CSE127, Computer Security

Application Security Defenses: Canaries, DEP, ASLR, and return-to-libc attacks

UC San Diego

Housekeeping

General course things to know

- PA1 was due
  - Grades are released! Mostly very good, though some non-instruction following
  - We'll be nice this time, but next time, we won't be so nice (expect huge penalties if you don't follow instructions)
- PA2 is released, due at **1/29** at 11:59
  - Note a two day preextension — this one is hard and my course staff suggested a little extra time might be appreciated
  - Discussion will give good hints and tips
  - Get started early (you can't really grind this one, you have to think a lot)

More course staff (so, more office hours)

Arul Mathur
Tutor
armathur@ucsd.edu

Previously on CSE127…

Other control flow vulnerabilities

- We talked about format string attacks, integer overflow attacks, and what you can do about them
  - Remember: any surface where your attacker can program your **weird machine** will lead to weird outcomes!

Today's lecture — Defenses + Advanced Attacks

Learning Objectives

- Understand how we **mitigate** buffer overflow attacks
- Understand the trade-offs of different mitigations
- Understand how these mitigations can be bypassed (sometimes)
  - Learn return-to-libc paradigm; return-oriented programming, why it's hard to handle and what to do about it
- We might overflow into next lecture

Mitigating Software Vulnerabilties

Defenses and Mitigation

Let's talk defender

- So far, we've been talking about attacks… but cybersecurity is about attacks **and** defenses
- Basic definitions:
  - What is a defense?
  - What is a mitigation?

Defenses and Mitigation

Let's talk defender

- So far, we've been talking about attacks… but cybersecurity is about attacks **and** defenses
- Basic definitions:
  - **What is a defense?**
  - What is a mitigation?

define: defense

- A **defense** (or countermeasure) is something put in place to *protect* unauthorized access or attack of a privileged resource
- Defenses usually come in two flavors
  - Proactive defenses (you put something in place to try and prevent broad classes of attacks)
  - Reactive defenses (you put something in place to try and prevent a **specific attack** you've seen recently)
- What are some defenses against car theft?

define: defense

- A **defense** (or countermeasure) is something put in place to *protect* unauthorized access or attack of a privileged resource
- Defenses usually come in two flavors
  - Proactive defenses (you put something in place to try and prevent broad classes of attacks)
  - Reactive defenses (you put something in place to try and prevent a **specific attack** you've seen recently)
- What are some defenses against car theft?
- Both are needed to secure your system. Why?

Defenses and Mitigation

Let's talk defender

- So far, we've been talking about attacks… but cybersecurity is about attacks **and** defenses
- Basic definitions:
  - What is a defense?
  - **What is a mitigation?**

define: mitigation

- A **mitigation** is a decision, action, or practice intended to reduce the level of risk associated with one or more threat events, threat scenarios, or vulnerabilities (from NIST SP 800-160 Vol. 2 Rev. 1)

- What's the difference between a defense and a mitigation?

define: mitigation

- A **mitigation** is a decision, action, or practice intended to reduce the level of risk associated with one or more threat events, threat scenarios, or vulnerabilities (from NIST SP 800-160 Vol. 2 Rev. 1)
- What's the difference between a defense and a mitigation?
  - Mitigations tend to be more reactive; they can also play at several levels beyond technical (technical, economic, policy)
- Mitigations will **not** stop all exploits, but they can make exploit development more difficult and costly
- Lots of folks throw around words like this interchangeably — worth paying attention to when one is used over another

In a perfect world…

- All developers and software engineers would write perfect code
  - But sadly, asking developers to **not insert vulnerabilities** doesn't really work
  - Even worse, not all vulnerabilities will be discovered prior to release
- So we build *defensive systems* and *mitigations* to prevent threats from getting out of hand

How to think about defenses…

- Even when we're learning about defenses, you should be thinking like an attacker
  - Challenge assumptions
  - Think about how you can circumvent each "solution"
- Recall: Security is a cat-and-mouse game
  - Developers introduce new features. Attackers exploit features. Developers defend those features *and* build a better system. Attackers adapt to the countermeasures… (and so it goes…)
  - …and both sides are employed indefinitely…

OK, back to software vulnerabilities…

- Think back to the attacks we've learned about so far: stack smashing, format string vulnerabilities, integer overflow vulnerabilities
- What kinds of outcomes do we want to prevent?

OK, back to software vulnerabilities…

- Think back to the attacks we've learned about so far: stack smashing, format string vulnerabilities, integer overflow vulnerabilities
- What kinds of outcomes do we want to prevent?
  - Overwriting the return address
  - Hijacking the control flow in any way
  - Unauthorized read / write of process memory

Today's defenses

- Try to detect overwriting control data (e.g., return address)
  - **Stack canaries / cookies**
- Try to make it difficult to redirect control flow to attacker code
  - **Memory protection (Data-execution prevention, W^X)**
  - **Address Space Layout Randomization (ASLR)**

Stack Canaries

Stack Canaries

- Basic idea to prevent buffer overflow attacks
- **Detect** overwriting of the return address
  - Place a special value (called a **canary** or **cookie**) between local variables and the saved frame pointer
  - Check that value before popping saved frame pointer and return address from the stack
  - Exception is triggered if the value is not correct!

Caller / Callee with canaries during `call`

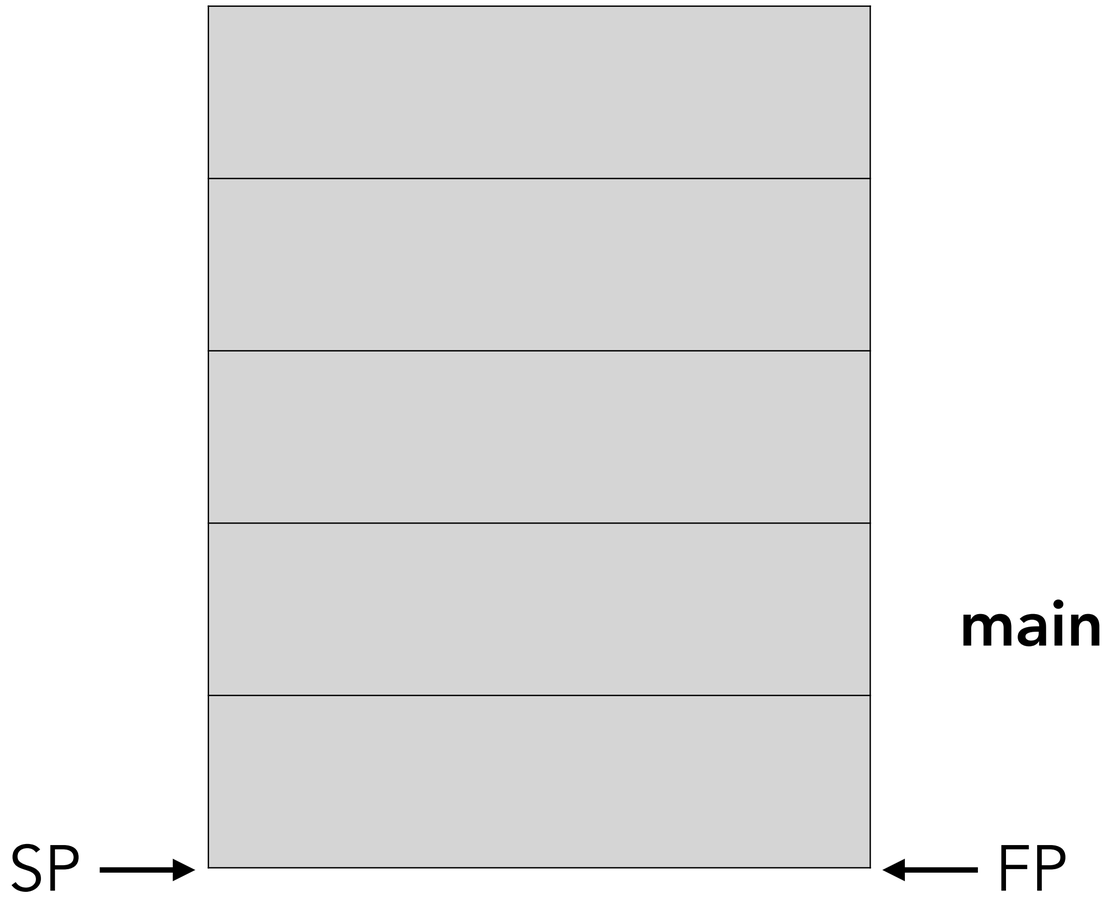
- What are the responsibilities of the caller?
  - Pass arguments, save return address, call new function
- What is the responsibility of the callee?
  - Save old FP, set FP = SP, allocate stack space for…
    - **Canary value, which is pushed onto the stack**
    - Local storage, which is pushed after canary

Caller / Callee with canaries during `ret`

- What does the callee do when returning?
  - Pop local storage
  - **Pop canary, check if canary matches expected, known value, otherwise throw exception**
  - Set SP = FP
  - Pop frame pointer
  - Pop return address and **ret**
- What does the caller do when returning?
  - Pop arguments and continue

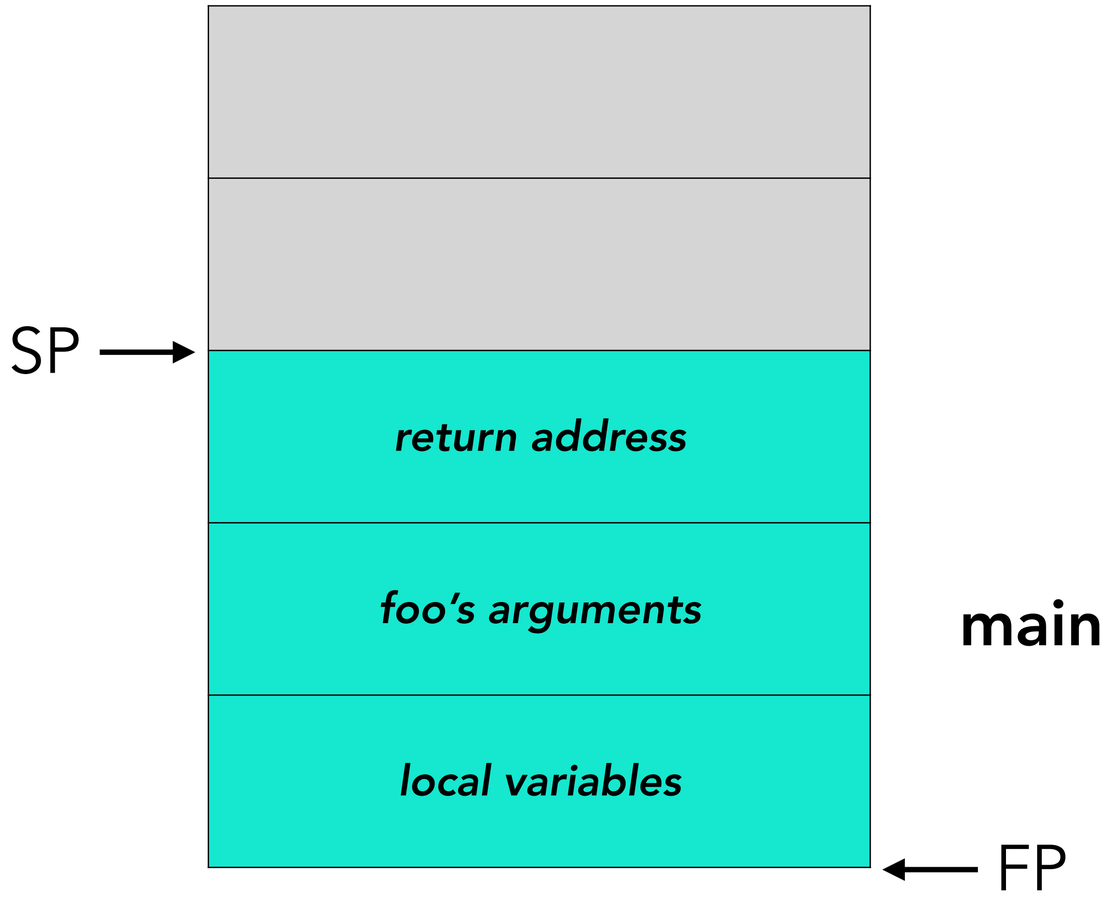
overflow.c – canaries

```
void foo (char* inp) {

    char buf1[4];
    strcpy(buf1, inp);

}



int main() {

    foo(getFromUser());

}
```



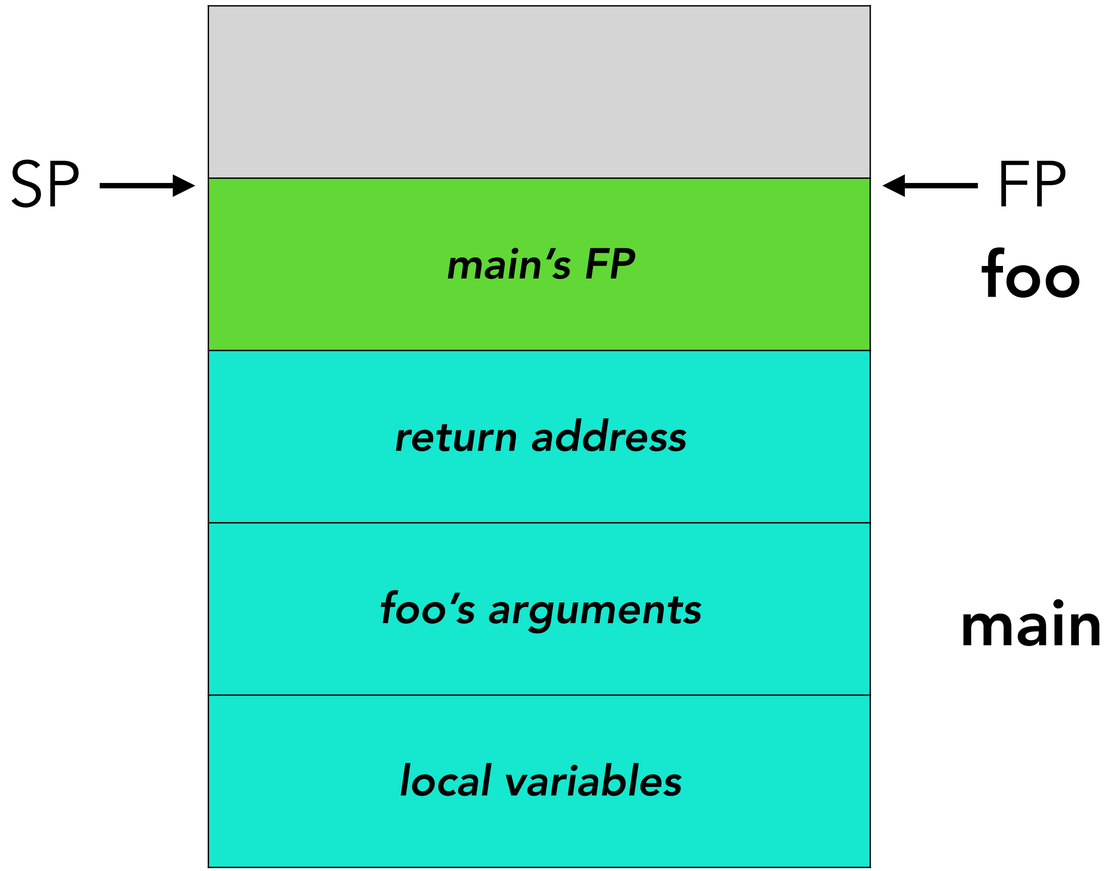
overflow.c – canaries

```
void foo (char* inp) {

    char buf1[4];
    strcpy(buf1, inp);

}


int main() {

    foo(getFromUser());

}
```



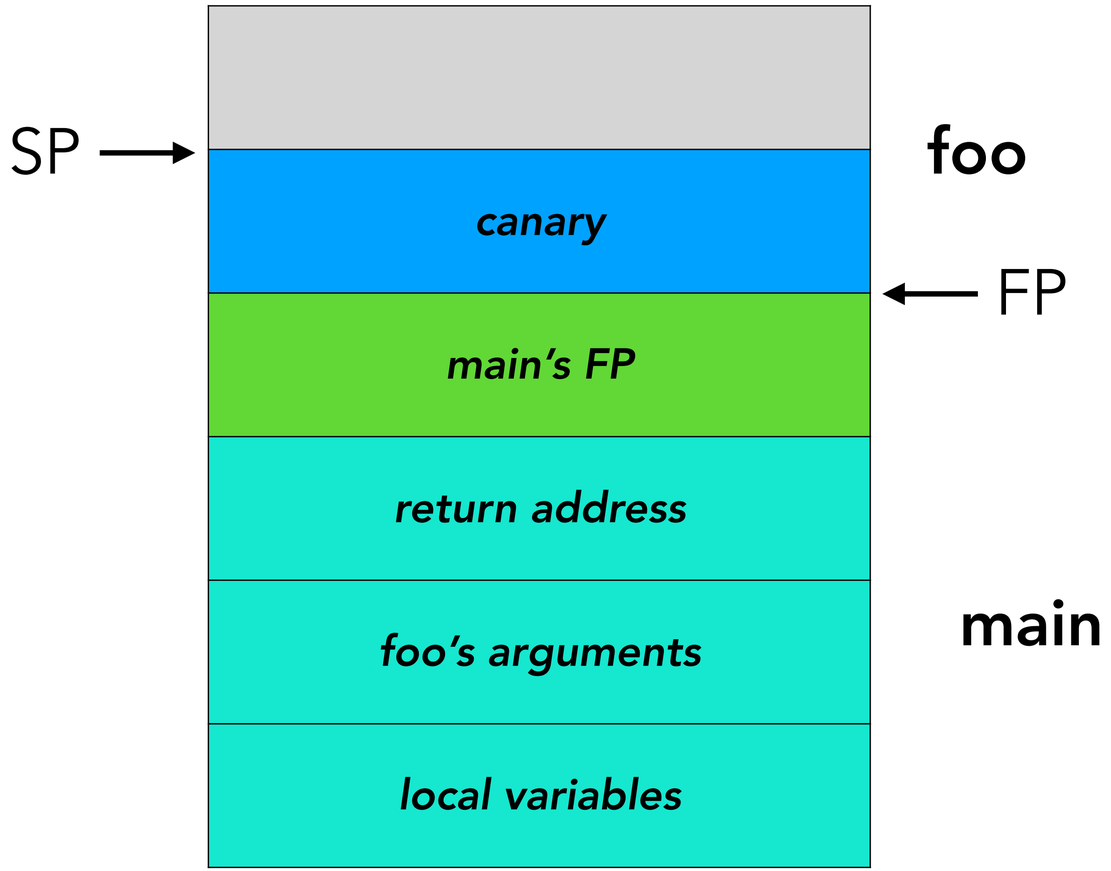
overflow.c – canaries

```
void foo (char* inp) {

    char buf1[4];
    strcpy(buf1, inp);

}


int main() {

    foo(getFromUser());

}
```



