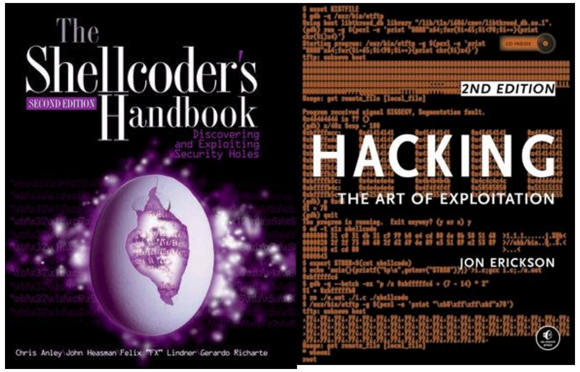
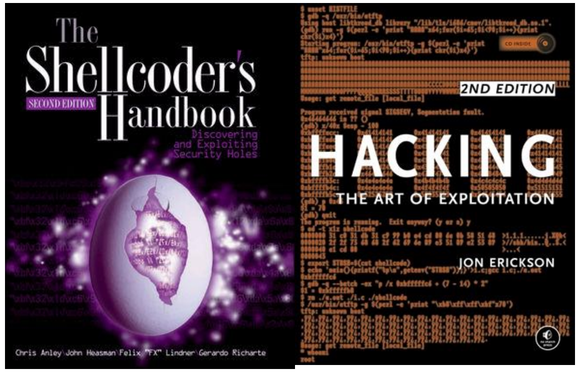
Software Exploitation

Computer Security Fall 2024

Georgios (George) Portokalidis

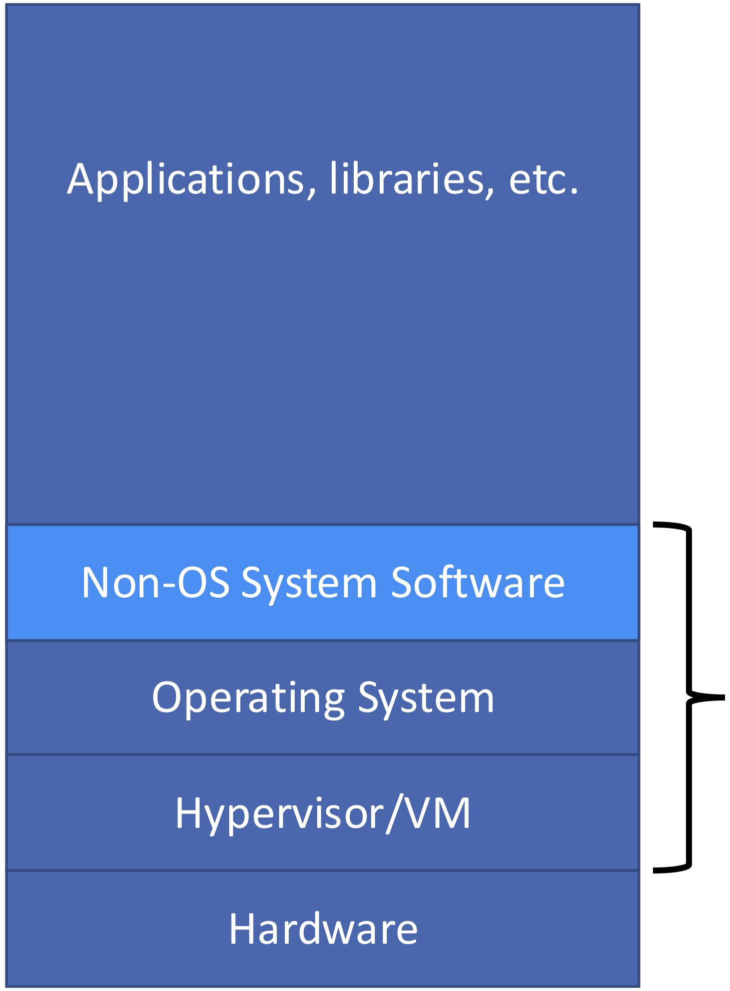


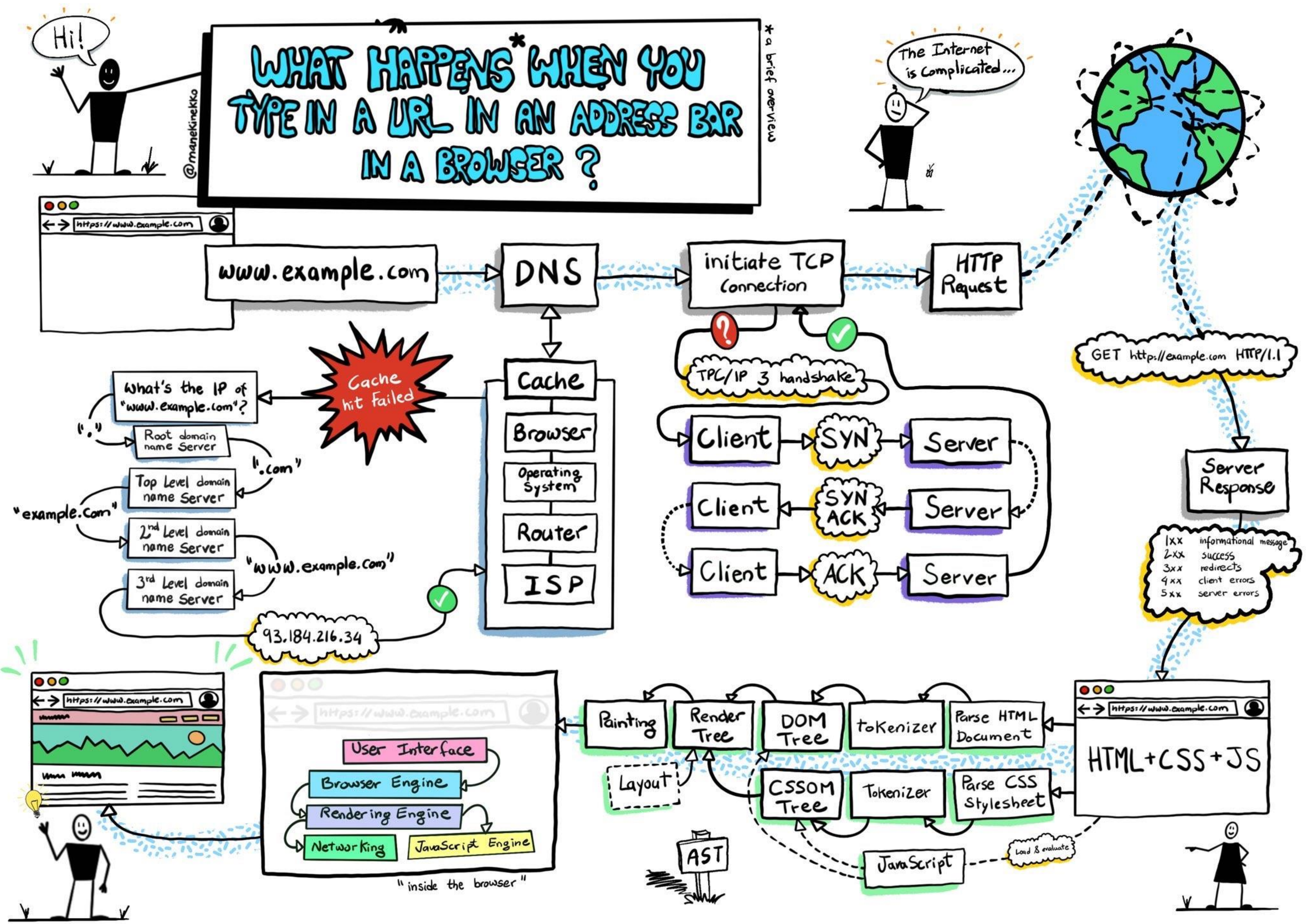


Part 1. Introduction

Which Software?

