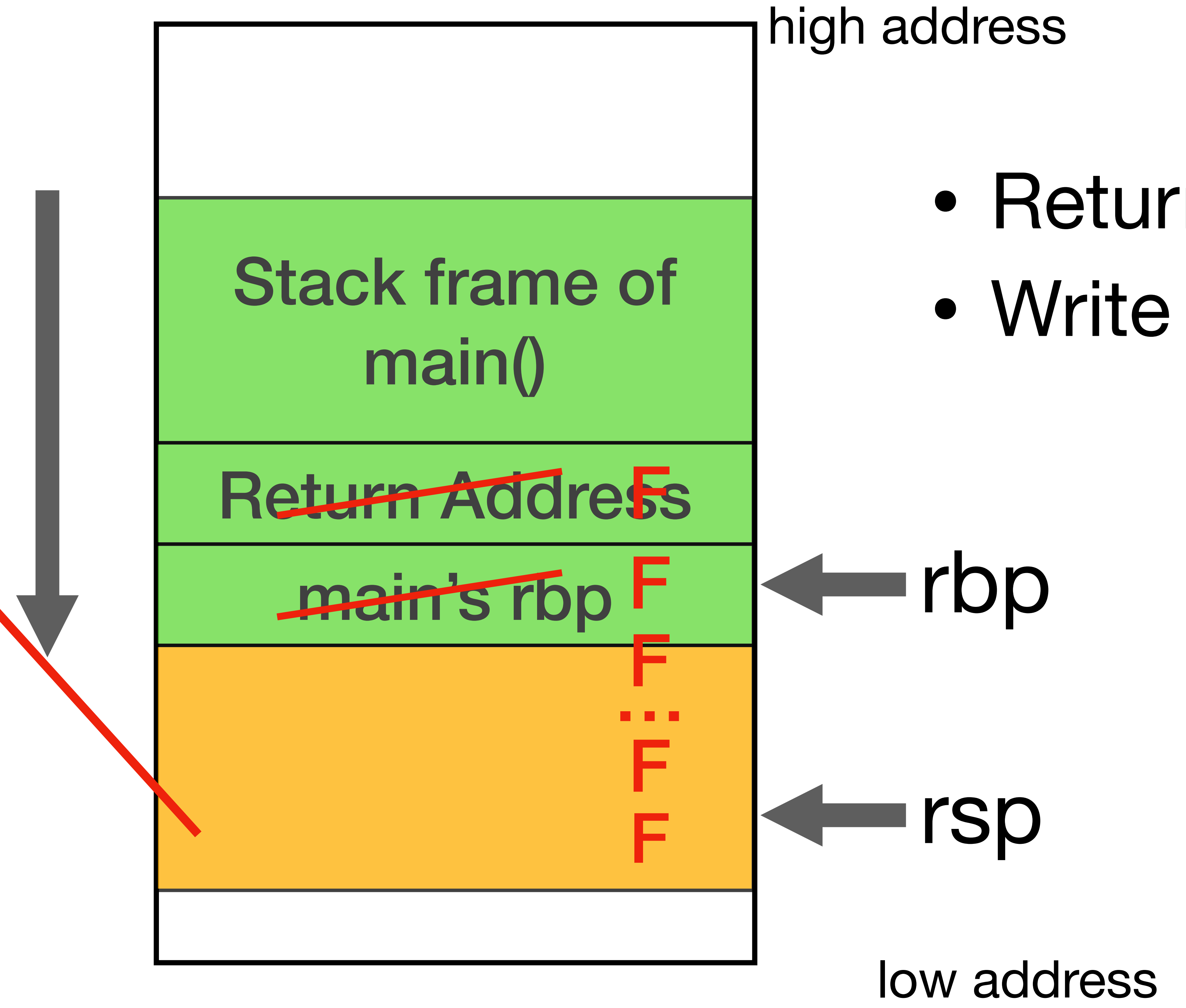


Smashing the Stack: What Happened?

```
void foo(int a, int b) {
      char buffer[12];
      gets(buffer);
      return;
}
```

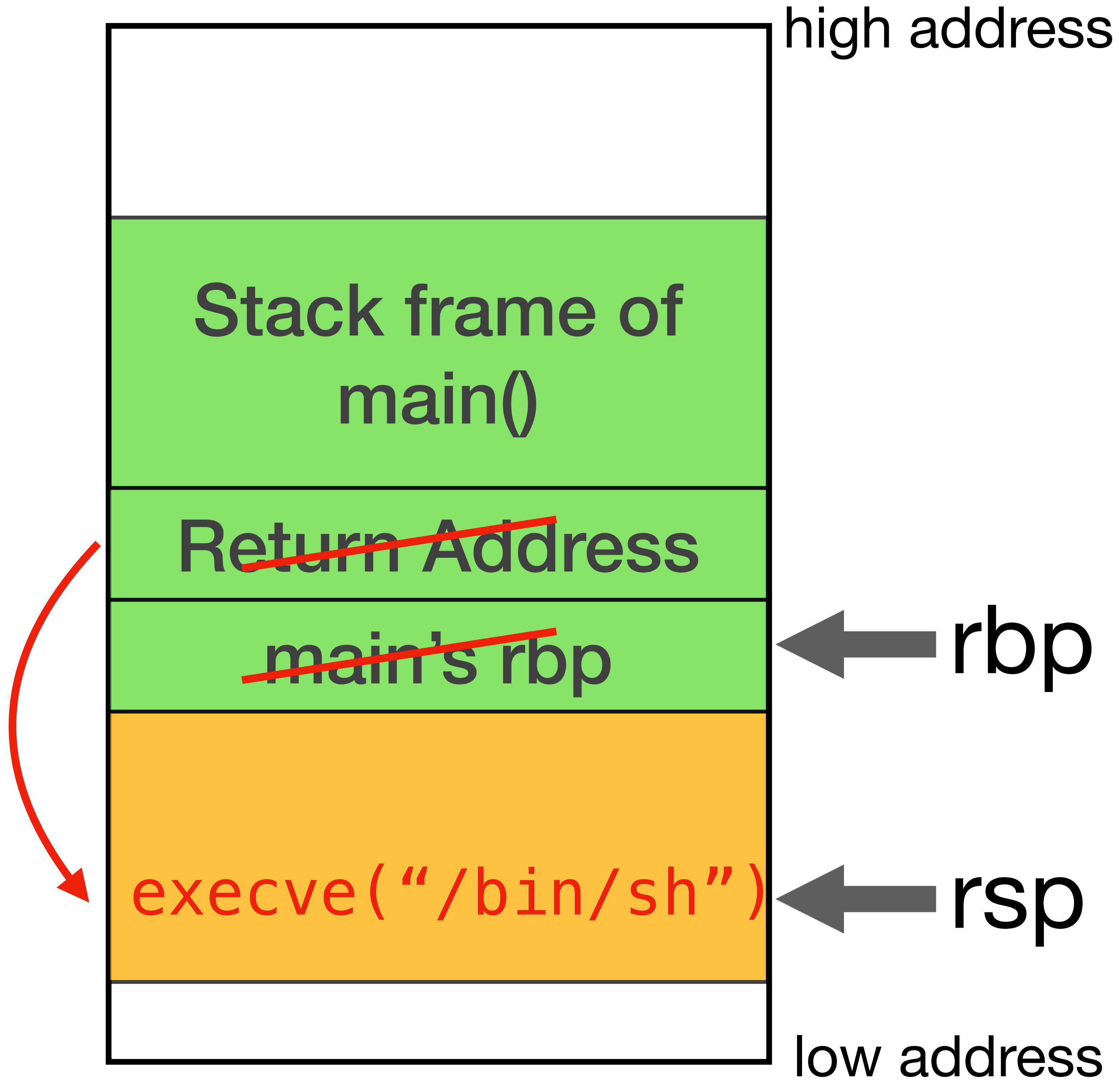
What happens if we input a large string?

./example ffffffffffffffffffffff…fffff



- Return to an invalid address
- Write to an unwritable address

Smashing the Stack: Injecting Shell Code

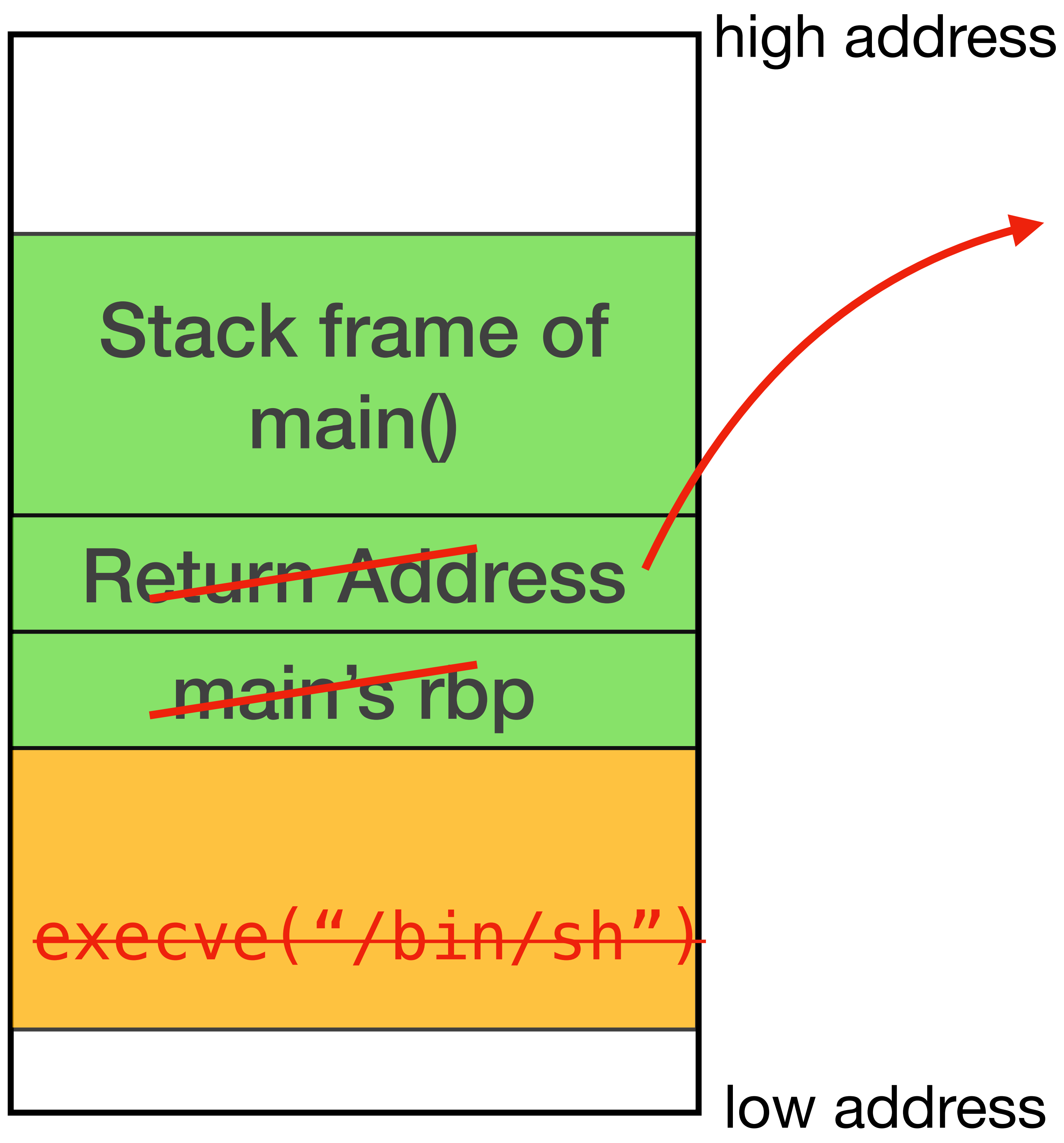


- This brings up a shell.
- Attackers can execute *any* command in the shell.
- The shell has the same privilege as the process.

- Good news:
  - C/C++ stack is not executable by default.

- Bad news:
  - Code injection works in other cases, e.g. JIT.

Exploiting Existing and Executable Code



"return" to `system()`

```
(gdb) disassemble system
Dump of assembler code for function system:
   0x00007ffff7c50d70 <+0>:     endbr64
   0x00007ffff7c50d74 <+4>:     test   %rdi,%rdi
   0x00007ffff7c50d77 <+7>:     je     0x7ffff7c50d80 <system+16>
   0x00007ffff7c50d79 <+9>:     jmp    0x7ffff7c50900
   0x00007ffff7c50d7e <+14>:    xchg   %ax,%ax
   0x00007ffff7c50d80 <+16>:    sub    $0x8,%rsp
   0x00007ffff7c50d84 <+20>:    lea    0x1878f5(%rip),%rdi
   0x00007ffff7c50d8b <+27>:    call   0x7ffff7c50900
   0x00007ffff7c50d90 <+32>:    test   %eax,%eax
   0x00007ffff7c50d92 <+34>:    sete   %al
   0x00007ffff7c50d95 <+37>:    add    $0x8,%rsp
   0x00007ffff7c50d99 <+41>:    movzbl %al,%eax
   0x00007ffff7c50d9c <+44>:    ret
```

Return-to-libc(ret2libc) Attack: Exploiting `system()`

- `system()` libc function

NAME top

system - execute a shell command

LIBRARY top

Standard C library (*libc*, *-lc*)

SYNOPSIS top

```
#include <stdlib.h>

int system(const char *command);
```

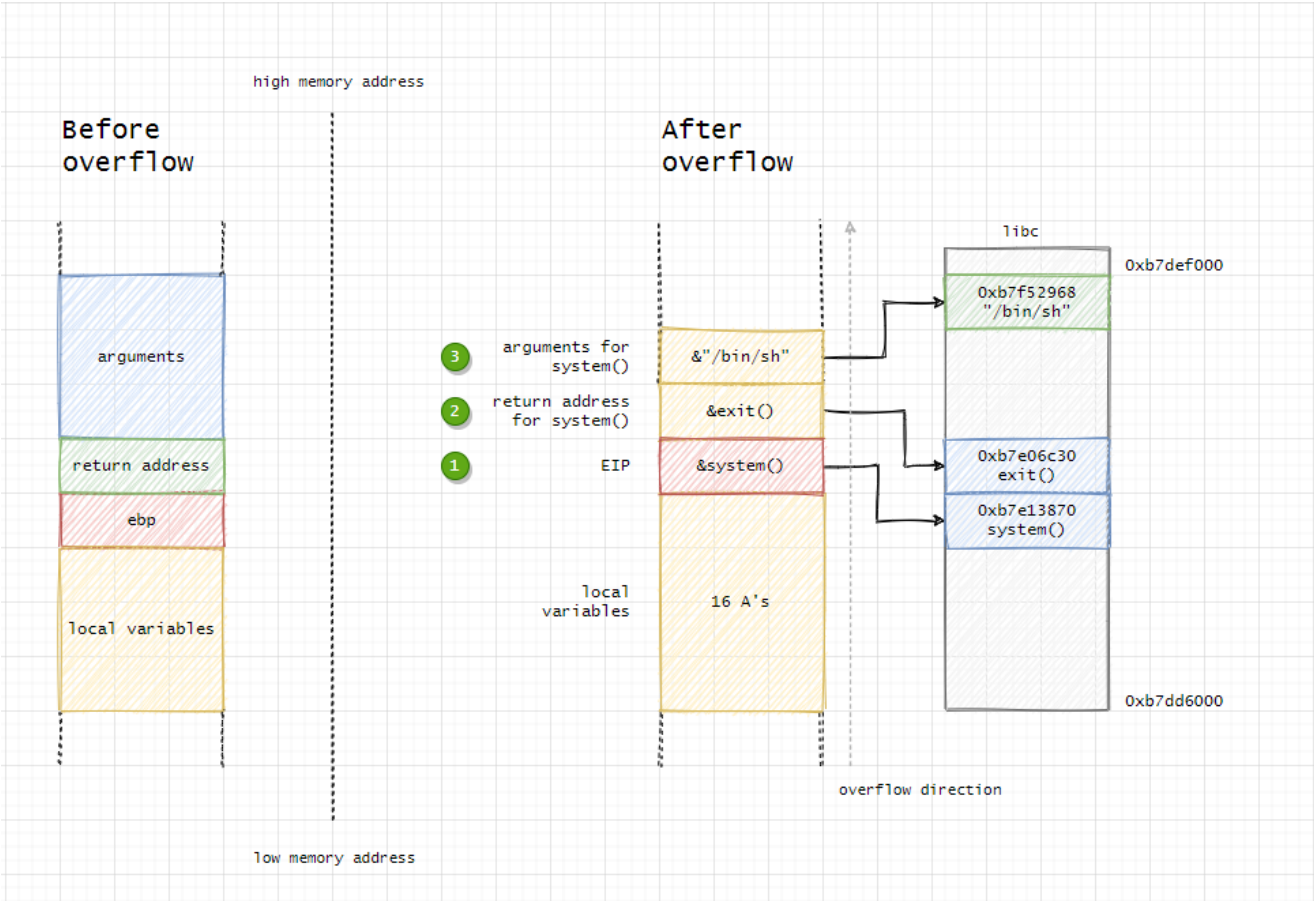
DESCRIPTION top

The **system**() library function behaves as if it used fork(2) to create a child process that executed the shell command specified in *command* using execl(3) as follows:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

system() returns after the command has been completed.

Exploiting ret2libc on x86-32



Stack memory layout of a 32-bit vulnerable program

https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/return-to-libc-ret2libc

System V AMD64 Calling Convention

How functions/subroutines pass arguments and return values at the macro-architecture level.

