SECURITY (COMP0141): BUFFER OVERFLOWS

FUNCTION CALLS

How does a function call work?

```
b = check_auth(“abc123”);
```

```
int check_auth(char *password) {
    int auth_flag = 0;
    char pass[16];
    strcpy(pass, password);
    if (strcmp(pass, "abc123") == 0)
        auth_flag = 1;
    return auth_flag;
}
```

How does the called function know where to return to?

Where is the return address stored?

Answer: the computer keeps track using a stack

FUNCTION CALLS

```
_check_auth:
100000e50:      55              pushq   %rbp
100000e51:      48 89 e5                movq    %rsp, %rbp
100000e54:      48 83 ec 30             subq    $48, %rsp
100000e58:      48 8d 45 e0             leaq    -32(%rbp), %rax
100000e5c:      48 89 7d f8             movq    %rdi, -8(%rbp)
100000e60:      c7 45 f4 00 00 00 00    movl    $0, -12(%rbp)
100000e67:      48 8b 75 f8             movq    -8(%rbp), %rsi
100000e6b:      48 89 c7                movq    %rax, %rdi
100000e6e:      48 89 45 d8             movq    %rax, -40(%rbp)
100000e72:      e8 c3 00 00 00          callq   195
100000e77:      48 8d 35 f0 00 00 00    leaq    240(%rip), %rsi
100000e7e:      48 8b 7d d8             movq    -40(%rbp), %rdi
100000e82:      48 89 45 d0             movq    %rax, -48(%rbp)
100000e86:      e8 a9 00 00 00          callq   169
100000e8b:      83 f8 00                cmpl    $0, %eax
100000e8e:      0f 85 07 00 00 00       jne     7 <_check_auth+0x4b>
100000e94:      c7 45 f4 01 00 00 00    movl    $1, -12(%rbp)
100000e9b:      8b 45 f4                movl    -12(%rbp), %eax
100000e9e:      48 83 c4 30             addq    $48, %rsp
100000ea2:      5d              popq    %rbp
100000ea3:      c3              retq
100000ea4:      66 66 66 2e 0f 1f 84 00 00 00 00 00     nopw    %cs:(%rax,%rax)
```