- This section of the course will focus on **systems software**
- **Systems software** is designed to provide a platform for other software
- Examples:
  - Operating systems
  - Hardware drivers
  - Language runtimes
  - Virtual machines
  - Frameworks
  - Performance critical software





Hi!
WHAT HAPPENS* WHEN YOU TYPE IN A URL IN AN ADDRESS BAR IN A BROWSER ?
* a brief overview
@manekinekko
The Internet is Complicated...
https://www.example.com
www.example.com
DNS
initiate TCP Connection
HTTP Request
Cache
Browser
Operating System
Router
ISP
Cache hit failed
What's the IP of "www.example.com"?
"."
Root domain name Server
".com"
Top Level domain name Server
"example.com"
2nd Level domain name Server
"www.example.com"
3rd Level domain name Server
93.184.216.34
TPC/IP 3 handshake
Client
SYN
Server
Client
SYN ACK
Server
Client
ACK
Server
GET http://example.com HTTP/1.1
Server Response
1xx informational message
2xx success
3xx redirects
4xx client errors
5xx server errors
HTML+CSS+JS
Parse HTML Document
tokenizer
DOM Tree
Render Tree
Painting
Layout
CSSOM Tree
Tokenizer
Parse CSS Stylesheet
JavaScript
Load & evaluate
AST
User Interface
Browser Engine
Rendering Engine
Networking
JavaScript Engine
"inside the browser"

C and C++ power the majority of systems software

C and C++

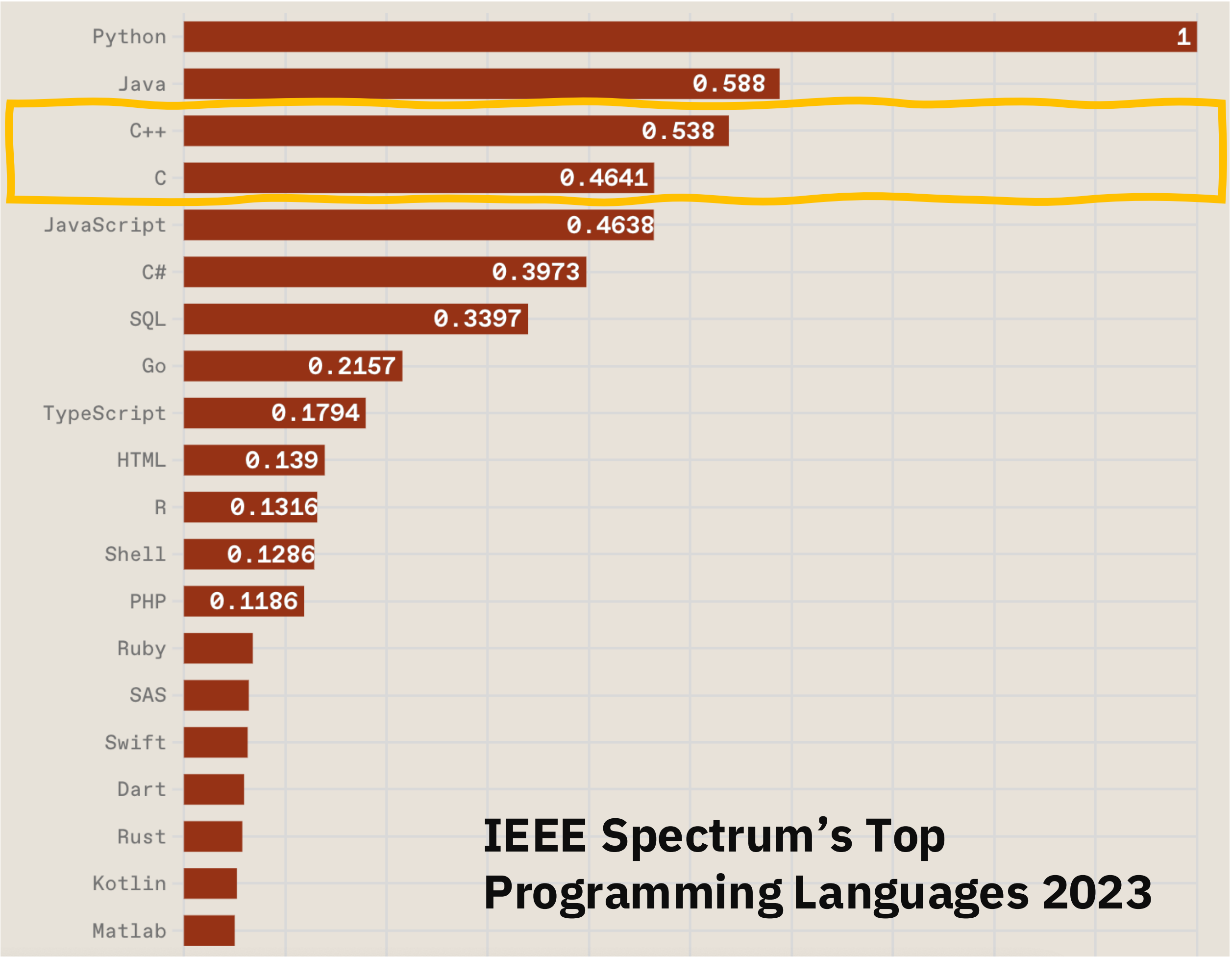
- Versatile and efficient! Pereira et al. [SLE'17]
  - It is fast
  - Uses the least energy
  - Has a small memory footprint

Table 4. Normalized global results for Energy, Time, and Memory

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

C and C++

- Versatile and efficient! Pereira et al. [SLE'17]
  - It is fast
  - Uses the least energy
  - Has a small memory footprint
- Popular