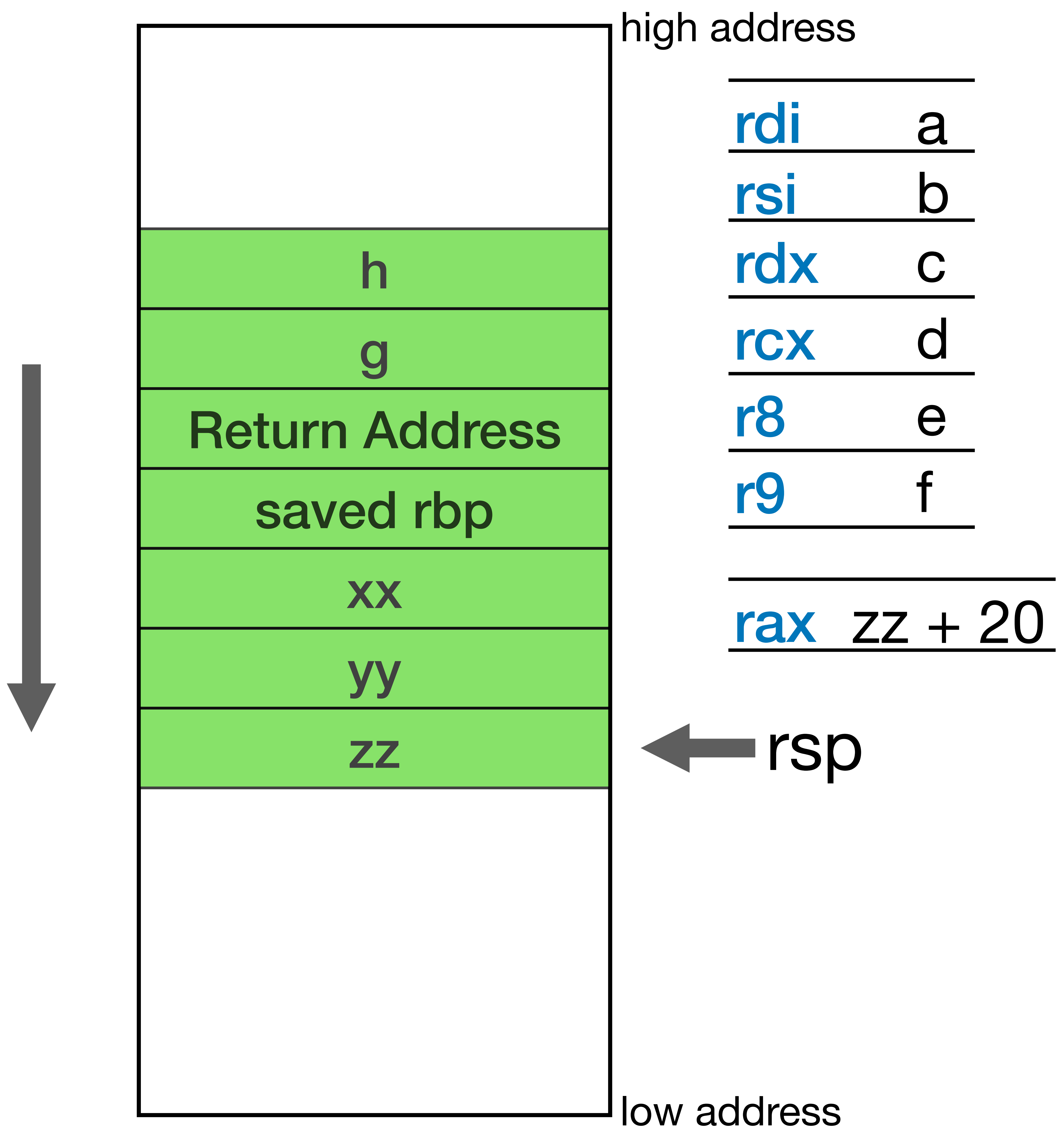
```
void foo() {
    ...
    bar(a, b, c, d, e, f, g, h);
    ...
}

long bar(long a, long b, long c, long d,
         long e, long f, long g, long h) {
   long xx = a * b * c * d * e * f * g * h;
   long yy = a + b + c + d + e + f + g + h;
   long zz = utilfunc(xx, yy, xx % yy);
   return zz + 20;
}
```

- Where to put all the arguments?
- Where to put the return value?

- Arguments are passed
  - in registers: rdi, rsi, rdx, rcx, r8, r9
  - then via stack
- Return value is passed via
  - in registers: rax, rdx
  - then via stack

x86-64/AMD64 Calling Convention

```
void foo() {
    ...
    bar(a, b, c, d, e, f, g, h);
    ...
}

long bar(long a, long b, long c, long d,
         long e, long f, long g, long h) {
   long xx = a * b * c * d * e * f * g * h;
   long yy = a + b + c + d + e + f + g + h;
   long zz = utilfunc(xx, yy, xx % yy);
   return zz + 20;
}
```



How to put malicious data in target registers?

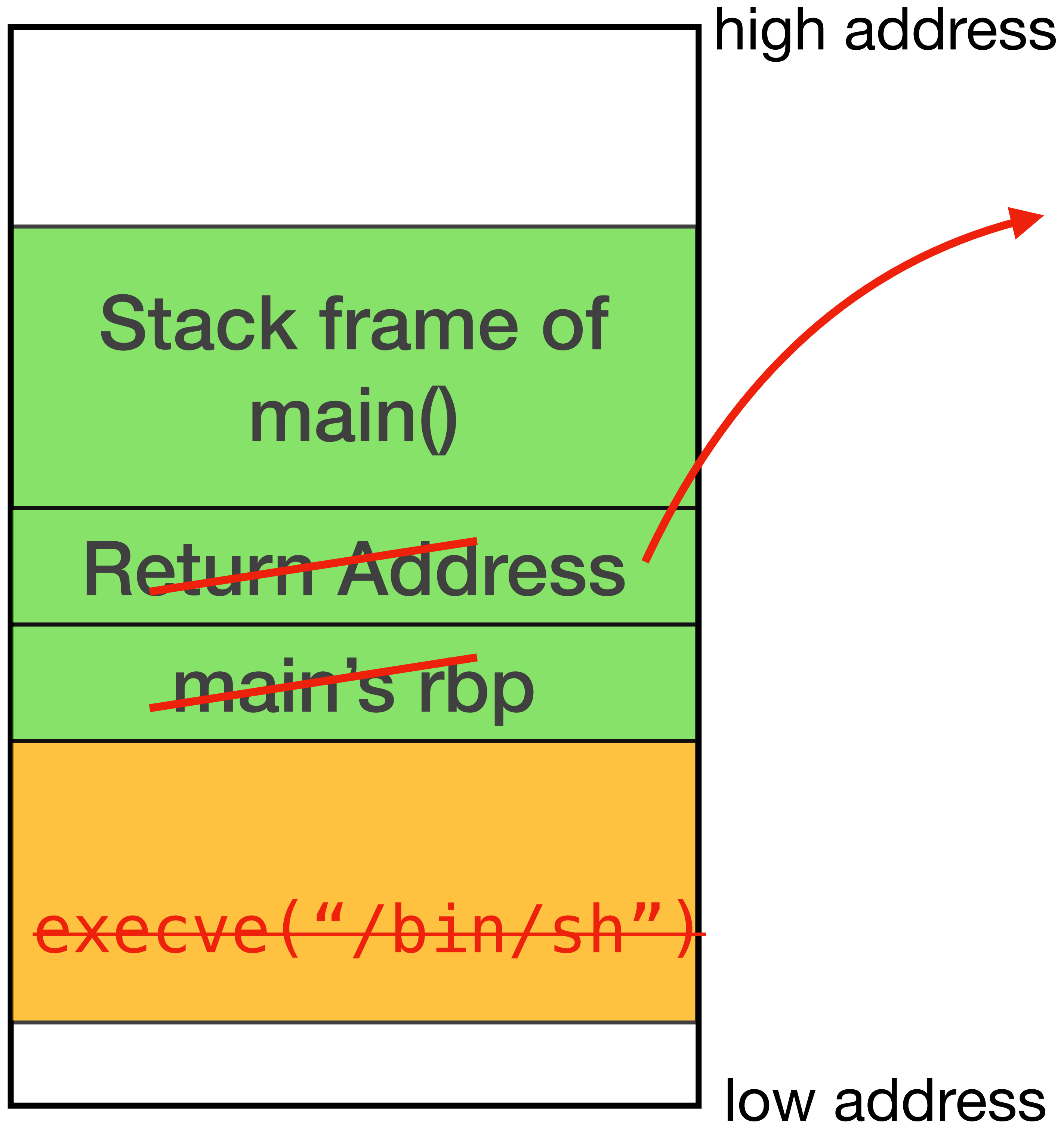
Limitations of `ret2libc` Attacks

- On AMD64 (and many other arch, e.g., AArch64), function arguments are first passed via registers instead of stack.
- Limited exploitable functions
  - `system()` and other "profitable" library functions could be removed.
- Can only execute straight-line code
  - Desired malicious computation may be invalidated by functions themselves.

Why do we need functions?

Functions facilitate software development,
but are not necessary for computations.

Exploiting Existing and Executable Code



"return" to `system()`

```
(gdb) disassemble system
Dump of assembler code for function system:
   0x00007ffff7c50d70 <+0>:     endbr64
   0x00007ffff7c50d74 <+4>:     test   %rdi,%rdi
   0x00007ffff7c50d77 <+7>:     je     0x7ffff7c50d80 <system+16>
   0x00007ffff7c50d79 <+9>:     jmp    0x7ffff7c50900
   0x00007ffff7c50d7e <+14>:    xchg   %ax,%ax
   0x00007ffff7c50d80 <+16>:    sub    $0x8,%rsp
   0x00007ffff7c50d84 <+20>:    lea    0x1878f5(%rip),%rdi
   0x00007ffff7c50d8b <+27>:    call   0x7ffff7c50900
   0x00007ffff7c50d90 <+32>:    test   %eax,%eax
   0x00007ffff7c50d92 <+34>:    sete   %al
   0x00007ffff7c50d95 <+37>:    add    $0x8,%rsp
   0x00007ffff7c50d99 <+41>:    movzbl %al,%eax
   0x00007ffff7c50d9c <+44>:    ret
```

How about setting the argument and executing the same instructions from other places?

Return-oriented Programming (ROP)

An exploit technique that allows arbitrary code execution without calling any functions.

- Exploiting memory corruption bugs
  - Often starting with a corrupted return address
- Chaining code sequences, called *gadgets*, that end with a `ret`
  - Generally, gadgets ending with control flow transfer instructions, e.g. `jmp`
- Turing-complete
  - Memory operations
  - Arithmetic and logic
  - Control flow

Return-oriented Programming (ROP)

- Use `ret` to jump to the "profitable" instructions to the attacker's interest

```
ret
```

⟺

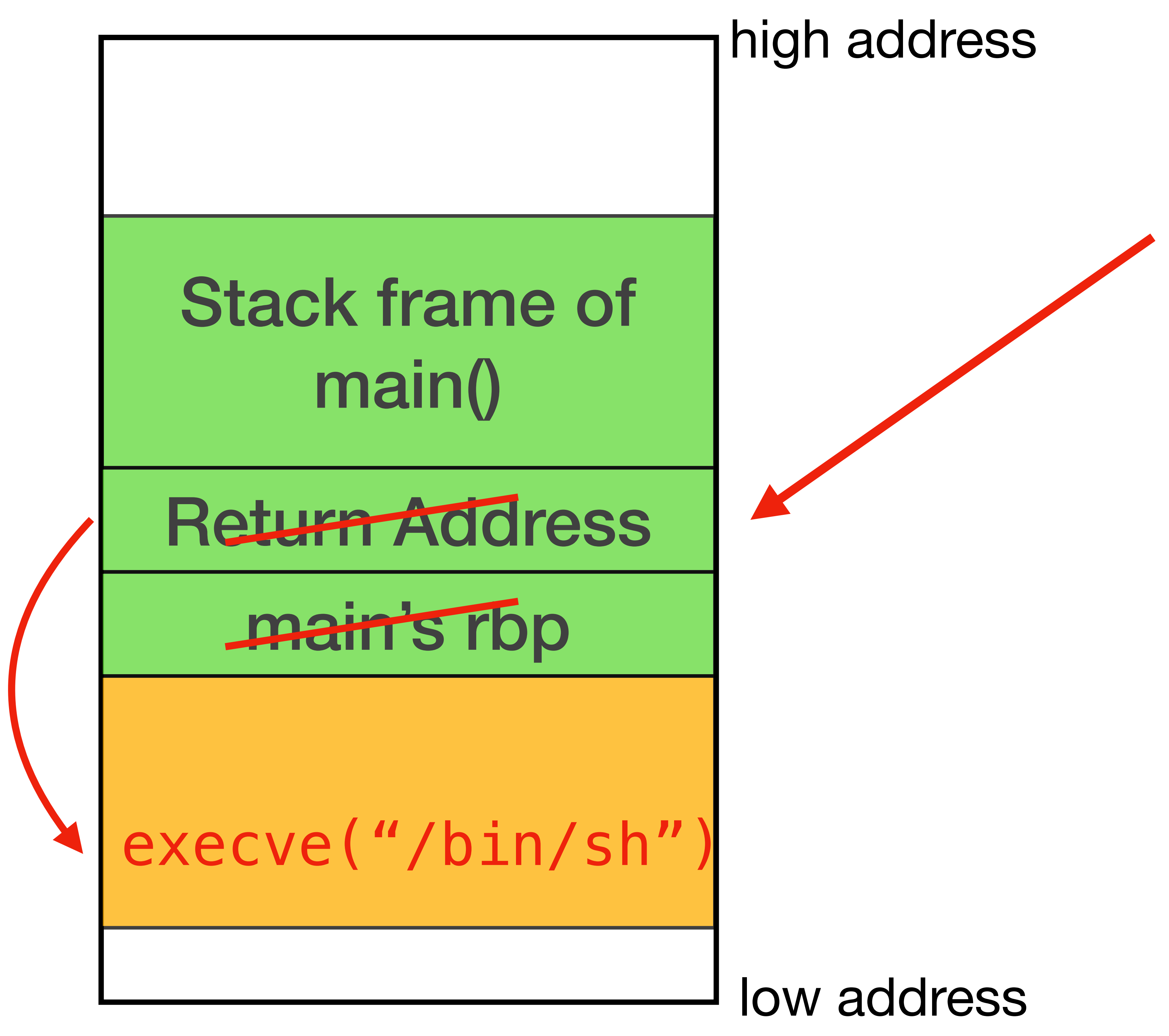
```
mov (%rsp), %rip
add $8, %rsp
```

Note this instruction is for demonstration. `rip` cannot be explicitly updated.

Controlling what `rsp` enables controlling the next to-be-executed instruction.

- Use `rsp` as a confused deputy for `rip`
  - Attackers use `rsp` to control the flow of the victim program.

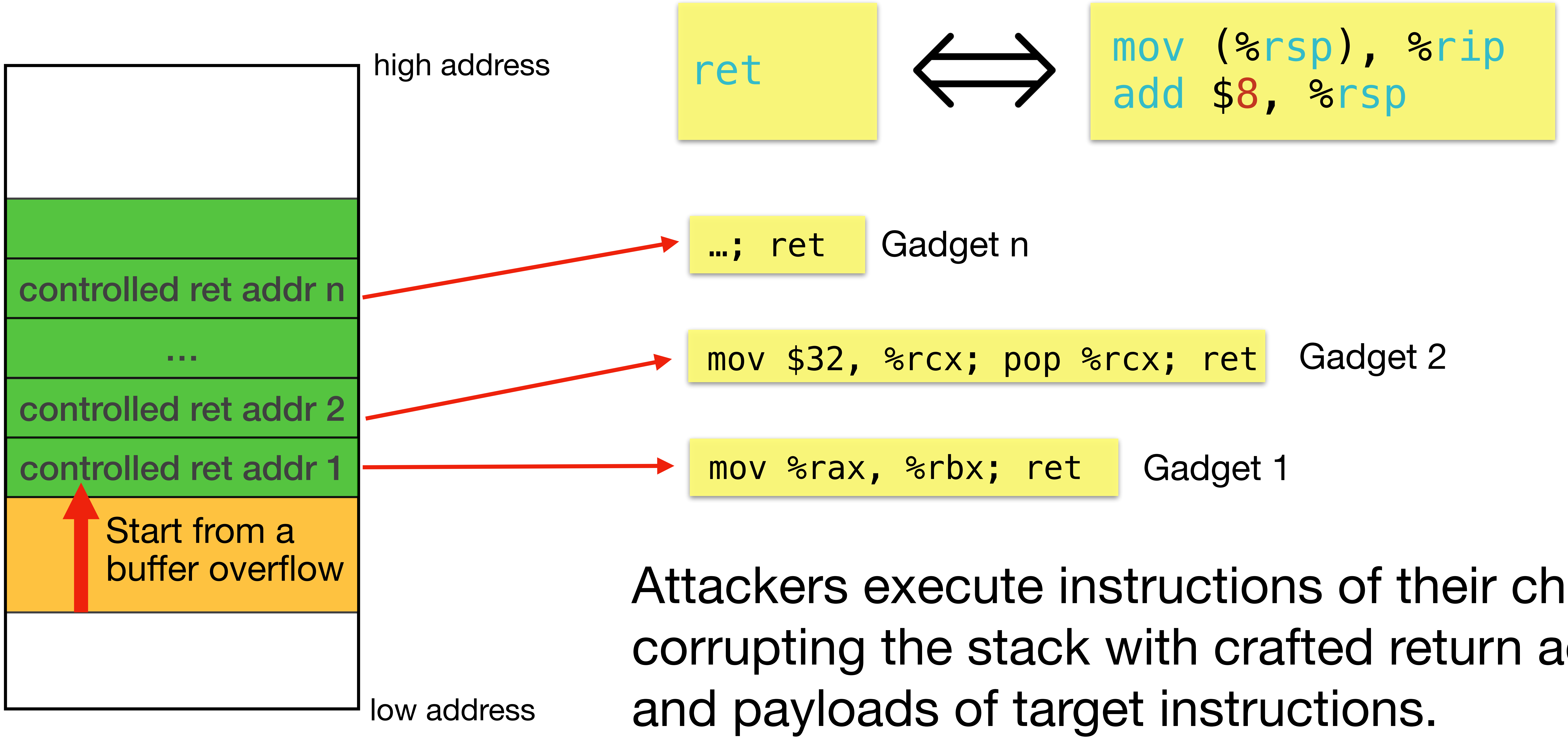
Exploiting Multiple `ret`



`ret2libc` exploits one `ret` instruction.

How about exploiting multiple ret?

Chaining Multiple ret

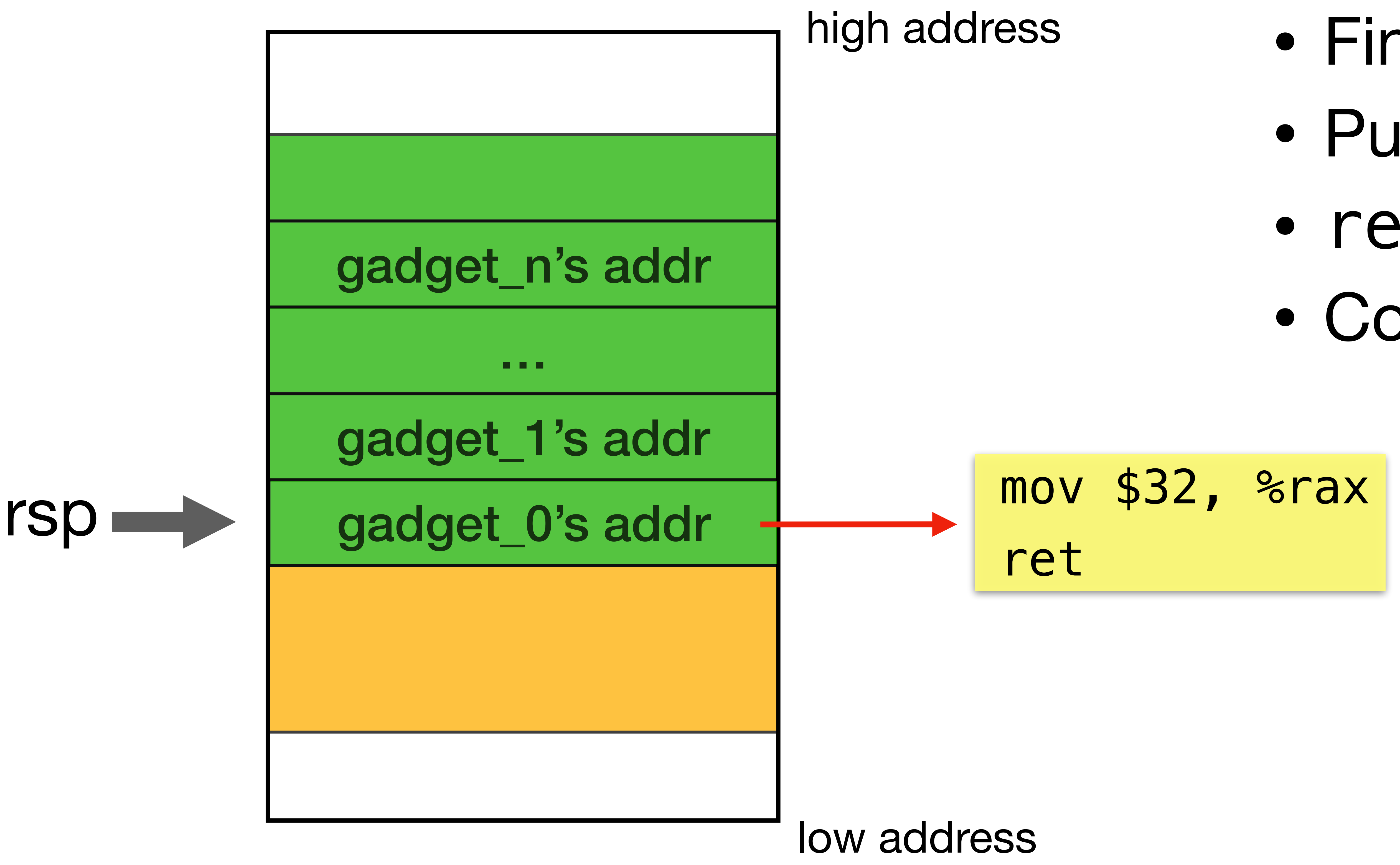


Attackers execute instructions of their choosing by corrupting the stack with crafted return addresses and payloads of target instructions.