FUNCTION CALLS

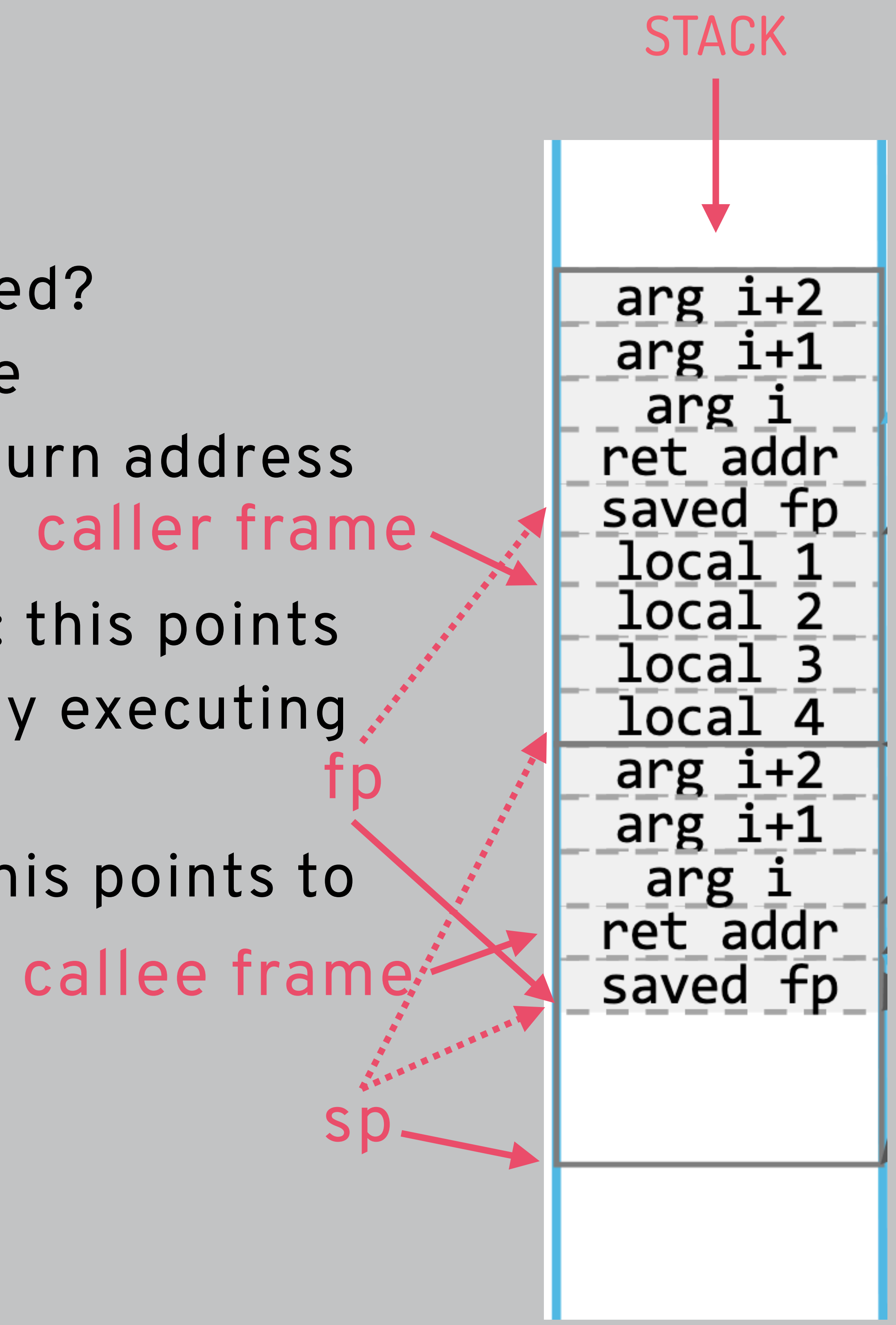
```c
#include <stdio.h>
#include <string.h>

int check_auth(char *password) {
    int auth_flag = 0;
    char pass[16];

    strcpy(pass, password);
    if (strcmp(pass, "abc123") == 0)
        auth_flag = 1;

    return auth_flag;
}
```

```asm
check_auth(char*):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 48
        mov     QWORD PTR [rbp-40], rdi
        mov     DWORD PTR [rbp-4], 0
        mov     rdx, QWORD PTR [rbp-40]
        lea     rax, [rbp-32]
        mov     rsi, rdx
        mov     rdi, rax
        call    strcpy
        lea     rax, [rbp-32]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, rax
        call    strcmp
        test    eax, eax
        jne     .L2
        mov     DWORD PTR [rbp-4], 1
.L2:
        mov     eax, DWORD PTR [rbp-4]
        leave
        ret
```

CALL FRAME

What happens when a function is called?