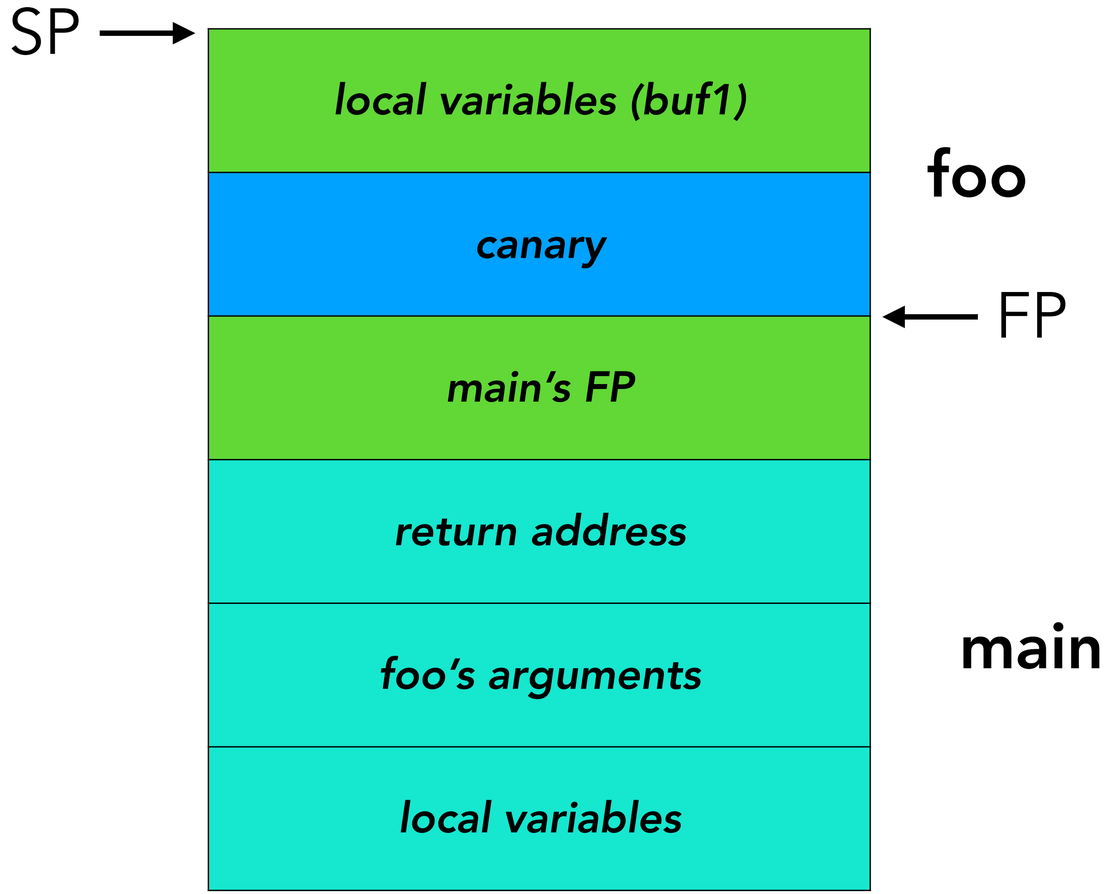
overflow.c – canaries

```
void foo (char* inp) {

    char buf1[4];
    strcpy(buf1, inp);

}


int main() {

    foo(getFromUser());

}
```



overflow.c – canaries

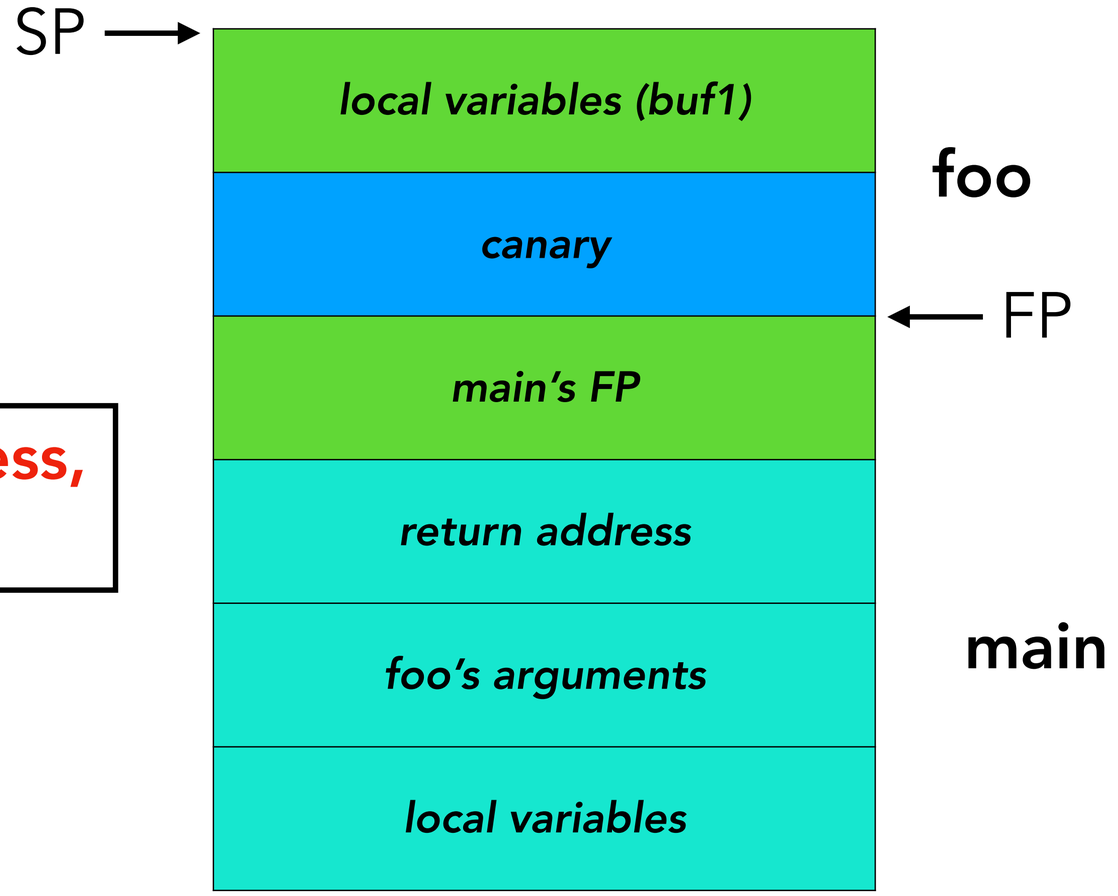
```
void foo (char* inp) {

    char buf1[4];
    strcpy(buf1, inp);

}


int main() {

    foo(getFromUser());

}
```



overflow.c – canaries

```
void foo (char* inp) {

    char buf1[4];
    strcpy(buf1, inp);

}

int main() {

    foo(getFromUser());

}
```

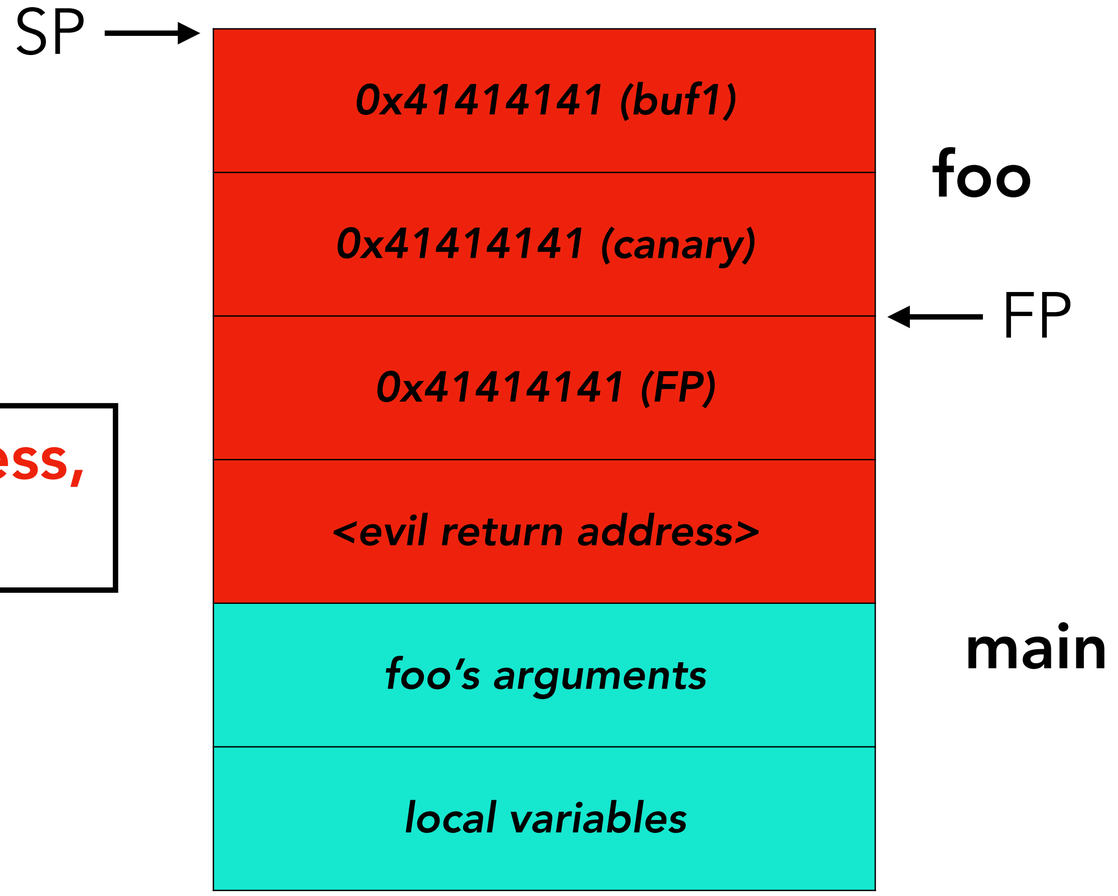
returns 0x41 until return address, and overwrites return



overflow.c – canaries

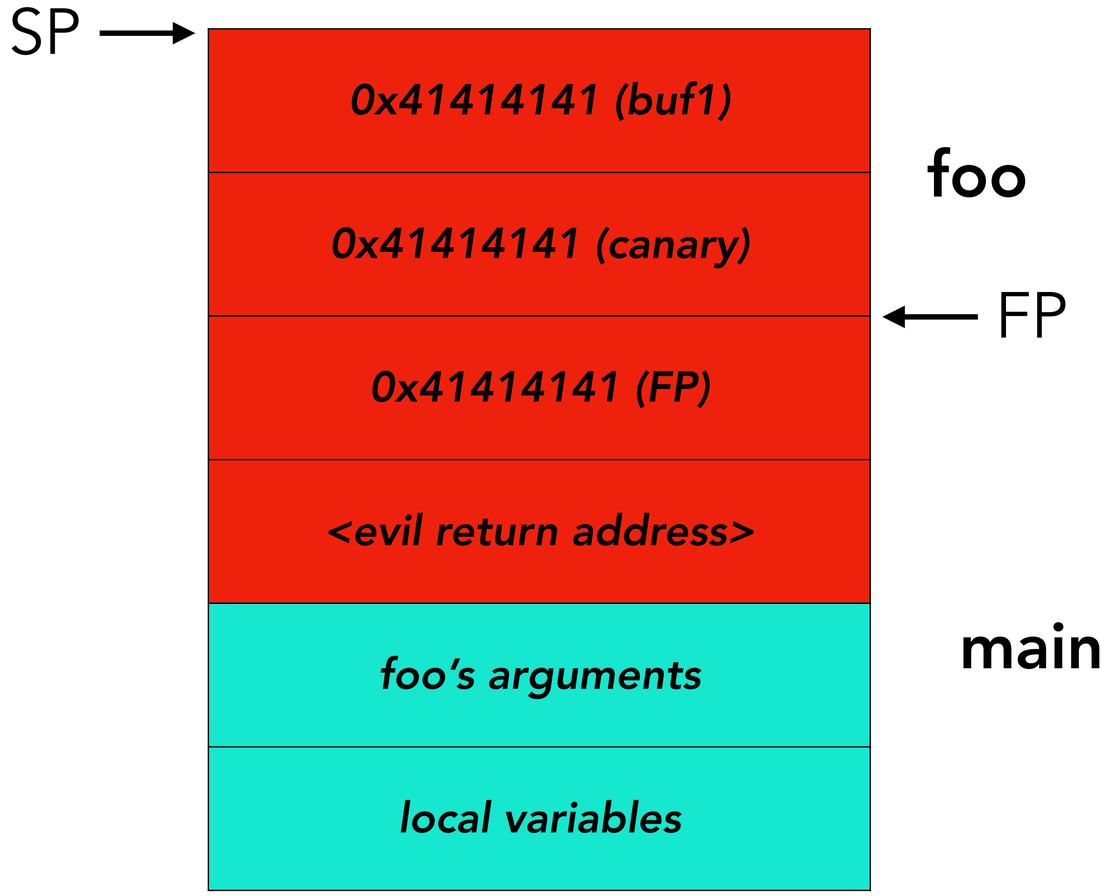
```
void foo (char* inp) {

    char buf1[4];
    strcpy(buf1, inp);

}

int main() {

    foo(getFromUser());

}
```

returns 0x41 until return address, and overwrites return



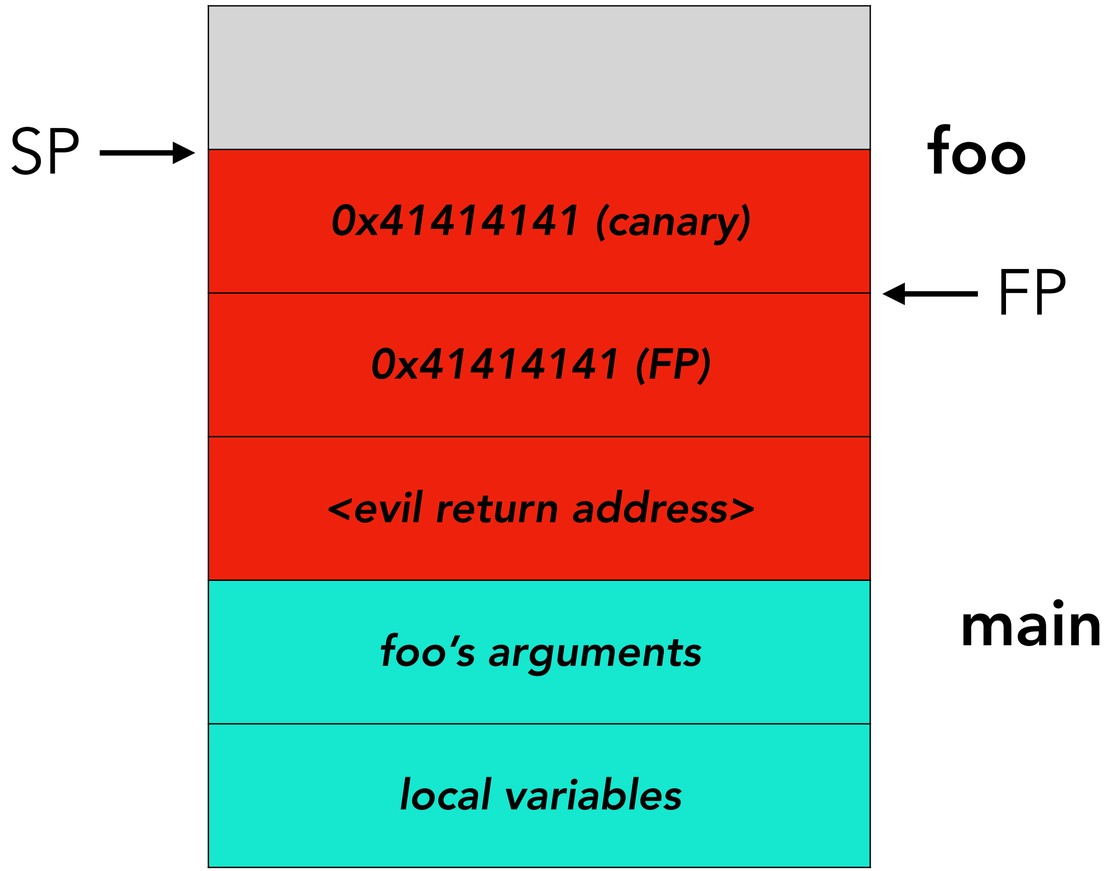
overflow.c – canaries returning

```
void foo (char* inp) {

    char buf1[4];
    strcpy(buf1, inp);

}


int main() {

    foo(getFromUser());

}
```



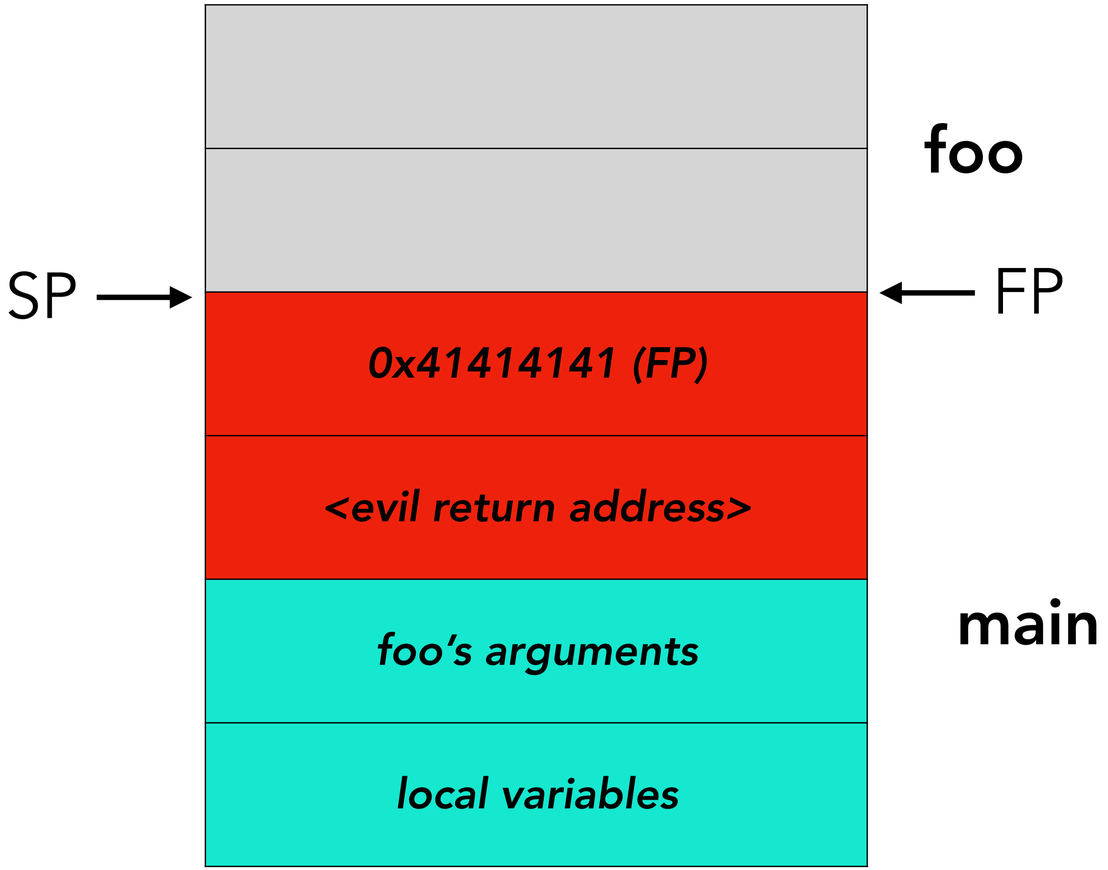
overflow.c – canaries returning

```
void foo (char* inp) {

    char buf1[4];
    strcpy(buf1, inp);

}


int main() {

    foo(getFromUser());

}
```



overflow.c – canaries returning

```
pop canary value


if (canary != <expected_value>){
      goto canary_fail;
}
*STACK SMASHING DETECTED!*
```



Why is called a canary…?

Miner's canary [edit]

Mice were used as sentinel species for use in detecting carbon monoxide in British coal mining from around 1896,[143] after the idea had been suggested in 1895 by John Scott Haldane.[144] Toxic gases such as carbon monoxide, asphyxiant gases such as carbon dioxide and explosive gases like methane[145] in the mine would affect small warm-blooded animals before affecting the miners, since their respiratory exchange is more rapid than in humans. A mouse will be affected by carbon monoxide within a few minutes, while a human will have an interval of 20 times as long.[146] Later, canaries were found to be more sensitive and a more effective indicator as they showed more visible signs of distress. Their use in mining is documented from around 1900.[147] The birds were sometimes kept in carriers which had small oxygen bottles attached to revive them.[148][149]

Mining foreman R. Thornburg shows a small cage with a canary used for testing carbon monoxide gas in 1928.

A canary in a coal mine: "An early warning sign of impending danger."

Good values for canaries

- What are some good values for canaries?

Good values for canaries

- What are some good values for canaries?
  - Canaries tend to include **null bytes**; force termination of unsafe functions (called *terminator canaries*)
    - e.g., **`0x000A0DFF`**
- What's the problem with using a fixed canary value?

Good values for canaries

- What are some good values for canaries?
  - Canaries tend to include **null bytes**; force termination of unsafe functions (called *terminator canaries*)
    - e.g., **`0x000A0DFF`**
- What's the problem with using a fixed canary value?
  - If attacker can guess a canary… it's still game over
- Modern systems try to make canaries **hard to guess**
  - ***Random canaries*** are great options so long as they remain secret

When do we add canaries?

- At compile time! Now a common feature of all popular compilers…
- `gcc` flags to add canaries
  - `-fstack-protector`
    - Functions with character buffers >= `ssp-buffer-size` (8 bytes default)
    - Functions with variable sized alloca() (dynamic stack allocation)
  - `-fstack-protector-strong`
    - Functions with local arrays of any size/type
    - Functions that have references to local stack variables
  - `-fstack-protector-all`
    - All functions!

Canary Tradeoffs

- **Pros**
  - Simple to implement and deploy
  - Can implement mitigation as a compiler pass, so **no input from developers is needed**
- **Cons**
  - Performance
  - Stack protection has a marginal cost

No stack protection

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $-559038737, -12(%ebp)
        subl    $8, %esp
        pushl   16(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        leave
        ret
```

`-fstack-protector-strong`

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    16(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -28(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L3
        call    __stack_chk_fail
.L3:
        leave
        ret
```

Limitations of Canaries

- What is the fundamental assumption canaries make about attacks it tries to protect against?

Limitations of Canaries

- What is the fundamental assumption canaries make about attacks it tries to protect against?
  - Impossible to overwrite return address without corrupting the canary.
- Is this always true?

Limitations of Canaries

- What is the fundamental assumption canaries make about attacks it tries to protect against?
  - Impossible to overwrite return address without corrupting the canary.
- Is this always true?
  - Is it possible to overwrite the canary value with a valid one, even if we don't know the value a priori?
  - What about non-protected data?
  - What about arbitrary writes (e.g., via printf?)

Limitations of Canaries

- What is the fundamental assumption canaries make about attacks it tries to protect against?
  - Impossible to overwrite return address without corrupting the canary.
- Is this always true?
  - **Is it possible to overwrite the canary value with a valid one, even if we don't know the value a priori?**
  - What about non-protected data?
  - What about arbitrary writes (e.g., via printf?)