Loading a Constant

How to load a constant (e.g. `0x32`) into a register?

- Option 1: Find a gadget like `mov $32, %rax; ret`

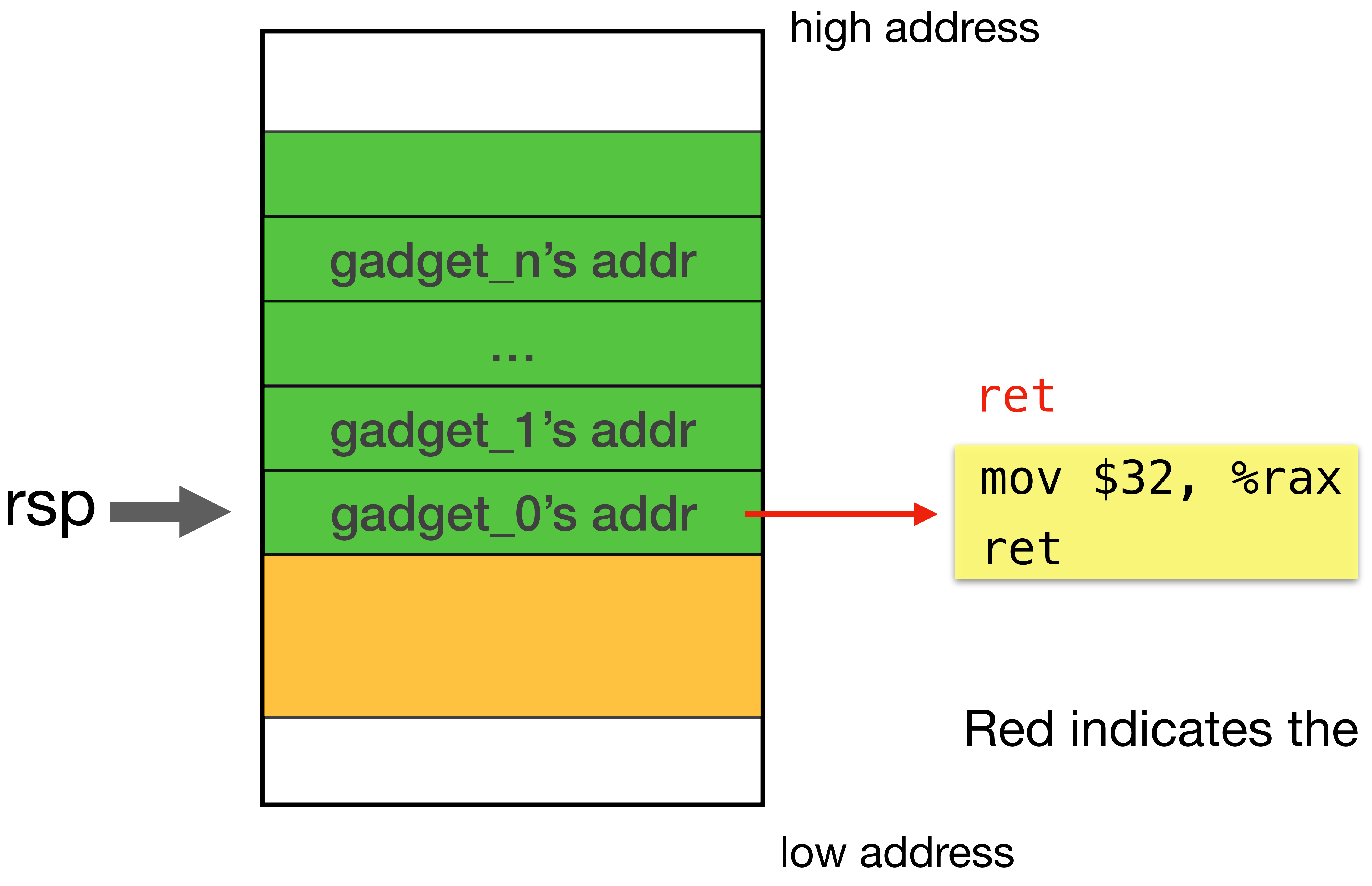


- Find this gadget in code
- Put the gadget's address on stack
- `ret` to the gadget
- Continue with the next gadget

Loading a Constant

How to load a constant (e.g. `0x32`) into a register?

- Option 1: Find a gadget like `mov $32, %rax; ret`



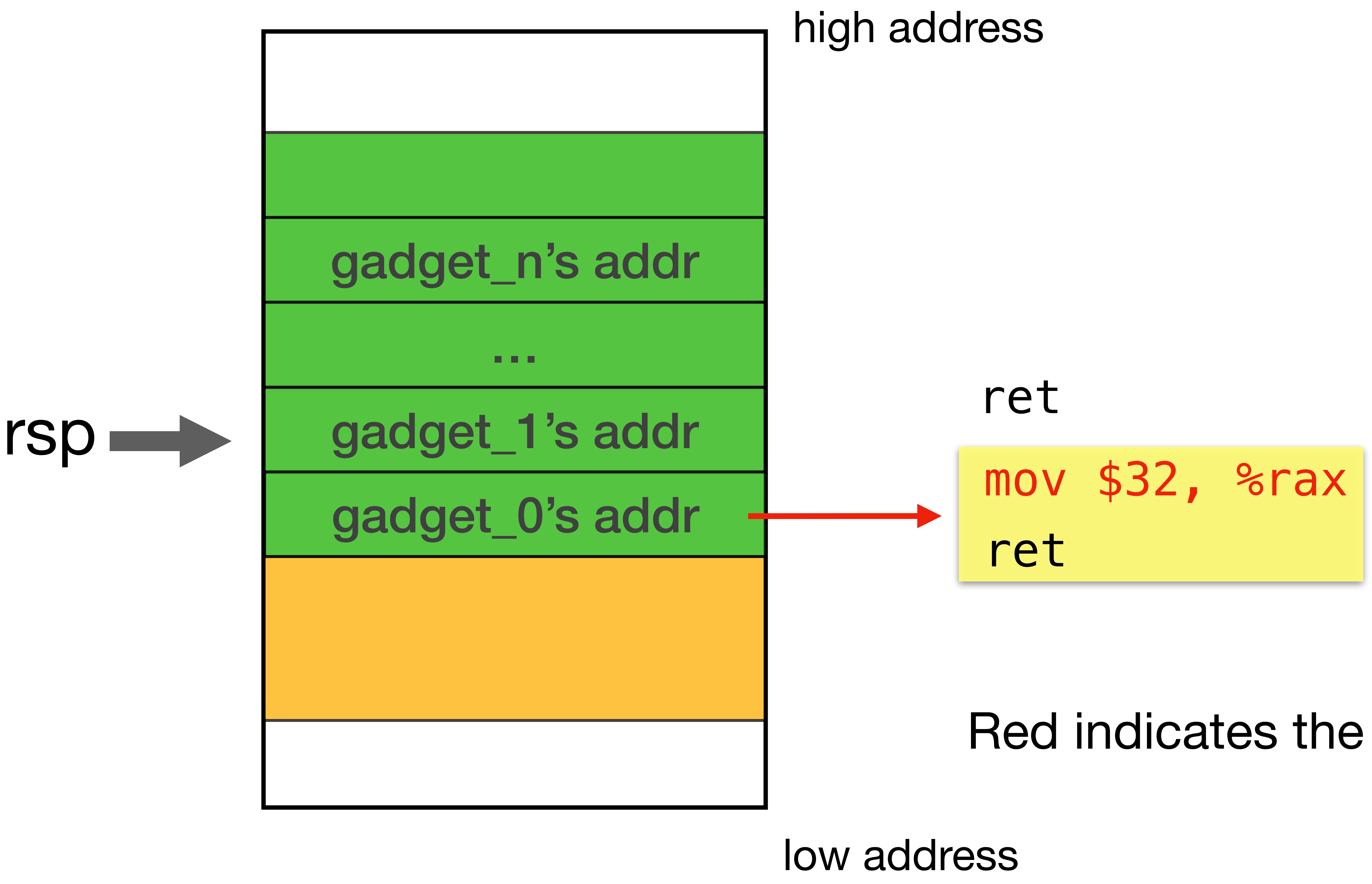
rax	123456
rsp	0x40000
rip	`ret`'s address

Red indicates the next instruction to execute.

Loading a Constant

How to load a constant (e.g. `0x32`) into a register?

- Option 1: Find a gadget like `mov $32, %rax; ret`



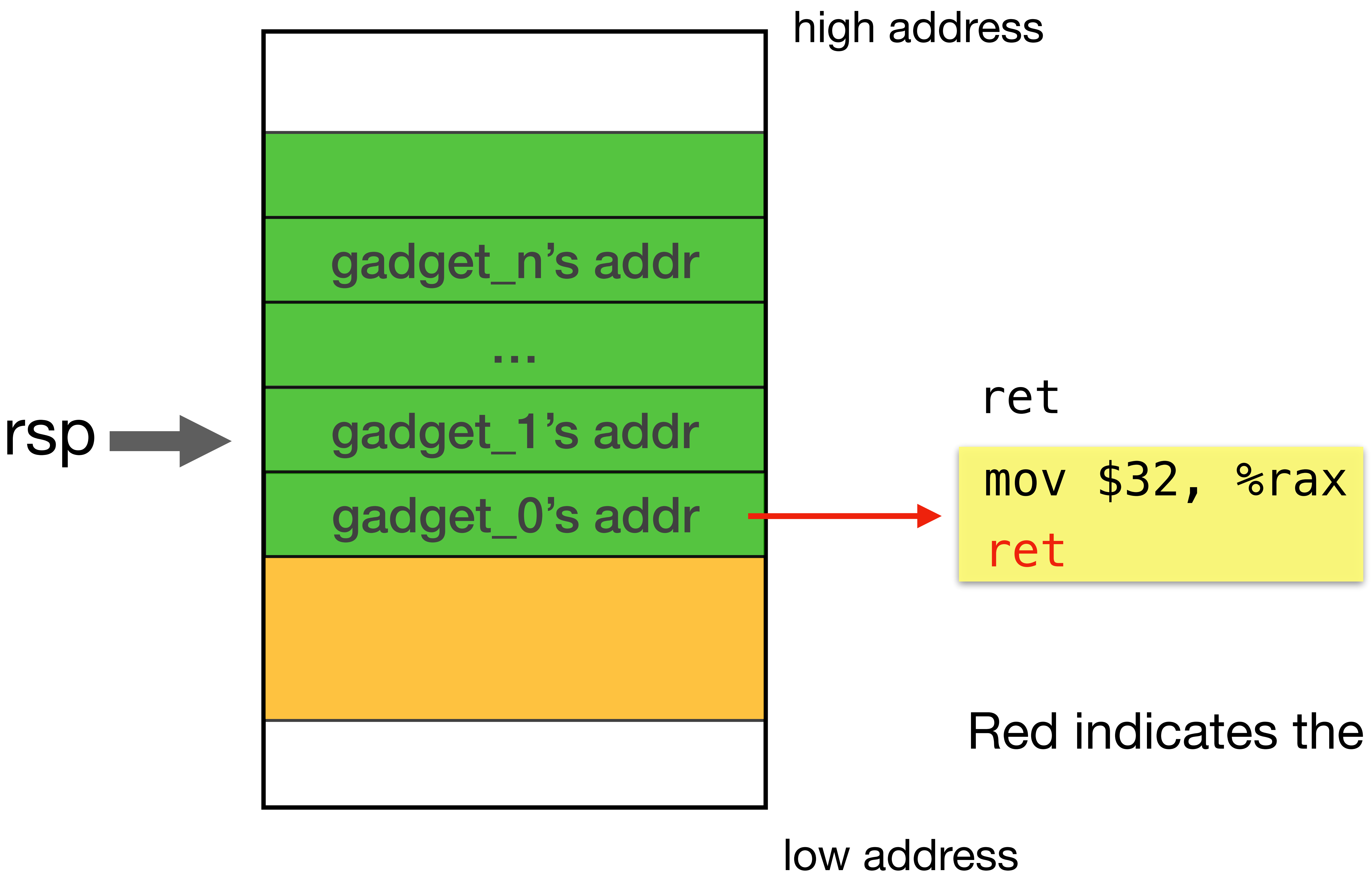
rax	123456
rsp	0x40008
rip	gadget_0's address

Red indicates the next instruction to execute.

Loading a Constant

How to load a constant (e.g. `0x32`) into a register?

- Option 1: Find a gadget like `mov $32, %rax; ret`



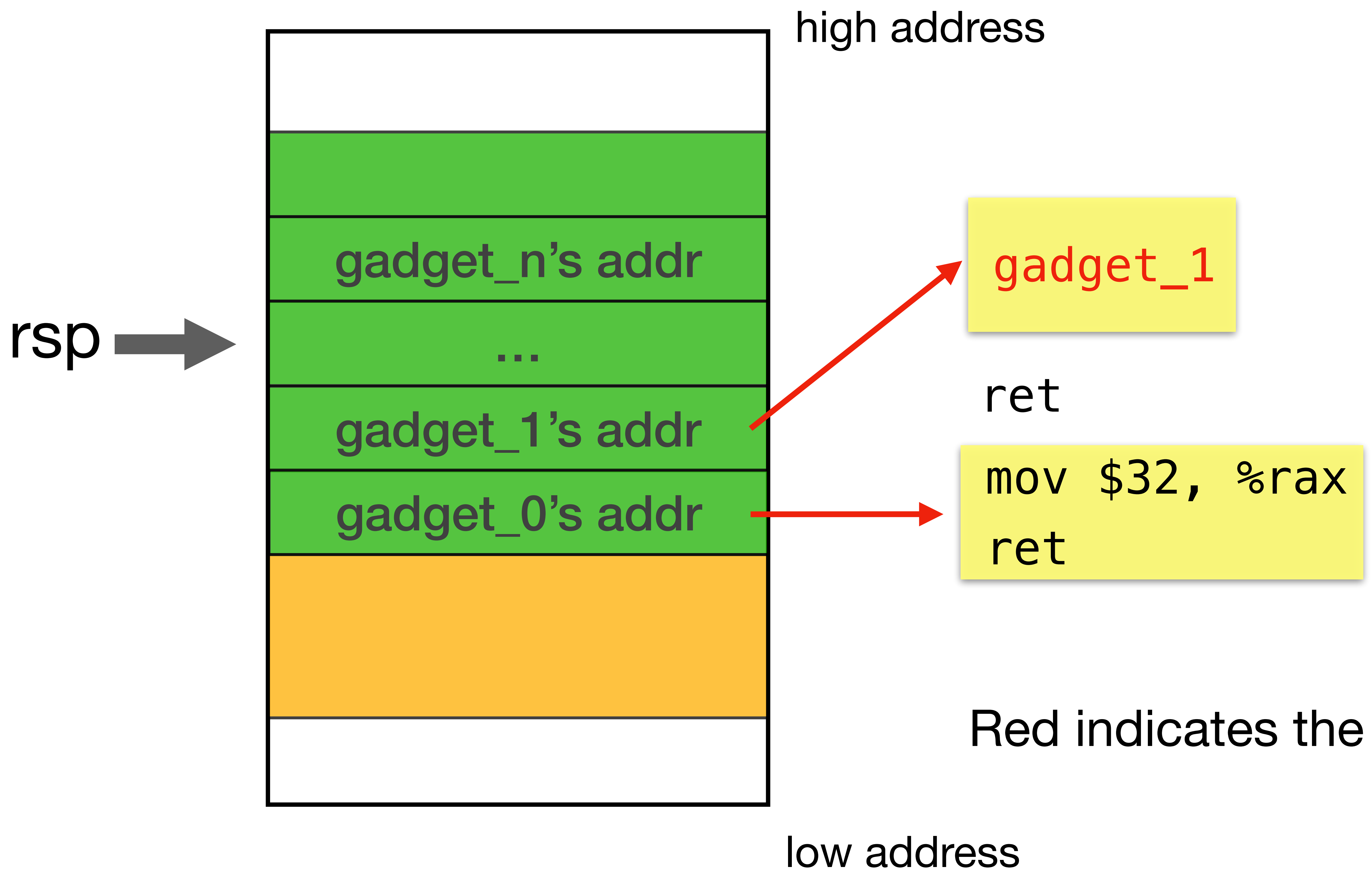
rax	32
rsp	0x40008
rip	gadget_0's ret addr

Red indicates the next instruction to execute.

Loading a Constant

How to load a constant (e.g. `0x32`) into a register?

- Option 1: Find a gadget like `mov $32, %rax; ret`



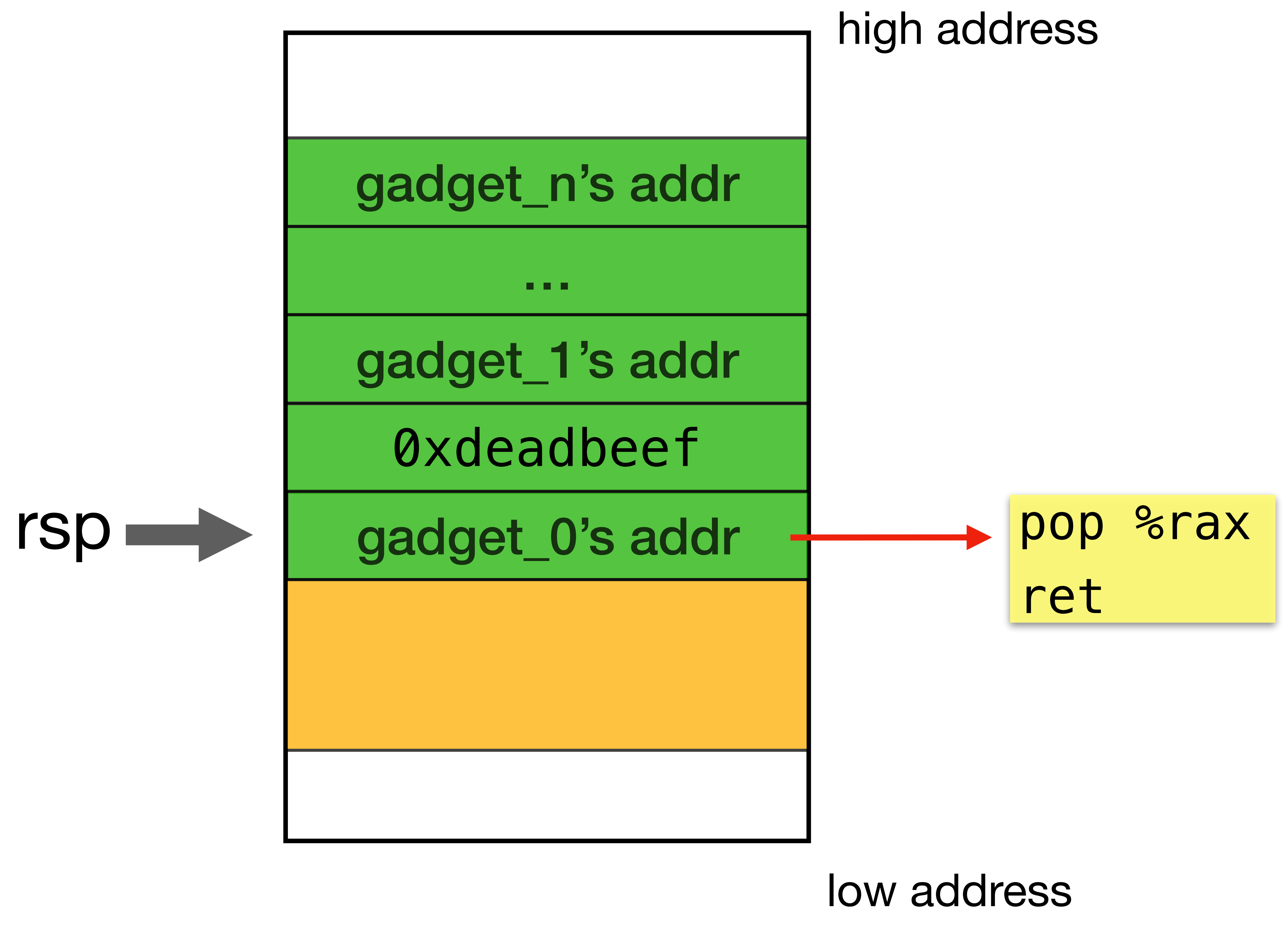
rax	32
rsp	0x40010
rip	gadget_1's address

Red indicates the next instruction to execute.