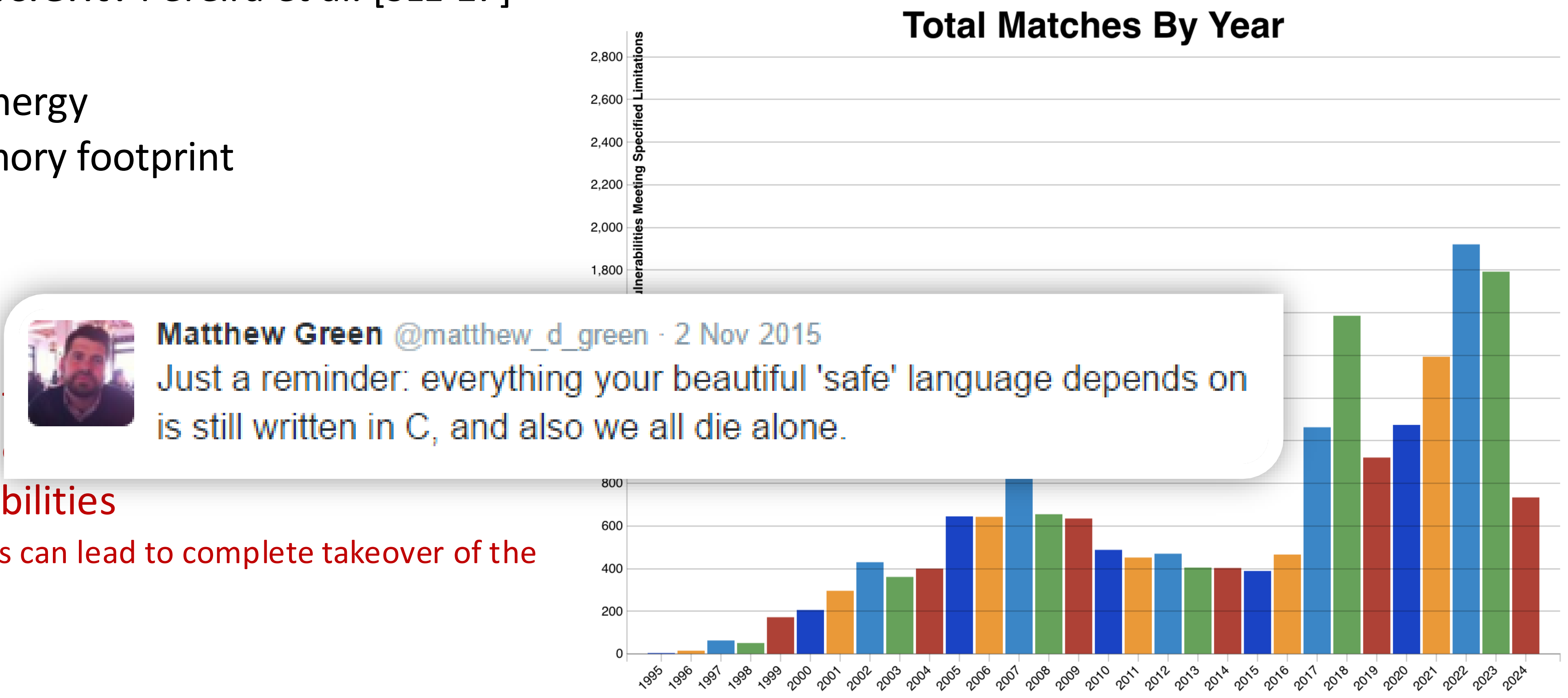


https://spectrum.ieee.org/the-top-programming-languages-2023

C and C++

- Versatile and efficient! Pereira et al. [SLE'17]
  - It is fast
  - Uses the least energy
  - Has a small memory footprint
- Popular
- Error prone!
  - No memory safe
    - Developer handl
  - Leads to vulnerabilities
    - Powerful exploits can lead to complete takeover of the system

Vulnerabilities matching "overflow"



https://nvd.nist.gov/vuln/search

Memory Safety

- What happens when line 6 executes?

```
1.  void foo()
2.  {
3.      int a;
4.      char buffer[4];
5.      …
6.      buffer[4] = ‘A’;
7.      …
8.  }
```

This is classified as ”undefined behavior”

Whatever you guessed may be correct

Memory Management

```
int mytest(char *str)
{
        char *buf = malloc(16);

        /* Do something with buf */

        free(buf);

        return 0;
}
```

Developers are responsible to allocate and de-allocate memory

Memory Management

```
int mytest(char *str)
{
        char *buf = malloc(16);

        /* Do something with buf */

        free(buf);
        buf[1] = '\0';
        return 0;
}
```

Developers are responsible to allocate and de-allocate memory

What happens if buf is used after free?

Memory Corruption

"Memory corruption occurs in a computer program when the contents of a memory location are unintentionally modified due to programming errors; this is termed **violating memory safety**.

When the corrupted memory contents are used later in that program, it leads either to program crash or to **strange and bizarre program behavior.** "

--wikipedia

Common Vulnerabilities

- Overflows: Writing beyond the end of a buffer
- Underflows: Writing beyond the beginning of a buffer
- Format string vulnerabilities: Evaluating input string as format string
- Uninitialized memory: Using pointer before initialization
- Null pointer dereferences: Using NULL pointers
- Use-after-free: Using memory after it has been freed
- Type confusion: Assume a variable/object has the wrong type

Why Are These Bugs (Serious) Vulnerabilities?

- **Arbitrary code execution (ACE)**, put simply, is a vulnerability that allows attackers to inject their own malicious code onto a target system without user awareness or permission

- A **Remote code execution (RCE)** attack is one where an attacker can run malicious code on an organization's computers or network