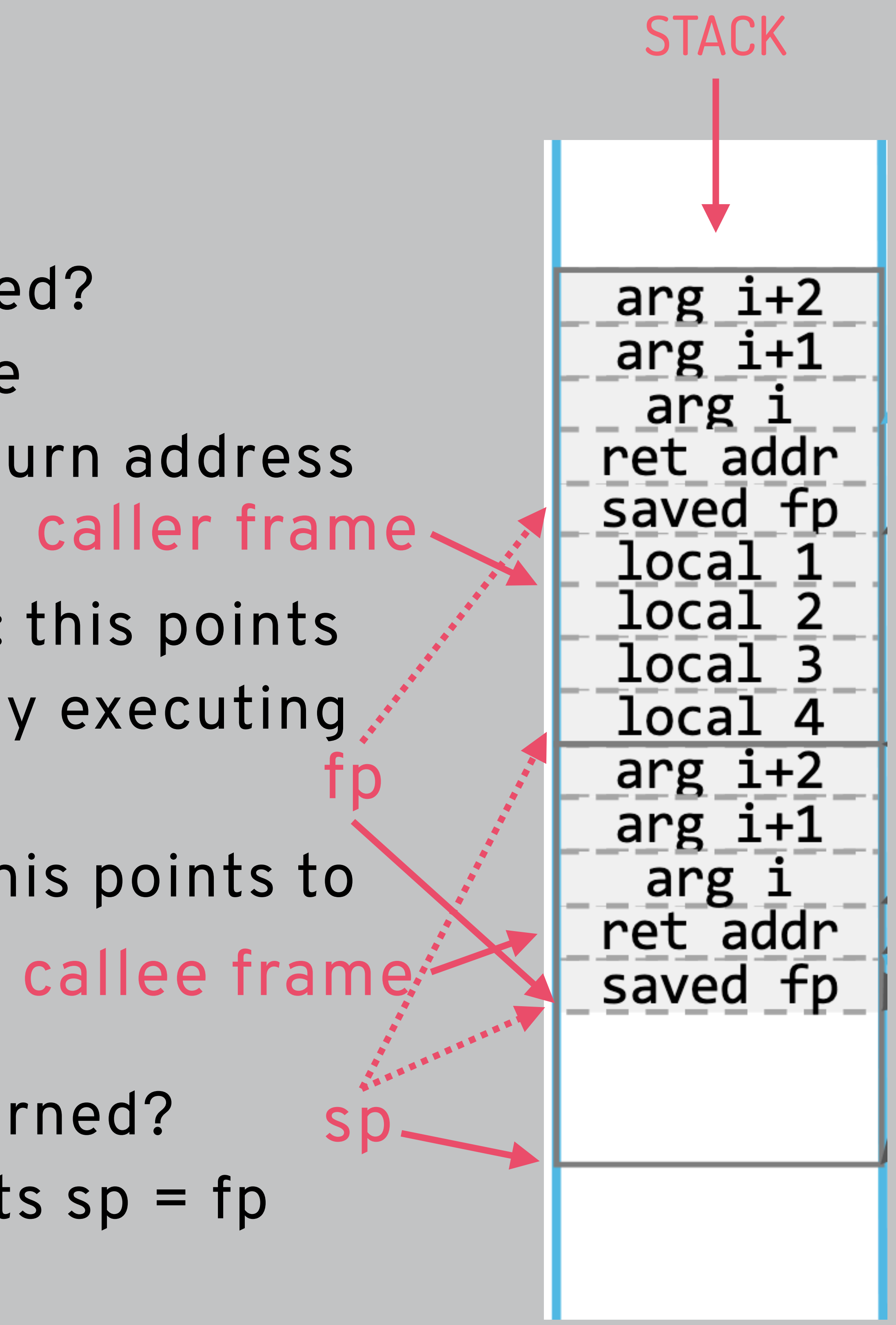
- Allocate new **frame** for the callee
- Caller pushes arguments and return address
- Callee:
  - pushes old **frame pointer** (fp): this points to bottom of frame of currently executing function
  - sets fp = sp (**stack pointer**): this points to top of stack)



CALL FRAME

What happens when a function is called?

- Allocate new **frame** for the callee
- Caller pushes arguments and return address
- Callee:
  - pushes old **frame pointer** (fp): this points to bottom of frame of currently executing function
  - sets fp = sp (**stack pointer**): this points to top of stack)



What happens when a function is returned?