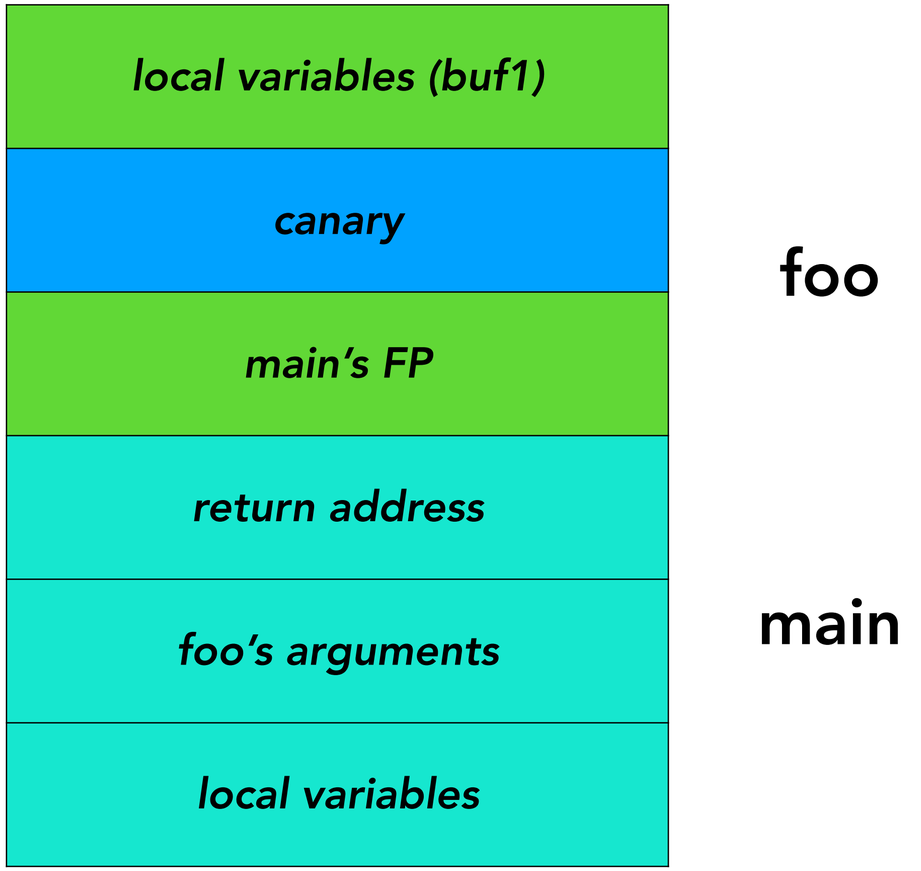
Learning Canaries through Brute Force

- If underlying software *leaks* information about the canary, you can use that to your advantage
- Canary value is selected *at process creation*, stays the same throughout the runtime of the process
- Consider long running processes… e.g., **web servers**
  - Main server process:
    - Establishes listening socket on the network
    - `fork` workers to handle requests; if any workers die, fork a new one
  - Worker process:
    - Accept connection on listening socket & process request

Long running processes can leak canary values…

- Forked process has same memory layout and contents as parent… **including canary values**

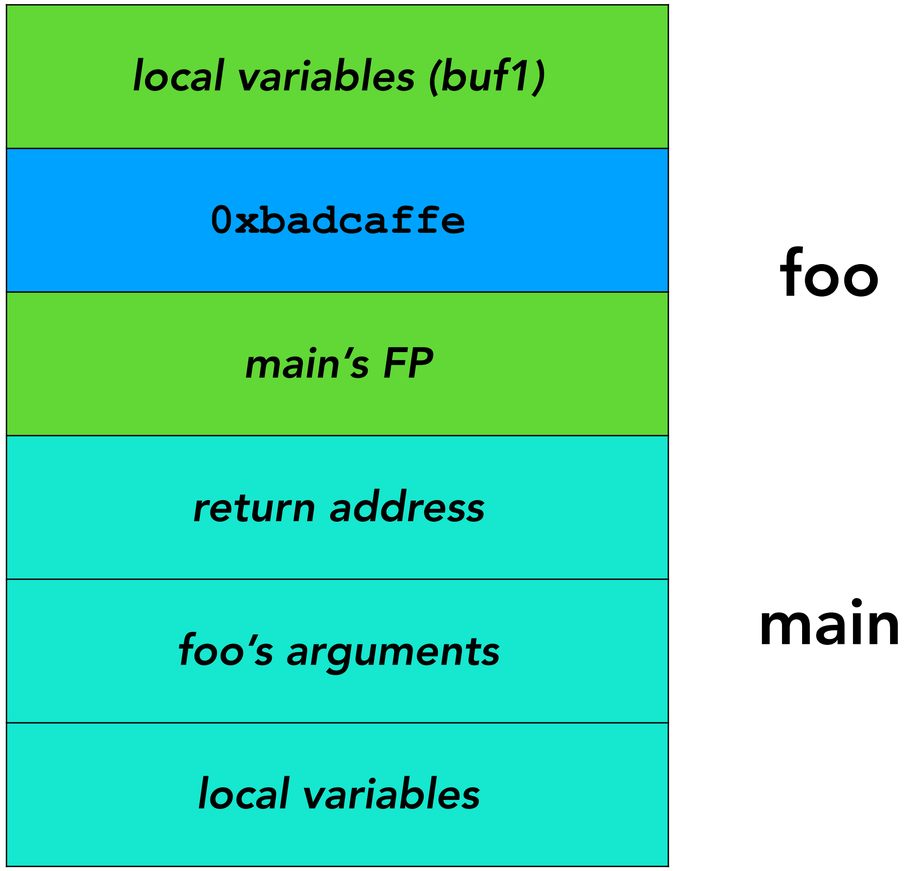
- The “fork on crash” lets us try different canary values at our wish…



Long running processes can leak canary values…

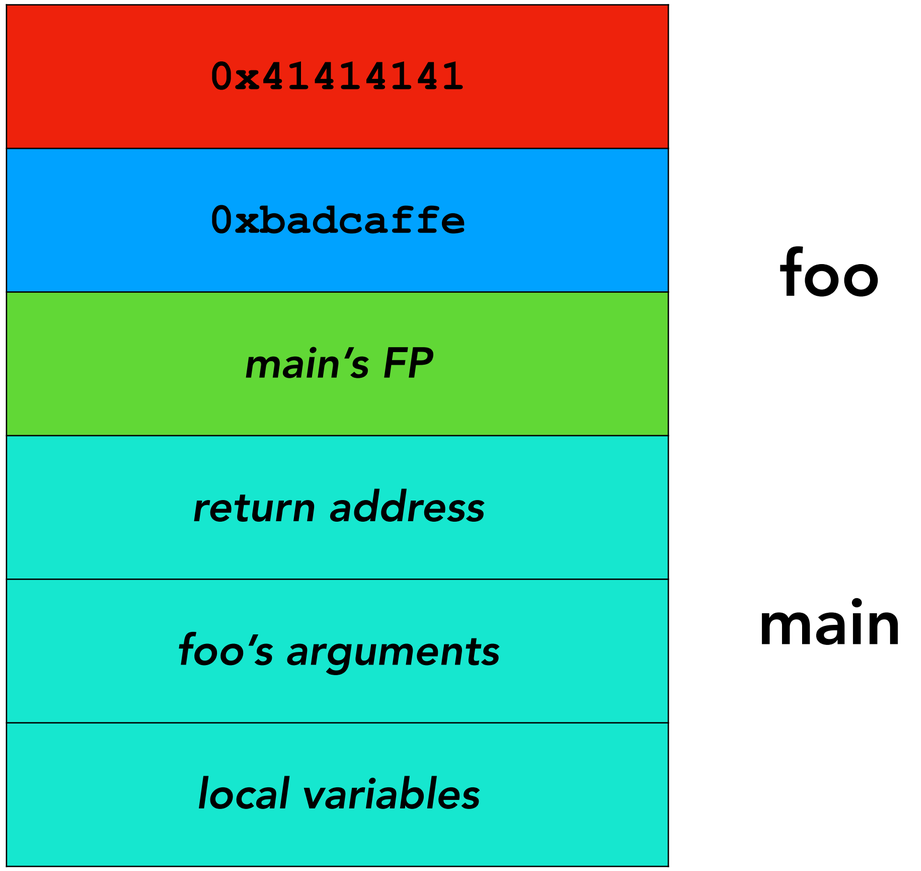
- Forked process has same memory layout and contents as parent… **including canary values**

- The “fork on crash” lets us try different canary values at our wish…



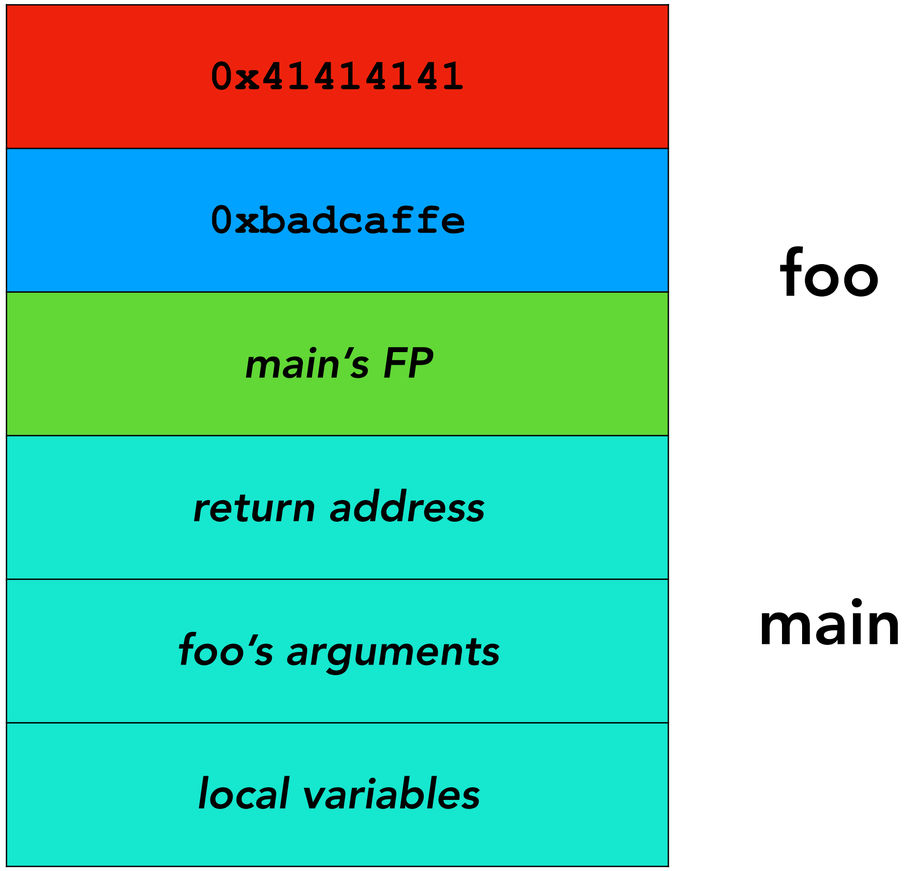
Long running processes can leak canary values…

- Forked process has same memory layout and contents as parent… **including canary values**
- The “fork on crash” lets us try different canary values at our wish…
- We know size of buffer, so fill it with nonsense



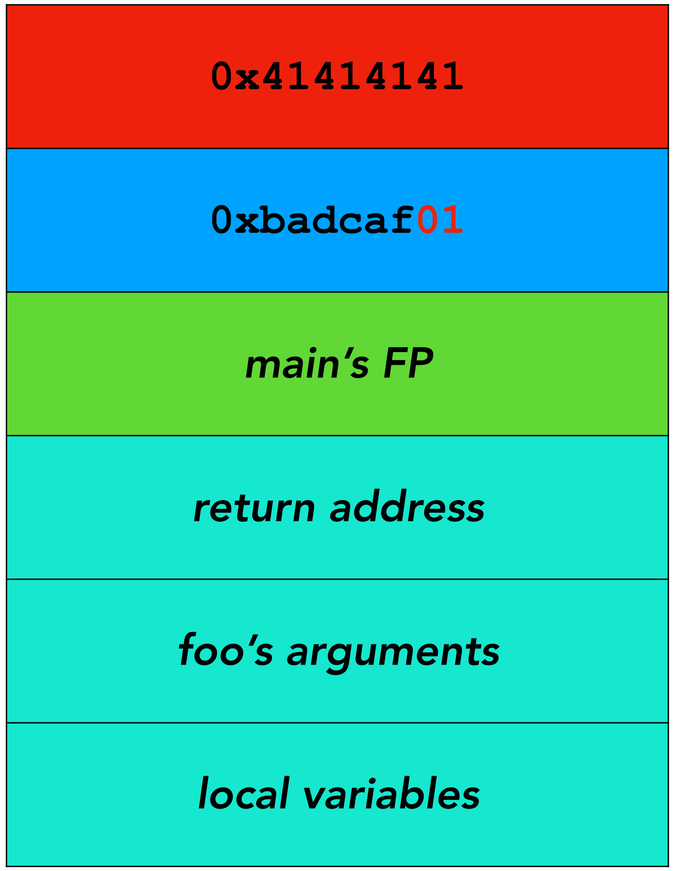
Long running processes can leak canary values…

- Forked process has same memory layout and contents as parent… **including canary values**
- The “fork on crash” lets us try different canary values at our wish…
- We know size of buffer, so fill it with nonsense
- Try modifying **one byte at a time**, see if the forked process crashes



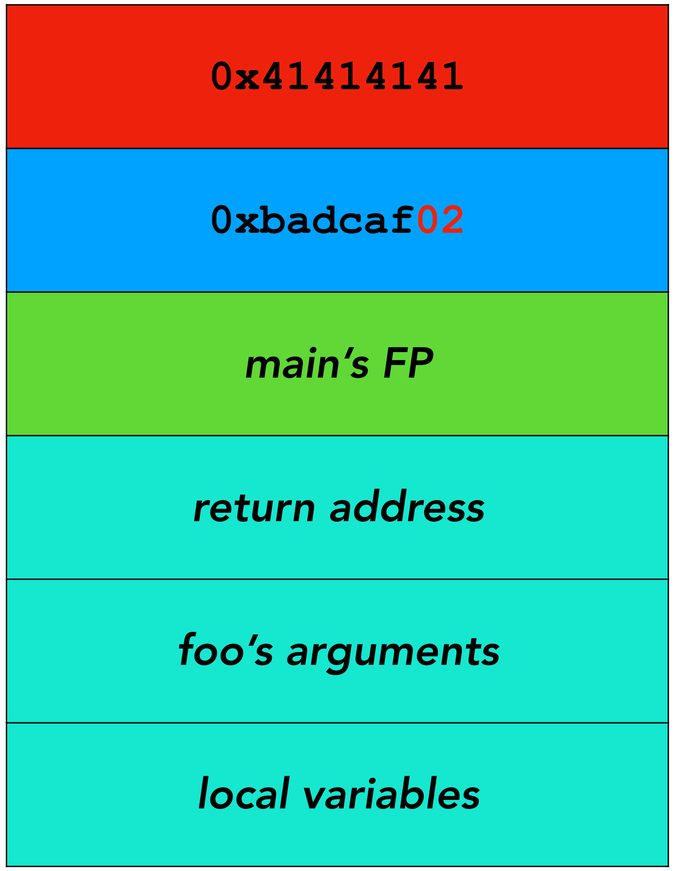
Long running processes can leak canary values…

- Forked process has same memory layout and contents as parent… **including canary values**
- The “fork on crash” lets us try different canary values at our wish…
- We know size of buffer, so fill it with nonsense
- Try modifying **one byte at a time**, see if the forked process crashes



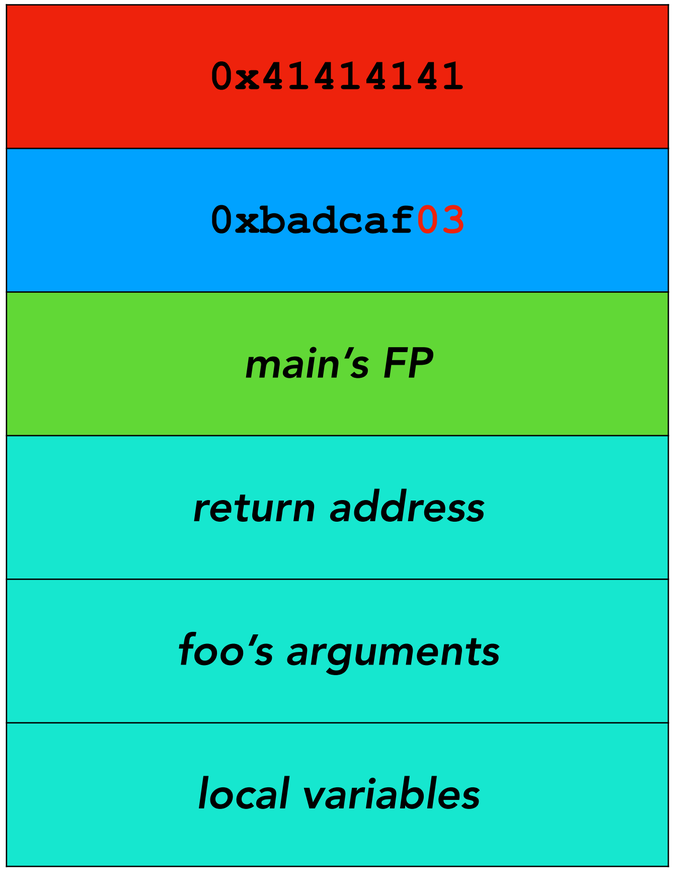
Long running processes can leak canary values…

- Forked process has same memory layout and contents as parent… **including canary values**
- The “fork on crash” lets us try different canary values at our wish…
- We know size of buffer, so fill it with nonsense
- Try modifying **one byte at a time**, see if the forked process crashes



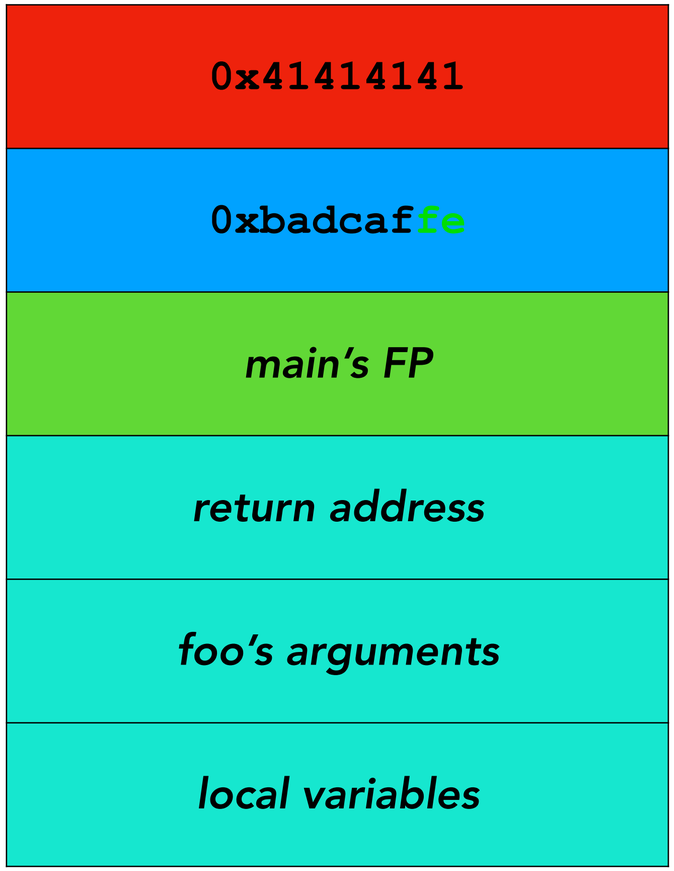
Long running processes can leak canary values…

- Forked process has same memory layout and contents as parent… **including canary values**
- The “fork on crash” lets us try different canary values at our wish…
- We know size of buffer, so fill it with nonsense
- Try modifying **one byte at a time**, see if the forked process crashes



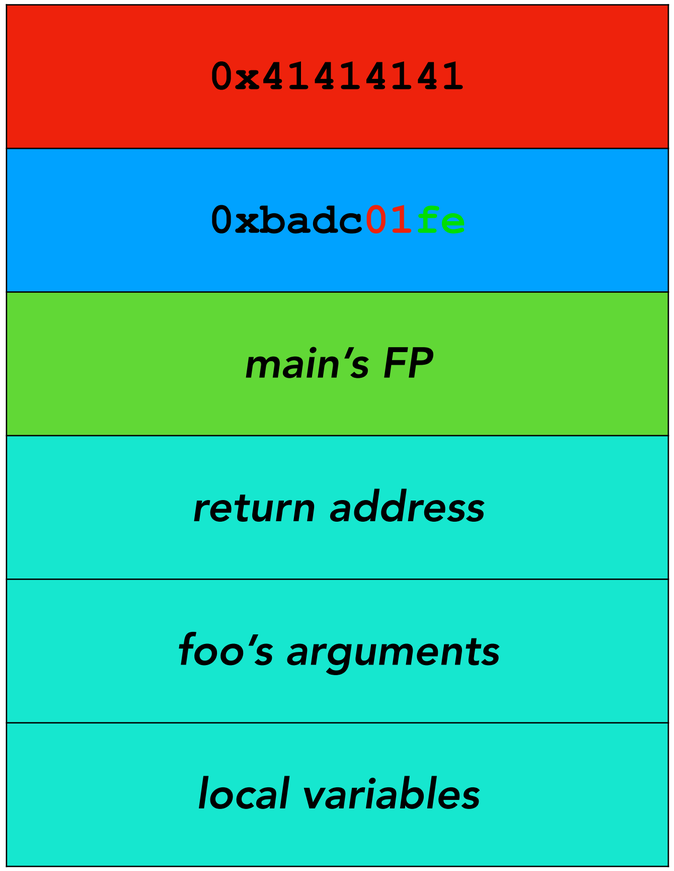
Long running processes can leak canary values…