Generic Attack Types

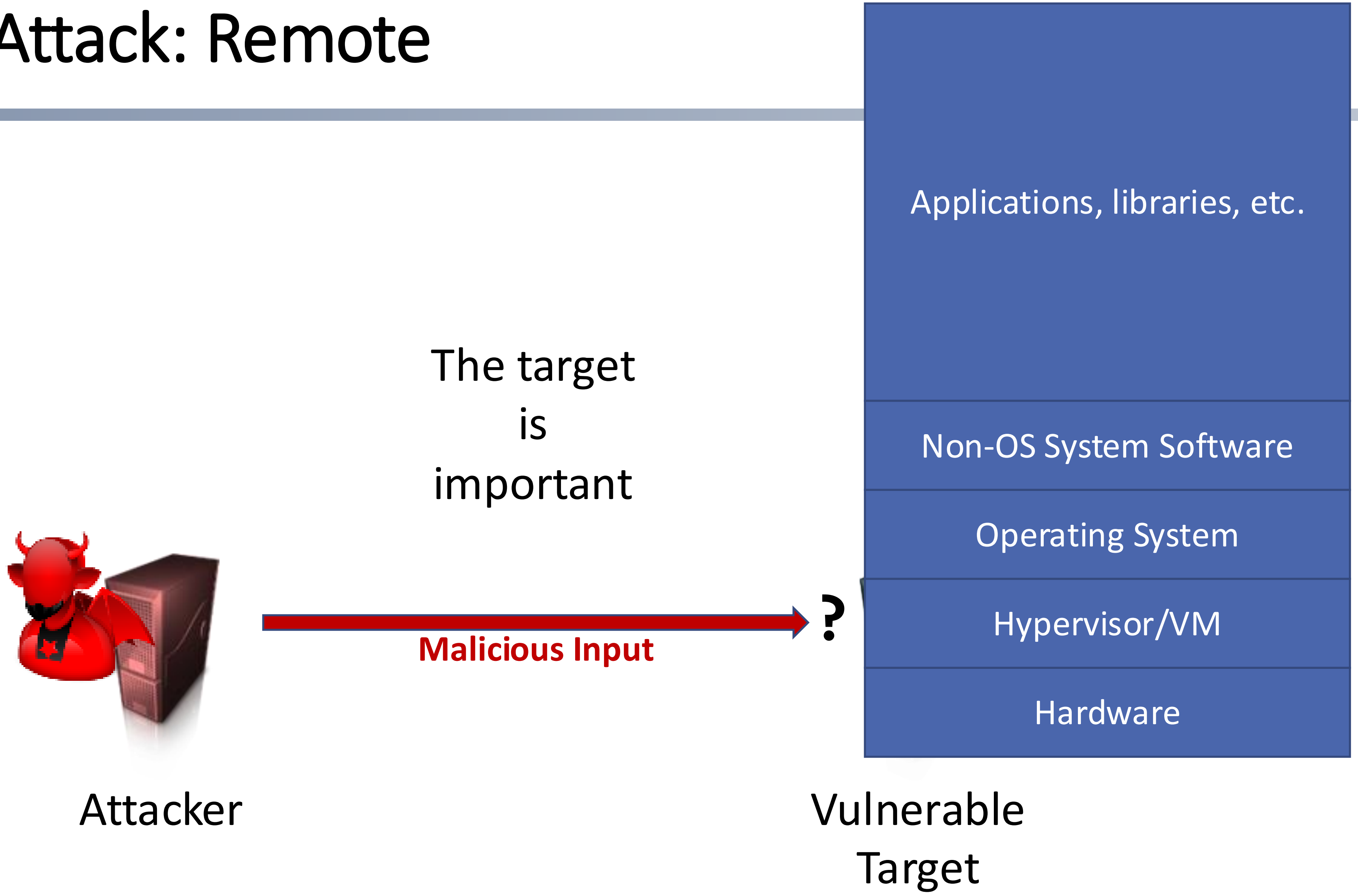
Local Attacks

Remote Attacks

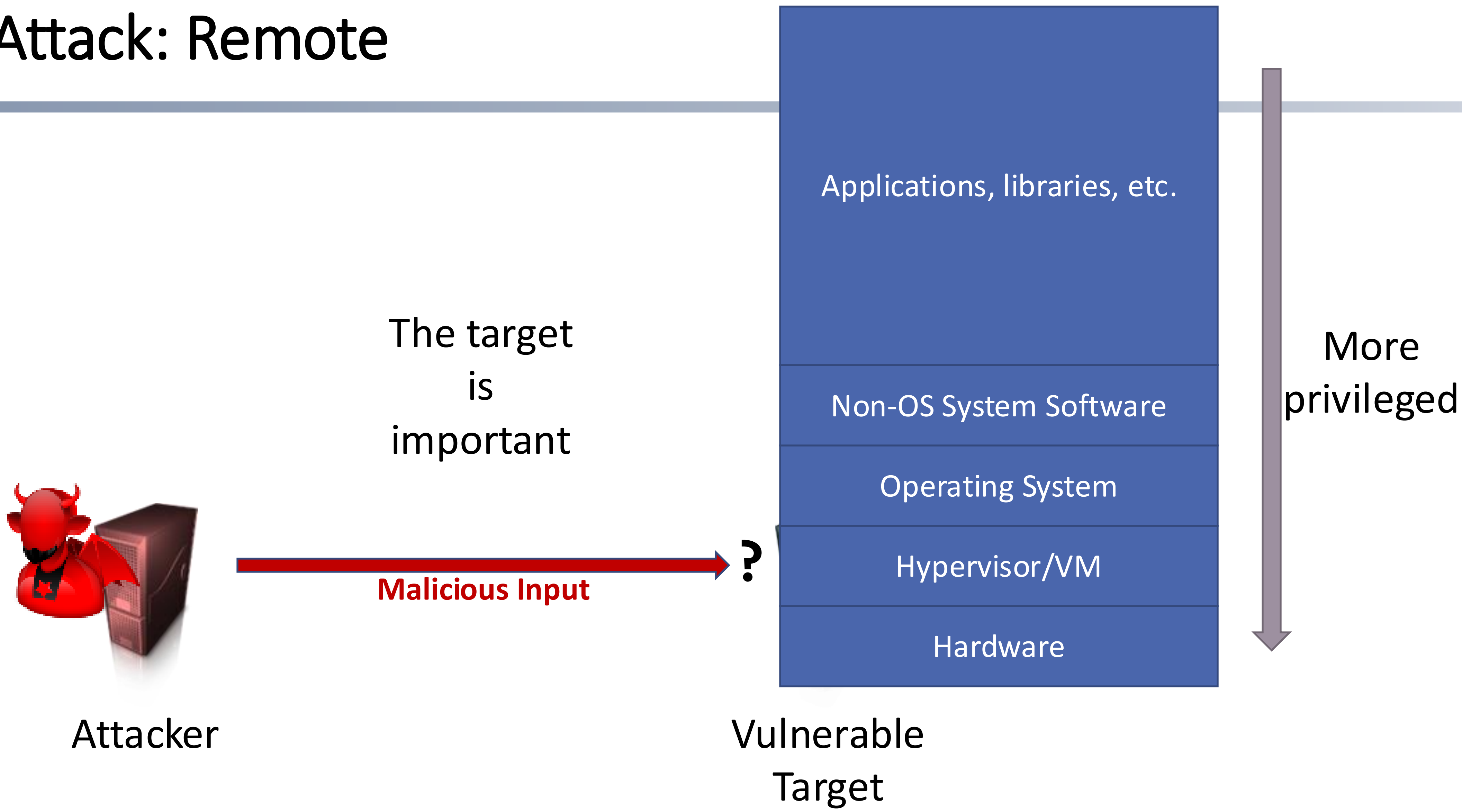
Types of Attack: Remote

Types of Attack: Remote

The target
is
important

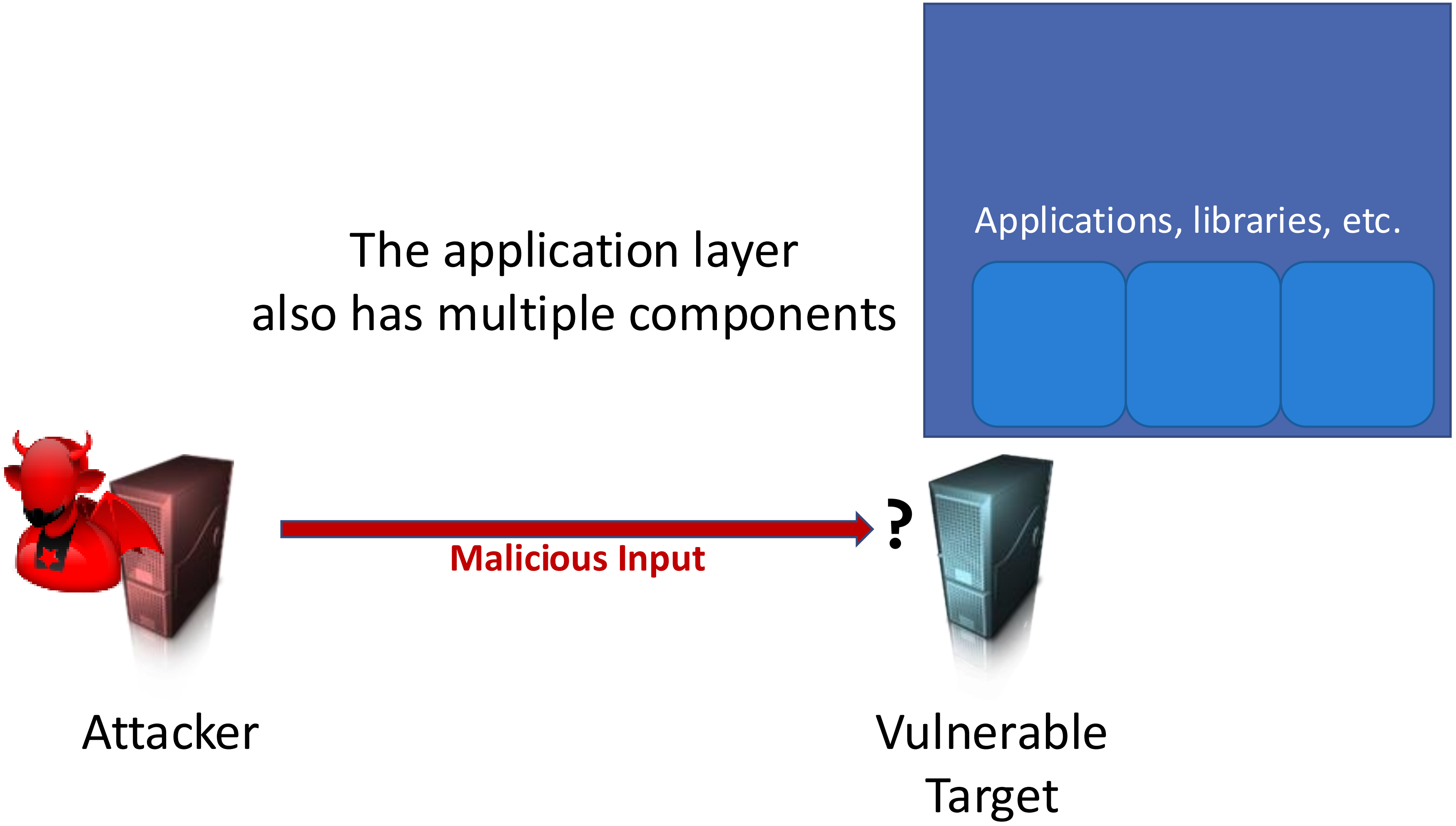


Types of Attack: Remote



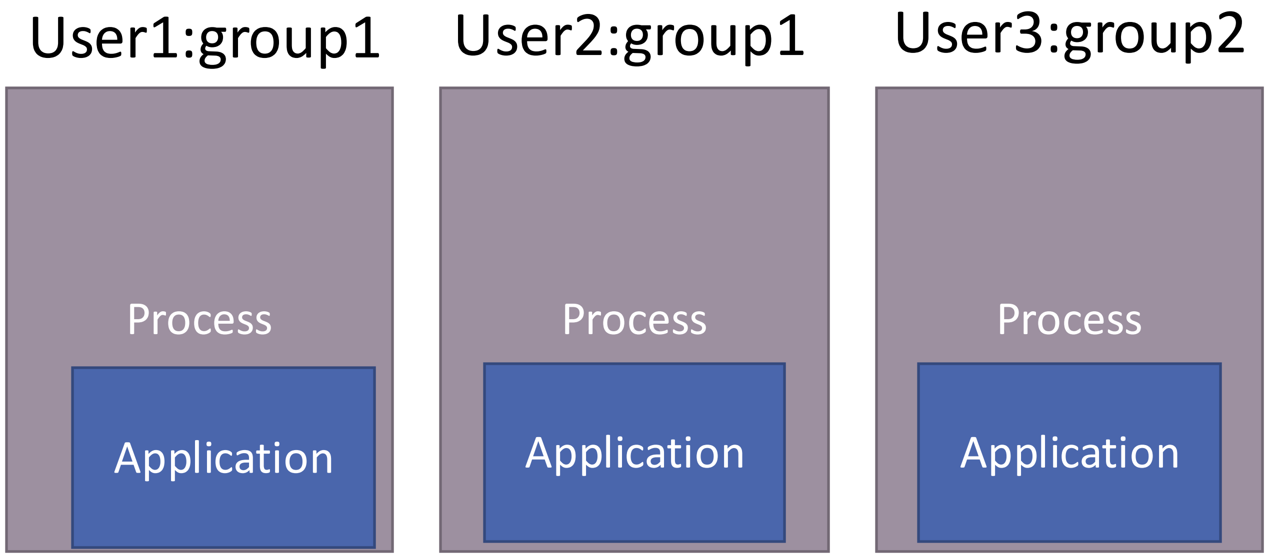