Loading a Constant

How to load an arbitrary constant (e.g. `0xdeadbeef`) into a register?

- Option 1: Pop the constant to the target register

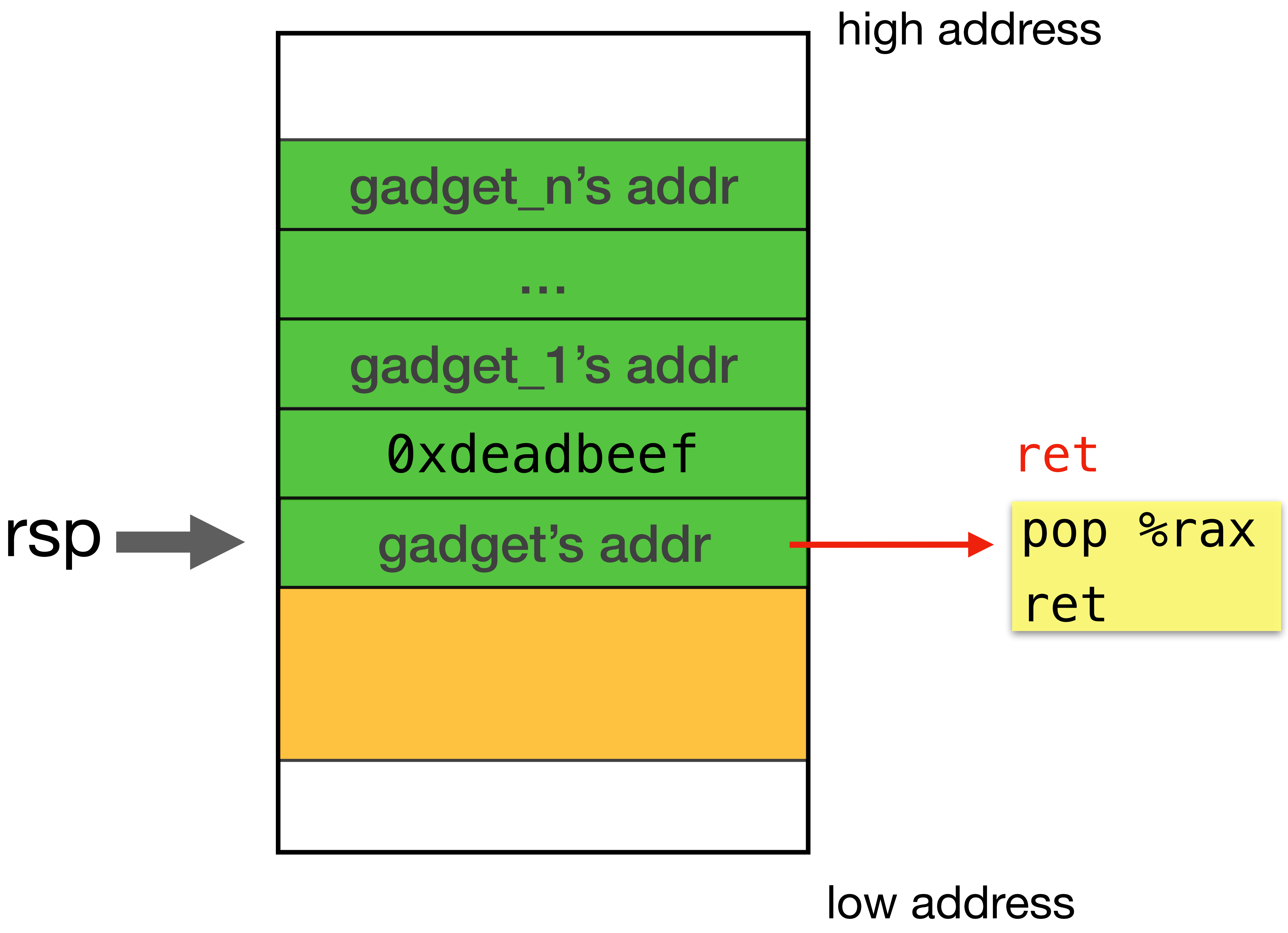


- Put gadget's address on stack
- Put target constant on stack (above `rsp`)
- `ret` makes `rsp` point to the constant
- `pop` loads the constant into the register

Loading a Constant

How to load an arbitrary constant (e.g. `0xdeadbeef`) into a register?

- Option 1: Pop the constant to the target register

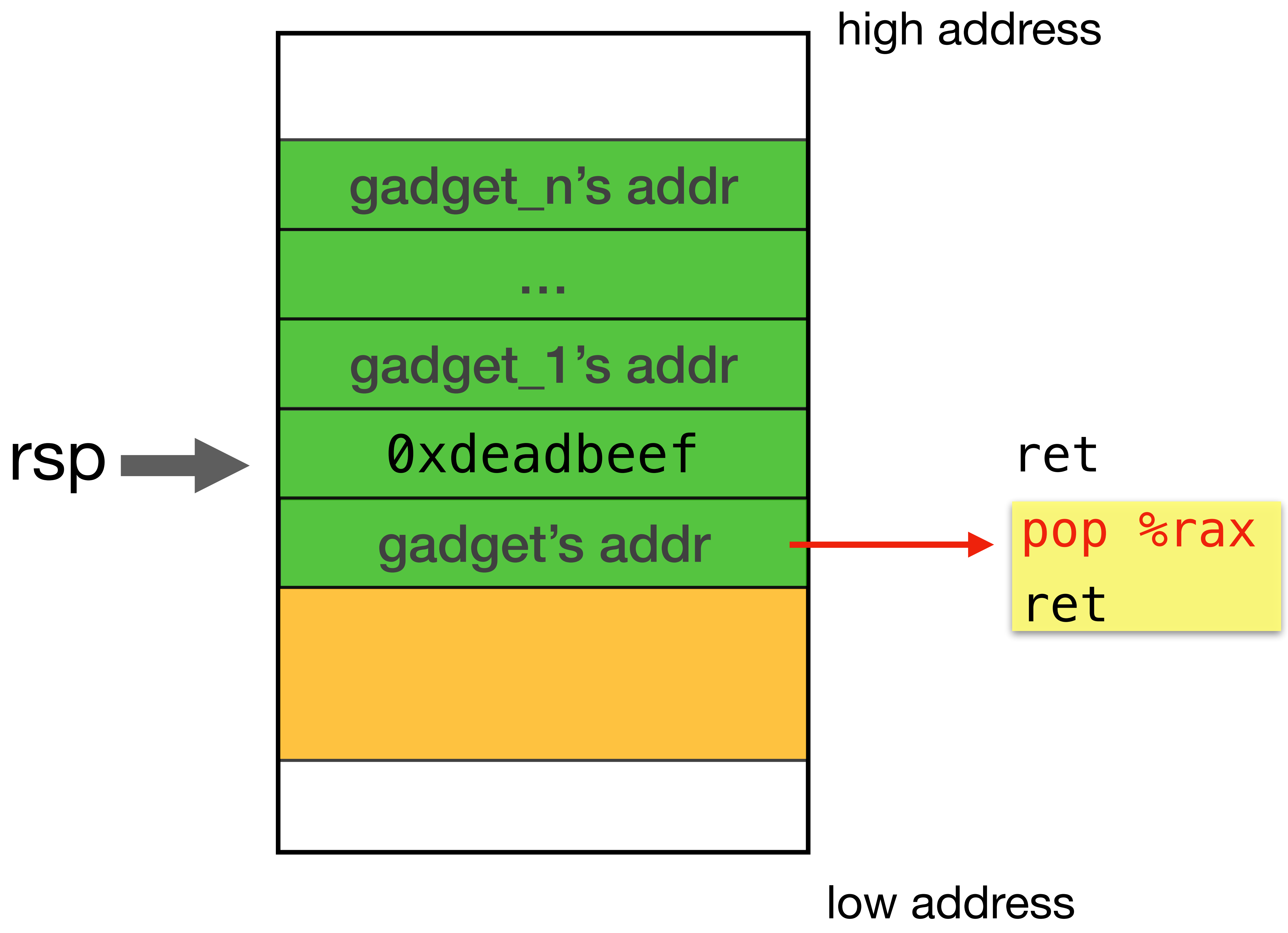


rax	123456
rsp	0x40000
rip	`ret`'s address

Loading a Constant

How to load an arbitrary constant (e.g. `0xdeadbeef`) into a register?

- Option 1: Pop the constant to the target register

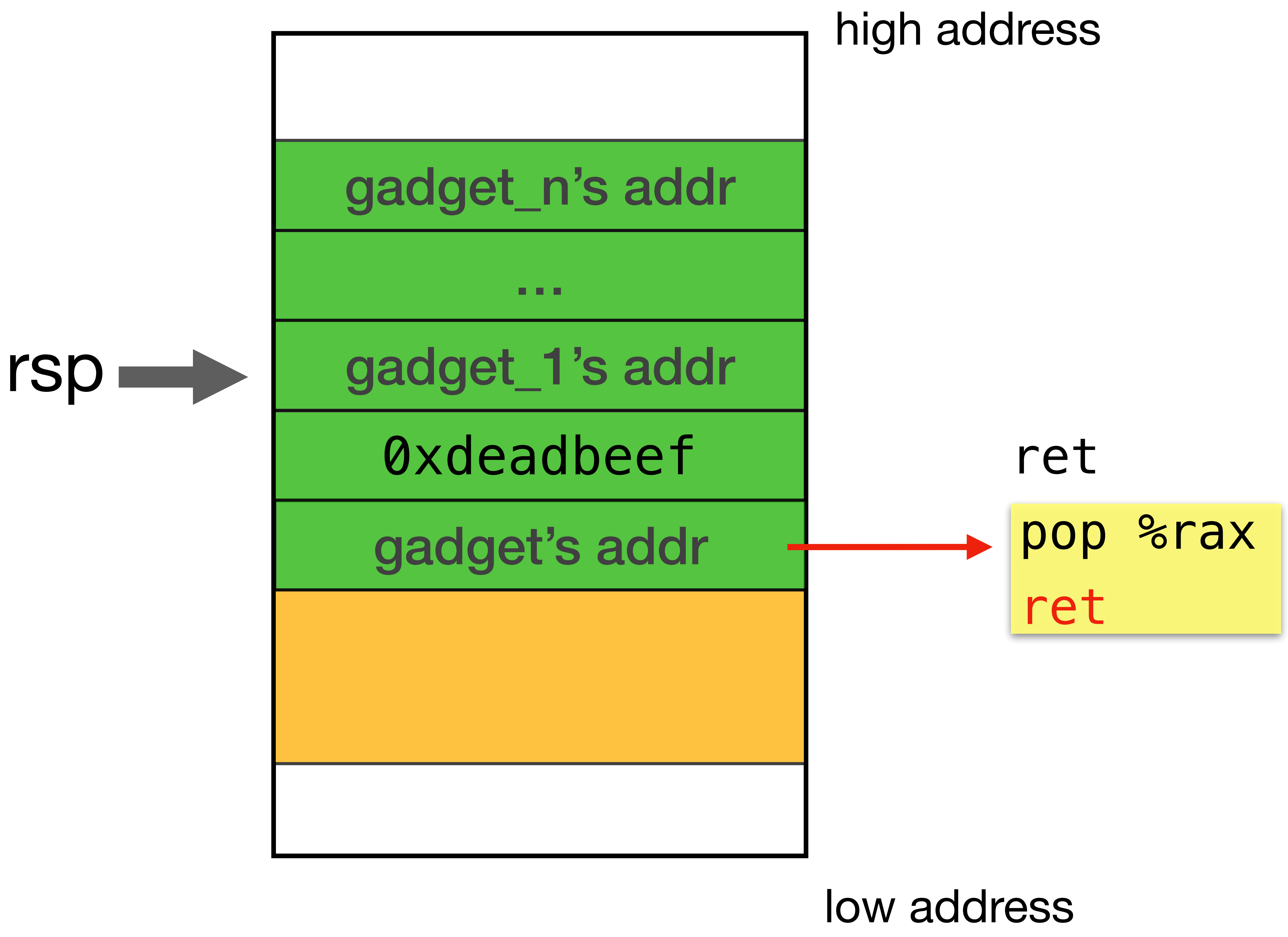


rax	123456
rsp	0x40008
rip	gadget_0's address

Loading a Constant

How to load an arbitrary constant (e.g. `0xdeadbeef`) into a register?

- Option 1: Pop the constant to the target register

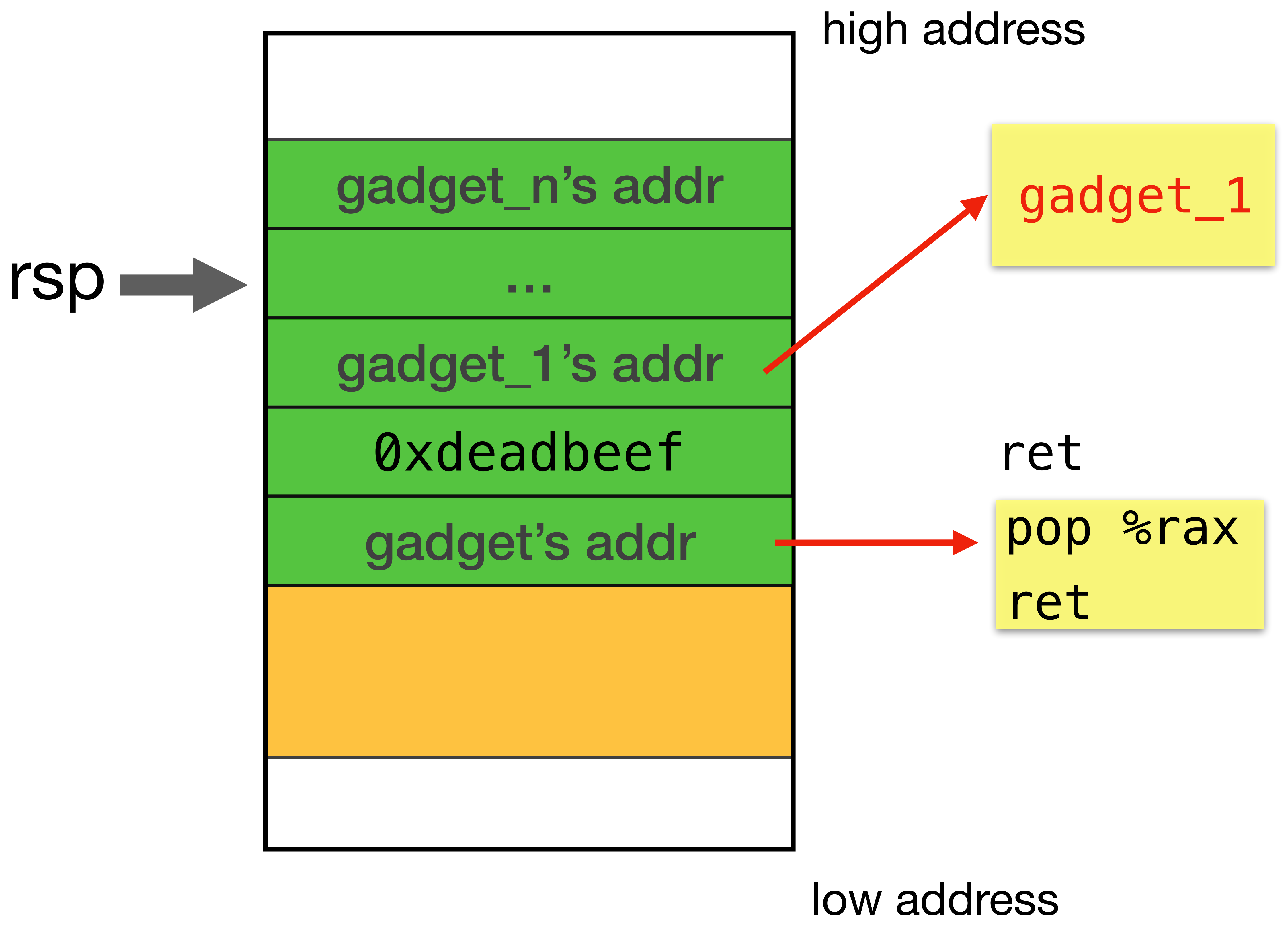


rax	0xdeadbeef
rsp	0x40010
rip	gadget_0's ret

Loading a Constant

How to load an arbitrary constant (e.g. `0xdeadbeef`) into a register?

- Option 1: Pop the constant to the target register



rax	0xdeadbeef
rsp	0x40018
rip	gadget_1's address

Loading From A Memory Address

How to load a value in memory into a register?

- e.g., want to load the value in address `0x400000` into `rbx`.