- Forked process has same memory layout and contents as parent… **including canary values**
- The “fork on crash” lets us try different canary values at our wish…
- We know size of buffer, so fill it with nonsense
- Try modifying **one byte at a time**, see if the forked process crashes



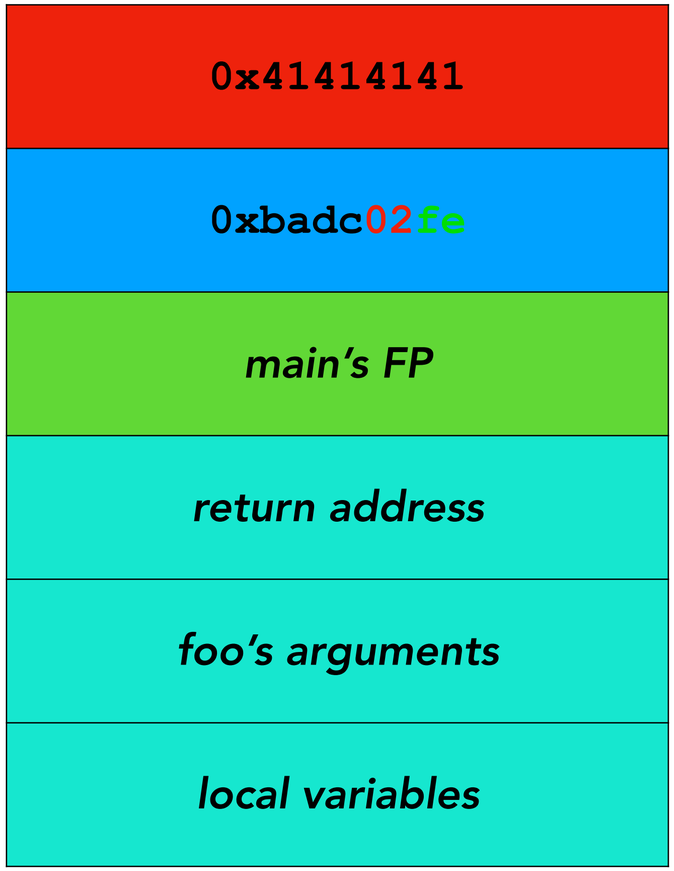
Long running processes can leak canary values…

- Forked process has same memory layout and contents as parent… **including canary values**
- The “fork on crash” lets us try different canary values at our wish…
- We know size of buffer, so fill it with nonsense
- Try modifying **one byte at a time**, see if the forked process crashes
- And so on…



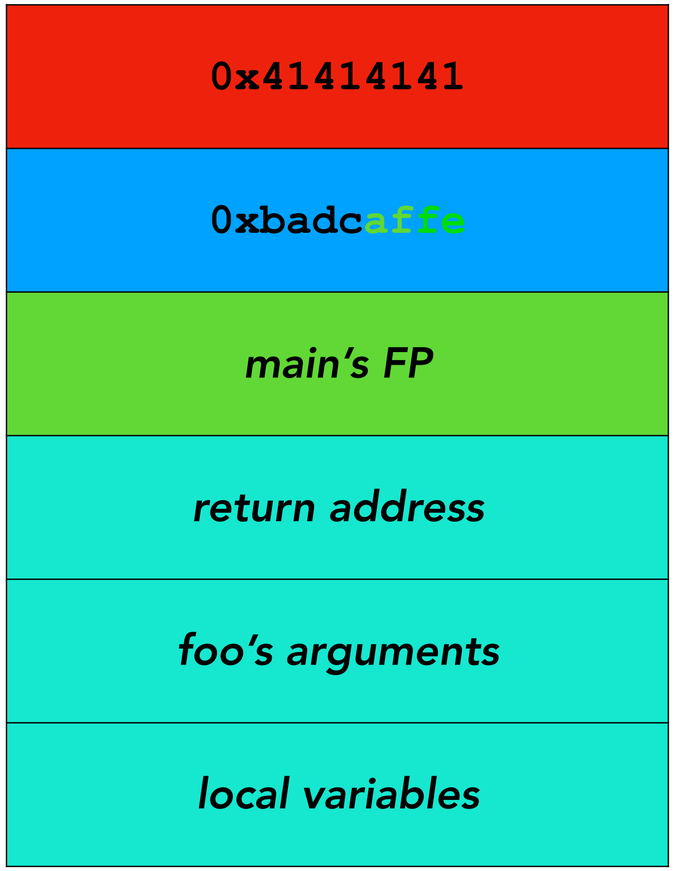
Long running processes can leak canary values…

- Forked process has same memory layout and contents as parent… **including canary values**
- The “fork on crash” lets us try different canary values at our wish…
- We know size of buffer, so fill it with nonsense
- Try modifying **one byte at a time**, see if the forked process crashes
- And so on…



Long running processes can leak canary values…

- Forked process has same memory layout and contents as parent… **including canary values**
- The “fork on crash” lets us try different canary values at our wish…
- We know size of buffer, so fill it with nonsense
- Try modifying **one byte at a time**, see if the forked process crashes
- And so on…



Limitations of Canaries

- What is the fundamental assumption canaries make about attacks it tries to protect against?
  - Impossible to overwrite return address without corrupting the canary.
- Is this always true?
  - Is it possible to overwrite the canary value with a valid one, even if we don't know the value a priori?
  - **What about non-protected data?**
  - **What about arbitrary writes (e.g., via printf?)**

totally_secure.c

```
0x08049b95  void evil() {
                printf("evil\n");
                exit(0);
            }

            int i = 42;

            void func(char *str) {
                int *ptr = &i;
                int val = 44;
                char buf[4];
                strcpy(buf, str);
                *ptr = val;
            }

            int main(int argc, char**argv) {
                func(argv[1]);
                return 0;
            }
```

totally_secure.c

```
0x08049b95  void evil() {
                printf("evil\n");
                exit(0);
            }

            int i = 42;

            void func(char *str) {
                int *ptr = &i;
                int val = 44;
                char buf[4];
                strcpy(buf, str);
                *ptr = val;
            }

            int main(int argc, char**argv) {
                func(argv[1]);
                return 0;
            }
```

Initializes a pointer to i
Creates another val = 44

totally_secure.c

```
0x08049b95  void evil() {
                printf("evil\n");
                exit(0);
            }

            int i = 42;

            void func(char *str) {
                int *ptr = &i;
                int val = 44;
                char buf[4];
                strcpy(buf, str);
                *ptr = val;
            }

            int main(int argc, char**argv) {
                func(argv[1]);
                return 0;
            }
```

makes a vulnerable call to `strcpy`

What can the attacker overwrite?

totally_secure.c

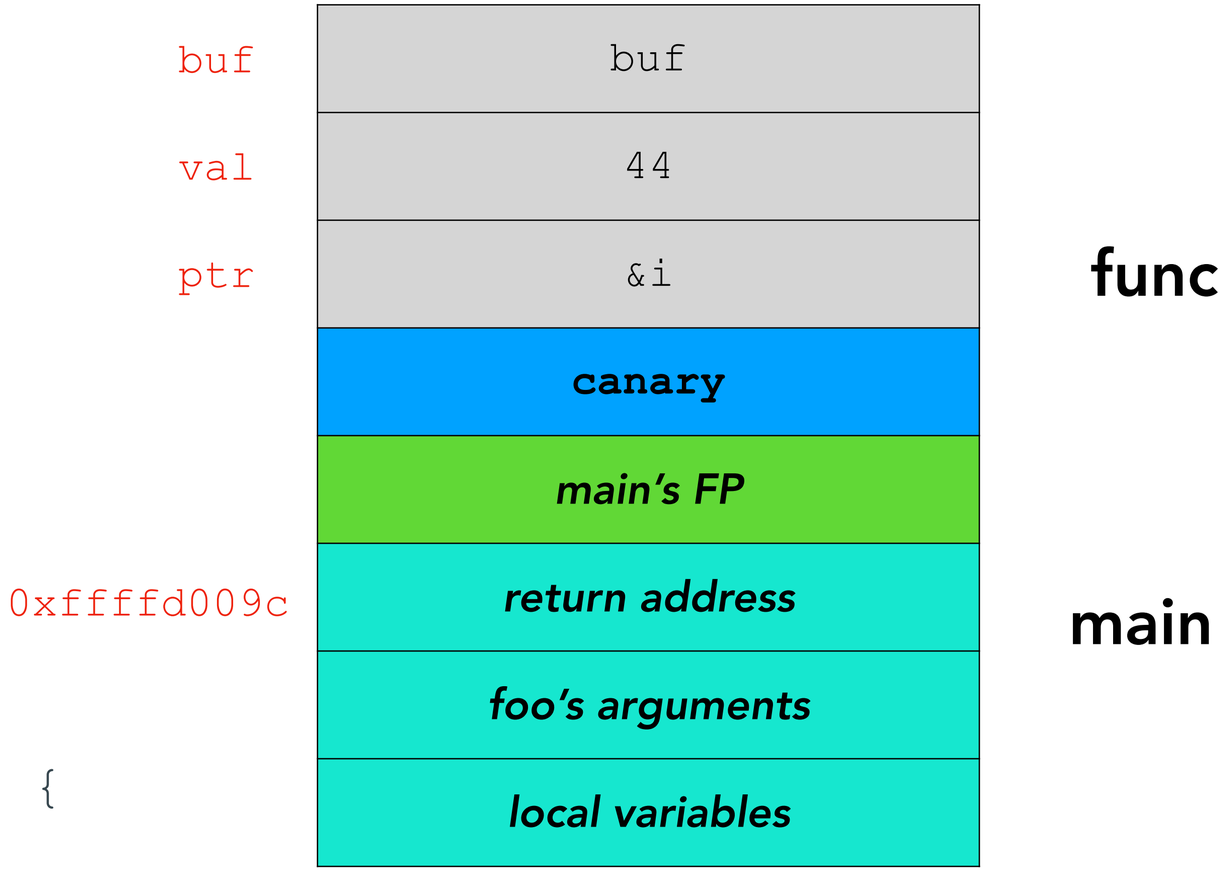
```
0x08049b95  void evil() {
                printf("evil\n");
                exit(0);
            }

            int i = 42;

            void func(char *str) {
                int *ptr = &i;
                int val = 44;
                char buf[4];
                strcpy(buf, str);
                *ptr = val;
            }

            int main(int argc, char**argv) {
                func(argv[1]);
                return 0;
            }
```

dereferences ptr, sets equal to val

both are in attacker's control.

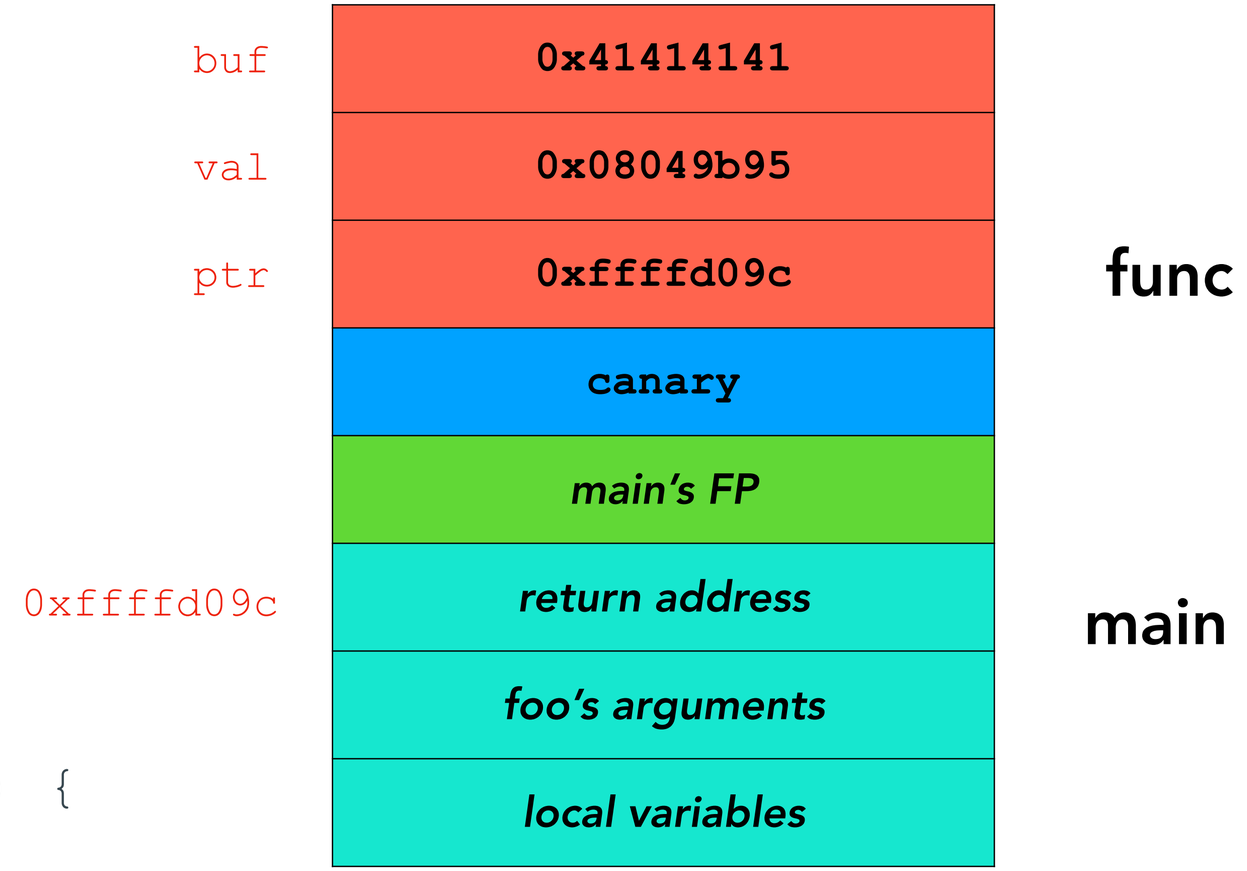
totally_secure.c

```
0x08049b95  void evil() {
                printf("evil\n");
                exit(0);
            }

            int i = 42;

            void func(char *str) {
                int *ptr = &i;
                int val = 44;
                char buf[4];
                strcpy(buf, str);
                *ptr = val;
            }

            int main(int argc, char**argv) {
                func(argv[1]);
                return 0;
            }
```



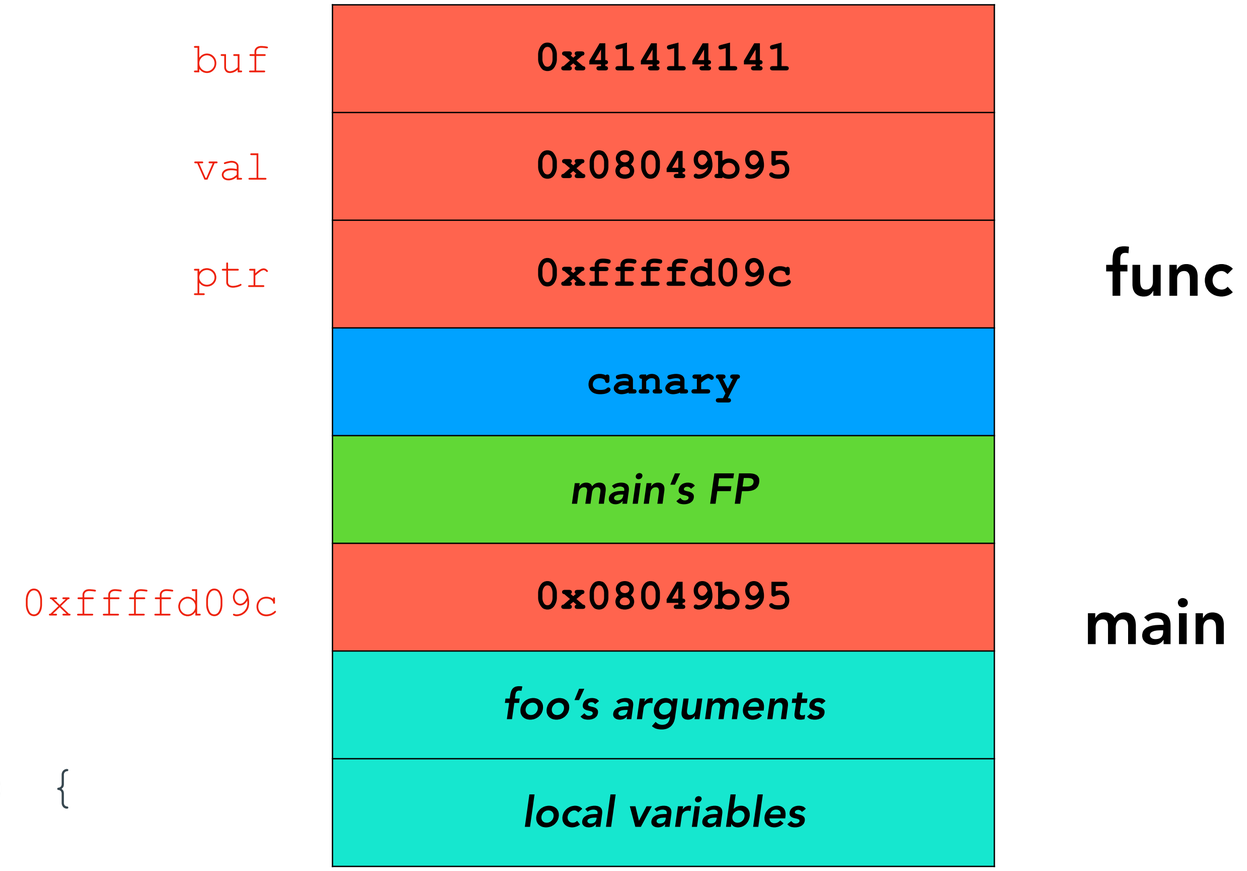
totally_secure.c

```
0x08049b95  void evil() {
                printf("evil\n");
                exit(0);
            }

            int i = 42;

            void func(char *str) {
                int *ptr = &i;
                int val = 44;
                char buf[4];
                strcpy(buf, str);
                *ptr = val;
            }

            int main(int argc, char**argv) {
                func(argv[1]);
                return 0;
            }
```



totally_secure.c

```
0x08049b95  void evil() {
                printf("evil\n");
                exit(0);
            }

            int i = 42;

            void func(char *str) {
                int *ptr = &i;
                int val = 44;
                char buf[4];
                strcpy(buf, str);
                *ptr = val;
            }

            int main(int argc, char**argv) {
                func(argv[1]);
                return 0;
            }
```



totally_secure.c

```
0x08049b95  void evil() {
                printf("evil\n");
                exit(0);
            }

            int i = 42;

            void func(char *str) {
                char buf[12];
                int val = 0xdeadbeef;
                int *ptr;

                strcpy(buf, str);
                *ptr = val;
            }

            int main(int argc, char**argv) {
                func(argv[1]);
                return 0;
            }
```

buf

val

ptr

0xffffd09c

Attacker overwrote return address without touching canary value!

“Pointer subterfuge” attack

Canaries are widely used today

- Stack canaries do not protect from non-sequential overwrites
- Stack canaries do not *prevent* the overwrite, they only attempt to detect when it happens
  - “Reactive mitigation” —> attack has happened; how do we recover from damage?
- **In spite of limitations, stack canaries still offer significant value for the performance tradeoff**
  - Considered essential mitigation on modern systems; *85% desktop binaries ship with stack canaries* (https://www.ndss-symposium.org/wp-content/uploads/2022-31-paper.pdf)

Data Execution Prevention

Data Execution Prevention (DEP)

- Goal: prevent execution of *shellcode on the stack*
- Modern processes can mark virtual memory pages with permission bits: Read, Write, and/or eXecute (RWX)
  - Idea: Mark stack pages as "nonexecutable" — any attempts to execute from stack will trigger memory access violation
- Can extend beyond stack too…
  - Make **all pages** either writable or executable, but not both
  - Stack + heap are writeable, not executable
  - Code is executable, but not writeable
  - Known as **W^X (Write XOR eXecute)**

DEP Tradeoffs

- **Pros**
  - No changes to application software (happens in runtime)
  - Little / no performance impact
- **Cons**
  - Requires hardware support (MPU, MMU, or SMMU)…
    - Might not be possible in low level, cheap embedded devices
  - Doesn’t work automatically for some programs…
    - E.g., self-modifying code, just-in-time compilation, etc.

Busting DEP

- What is the core assumption the DEP relies on?

Busting DEP

- What is the core assumption the DEP relies on?
  - Attacker wants to transfer control to something on the stack. Is this always true?