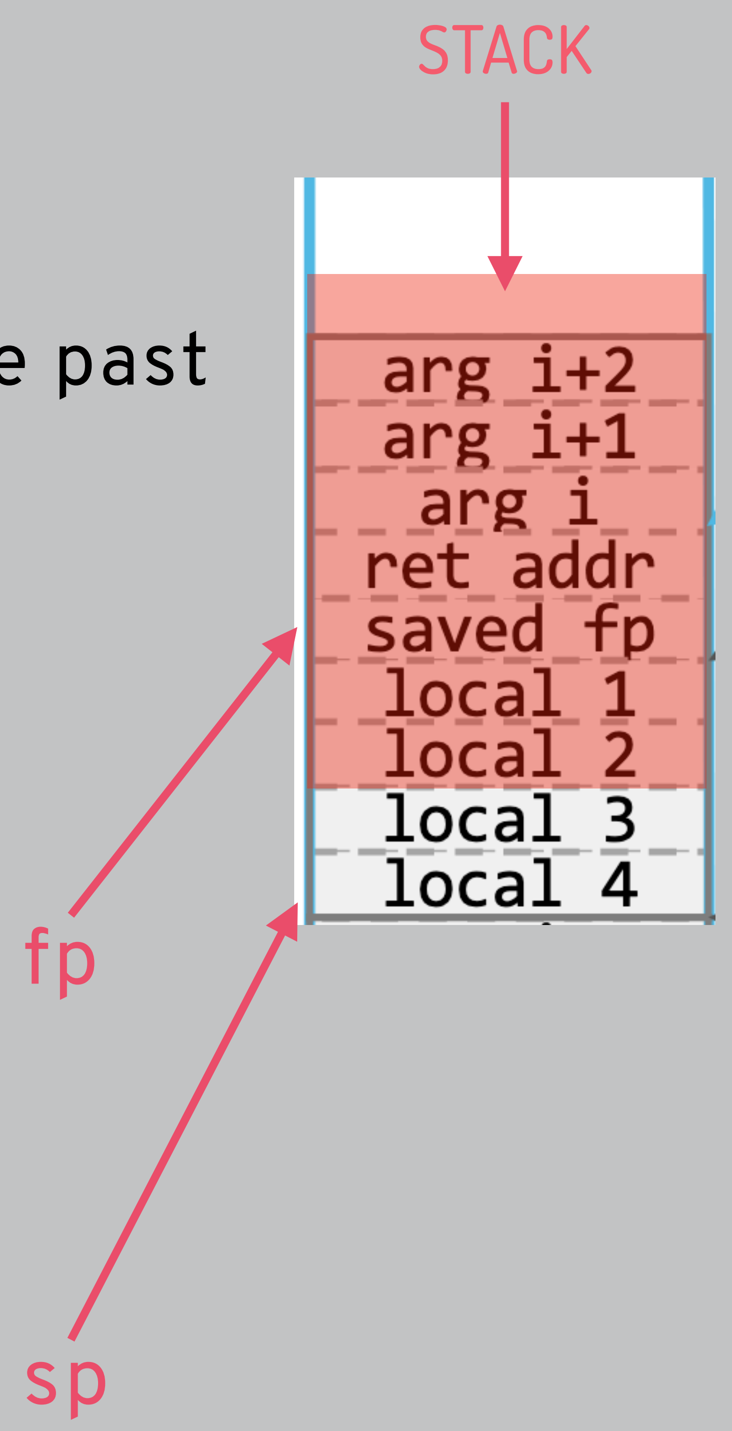
- Callee pops local storage and sets sp = fp
- Callee pops frame pointer
- Callee pops return address and returns to next instruction in caller frame

SMASHING THE STACK

What happens if you overwrite a malicious value past the bounds of a local variable?

Could overwrite:

- Another local variable
- Saved fp
- Return address
- Function argument
- Deeper stack frames
- Exception control data



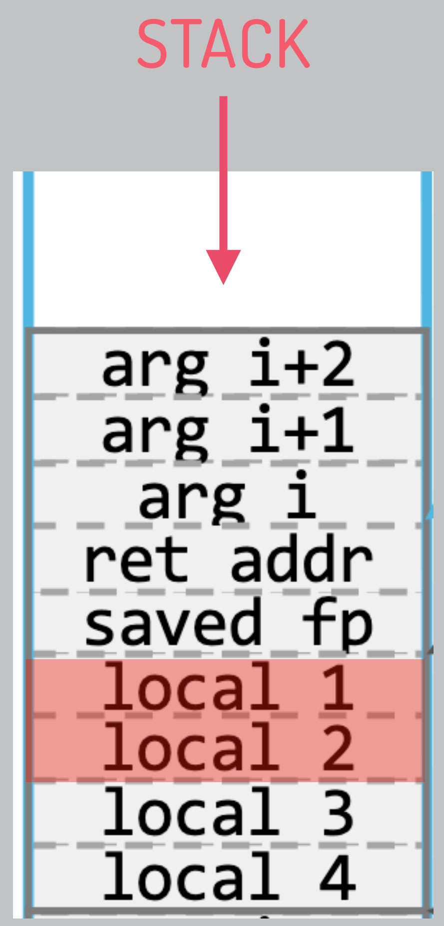
LOCAL VARIABLES

What happens if you overwrite another local variable?