- Common load: `mov offset(%rax), %rbx`

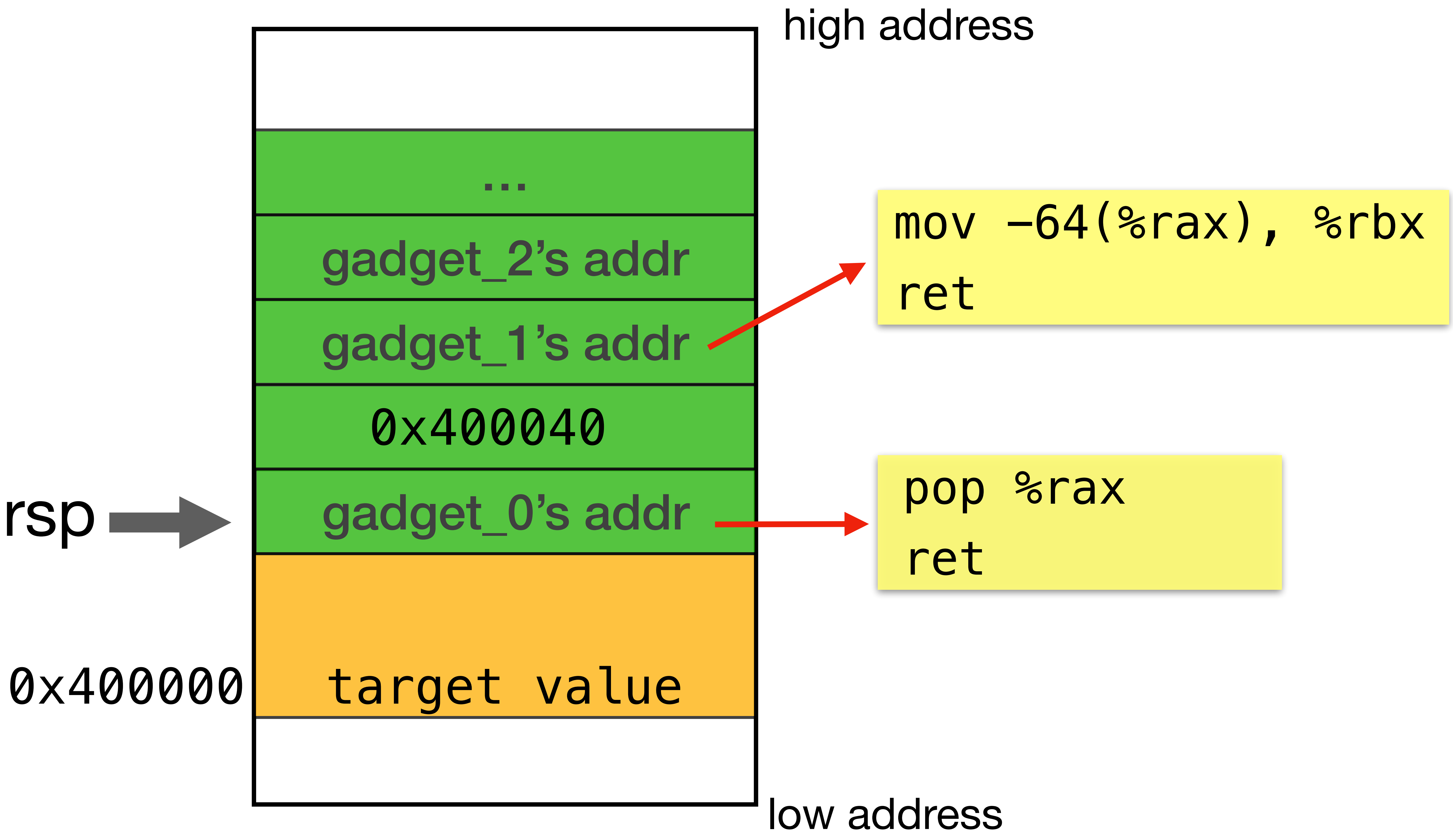
1. Set up the target address to one register
2. Load the value from the address into the target register

  - We need two gadgets.

Loading From A Memory Address

How to load a value in memory into a register?

- e.g., want to load the value in address `0x400000` into `rbx`.

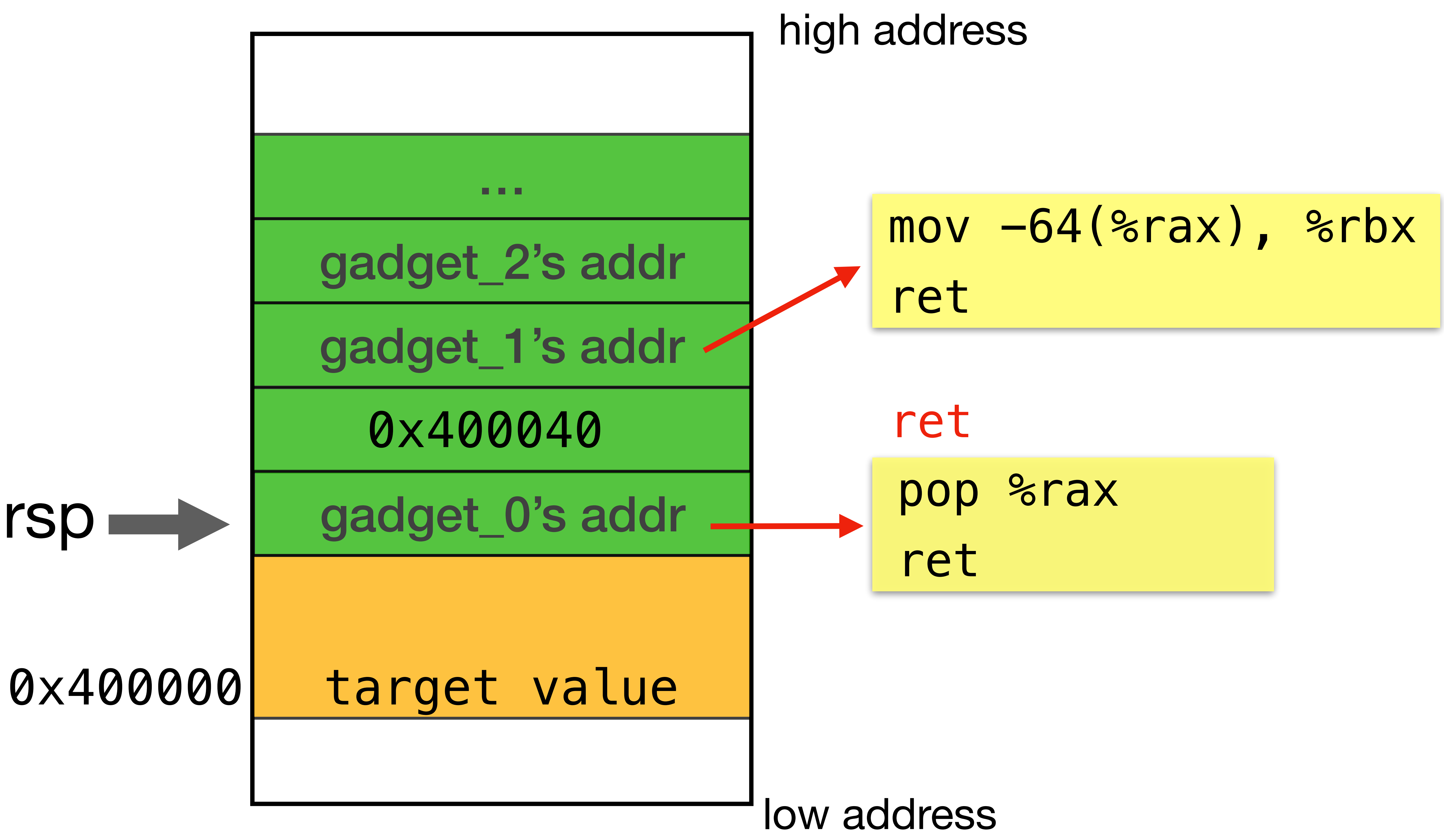


- Put gadgets' addresses on stack
- Put payload addresses on stack
- gadget_0 prepares target address 1
- gadget_1 loads from adjusted address

Loading From A Memory Address

How to load a value in memory into a register?

- e.g., want to load the value in address `0x400000` into `rbx`.

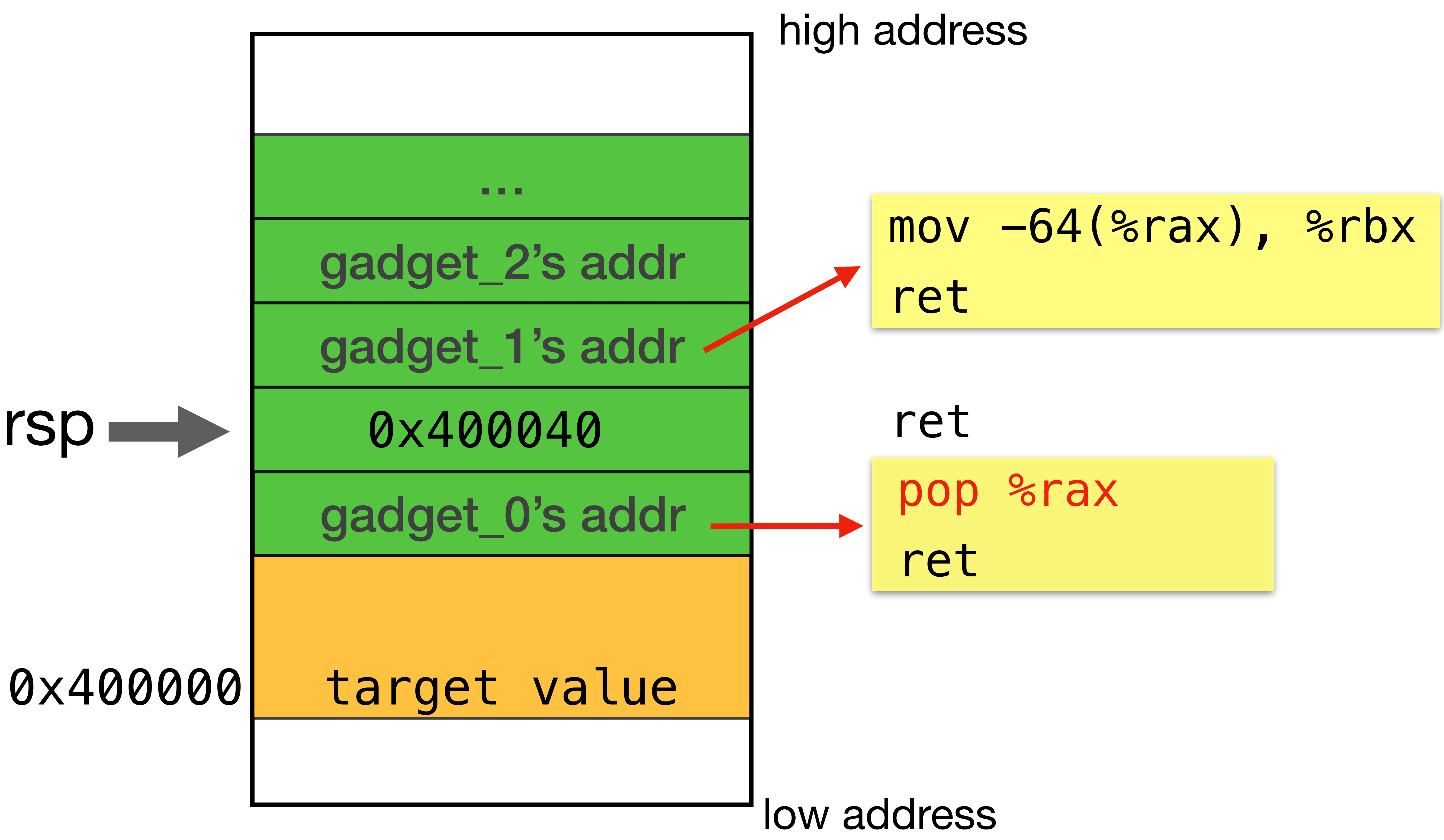


rax	0x123456
rbx	0x7890ab
rsp	0x400020
rip	ret's address

Loading From A Memory Address

How to load a value in memory into a register?

- e.g., want to load the value in address `0x400000` into `rbx`.

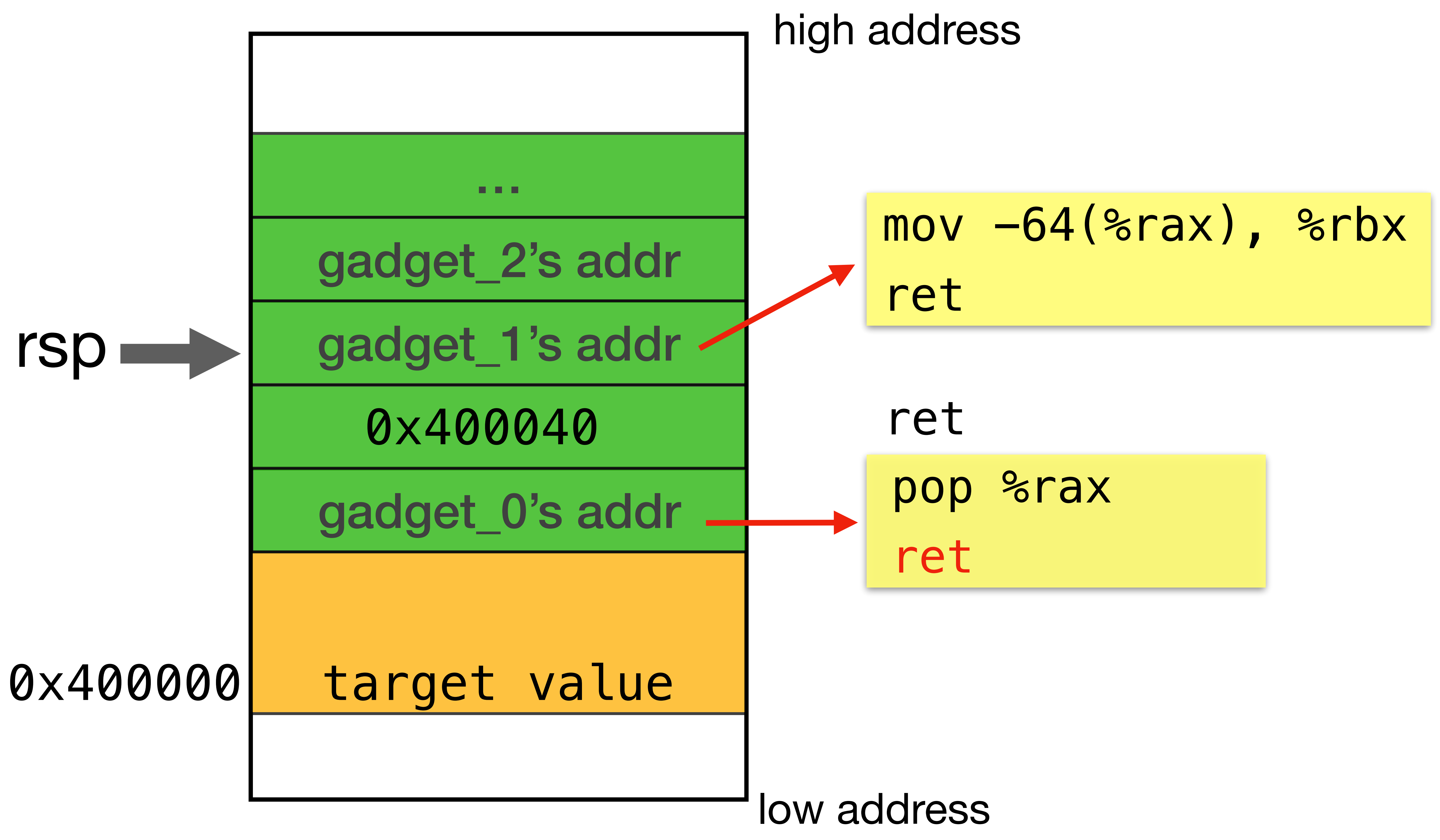


rax	0x123456
rbx	0x7890ab
rsp	0x400028
rip	gadget_0's address

Loading From A Memory Address

How to load a value in memory into a register?

- e.g., want to load the value in address `0x400000` into `rbx`.

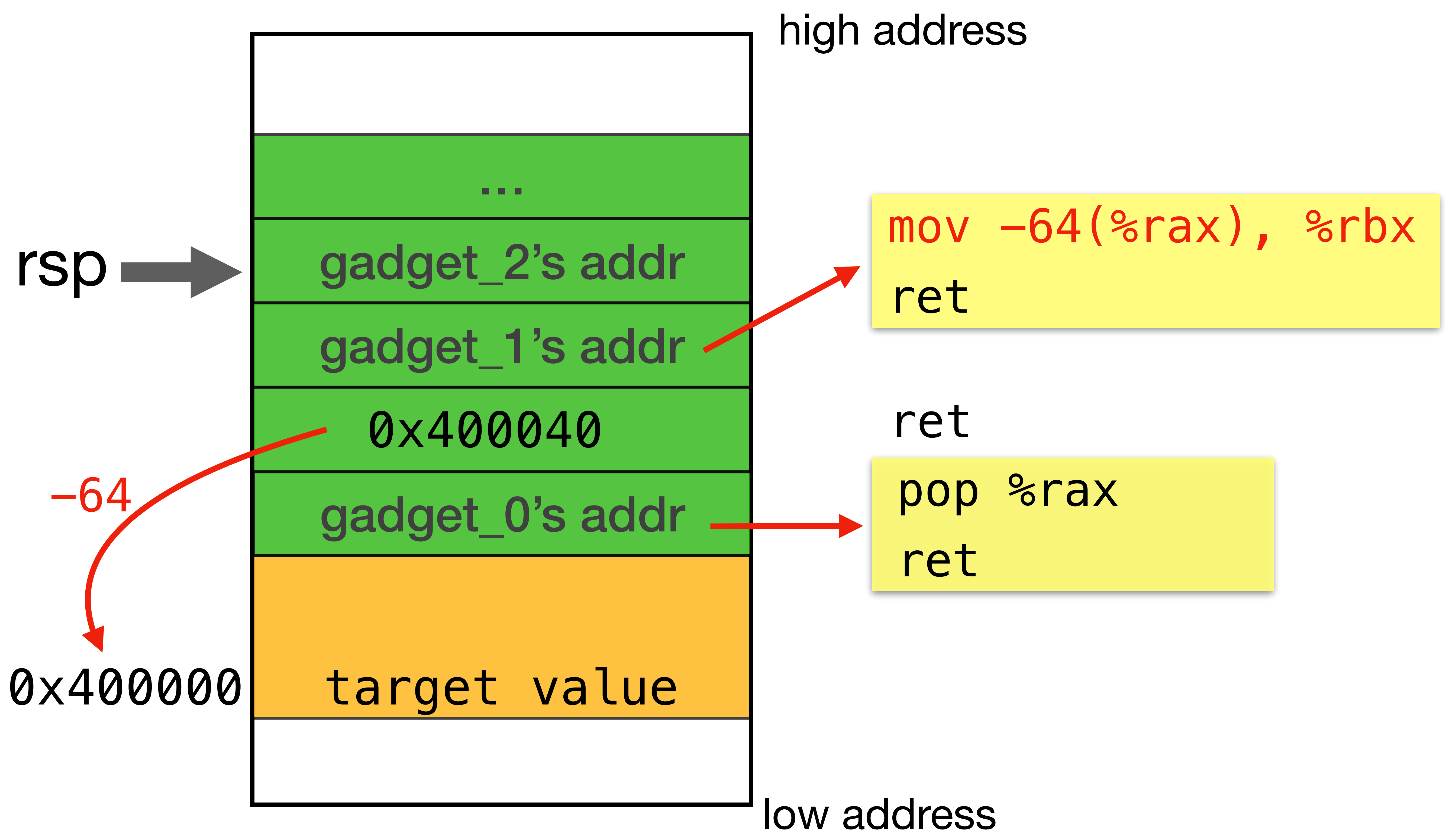


rax	0x400040
rbx	0x7890ab
rsp	0x400030
rip	gadget_0 ret's address

Loading From A Memory Address

How to load a value in memory into a register?

- e.g., want to load the value in address `0x400000` into `rbx`.