Busting DEP

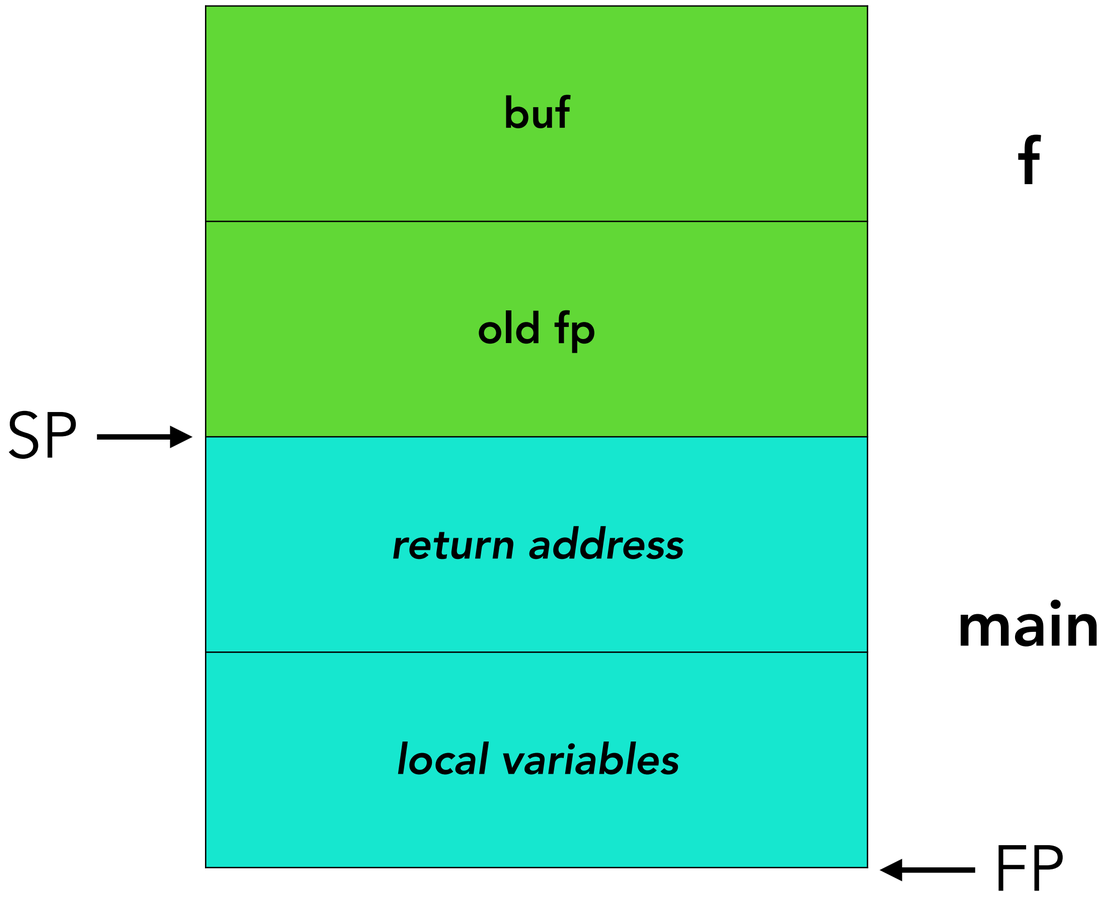
- What is the core assumption the DEP relies on?
  - Attacker wants to transfer control to something on the stack. Is this always true?
- No!
  - Is there any useful executable code you can repurpose…?
  - Recall… **parts of `libc` is loaded in process memory**
  - Remember `execve()`? If we can return control there, we can start any process…
    - Called a **return-to-libc** attack

Return-to-libc attacks

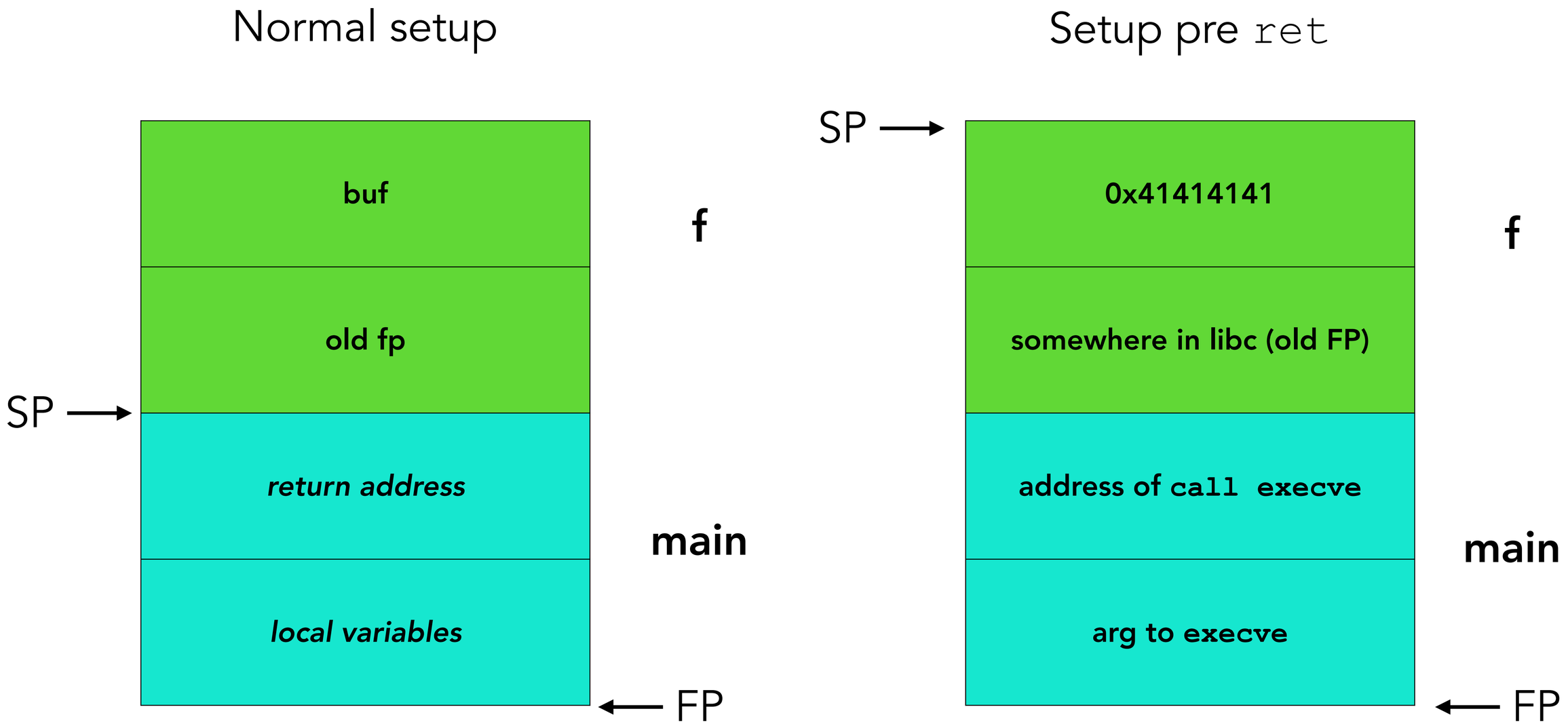
- Fundamental idea: **return control to a call to an executable function, usually in libc**
  - But… the stack needs to be set up the right way to work properly
- Core concept: Trick runtime into thinking the **`ret`** functions like a **`call`**