Depends!

Bad if:

- Results of a security check (`isValid`)
- Variable used in security check (`buff_size`)
- Data pointer (potential for further corruption)
- Function pointer (direct transfer of control when called)



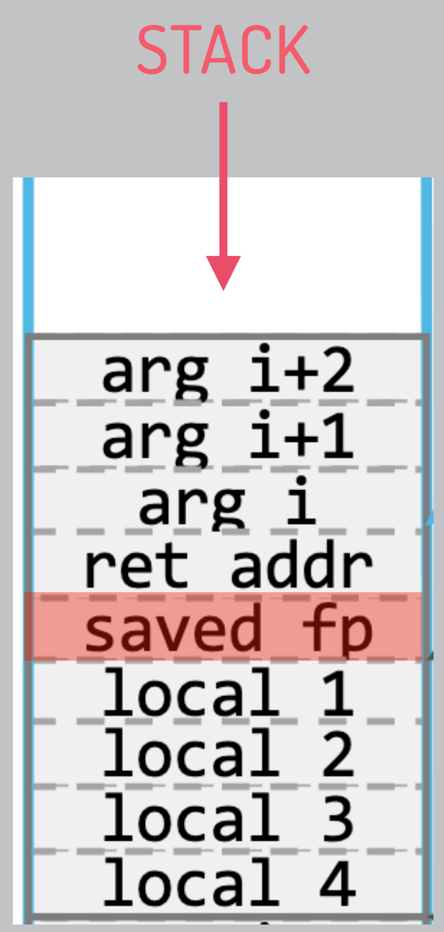
SAVED FP

What happens if you overwrite the saved fp?

Probably terrible things!

When the function returns, the stack moves to an attacker-supplied address ⇒ complete control of execution

Even a single byte may be enough for this attack



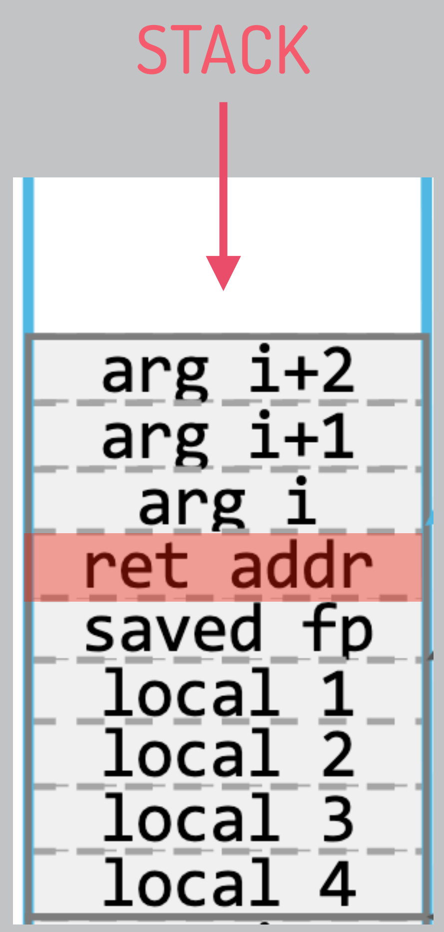
RETURN ADDRESS

What happens if you overwrite the return address?

Terrible things!