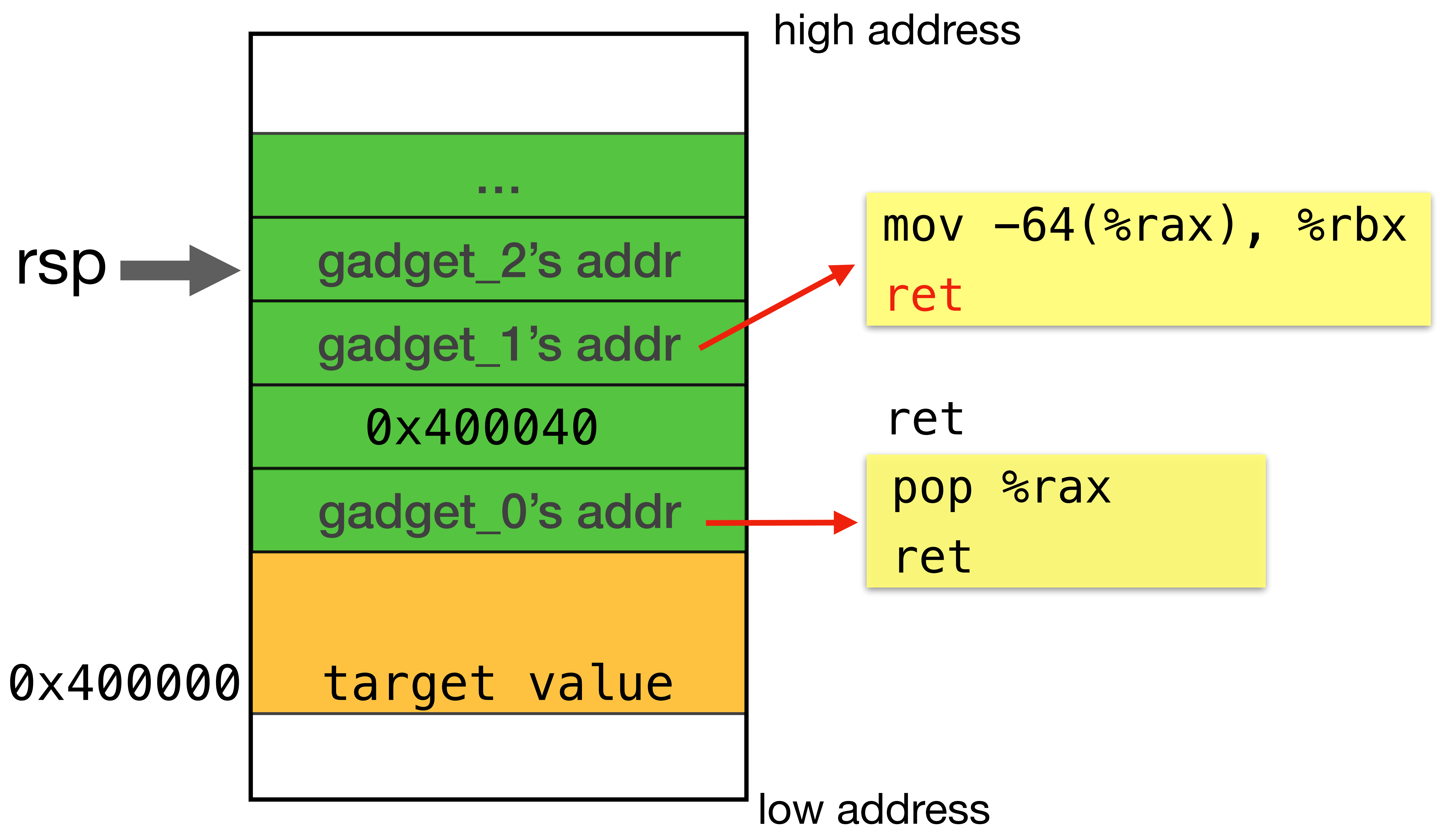


rax	0x400040
rbx	0x7890ab
rsp	0x400038
rip	gadget_1's address

Loading From A Memory Address

How to load a value in memory into a register?

- e.g., want to load the value in address `0x400000` into `rbx`.

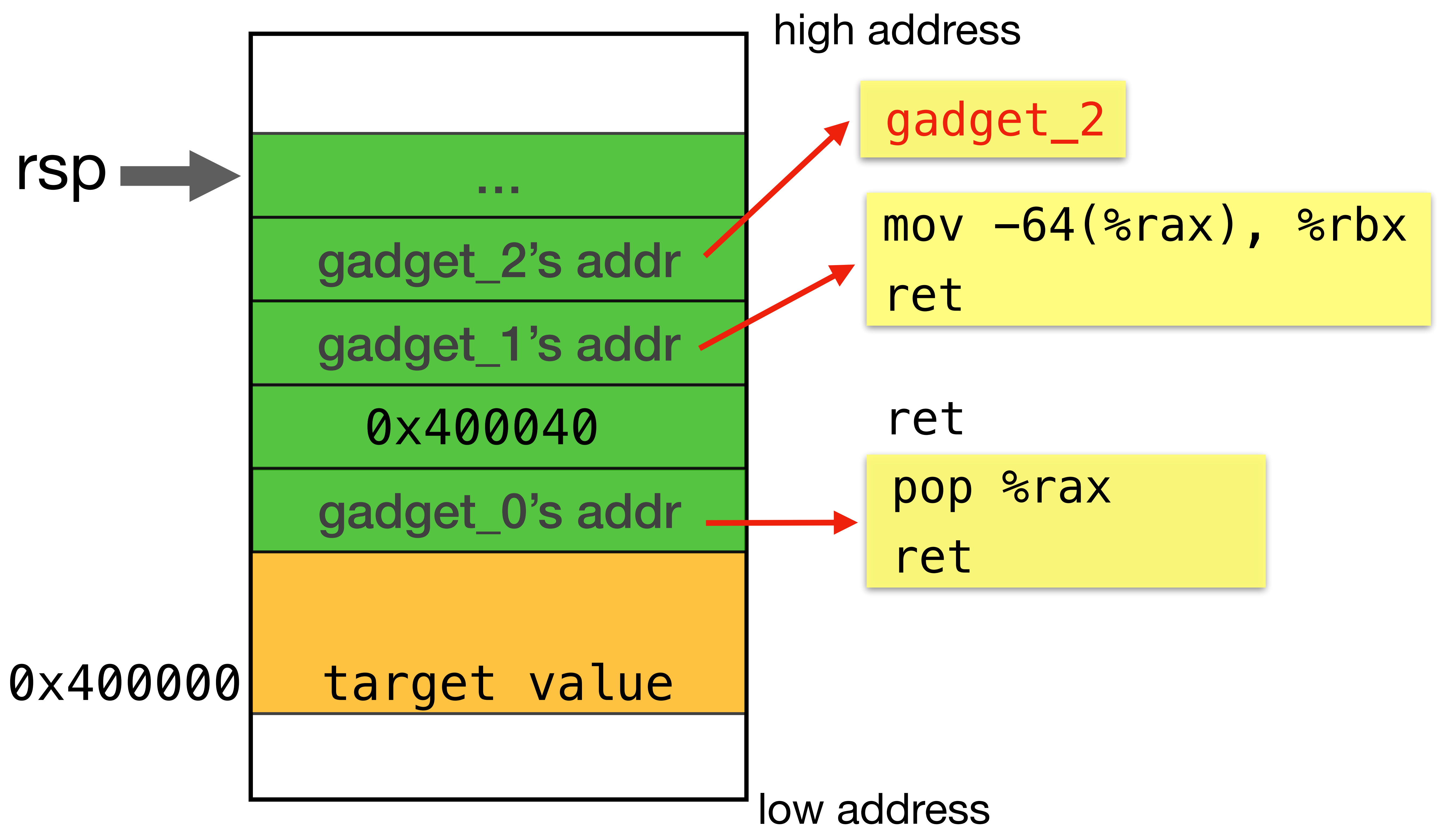


rax	0x400040
rbx	0x200000
rsp	0x400038
rip	gadget_1's ret address

Loading From A Memory Address

How to load a value in memory into a register?

- e.g., want to load the value in address `0x400000` into `rbx`.



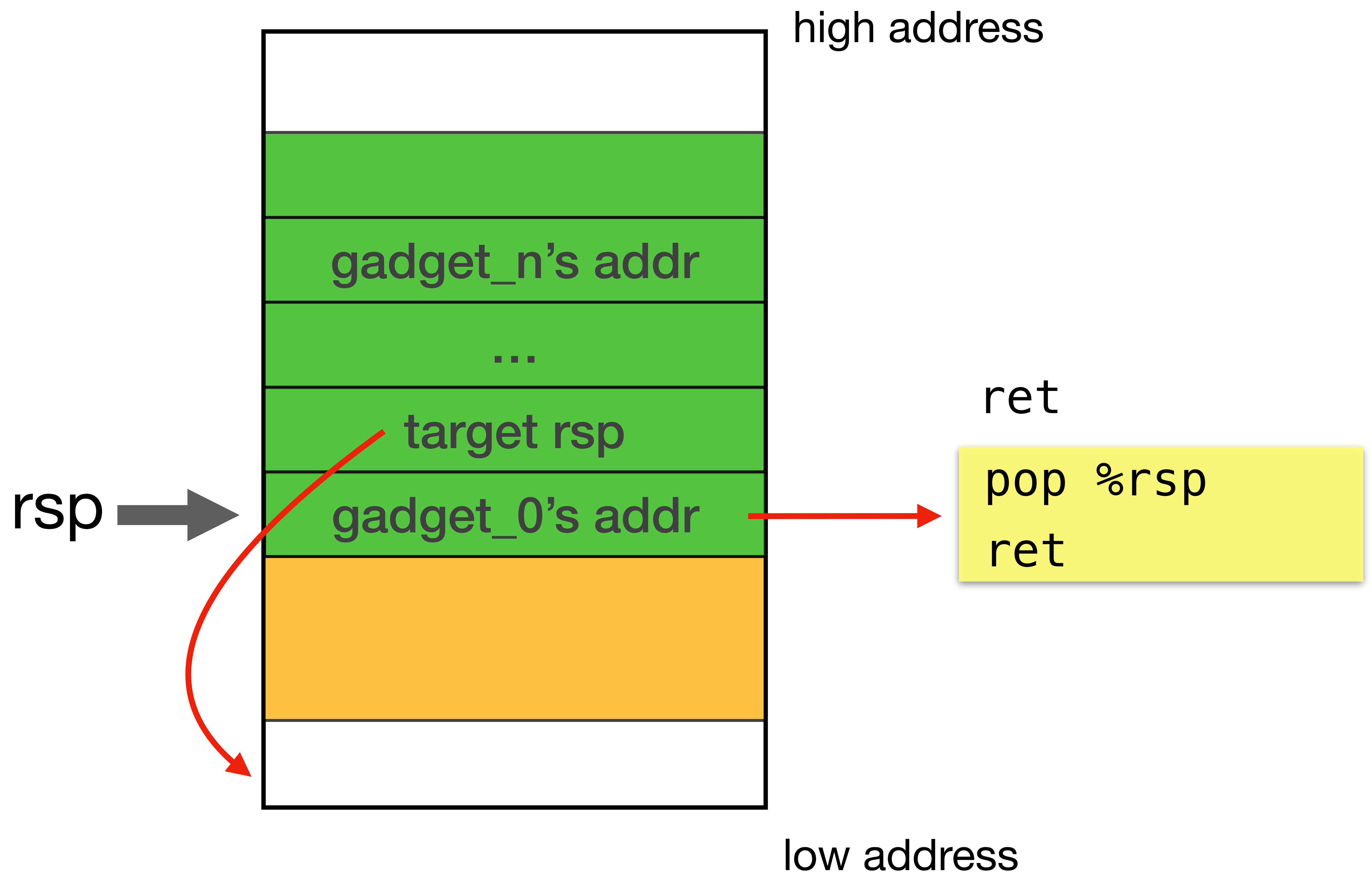
rax	0x400040
rbx	0x200000
rsp	0x400040
rip	gadget_2's address

What if the controllable stack space is insufficient for gadgets and their payloads?

Stack Pivoting

How to enable a larger "stack"?

- Pop the target address to `rsp`, and `ret`.

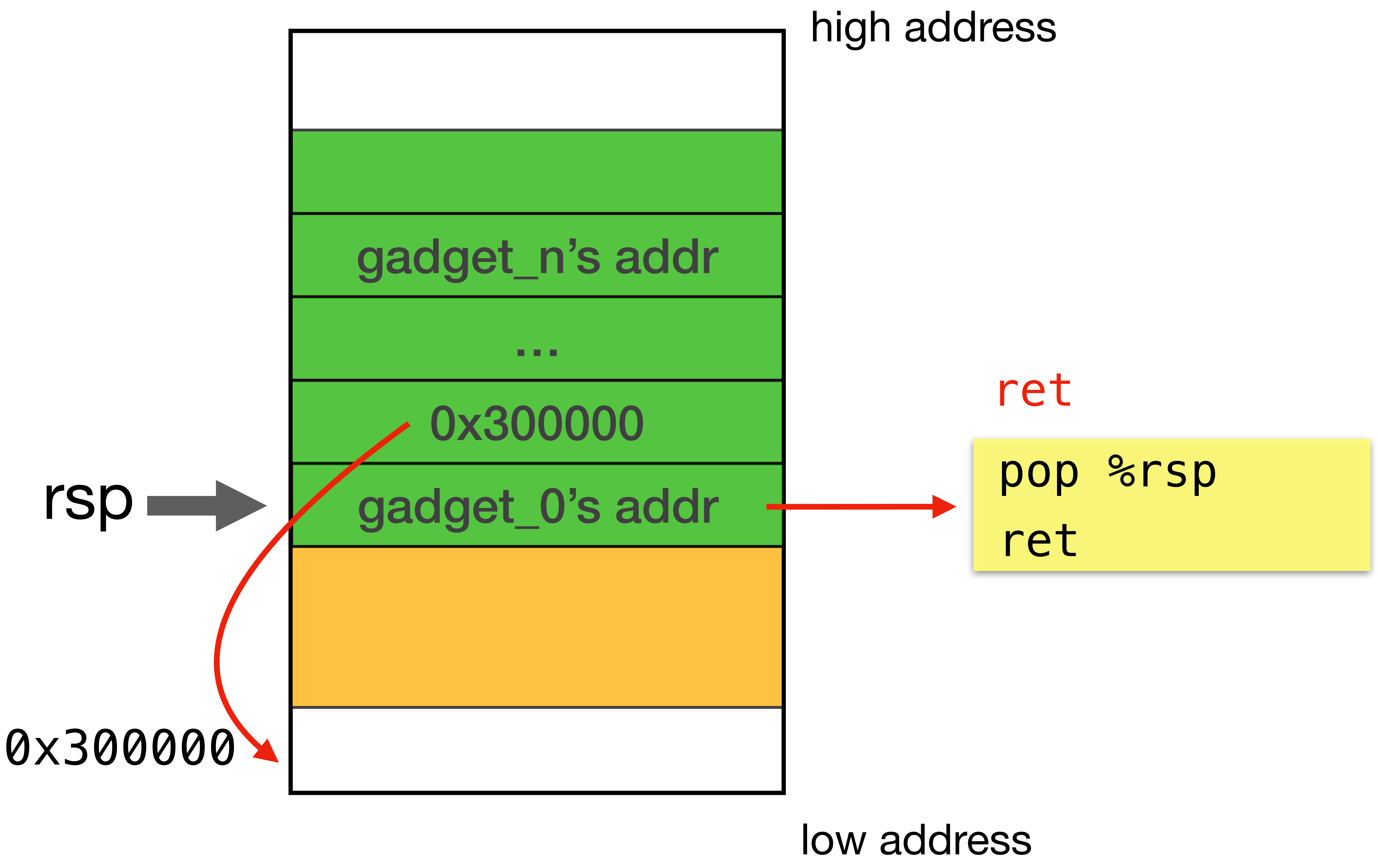


- Put gadget's address on stack
- Put target `rsp`'s address above
- Pop the target address to `rsp`

Stack Pivoting

How to enable a larger "stack"?

- Pop the target address (e.g. `0x300000`) to `rsp`, and `ret`.

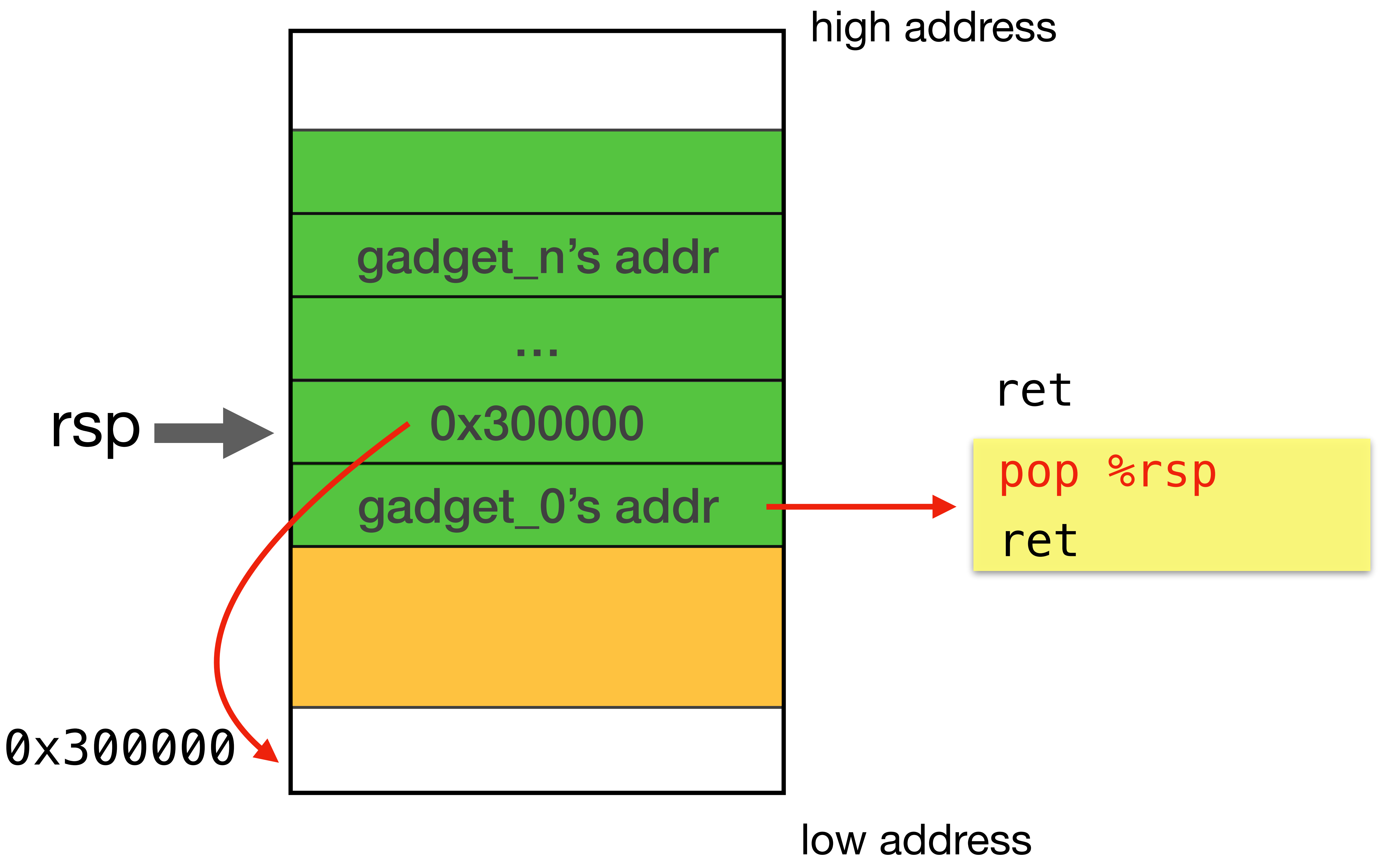


rsp	0x400000
rip	ret's address

Stack Pivoting

How to enable a larger “stack”?