Return-to-libc attacks — function setup

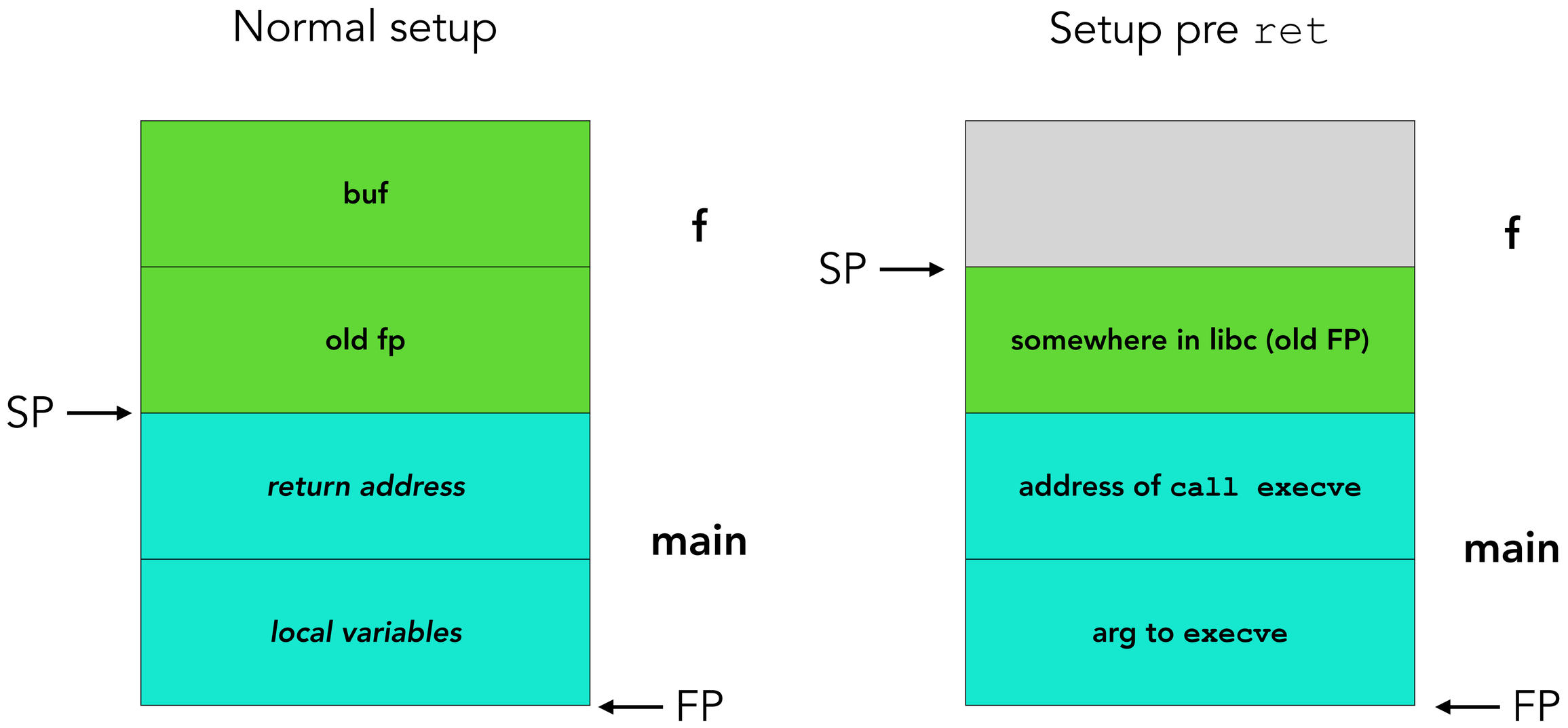
Normal setup



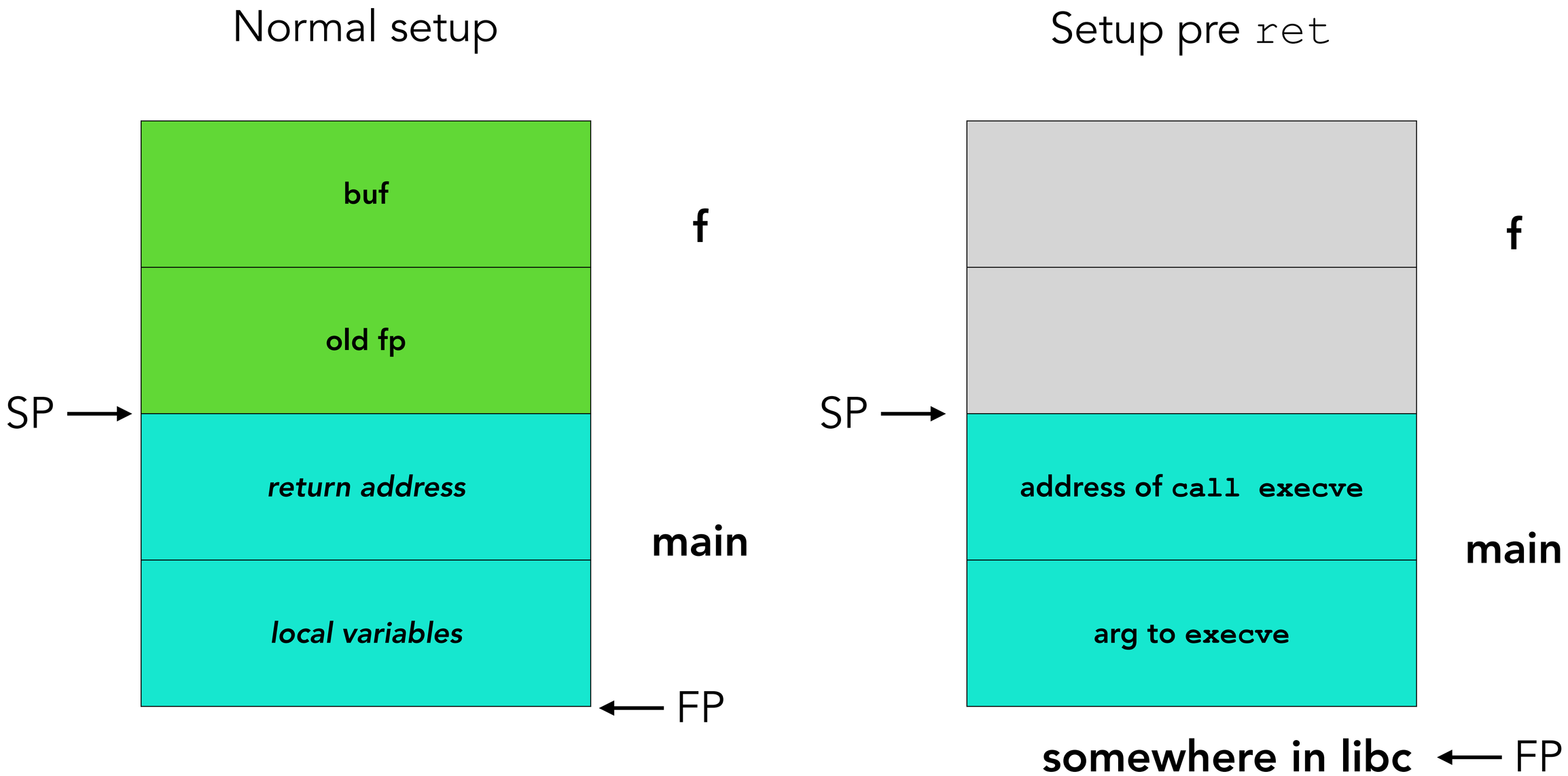
Return-to-libc attacks — function setup



Return-to-libc attacks — function setup

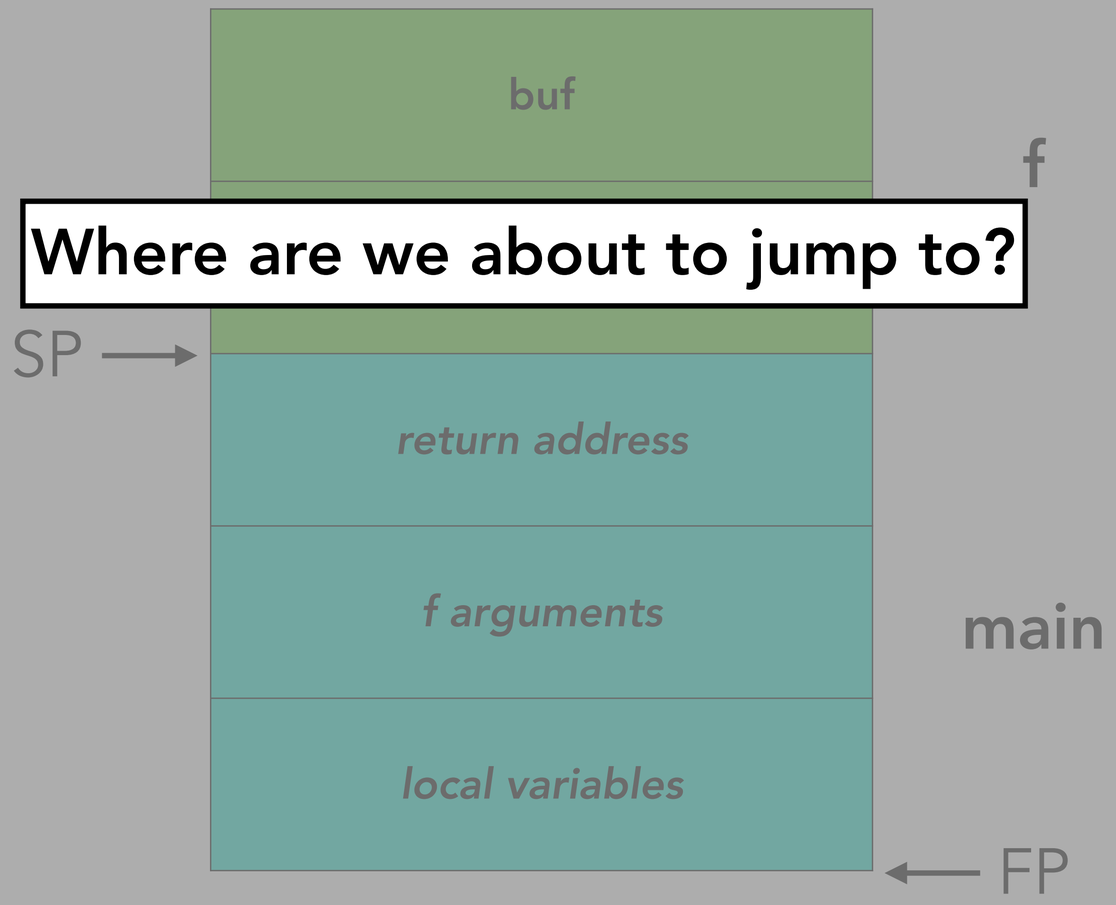


Return-to-libc attacks — function setup

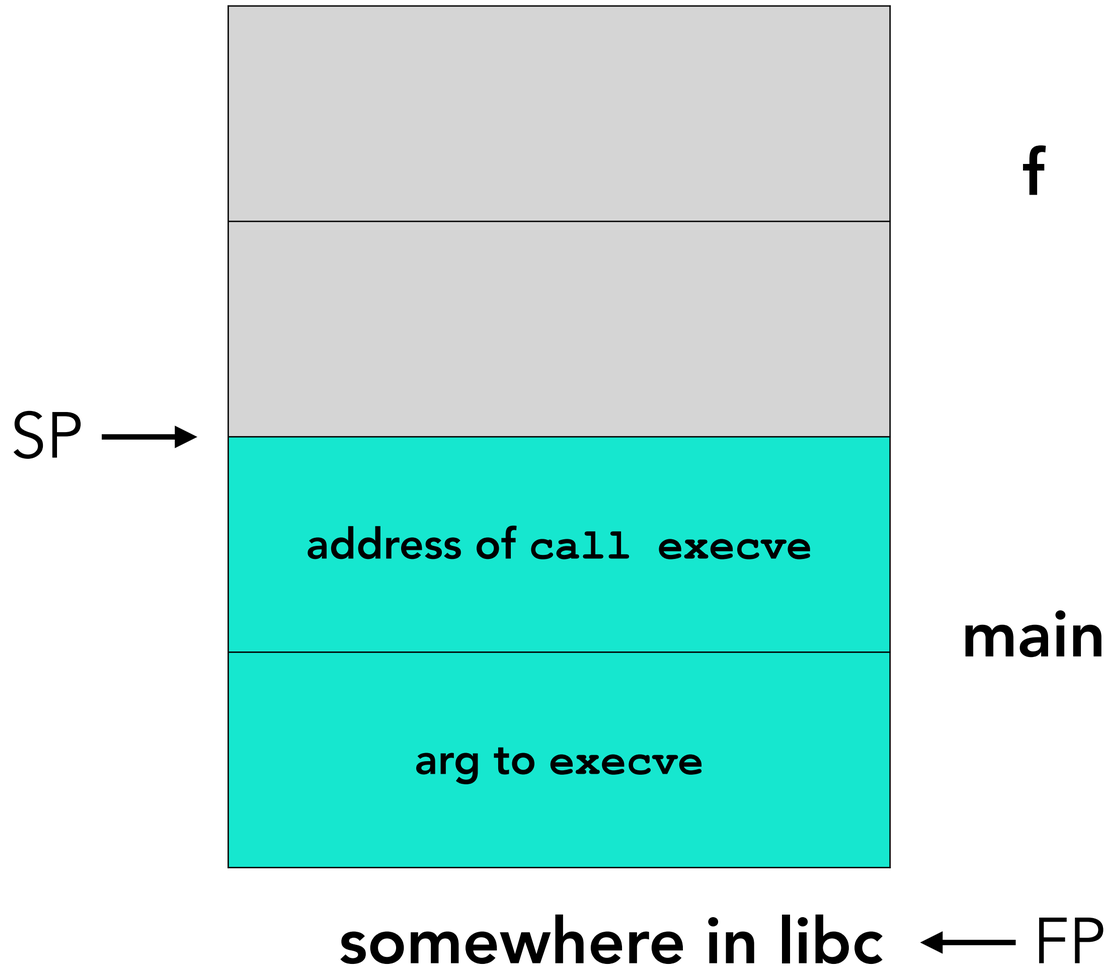


Return-to-libc attacks — function setup

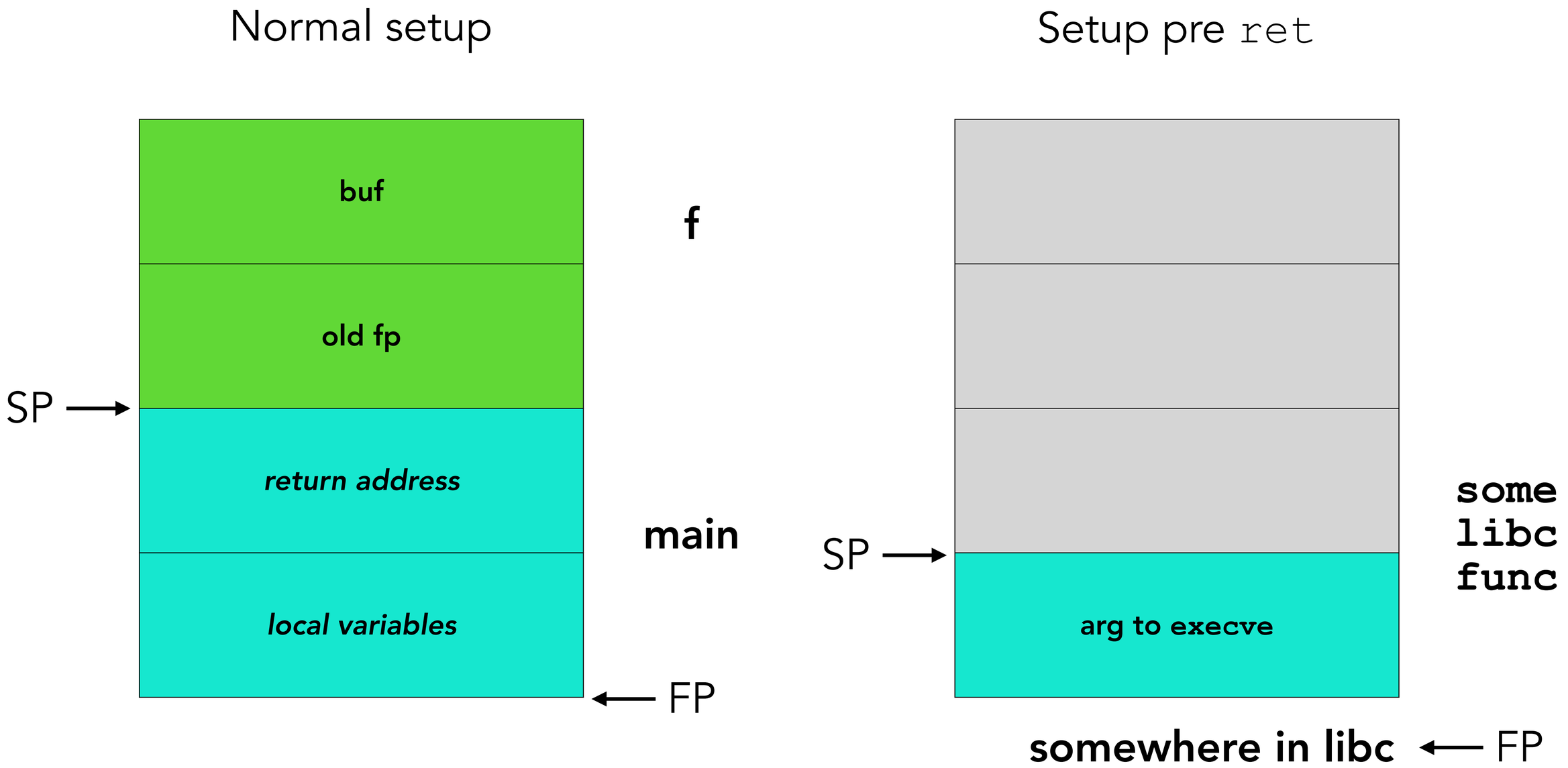
Normal setup pre `call`



Setup pre `ret`

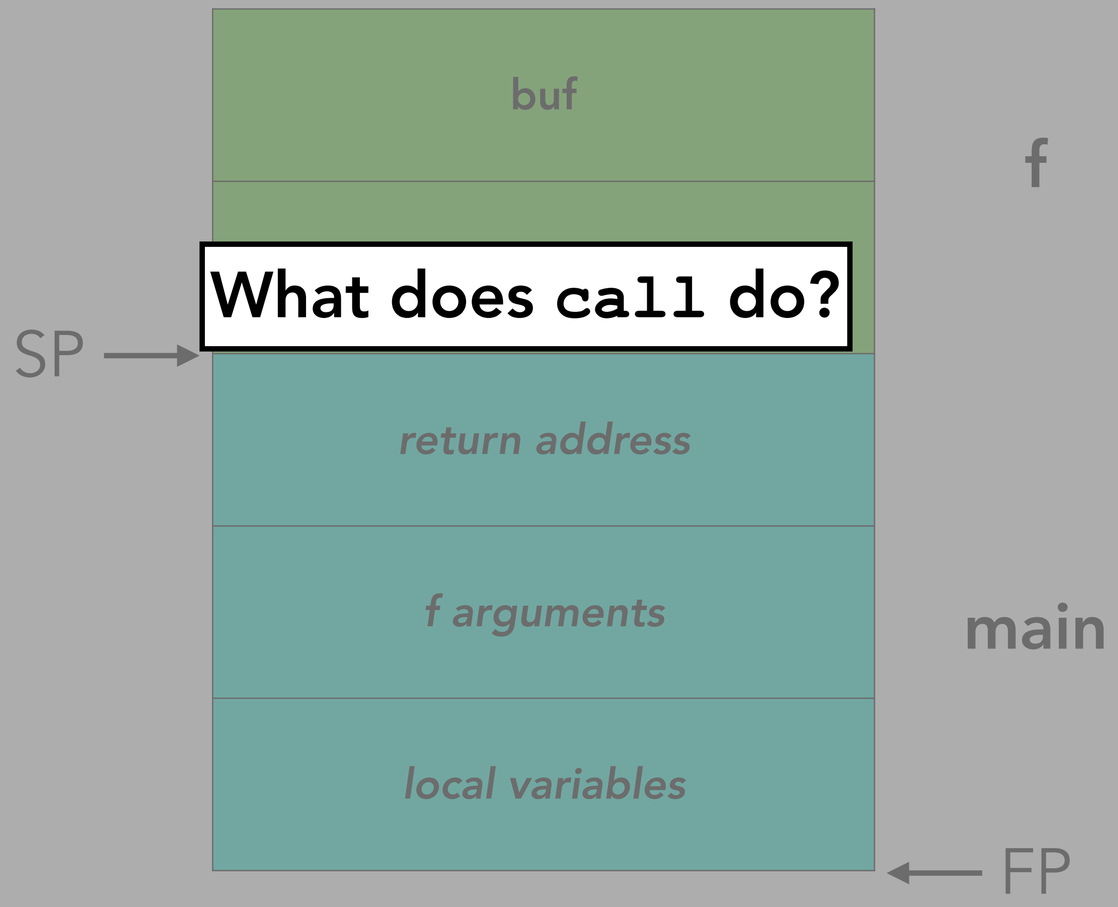


Return-to-libc attacks — function setup

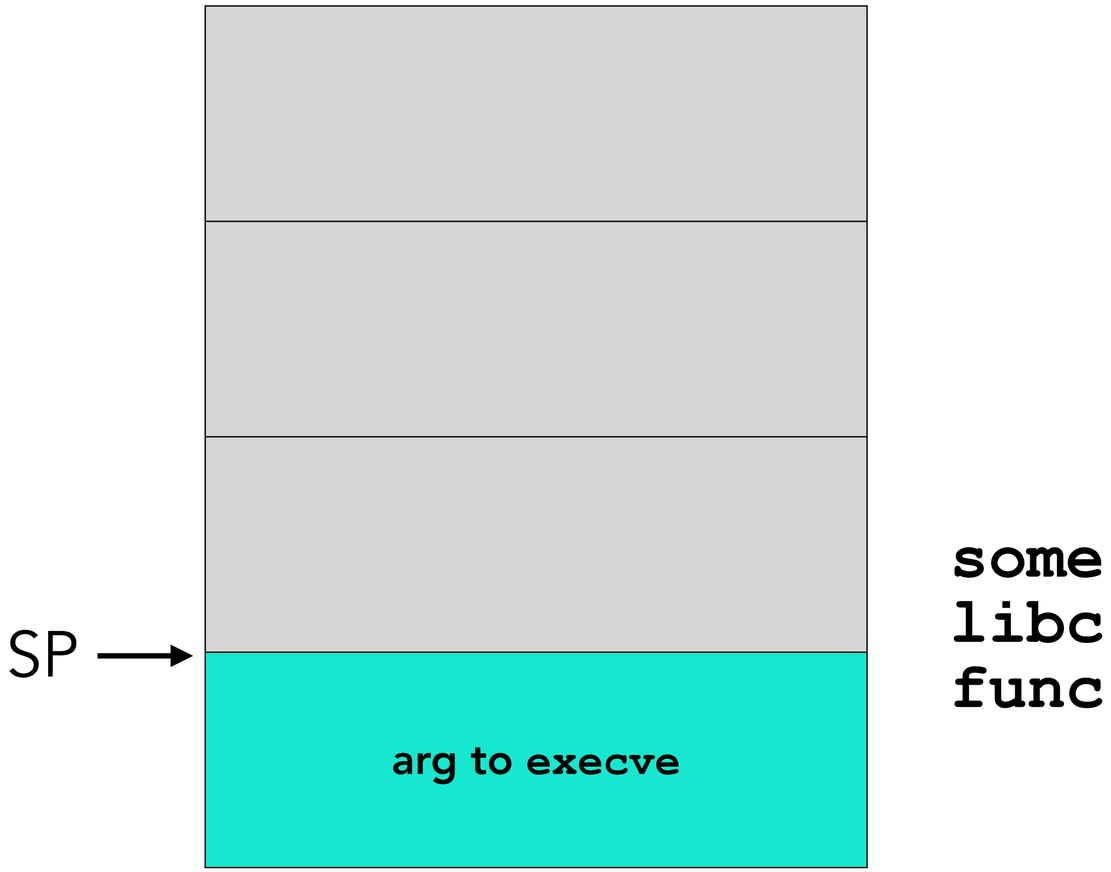


Return-to-libc attacks — function setup

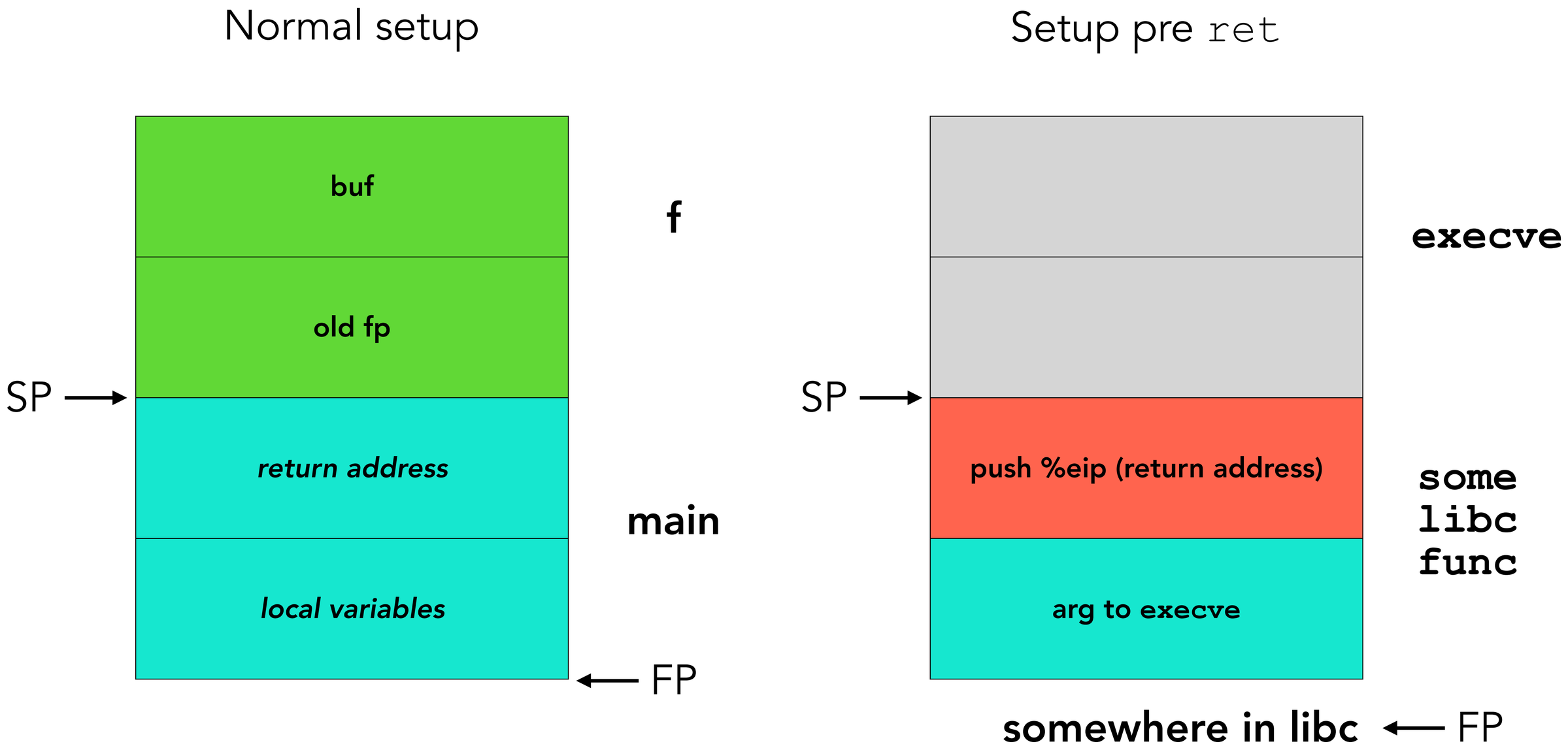
Normal setup pre `call`



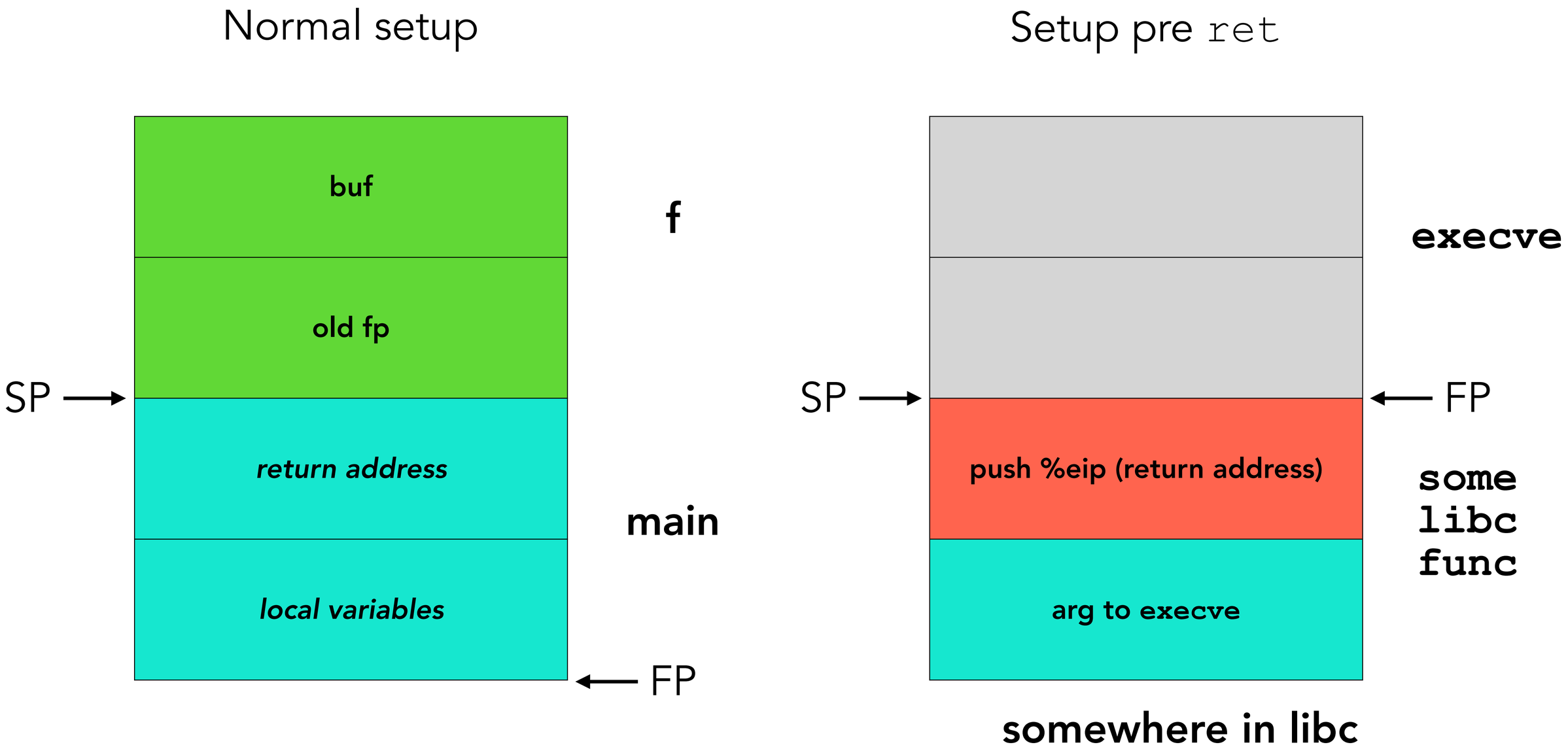
Setup pre `ret`



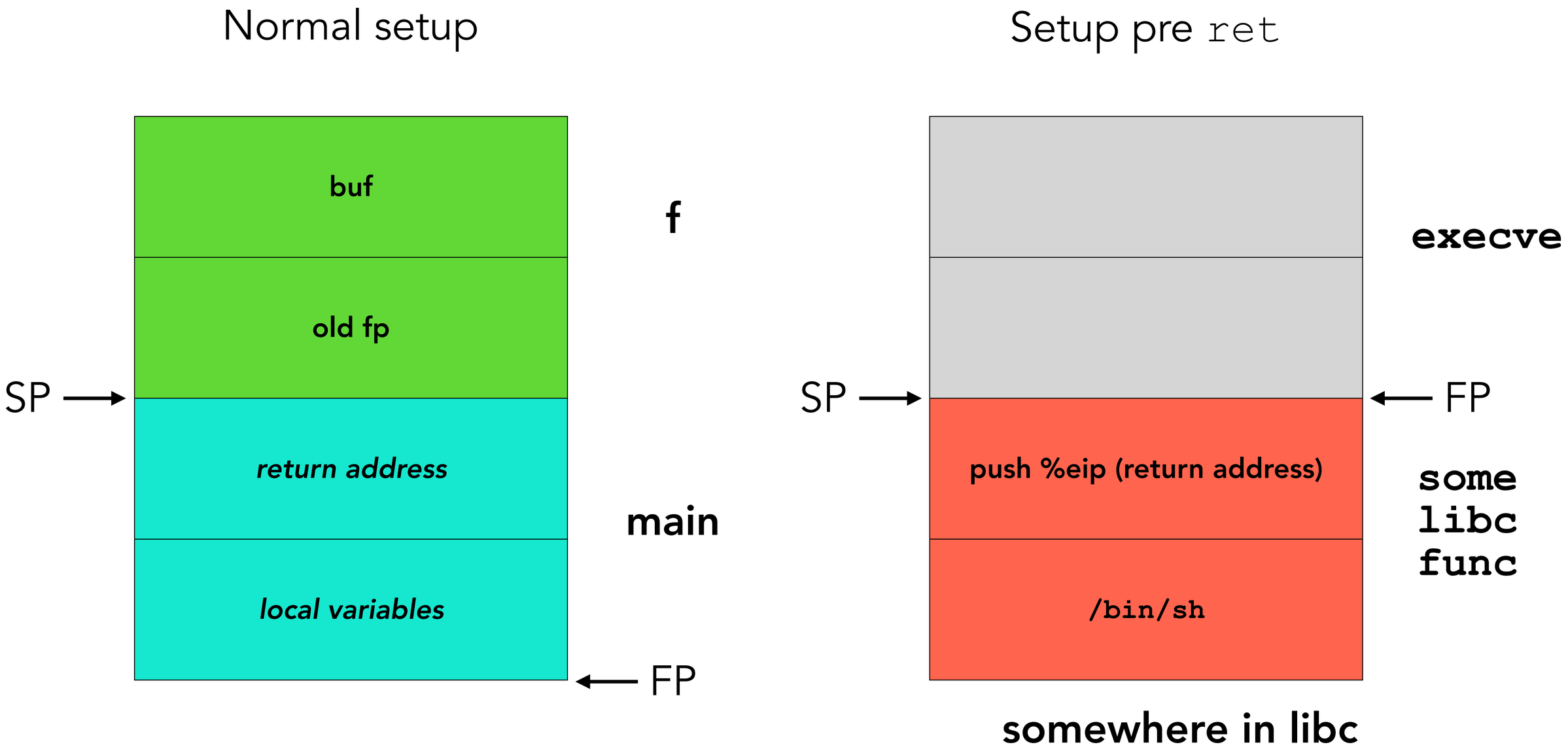
Return-to-libc attacks — function setup



Return-to-libc attacks — function setup

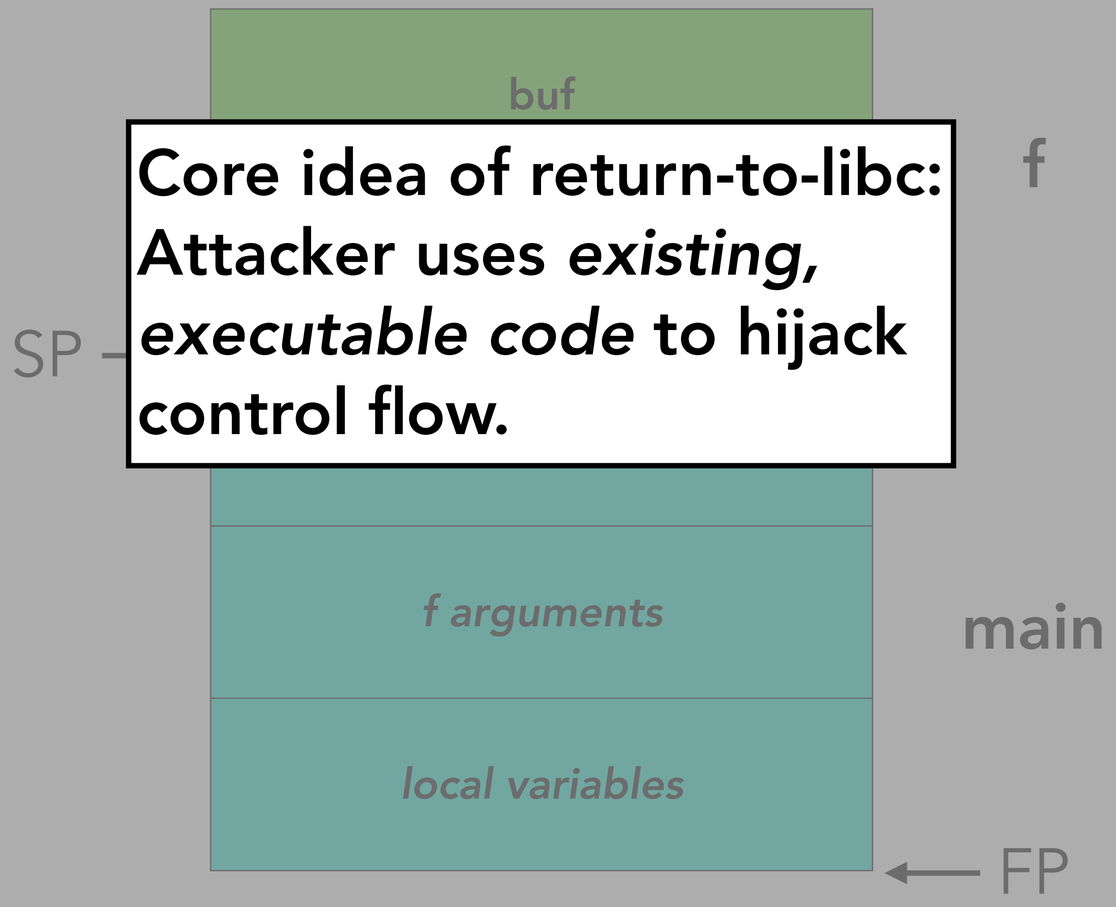


Return-to-libc attacks — function setup



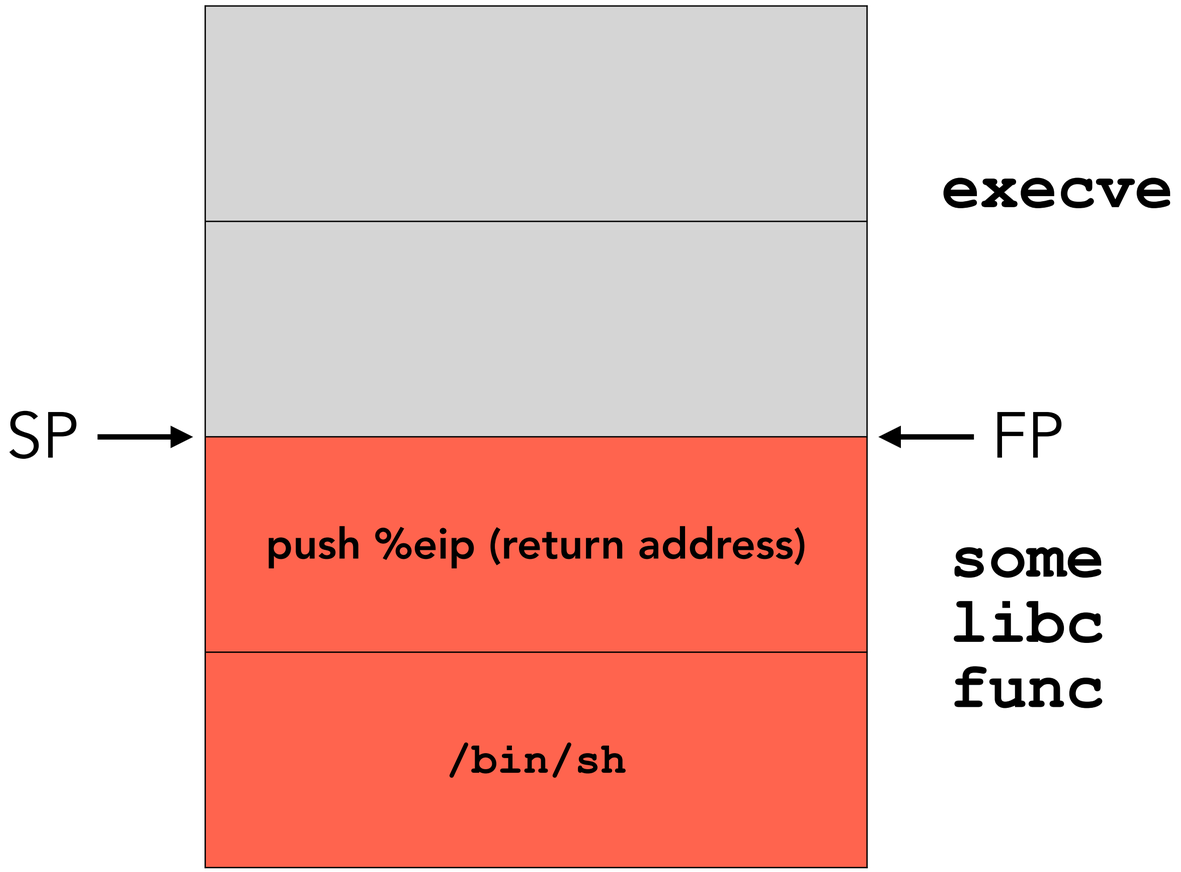
Return-to-libc attacks — function setup

Normal setup pre `call`



Core idea of return-to-libc: Attacker uses *existing, executable code* to hijack control flow.

Setup pre `ret`



somewhere in libc

How to handle return-to-libc attacks?

- Remove all code you don’t need!
  - If you don’t use `execve`, don’t load it in process memory
  - **This can be done at compile time**
- But not perfect…
  - Some dependencies are really challenging to find and remove and reduce functionality
  - Fundamental tradeoff between flexibility and security

Limitations with return-to-libc attacks

- “Straight line limited”
  - Means you can only enter into one libc function after another
- “Removal limited”
  - If you remove libc function that aren’t useful, you can seriously hamper attackers

Limitations with return-to-libc attacks

- “Straight line limited”
  - Means you can only enter into one libc function after another
- “Removal limited”
  - If you remove libc function that aren’t useful, you can seriously hamper attackers
- **…turns out… those assumptions are wrong, you don’t even need functions!**

Return-Oriented Programming

- Key idea: You don’t even need function calls!
- All you need are *micro sequences of instructions* (called gadgets) to mess with control flow of a program
- This is very possible in x86. Why?
  - x86 instructions are **ambiguous** and **dense**, so shifting by a single byte often leads to different strings of instructions
  - All you need is **ret** to chain gadgets together
- This is a UCSD paper: https://hovav.net/ucsd/dist/geometry.pdf
  - “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)”

Return-Oriented Programming

- Goal: make complex shellcode out of **existing application code**
- Stitch together arbitrary programs out of code "gadgets" already present in the target binary
  - **ROP Gadgets:** sets of code sequences that end in **`ret`**
  - Why **ret**?
    - It's everywhere (always at the end of every function, plus the `c3` byte is always other functions)
    - It changes control flow (and if you can write on the stack, you can control where it goes)

```
$otool -t /bin/ls |grep c3
0000000100000f70        39 48 38 7f 07 b8 ff ff ff ff 7d 02 5d c3 48 83
0000000100000fc0        00 00 7d 02 5d c3 48 83 c6 68 49 83 c0 68 48 89
0000000100001010        c3 48 83 c7 68 48 83 c6 68 5d e9 6b 35 00 00 55
0000000100001050        b8 01 00 00 00 7d 02 5d c3 48 83 c6 68 49 83 c0
00000001000010a0        7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 d8 34
00000001000010e0        48 7f 07 b8 01 00 00 00 7d 02 5d c3 48 83 c6 68
0000000100001120        7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 58 34
0000000100001150        b8 01 00 00 00 7d 02 5d c3 48 83 c6 68 48 83 c1