Types of Attack: Remote

The application layer
also has multiple components



Processes and Permissions

- Example shows UNIX-style permissions

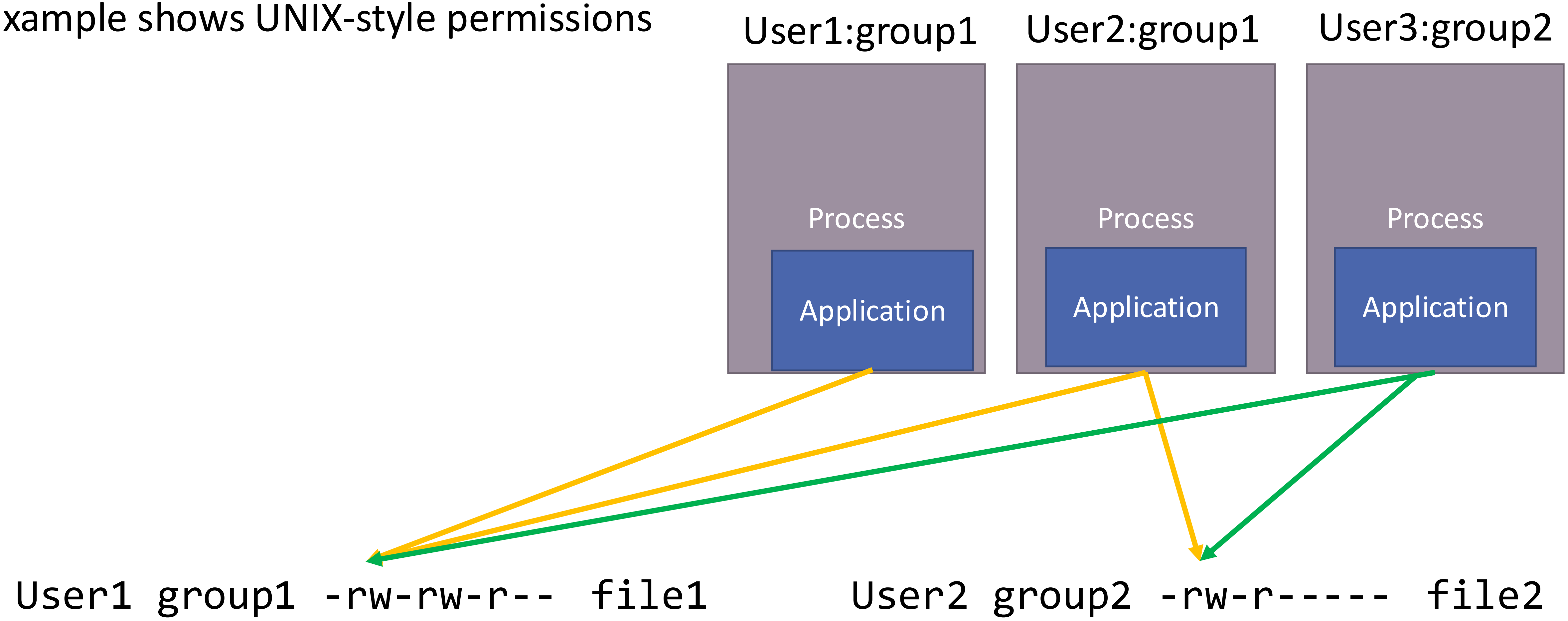


`User1 group1 -rw-rw-r--  file1`

`User2 group2 -rw-r-----  file2`

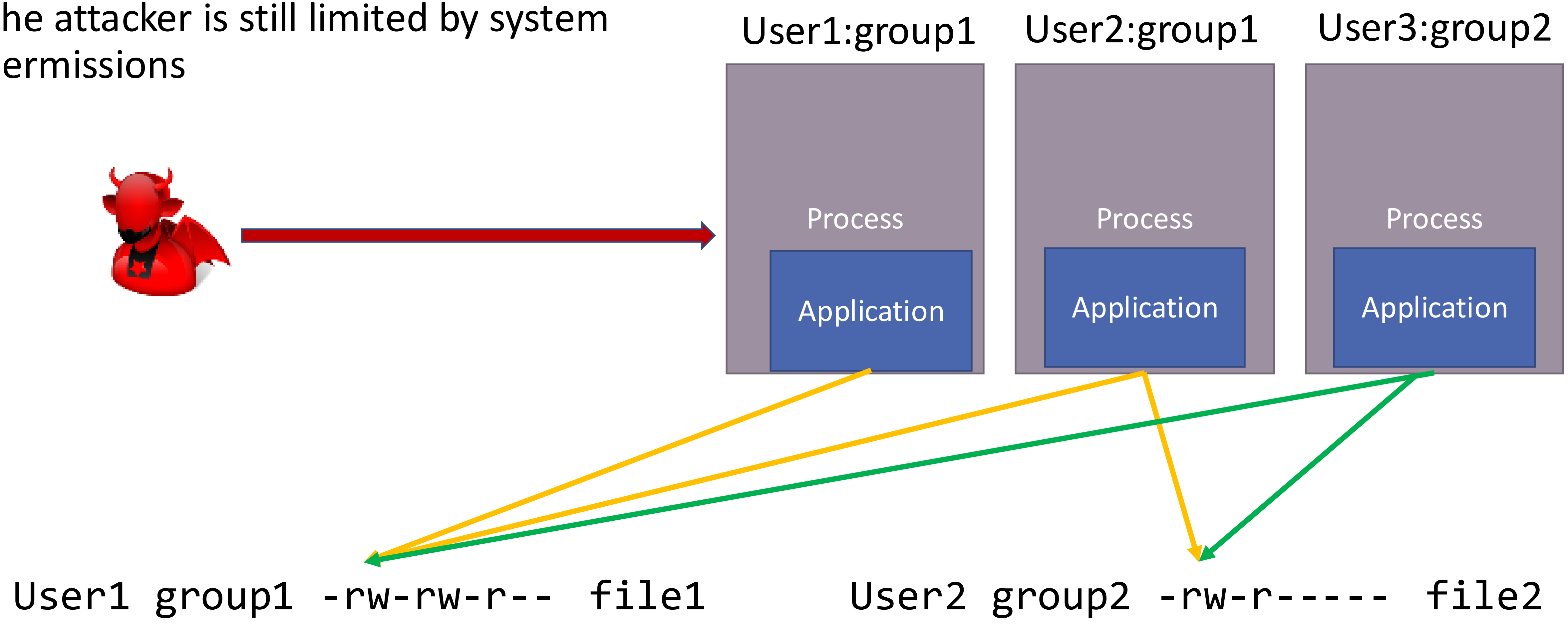
Processes and Permissions