- Pop the target address (e.g. `0x300000`) to `rsp`, and `ret`.



rsp	0x400008
rip	gadget’s address

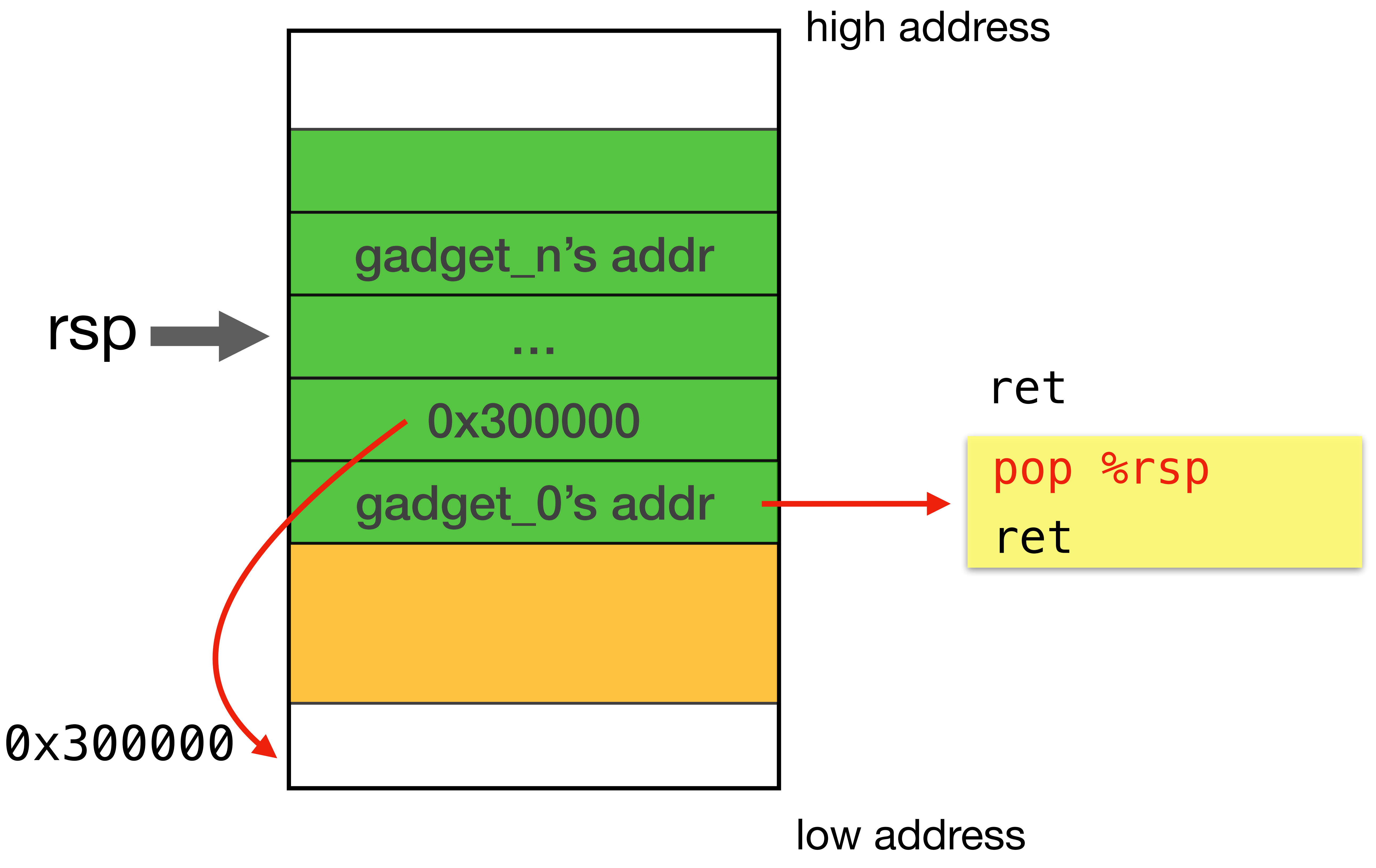
`pop %rsp` is special:

- `rsp` gets incremented by one word.
- Data pointed by old `rsp` is loaded to `rsp`.

Stack Pivoting

How to enable a larger "stack"?

- Pop the target address (e.g. `0x300000`) to `rsp`, and `ret`.



rsp	0x400010
rip	gadget's address

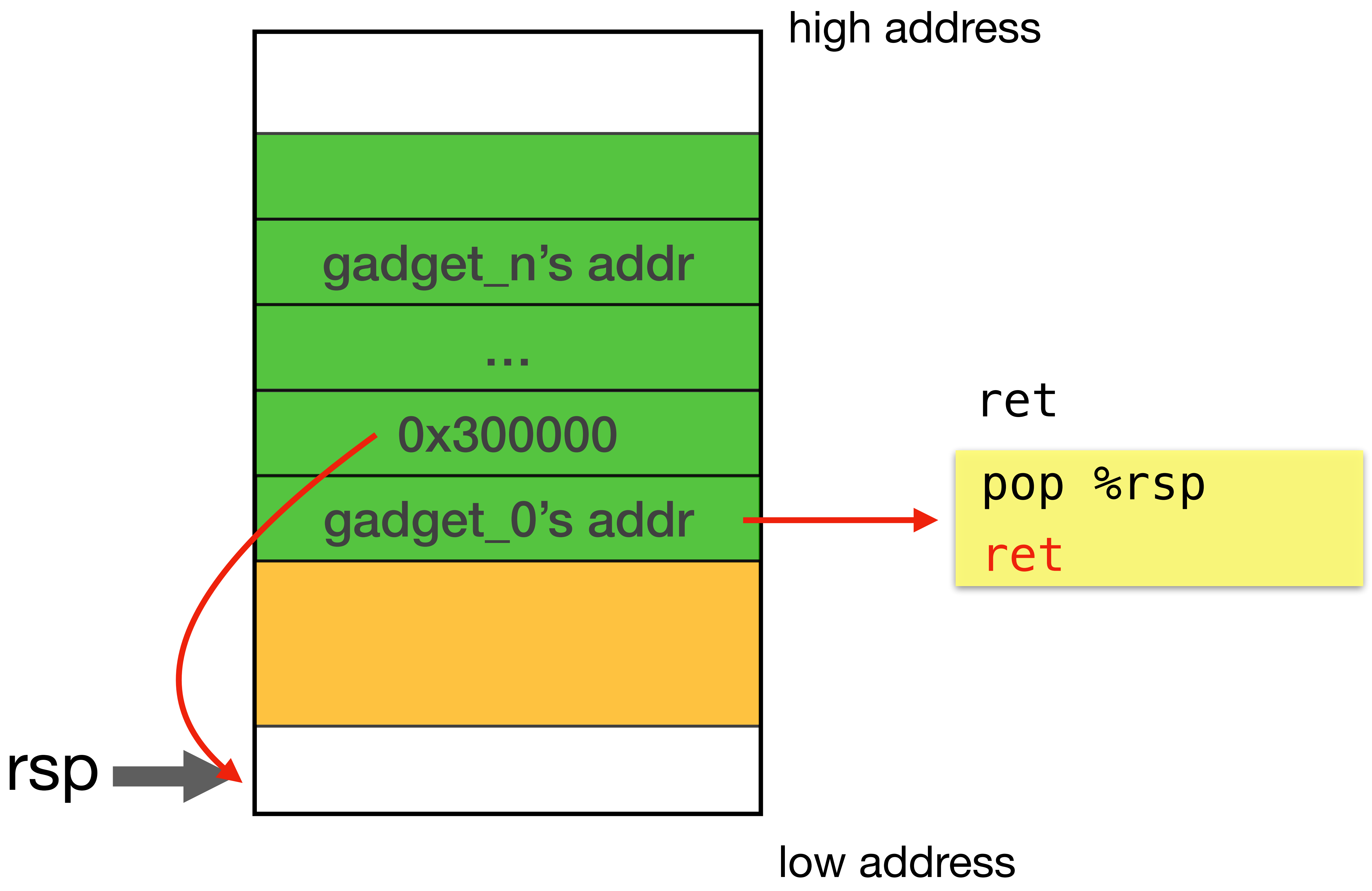
`pop %rsp` is special:

- `rsp` gets incremented by one word.
- Data pointed by old `rsp` is loaded to `rsp`.

Stack Pivoting

How to enable a larger "stack"?

- Pop the target address (e.g. `0x300000`) to `rsp`, and `ret`.



rsp	0x300000
rip	gadget ret's address

`pop %rsp` is special:

- `rsp` gets incremented by one word.
- Data pointed by old `rsp` is loaded to `rsp`.

Finding ROP Gadgets

ROP gadgets: Instructions sequences ending with a `ret`.

```c
void foo(int a, int b) {
      char buffer[12];
      gets(buffer);
      return;
}

int main() {
      int x;
      x = 0;
      foo(1,2);
      x = 1;
      printf(“%d\n",x);
      return 0;
}
```

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000001170 <+0>:     push   %rbp
   0x0000000000001171 <+1>:     mov    %rsp,%rbp
   0x0000000000001174 <+4>:     sub    $0x10,%rsp
   0x0000000000001178 <+8>:     movl   $0x0,-0x4(%rbp)
   0x000000000000117f <+15>:    movl   $0x0,-0x8(%rbp)
   0x0000000000001186 <+22>:    mov    $0x1,%edi
   0x000000000000118b <+27>:    mov    $0x2,%esi
   0x0000000000001190 <+32>:    call   0x1150 <foo>
   0x0000000000001195 <+37>:    movl   $0x1,-0x8(%rbp)
   0x000000000000119c <+44>:    mov    -0x8(%rbp),%esi
   0x000000000000119f <+47>:    lea    0xe5e(%rip),%rdi
   0x00000000000011a6 <+54>:    mov    $0x0,%al
   0x00000000000011a8 <+56>:    call   0x1030 <printf@plt>
   0x00000000000011ad <+61>:    xor    %eax,%eax
   0x00000000000011af <+63>:    add    $0x10,%rsp
   0x00000000000011b3 <+67>:    pop    %rbp
   0x00000000000011b4 <+68>:    ret
```

Finding ROP Gadgets

ROP gadgets: Instructions sequences ending with a `ret`.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000001170 <+0>:     push   %rbp
   0x0000000000001171 <+1>:     mov    %rsp,%rbp
   0x0000000000001174 <+4>:     sub    $0x10,%rsp
   0x0000000000001178 <+8>:     movl   $0x0,-0x4(%rbp)
   0x000000000000117f <+15>:    movl   $0x0,-0x8(%rbp)
   0x0000000000001186 <+22>:    mov    $0x1,%edi
   0x000000000000118b <+27>:    mov    $0x2,%esi
   0x0000000000001190 <+32>:    call   0x1150 <foo>
   0x0000000000001195 <+37>:    movl   $0x1,-0x8(%rbp)
   0x000000000000119c <+44>:    mov    -0x8(%rbp),%esi
   0x000000000000119f <+47>:    lea    0xe5e(%rip),%rdi
   0x00000000000011a6 <+54>:    mov    $0x0,%al
   0x00000000000011a8 <+56>:    call   0x1030 <printf@plt>
   0x00000000000011ad <+61>:    xor    %eax,%eax
   0x00000000000011af <+63>:    add    $0x10,%rsp
   0x00000000000011b3 <+67>:    pop    %rbp
   0x00000000000011b4 <+68>:    ret
```

```
Dump of assembler code for function foo:
   0x0000000000001150 <+0>:     push   %rbp
   0x0000000000001151 <+1>:     mov    %rsp,%rbp
   0x0000000000001154 <+4>:     sub    $0x20,%rsp
   0x0000000000001158 <+8>:     mov    %edi,-0x4(%rbp)
   0x000000000000115b <+11>:    mov    %esi,-0x8(%rbp)
   0x000000000000115e <+14>:    lea    -0x14(%rbp),%rdi
   0x0000000000001162 <+18>:    mov    $0x0,%al
   0x0000000000001164 <+20>:    call   0x1040 <gets@plt>
   0x0000000000001169 <+25>:    add    $0x20,%rsp
   0x000000000000116d <+29>:    pop    %rbp
   0x000000000000116e <+30>:    ret
```

How many gadgets can you find in these two functions?

ROP Gadgets Are Abundant

ROP gadgets: Instructions sequences ending with a `ret`.

- Linked libraries provide a plethora of instructions.
- x86 ISA uses variable-length instructions.
  - Allows *unintended* instruction sequences

ROP Gadgets Are Abundant

ROP gadgets: Instructions sequences ending with a `ret`.

- x86 ISA uses variable-length instructions.
  - Allows *unintended* instruction sequences
- **Linked libraries provide a plethora of instructions.**

7 English words

Linkedlibrariesprovideaplethoraofinstructions.

How many words can you find?

linked vid let fin ion any scan

link

ink

Unintended words become available.

ROP Gadgets Are Abundant

ROP gadgets: Instructions sequences ending with a `ret`.

- Linked libraries provide a plethora of instructions.
- x86 ISA uses variable-length instructions.
  - Allows *unintended* instruction sequences

ret is encoded as `0xc3` in hexadecimal format.

```
Dump of assembler code for function foo:
   0x0000000000001150 <+0>:     push   %rbp
   0x0000000000001151 <+1>:     mov    %rsp,%rbp
   0x0000000000001154 <+4>:     sub    $0x20,%rsp
   0x0000000000001158 <+8>:     mov    %edi,-0x4(%rbp)
   0x000000000000115b <+11>:    mov    %esi,-0x8(%rbp)
   0x000000000000115e <+14>:    lea    -0x14(%rbp),%rdi
   0x0000000000001162 <+18>:    mov    $0x0,%al
   0x0000000000001164 <+20>:    call   0x1040 <gets@plt>
   0x0000000000001169 <+25>:    add    $0x20,%rsp
   0x000000000000116d <+29>:    pop    %rbp
   0x000000000000116e <+30>:    ret

(gdb) x/4xb 0x000000000000116e
0x116e <foo+30>:        0xc3    0x90    0x55    0x48
```

ROP Gadgets Are Abundant

ROP gadgets: Instructions sequences ending with a `ret`.

- Linked libraries provide a plethora of instructions.
- x86 ISA uses variable-length instructions.
  - Allows *unintended* instruction sequences

ret is encoded as 0xc3 in hexadecimal format.

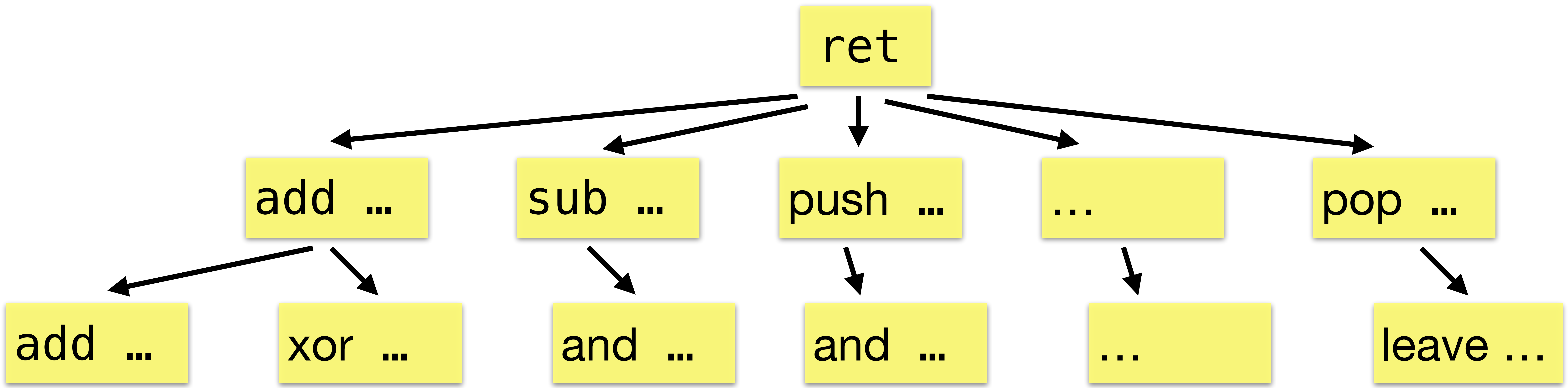
`f7 c7 07 00 00 00`	test \$0x00000007, %edi
`0f 95 45 c3`	setnzb -61(%ebp)

Starting one byte later, the attacker instead obtains

`c7 07 00 00 00 0f`	movl \$0x0f000000, (%edi)
`95`	xchg %ebp, %eax
`45`	inc %ebp
`c3`	ret

How to Find ROP Gadgets

- Start from a `ret` (`0xc3`) and backtrack to find gadgets.
  - Check whether the previous n bytes (n <= 15 for AMD64) form an instruction
  - Recurse from the previously found instruction



How to Find ROP Gadgets

- Start from a ret (0xc3) and backtrack to find gadgets.
  - Check whether the previous n bytes (n <= 15 for AMD64) form an instruction
  - Recurse from the previously found instruction

```
f7 c7 07 00 00 00      test $0x00000007, %edi
0f 95 45 c3            setnzb -61(%ebp)
```

Starting one byte later, the attacker instead obtains

```
c7 07 00 00 00 0f      movl $0x0f000000, (%edi)
95                     xchg %ebp, %eax
45                     inc %ebp
c3                     ret
```

ROP Thesis

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), by Hovav Shacham.

“In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake ***arbitrary*** *computation.”*

☹️ Also true in almost all other major architectures.

Find ROP Gadgets

ROPGagdet: A tool that examines binaries to find code-reuse gadgets.

```c
void foo(int a, int b) {
      char buffer[12];
      gets(buffer);
      return;
}

int main() {
      int x;
      x = 0;
      foo(1,2);
      x = 1;
      printf("%d\n",x);
      return 0;
}
```

How many ret gadgets in this program?

```
$ ROPgadget --binary demo | grep ret
4:0x00000000000010b3 : add byte ptr [rax], 0 ; add byte ptr [rax], al ; ret
7:0x00000000000010b4 : add byte ptr [rax], al ; add byte ptr [rax], al ; ret
8:0x0000000000001130 : add byte ptr [rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop dword ptr [rax] ; ret
13:0x00000000000010b6 : add byte ptr [rax], al ; ret
17:0x00000000000010f5 : add byte ptr [rax], r8b ; ret
18:0x0000000000001131 : add byte ptr [rcx], al ; pop rbp ; ret
20:0x0000000000001132 : add dword ptr [rbp - 0x3d], ebx ; nop dword ptr [rax] ; ret
21:0x000000000000112e : add eax, 0x100002f ; pop rbp ; ret
23:0x00000000000011b0 : add esp, 0x10 ; pop rbp ; ret
24:0x000000000000116a : add esp, 0x20 ; pop rbp ; ret
25:0x0000000000001017 : add esp, 8 ; ret
26:0x00000000000011af : add rsp, 0x10 ; pop rbp ; ret
27:0x0000000000001169 : add rsp, 0x20 ; pop rbp ; ret
28:0x0000000000001016 : add rsp, 8 ; ret
31:0x00000000000011bb : cli ; sub rsp, 8 ; add rsp, 8 ; ret
42:0x00000000000010f1 : loopne 0x1159 ; nop dword ptr [rax + rax] ; ret
43:0x000000000000112c : mov byte ptr [rip + 0x2f05], 1 ; pop rbp ; ret
44:0x00000000000010f3 : nop dword ptr [rax + rax] ; ret
45:0x00000000000010b1 : nop dword ptr [rax] ; ret
46:0x00000000000010f2 : nop word ptr [rax + rax] ; ret
47:0x00000000000010ef : or bh, bh ; loopne 0x1159 ; nop dword ptr [rax + rax] ; ret
48:0x0000000000001133 : pop rbp ; ret
51:0x000000000000101a : ret
52:0x0000000000001011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
53:0x00000000000011bd : sub esp, 8 ; add rsp, 8 ; ret
54:0x00000000000011bc : sub rsp, 8 ; add rsp, 8 ; ret
61:0x00000000000011ad : xor eax, eax ; add rsp, 0x10 ; pop rbp ; ret
```

Find ROP Gadgets

ROPGagdet: A tool that examines binaries to find code-reuse gadgets.

```
void foo(int a, int b) {
      char buffer[12];
      gets(buffer);
      return;
}

int main() {
      int x;
      x = 0;
      foo(1,2);
      x = 1;
      printf("%d\n",x);
      return 0;
}
```

How many ret gadgets in this program with statically-linked libraries?

```
[$ ROPgadget --binary demo.static | grep ret | wc -l
8375
```

Find ROP Gadgets

ROPGagdet: A tool that examines binaries to find code-reuse gadgets.