00000001000011a0        7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 d8 33
00000001000011e0        58 7f 07 b8 01 00 00 00 7d 02 5d c3 48 83 c6 68
0000000100001870        c0 09 c8 8a 0d ab 3c 00 00 89 c3 81 cb 80 00 00
0000000100001b70        5d d4 89 de e8 57 29 00 00 48 89 c3 48 85 db 0f
0000000100001c30        03 39 00 00 01 e9 52 01 00 00 0f b7 c0 83 f8 0d
0000000100001dd0        c3 48 8d 35 91 2d 00 00 eb 07 48 8d 35 c0 2d 00
0000000100001e20        36 0f b7 56 58 83 fa 07 75 02 5d c3 44 0f b7 c9
0000000100001ec0        00 48 8d 3d e2 2c 00 00 e8 21 26 00 00 48 89 c3
0000000100001f70        34 48 83 c3 02 80 f9 3a 75 19 80 7b fe 3a 75 13
0000000100001fa0        c3 84 c9 75 d0 44 89 b5 78 fb ff ff 45 89 e6 80
00000001000023b0        fb ff ff 74 5c 8b 78 74 e8 ef 20 00 00 48 89 c3
00000001000023e0        00 00 48 89 c3 48 85 db 0f 84 9a 04 00 00 48 89
0000000100002520        66 18 4d 8b 7e 20 41 8b 5e 30 48 63 c3 48 8d 34
0000000100002560        20 49 63 4e 30 41 89 04 8f 41 8b 5e 30 ff c3 41
0000000100002870        38 05 00 00 5b 41 5c 41 5d 41 5e 41 5f 5d c3 48
0000000100002970        c3 48 8d 3d 9e 22 00 00 48 8d 35 a1 22 00 00 48
0000000100002a30        ed 48 83 c3 68 48 89 df e8 4f 0b 00 00 89 c3 45
0000000100002a90        0f b7 7c 24 04 e8 28 0b 00 00 01 c3 89 d8 48 83
0000000100002aa0        c4 08 5b 41 5c 41 5d 41 5e 41 5f 5d c3 55 48 89
0000000100002c30        4f 28 48 8b 46 08 eb 0f 85 c0 45 8b 4f 38 48 8b
0000000100003200        00 48 83 c3 18 48 81 fb a8 01 00 00 75 84 bb 10
0000000100003260        45 89 fd 4c 8d bd b0 f7 ff ff 48 83 c3 18 48 83
00000001000032e0        5b 41 5c 41 5d 41 5e 41 5f 5d c3 48 8d 35 c5 18
0000000100003350        48 83 c4 08 5b 5d c3 48 8d 3d c6 1b 00 00 31 c0
00000001000034a0        c4 70 5b 41 5e 5d c3 e8 a0 0f 00 00 55 48 89 e5
0000000100003550        00 89 d8 48 83 c4 08 5b 5d c3 66 90 7e ff ff ff
00000001000035f0        00 00 00 5d c3 81 c1 00 60 00 00 81 e1 00 f0 00
00000001000036a0        75 06 48 83 c3 10 eb 69 48 8d 7b 68 e8 f7 0e 00
0000000100003740        5e 41 5f 5d c3 55 48 89 e5 41 57 41 56 41 55 41
0000000100003930        5c f0 ff ff 89 c3 48 8d 05 1b 1d 00 00 8b 08 85
0000000100003970        7c 04 85 c9 75 40 41 89 d4 89 c3 48 8d 05 b6 1c
0000000100003990        45 f8 e8 c3 0b 00 00 42 8d 04 2b 23 45 c8 44 89
00000001000039f0        83 c4 38 5b 41 5c 41 5d 41 5e 41 5f 5d c3 31 ff
0000000100003ac0        00 00 48 89 c3 8a 04 1a 88 45 d6 48 83 ca 01 48
0000000100003b00        f8 80 f9 30 75 36 83 c3 d0 41 89 1f 66 bb 01 00
0000000100003b40        9f 80 f9 07 77 08 83 c3 9f 41 89 1f eb 4e 89 c1
0000000100003b50        80 c1 bf 80 f9 07 77 12 83 c3 bf 41 89 1f 48 8b
0000000100003bd0        41 5d 41 5e 41 5f 5d c3 55 48 89 e5 41 56 53 41
0000000100003c30        c6 08 00 00 89 c7 44 89 f6 5b 41 5e 5d e9 e2 08
0000000100003c60        31 c0 48 83 c4 10 5d c3 55 48 89 e5 e8 e9 08 00
0000000100003c70        00 31 c0 5d c3 55 48 89 e5 41 56 53 89 f8 48 8d
0000000100003d10        5e 5d e9 b5 08 00 00 5b 41 5e 5d c3 55 48 89 e5
0000000100003e40        ff ff 4c 89 e6 4c 89 f9 e8 f5 06 00 00 48 89 c3
```

Return-Oriented Programming

- Basic idea:
  - Overwrite saved return address on the stack to point to first gadget, the following word to point to the second gadget, etc.
  - **The stack pointer acts as a sort of "instruction pointer" in this new world**
- What can we do with this?
  - Turing-complete computation
    - Load and store gadgets
    - Arithmetic and logic gadgets
    - Control flow gadgets
- This is target7 in PA2; extra credit

Simple example

What does this piece of assembly do?

```
mov $5, %edx
```

0x00 0xffffffff

stolen w/ love from UMD

Simple example

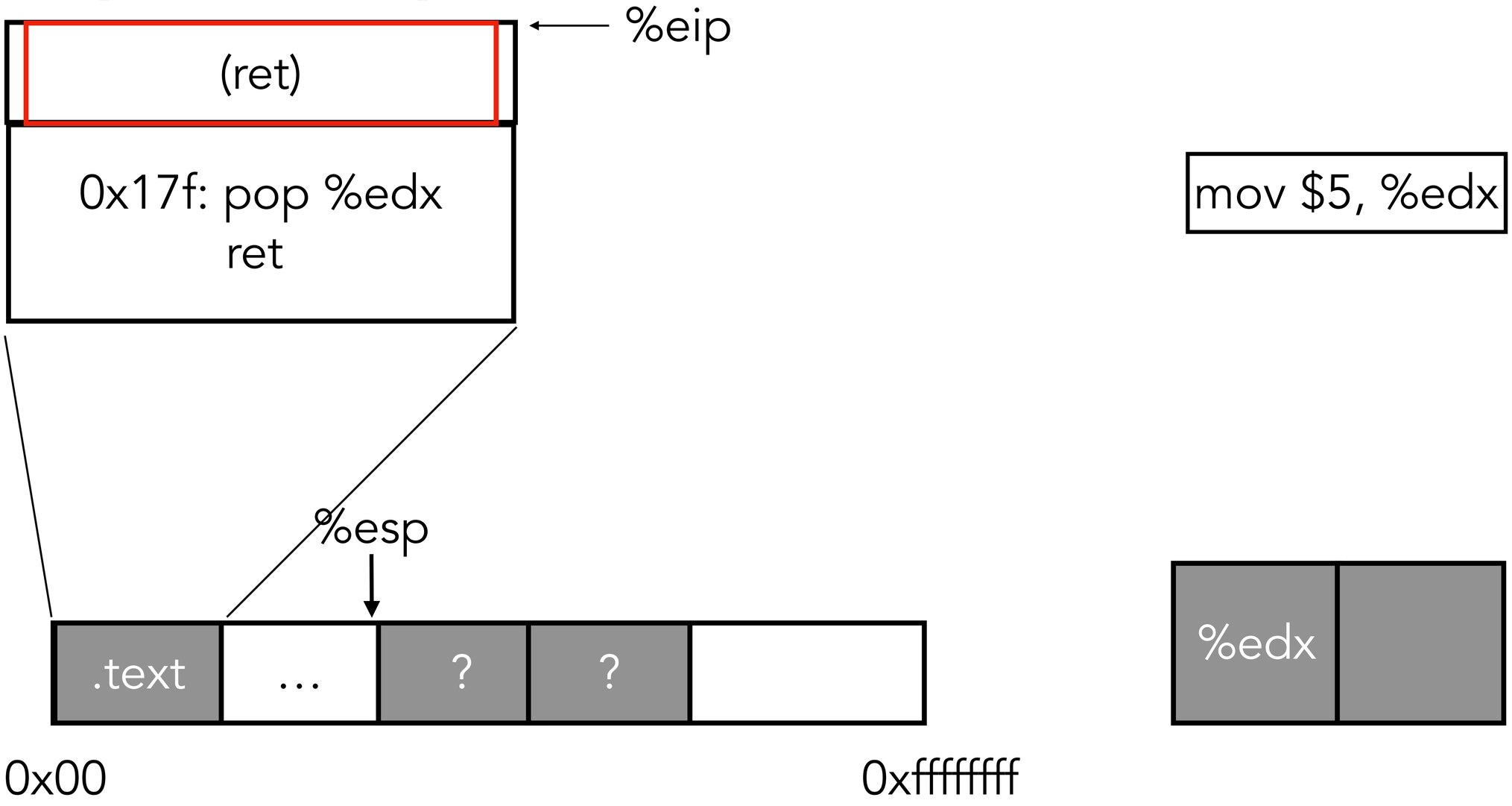
What does this piece of assembly do?

```
mov $5, %edx
```