- Example shows UNIX-style permissions



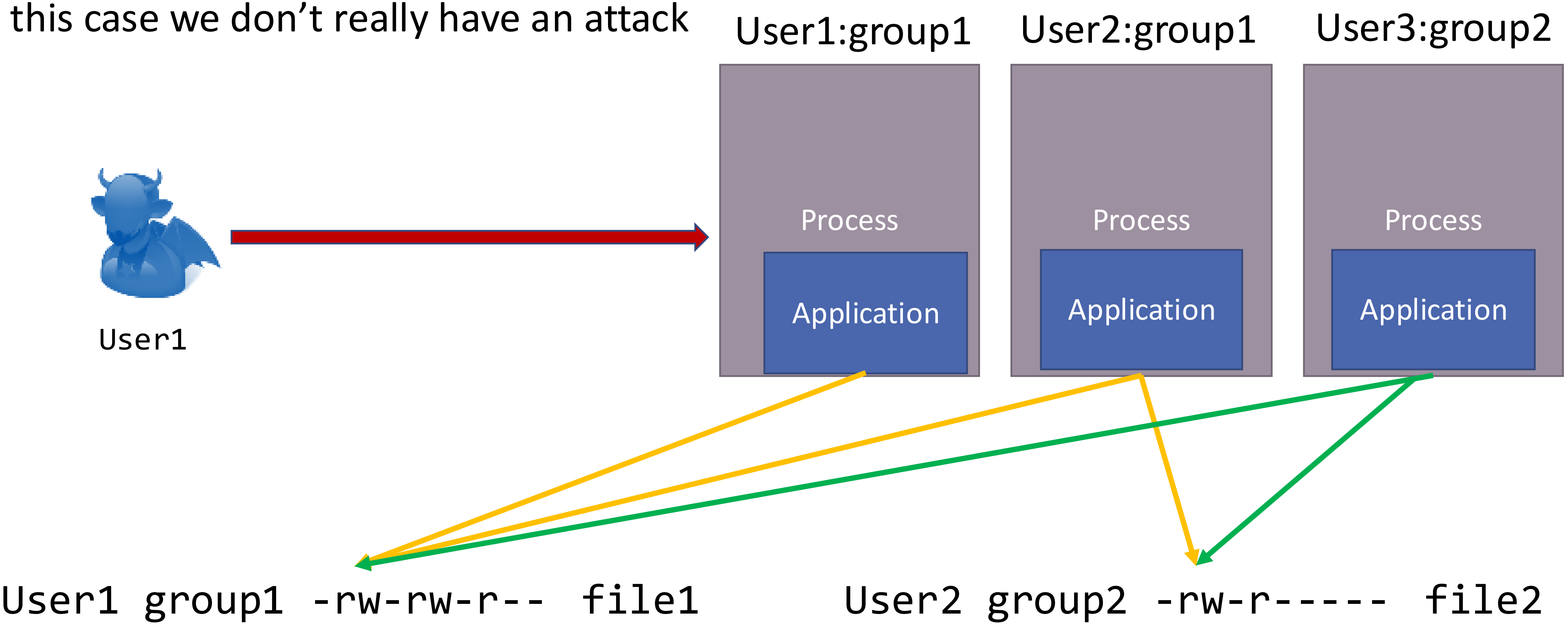
Processes and Permissions

- The attacker is still limited by system permissions



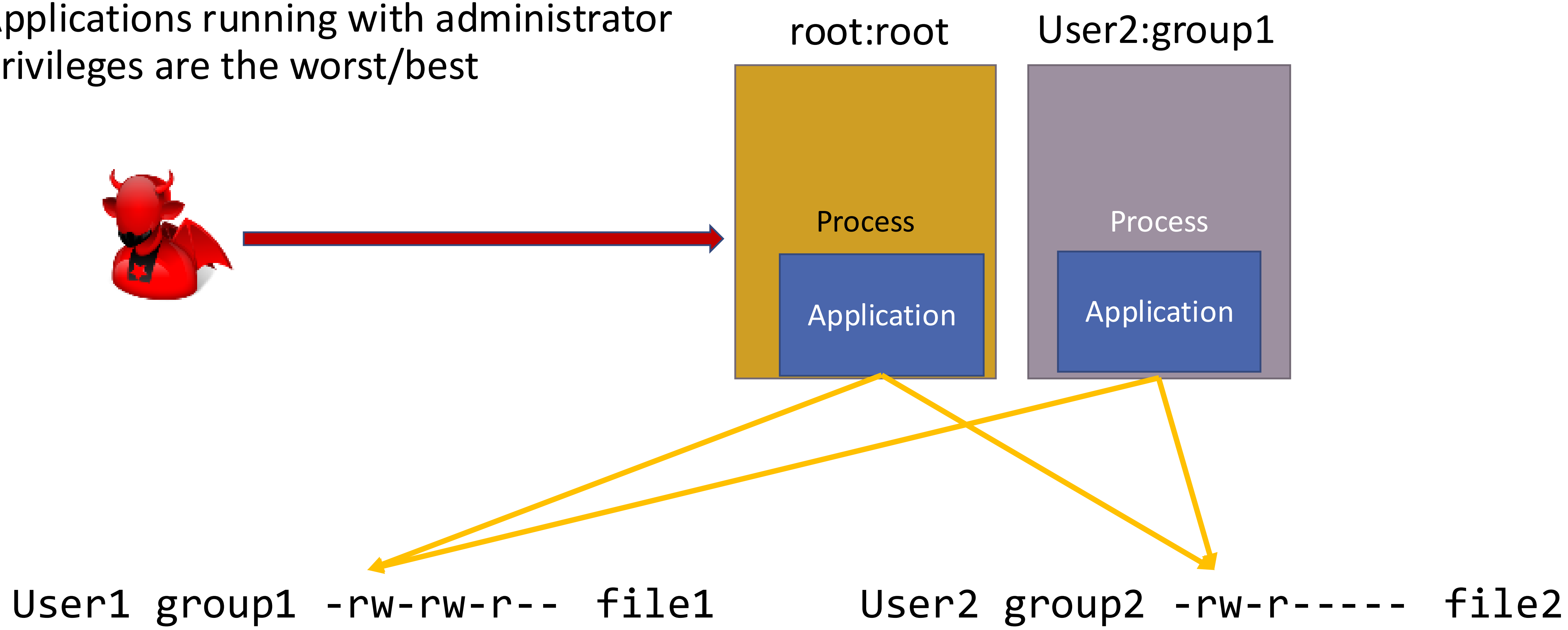
Processes and Permissions

- In this case we don't really have an attack



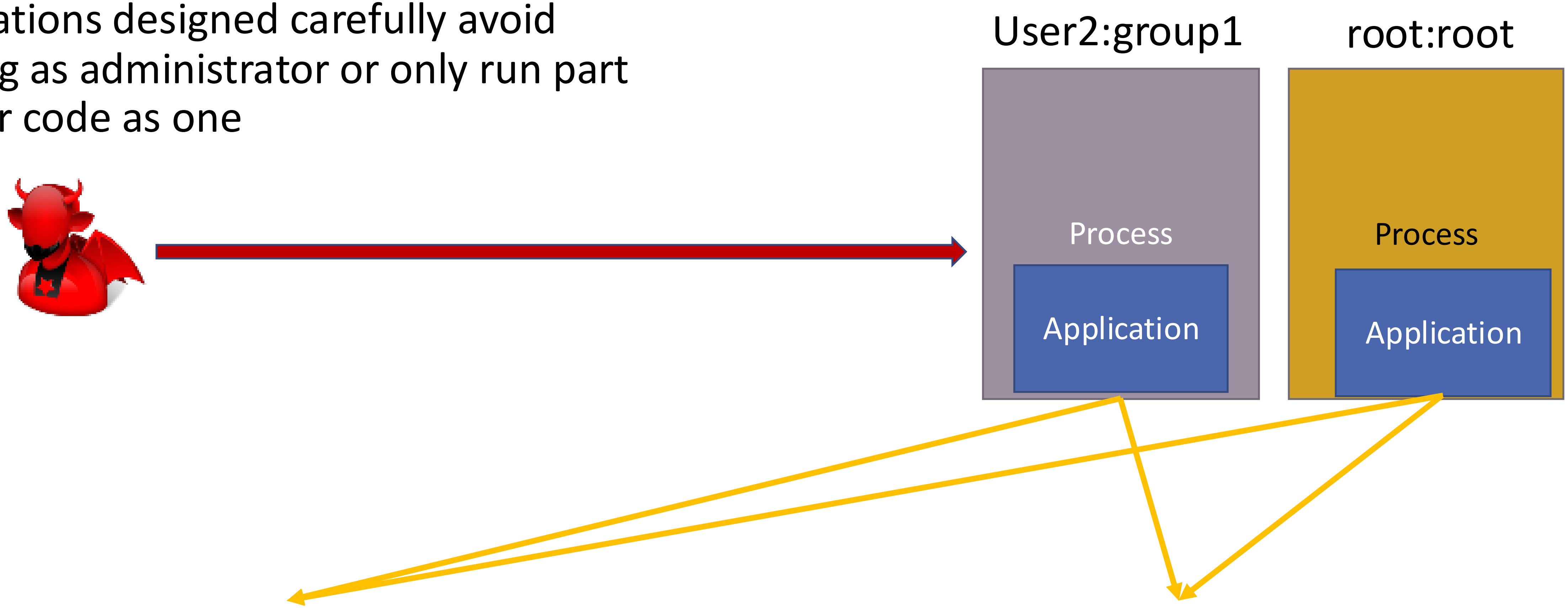