When the function returns, control is transferred to an attacker-supplied address ⇒ complete control of arbitrary code execution (re-direct to their own code)

This is often called **return-oriented programming**

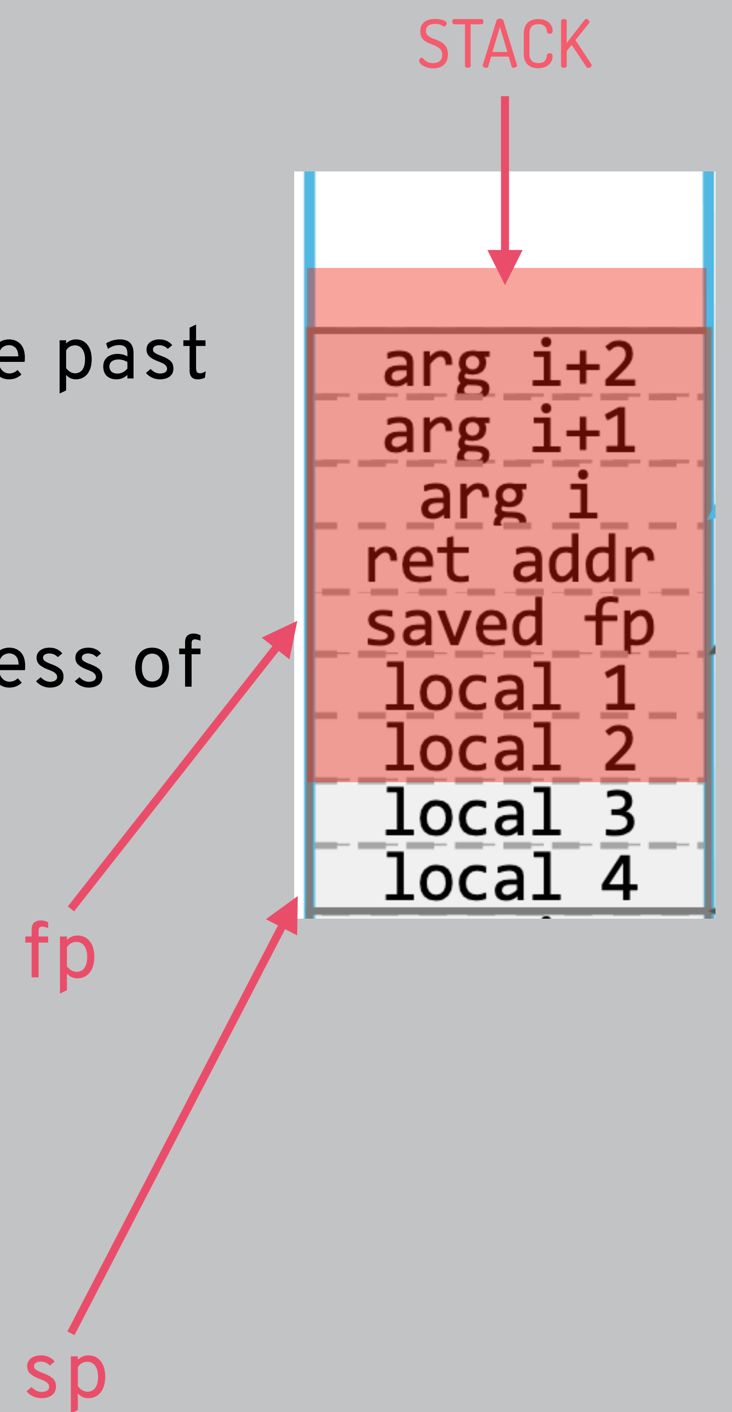


SMASHING THE STACK

What happens if you overwrite a malicious value past the bounds of a local variable?

Worst case: you can transfer control to an address of your choice

Now what?

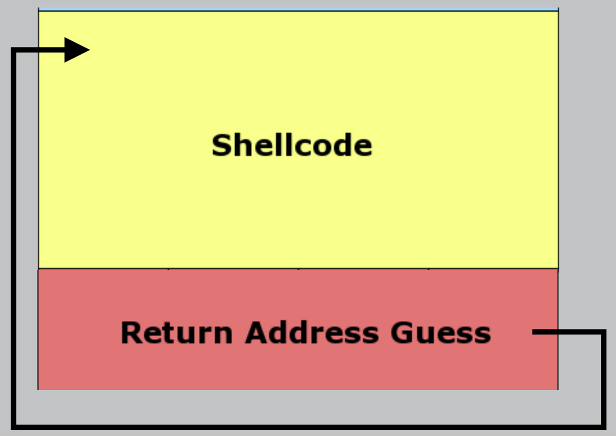


SHELLCODE

The best thing an attacker can do is launch the shell, because that allows them to execute arbitrary code (with higher privileges)