0x00 0xffffffff

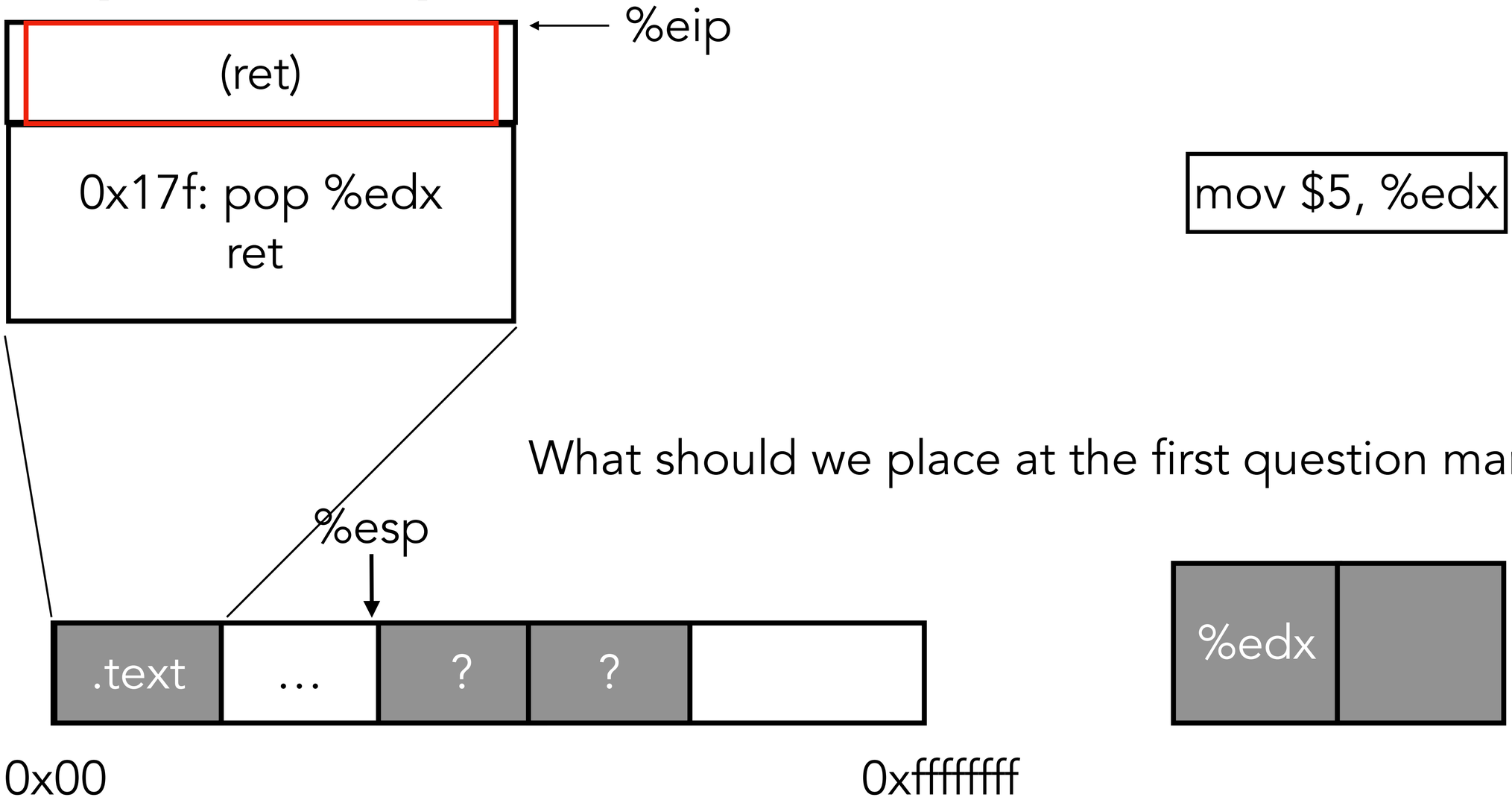
stolen w/ love from UMD

Simple example



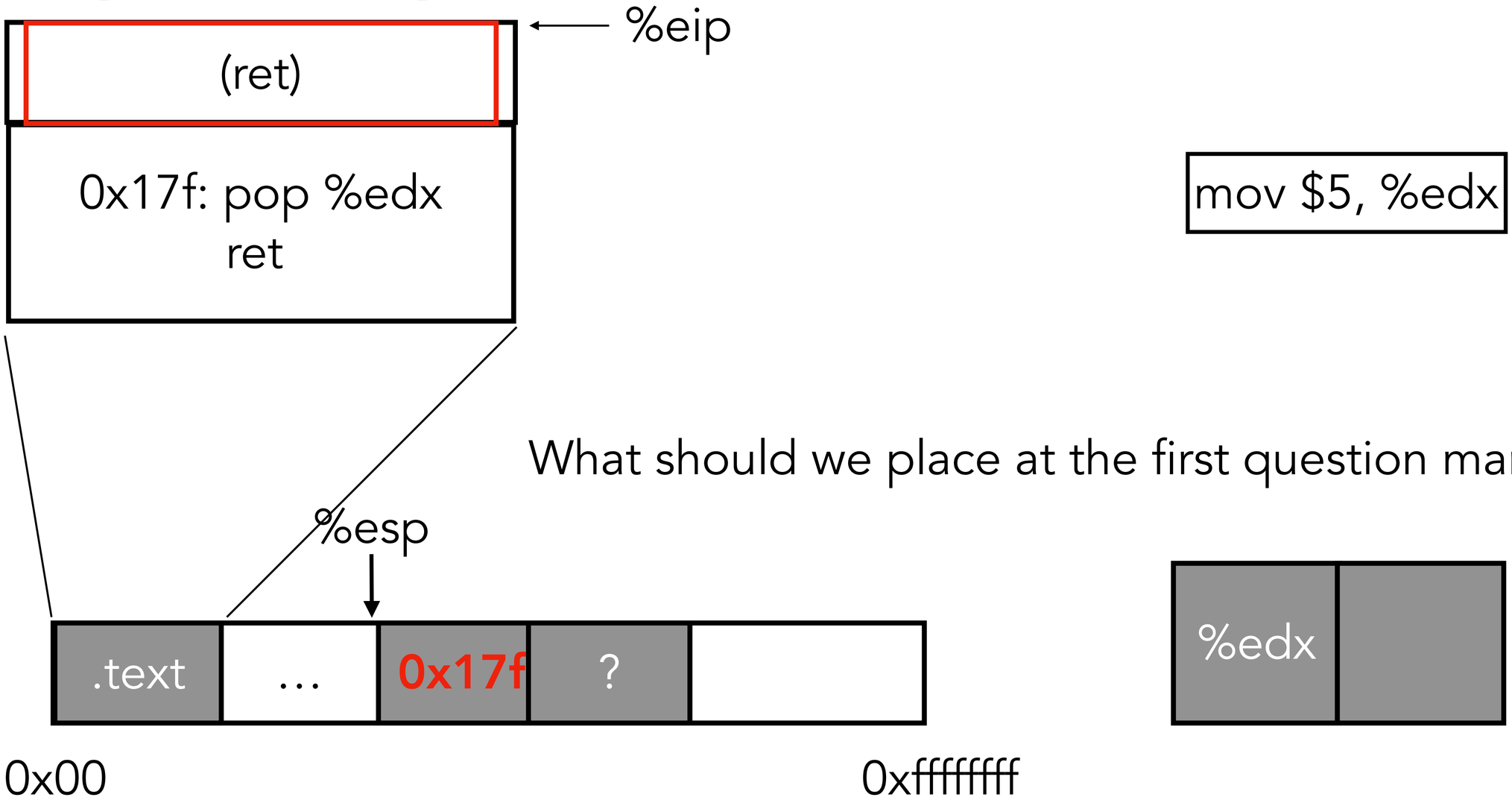
stolen w/ love from UMD

Simple example



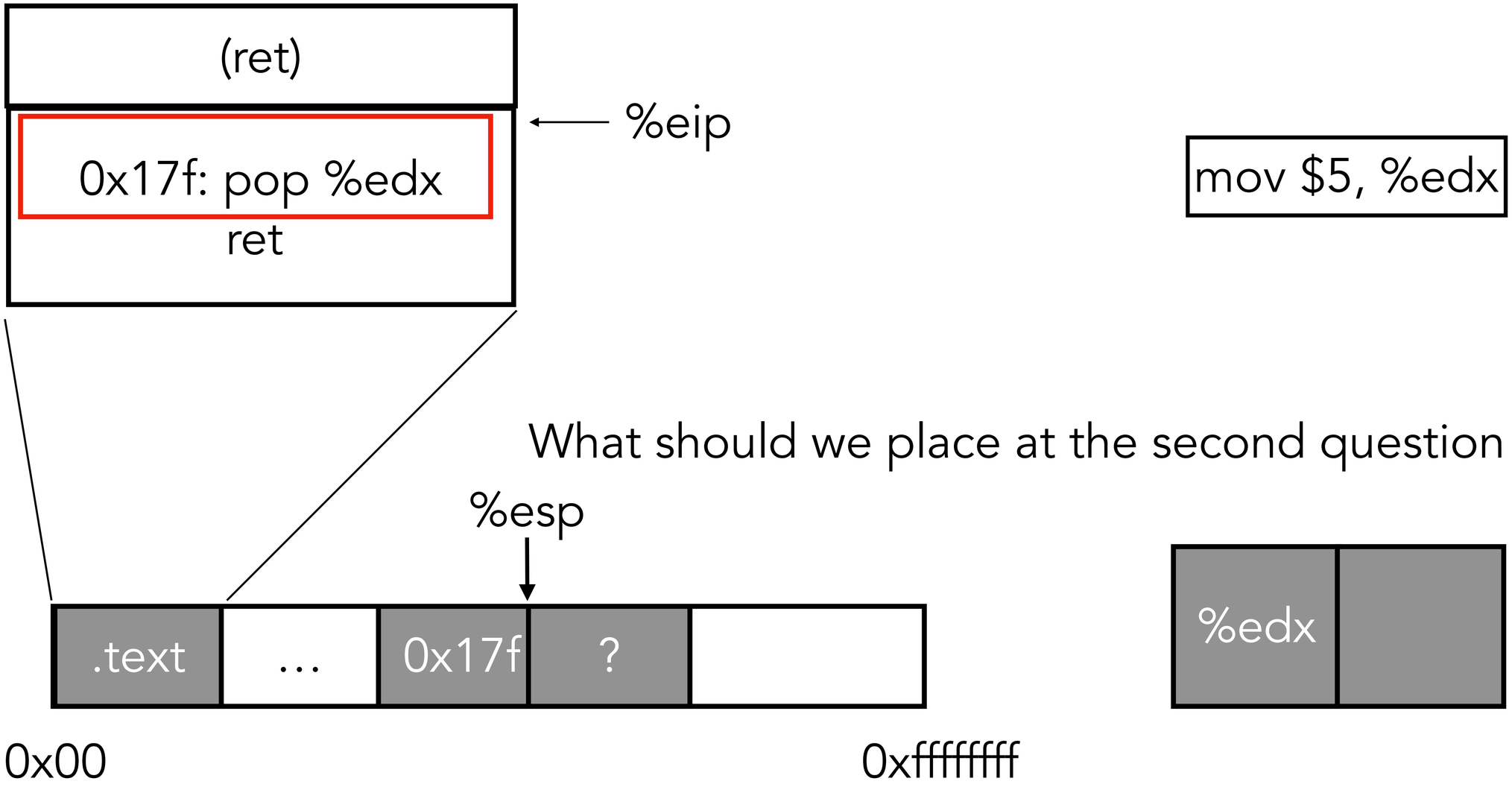
stolen w/ love from UMD

Simple example



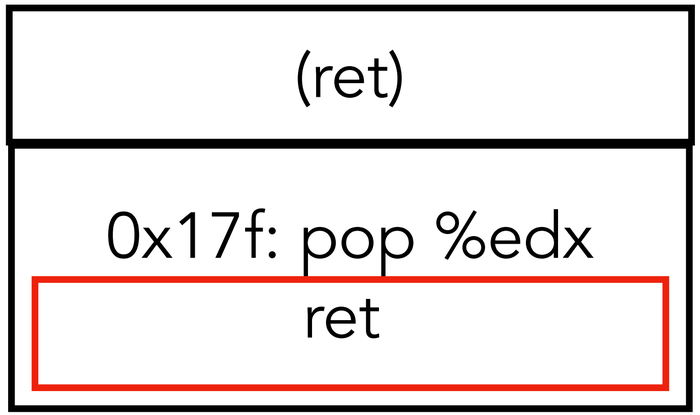
stolen w/ love from UMD

Simple example



stolen w/ love from UMD

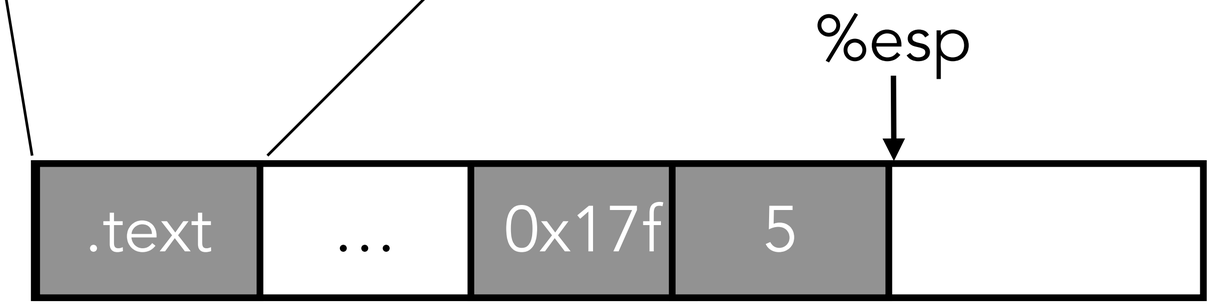
Simple example



%eip

mov $5, %edx

What should we place at the second question mark?

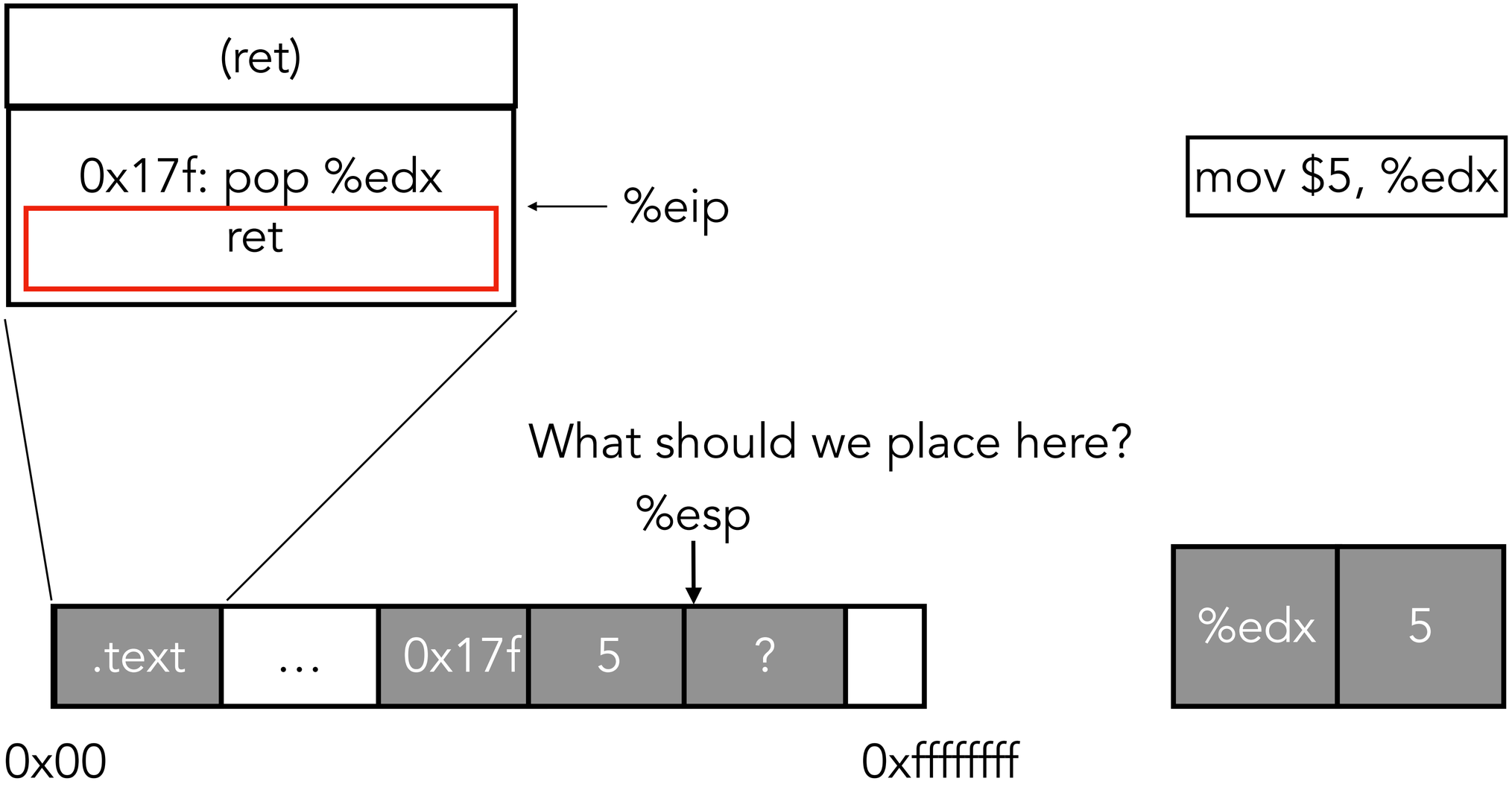


0x00

0xffffffff

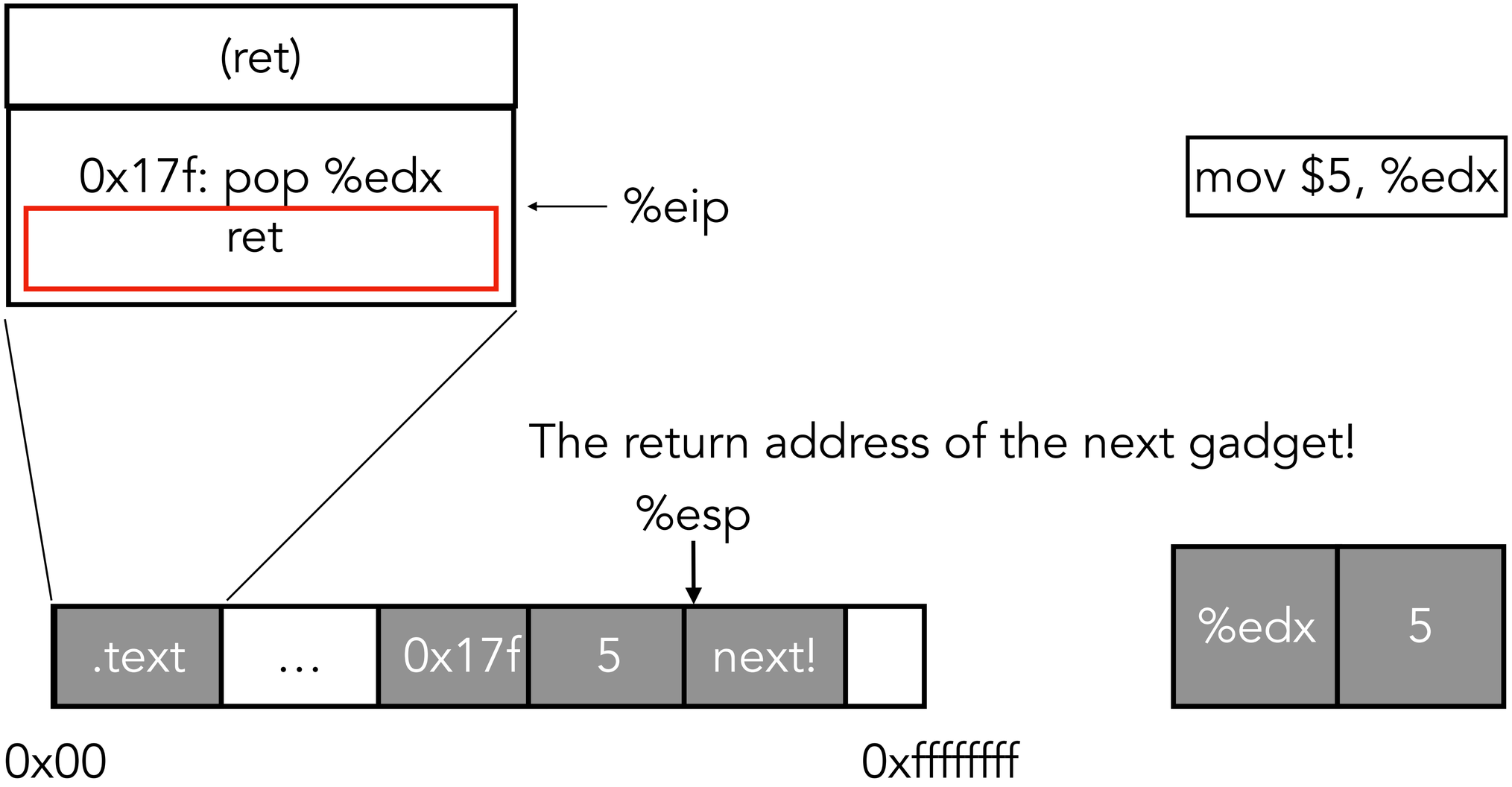
stolen w/ love from UMD

Simple example



stolen w/ love from UMD

Simple example



stolen w/ love from UMD


Return-Oriented Programming
is a lot like a ransom note, but instead of cutting cut letters from magazines, you are cutting out instructions from .text segments

Address Space Layout Randomization

Making ROP Hard

- What are some assumptions made about the *location* of libc functions that make ROP possible?

Making ROP Hard w/ Address Space Layout Randomization

- What are some assumptions made about the *location* of libc functions that make ROP possible?
  - libc is in a fixed location: **not true with Address Space Layout Randomization (ASLR)**
- Basic idea: Add a random offset to stack base
  - Make it harder for the attacker to guess location of libc, shellcode, etc.

ASLR Requirements

- Needs compiler, linker, and loader support
- Side effects
  - Increasing code size w/ minor performance overhead
  - Random number generator dependency
  - Potential load time impact for shared library relocation
- But despite this, **most modern systems rely on a combo of ASLR, DEP, and canaries for runtime protection**

Getting around ASLR?

- One mechanism: NOP sled
  - Basic idea… if you don't know exactly where the address of the buffer is, you can pad the buffer with **nop** (no operation) instructions to increase the chance of hitting shellcode
- You will have to figure out how to implement this in PA2 :)

Summary

- How do we protect software?
  - **Stack canaries**: detect overwrite of stack into control data
  - **DEP / W^X:** mark stack and heap pages are non-executable (with hardware support) to prevent code from executing there
  - **ASLR**: randomize location of libraries, data structures to prevent attacks from knowing where they are
- Nothing provides perfect security (everything can be bypassed)
  - **Theory:** make reliable exploits expensive and hard to implement
  - **Practice:** as new bypasses are developed, need to update the idea
- You will have to implement everything we talked about today (except ROP) in your PA2!

Next time…

- Shift away from application security —> towards system security
  - Thinking about the OS as a broader system, other mechanisms of leaking information (e.g., side channels, covert channels)
- Note — **there is a lot in application security we didn't cover, e.g.,**
  - Heap smashing
  - Control flow integrity
  - Fuzzing, static analysis, dynamic analysis
  - Heap spraying
  - The list goes on… lots to learn about if you're interested
- Work on your PA! It's not easy!