How many `ret` gadgets are in libc?

```
[$ ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 | grep ret | wc -l
19219
```

Return-oriented Programming (ROP)

An exploit technique that allows arbitrary code execution without calling any functions.

- Exploiting memory corruption bugs
  - Often starting with a corrupted return address
- Chaining code sequences, called *gadgets*, that end with a `ret`
  - Generally, gadgets ending with control flow transfer instructions, e.g. `jmp`
- Turing-complete
  - Memory operations
  - Arithmetic and logic
  - Control flow

What are the lessons we can learn from ROP?


TOM and JERRY
downloaded from pickywallpapers.com

“A single flower contains a whole world;
a single leaf embodies enlightenment.”

—Buddhāvataṃsaka Sūtra

“一花一世界，一叶一菩提。” —《华严经》

Integer Overflows

Signed vs. Unsigned Numbers

```c
char buf[N];
int len;
...
if (len > N) {
    error("Invliad length");
    return;
}
read(fd, buf, len);
```

What if len is negative?

```c
ssize_t read(int fd, void *buf, size_t count);
```

len will be cast to unsigned and negative length overflows, e.g., -1 -> 2^32 - 1 = 4294967295

Integer Overflows

An integer overflow occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value.

- Standard integer types (signed)
  - signed char, short int, int, long int, long long int
- Signed overflow vs unsigned overflow
  - A signed overflow occurs when a value is carried over to the sign bit.
  - An unsigned overflow occurs when the underlying representation can no longer represent an integer value.

Integer Overflow Examples

```
unsigned int ui;
signed int si;
ui = UINT_MAX; // 2^32 – 1 = 4,294,967,295
ui++;
printf("ui = %u\n", ui);

si = INT_MAX; // 2^31 – 1 = 2,147,483,647
si++;
printf("si = %d\n”, si);
```

What does it print?

0

What does it print?

-2^31 = -2,147,483,648

Integer Overflow Examples

```c
unsigned int ui;
signed int si;
ui = 0;
ui--;
printf("ui = %u\n", ui);

si = INT_MIN; // -2^31 = -2,147,483,648
si--;
printf("si = %d\n”, si);
```

What does it print?

2^32 - 1 = 4,294,967,295

What does it print?

2^31 - 1= 2,147,483,647

Security Threats of Integer Overflows

Take two strings from user input, and concatenate them on heap.

```
int main(int argc, char *const *argv) {
    unsigned short int total;
    total = strlen(argv[1]) + strlen(argv[2]) + 1;
    char *buff = (char *) malloc(total);
    strcpy(buff, argv[1]);
    strcat(buff, argv[2]);
}
```

What if the `total` variable overflows because of the addition operation?

Vulnerability: JEPG Example

Based on a real-world vulnerability in the handling of the comment field in JPEG files

```
void getComment(unsigned int len, char *src) {
     unsigned int size;
     size = len;
     char *comment = (char *)malloc(size + 1);
     memcpy(comment, src, size);
     return;
}
```

Will overflow to 0 if size is INT_MAX

How to fix it?

Vulnerability: JEPG Example

Based on a real-world vulnerability in the handling of the comment field in JPEG files

```
void getComment(unsigned int len, char *src) {
      unsigned int size;
      size = len - 2;
      char *comment = (char *)malloc(size + 1);
      memcpy(comment, src, size);
      return;
}
```

Any problem?

What if we do “`getComment(1, “My Comment”);`”?

- Overflow to cause `malloc` to allocate zero bytes.

Vulnerability: Truncation Errors

```
int func(char *name, unsigned int cbBuf) {
    unsigned short bufSize = cbBuf;
    char *buf = (char *)malloc(bufSize);
    if (buf) {
        memcpy(buf, name, cbBuf);
        …
        free(buf);
        return 0;
    }
    return 1;
}
```

What if we call the function with `cbBuf` greater than `USHRT_MAX`?

Heap Overflows

Buffer Overflows

- Stack overflow: overflowing a memory region on the stack (e.g., overwriting a return address)
- Heap overflow: overflowing a memory region dynamically allocated on the heap

```
char *packet = (char *)malloc(1000);
while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
if (authenticated)
   ProcessPacket(packet);
```

What happens if `PacketRead` overflows the packet buffer and overwrite important data in memory?

- e.g., `authenticated` is on the heap and corrupted

Overflowing Heap Critical User Data

```
typedef struct chunk {
    char inp[64];             /* vulnerable input buffer */
    void (*process)(char *);  /* pointer to function */
} chunk_t;

void showlen(char *buf) {
    int len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[]) {
    chunk_t *next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

Overflow the buffer on the heap to set the function pointer to an arbitrary address.

Overflow Heap Metadata

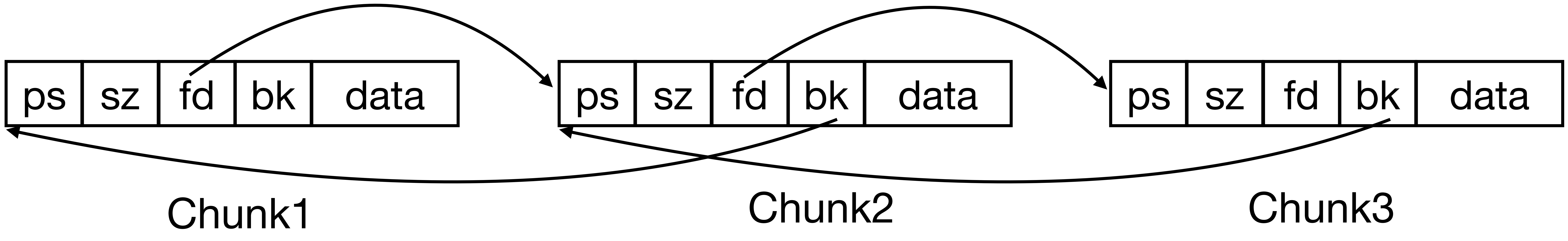
- Heap allocators (i.e., heap memory managers)
  - What regions have been allocated and their sizes
  - What regions are available for allocation
- Heap allocators maintain metadata such as chunk size, previous, and next pointers to other chunks.
  - Metadata are adjusted during heap-management functions.
    - `malloc()`, `callaoc()`, `realloc()`, etc. and `free()`
  - Heap metadata are often adjacent to heap user data

Example Heap Allocator

- Maintain a doubly-linked list of allocated and free chunks
- `malloc()` and `free()` modify this list

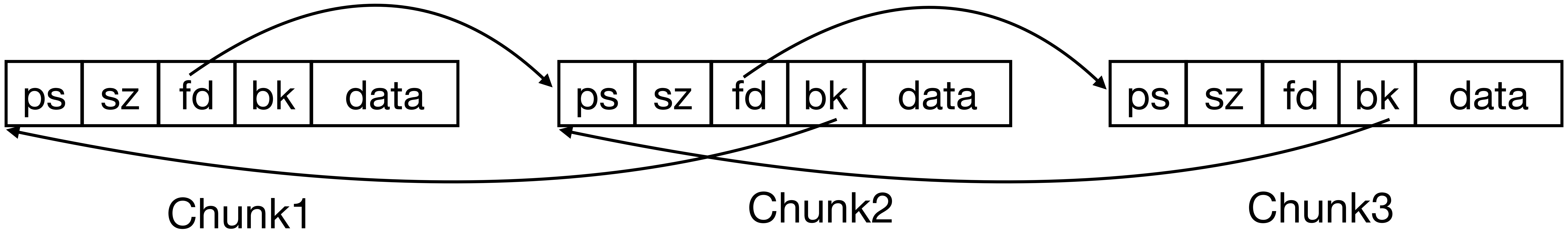
```
struct chunk {
    size_t   ps;      // prev_size
    size_t   sz;      // size
    chunk_t *fd;      // forward pointer
    chunk_t *bk;      // backward pointer
    unsigned char data[]; // allocated space for user data
};
```

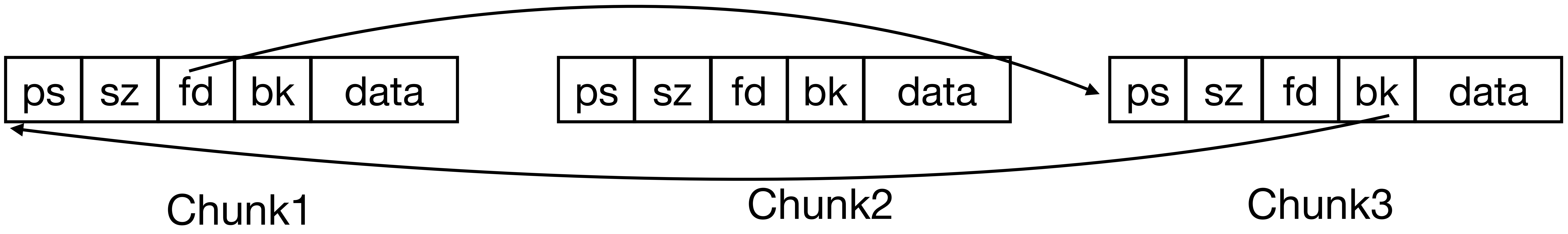
Example Heap Allocator



- ps: prev_size
- sz: size
- fd: forward pointer
- bk: backward pointer
- data: allocated space for user data

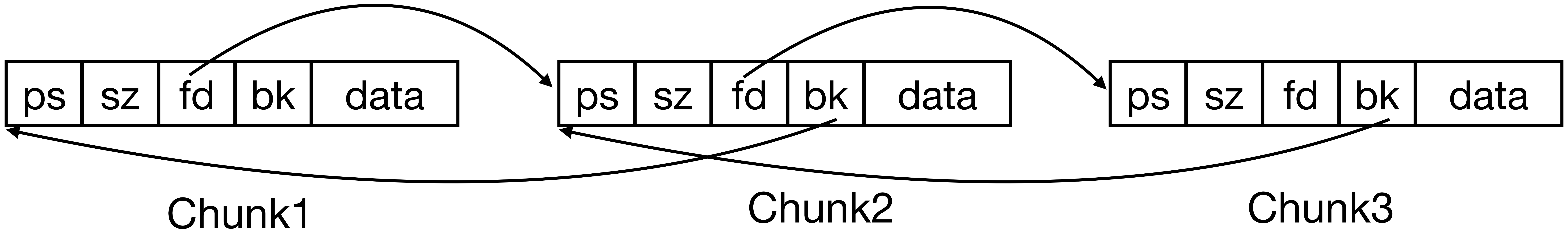
Example Heap Allocator





- `malloc()` removes a chunk from free list
  - `chunk2->bk->fd = chunk2->fd`
  - `chunk2->fd->bk = chunk2->bk`

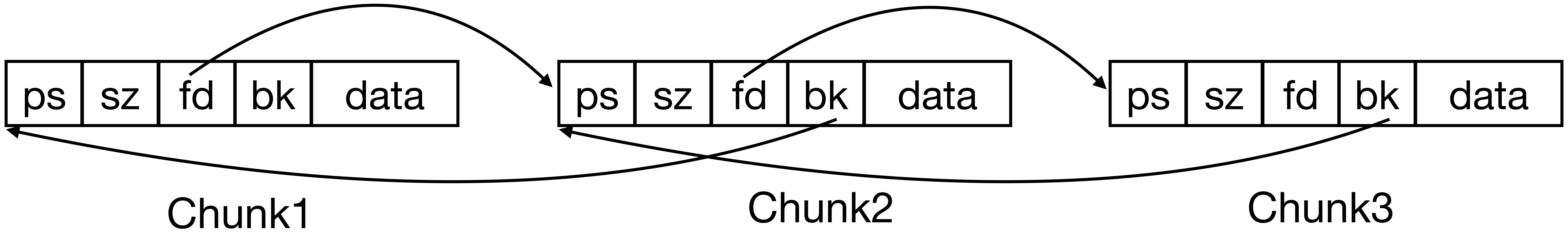
Attacking the Example Heap Allocator



- `malloc()` removes a chunk from allocated list
  - `chunk2->bk->fd = chunk2->fd`
  - `chunk2->fd->bk = chunk2->bk`
- By overflowing `chunk2`, attacker controls `bk` and `fd` of `chunk2`

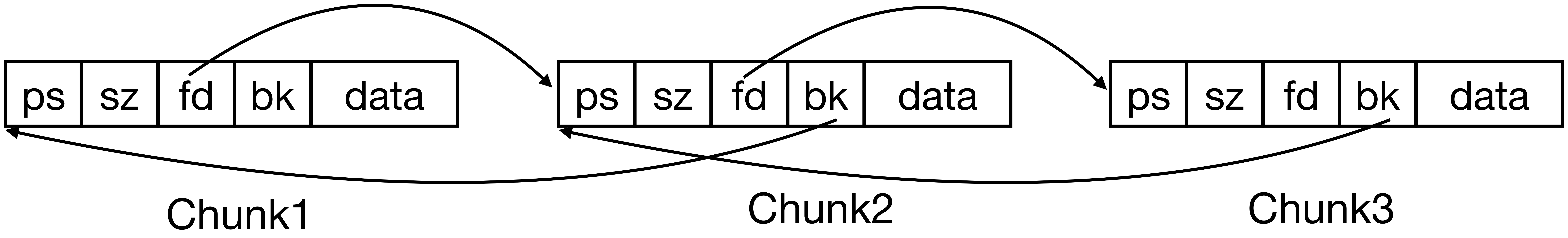
  How to exploit this vulnerability for arbitrary writes (write-where-what vul.)?

Attacking the Example Heap Allocator



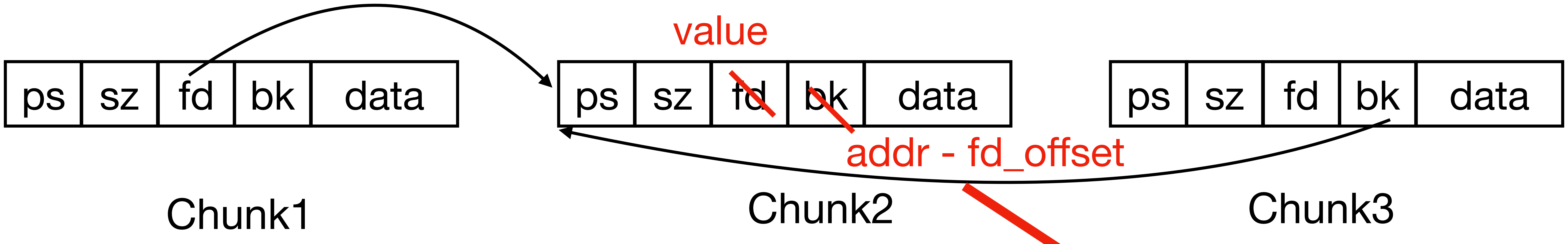
- `malloc()` removes a chunk from allocated list
  - `chunk2->bk->fd = chunk2->fd`
  - `chunk2->fd->bk = chunk2->bk`
- By overflowing `chunk2`, attacker controls `bk` and `fd` of `chunk2`
- Suppose the attacker wants to write `value` to memory address `addr`
  - Set `chunk2->fd` to be `value`
  - Set `chunk2->bk` to be `addr - fd_offset`, where `fd_offset` is the offset of the `fd` field in the chunk structure.

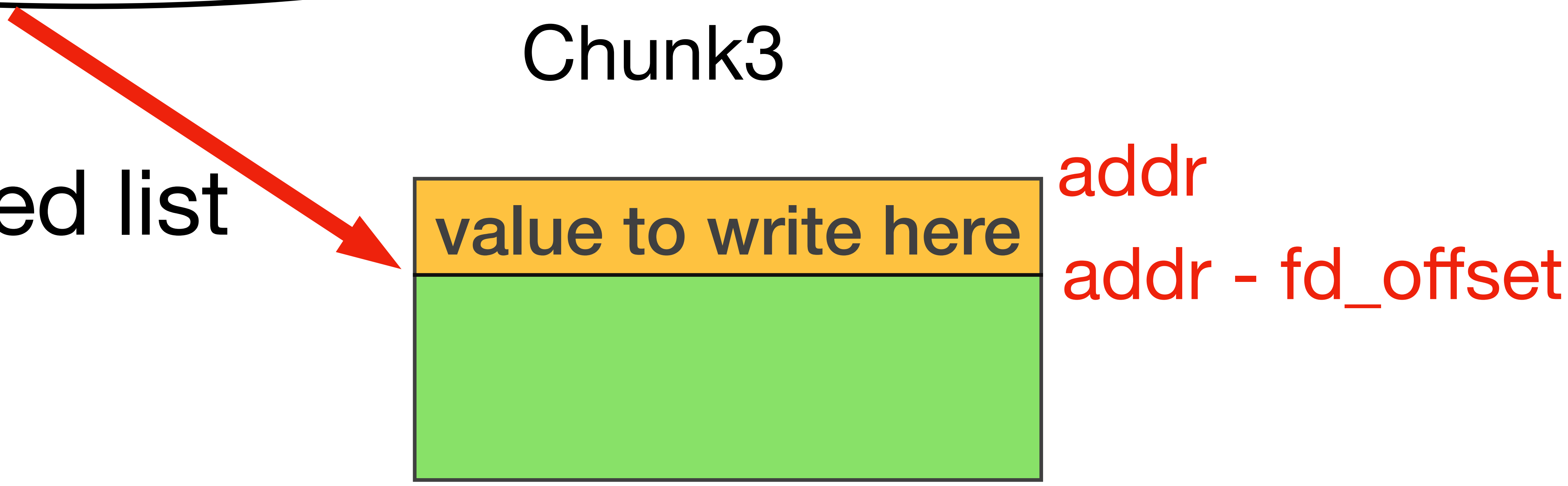
Attacking the Example Heap Allocator



- `malloc()` removes a chunk from allocated list
  - `chunk2->bk->fd = chunk2->fd`
  - `chunk2->fd->bk = chunk2->bk`
- By overflowing `chunk2`, attacker controls `bk` and `fd` of `chunk2`
- `malloc()` changes the program as follows:
  - `(addr - fd_offset)->fd = value`, the same as `(*addr) = value`
  - `value->bk = addr - offset`

Attacking the Example Heap Allocator





- `malloc()` removes a chunk from allocated list
  - `chunk2->bk->fd = chunk2->fd`
  - `chunk2->fd->bk = chunk2->bk`
- By overflowing `chunk2`, attacker controls `bk` and `fd` of `chunk2`
- `malloc()` changes the program as follows:
  - `(addr - fd_offset)->fd = value`, the same as `(*addr) = value`
  - `value->bk = addr - offset`

Enables arbitrary memory write!