Processes and Permissions

- Applications running with administrator privileges are the worst/best



Processes and Permissions

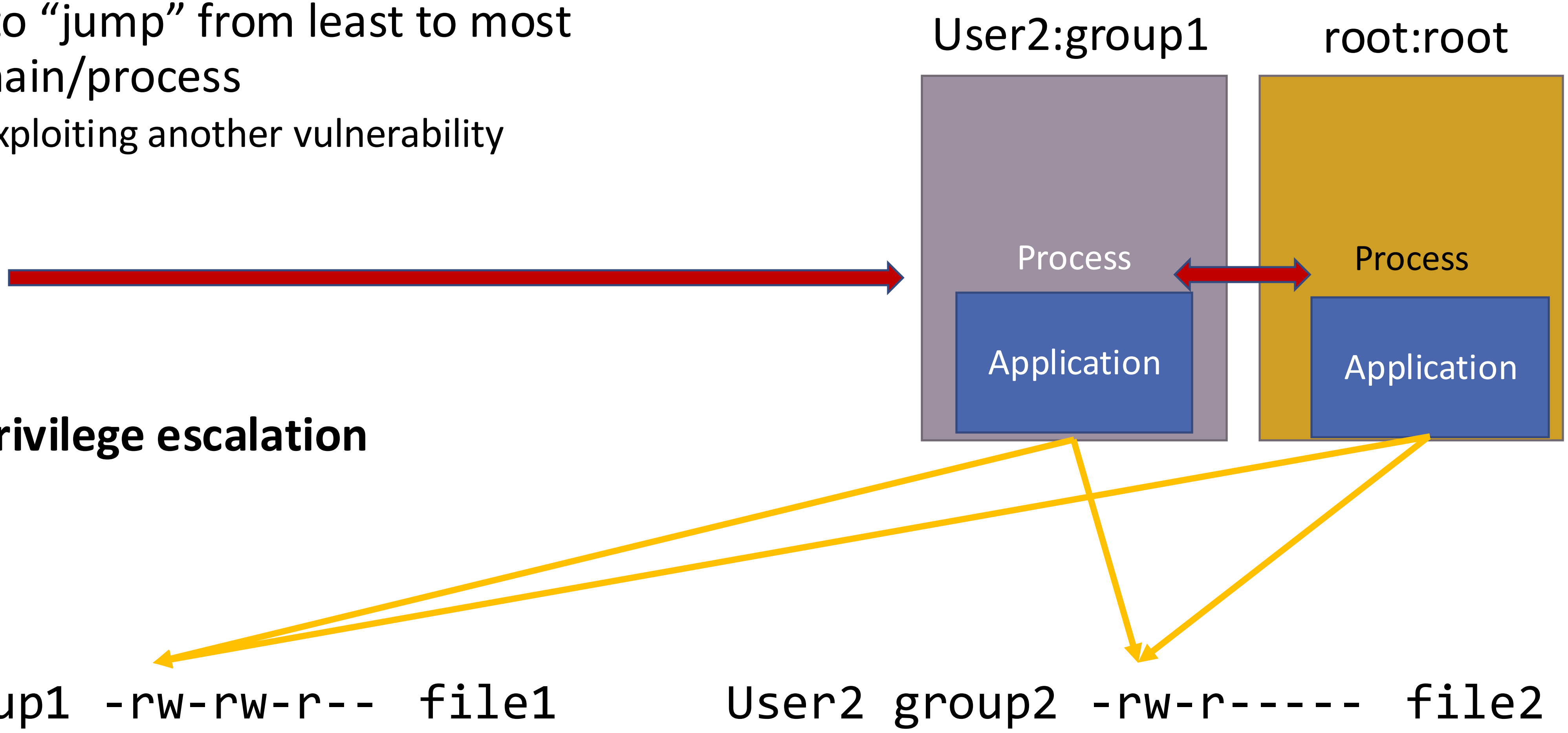
- Applications designed carefully avoid running as administrator or only run part of their code as one




```