The payload is thus often called **shellcode**

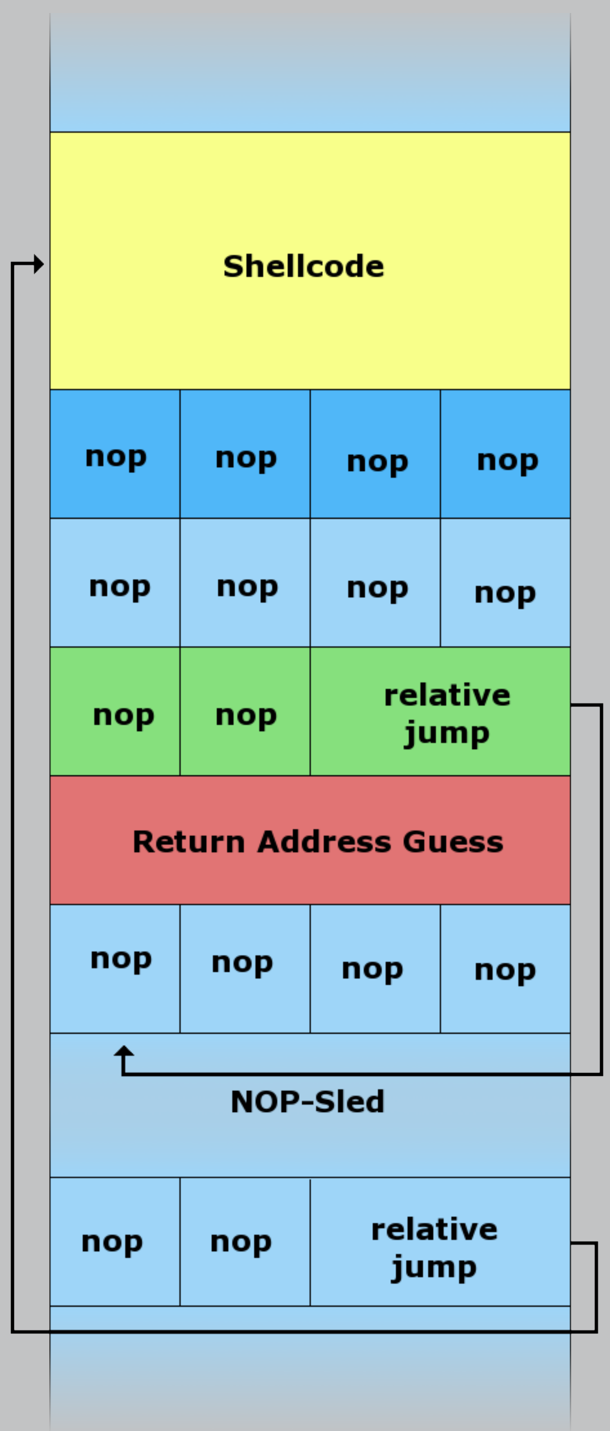
Attacker ensures shellcode is somewhere in the stack before overwriting return address but they might not know exactly where it is



NOP SLEDS

Instead, attacker can rely on the NOP ("no-op") instruction to create something called a **NOP sled** (or a NOP slide)

As long as the attacker's guess lands somewhere in this sequence of NOPs they can jump at the end to the start of the shellcode as desired



ADDRESSING BUFFER OVERFLOWS

Thinking like an attacker:

- Does the code check for bounds on memory access?
- Is the test invoked along every path leading up to the actual access (complete mediation)?
- Is the test correct? Can the test itself be attacked?

Investigate security aspects of tools, frameworks, libraries, APIs that you use and understand how to use them safely. The default way of doing something is often insecure!

Use `strlcpy` instead of `strcpy`, etc.

Lots of other techniques (stack canaries, ASLR, non-executable stack, etc.) that we won't cover in this module