User1 group1 -rw-rw-r--  file1        User2 group2 -rw-r-----  file2
```

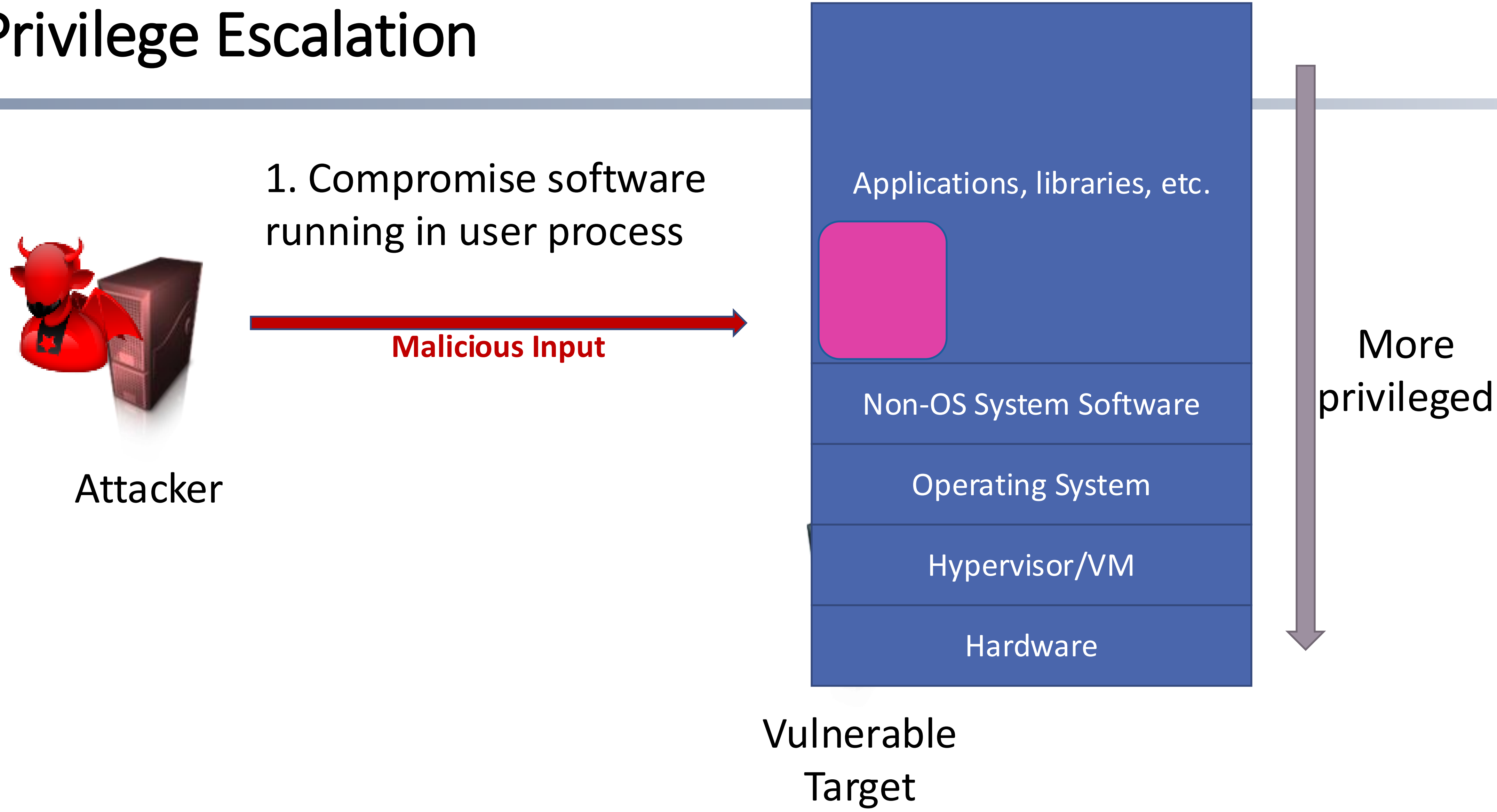
Processes and Permissions

- Attackers aim to “jump” from least to most privileged domain/process
  - Example: by exploiting another vulnerability



- This is called **privilege escalation**

Typical Privilege Escalation

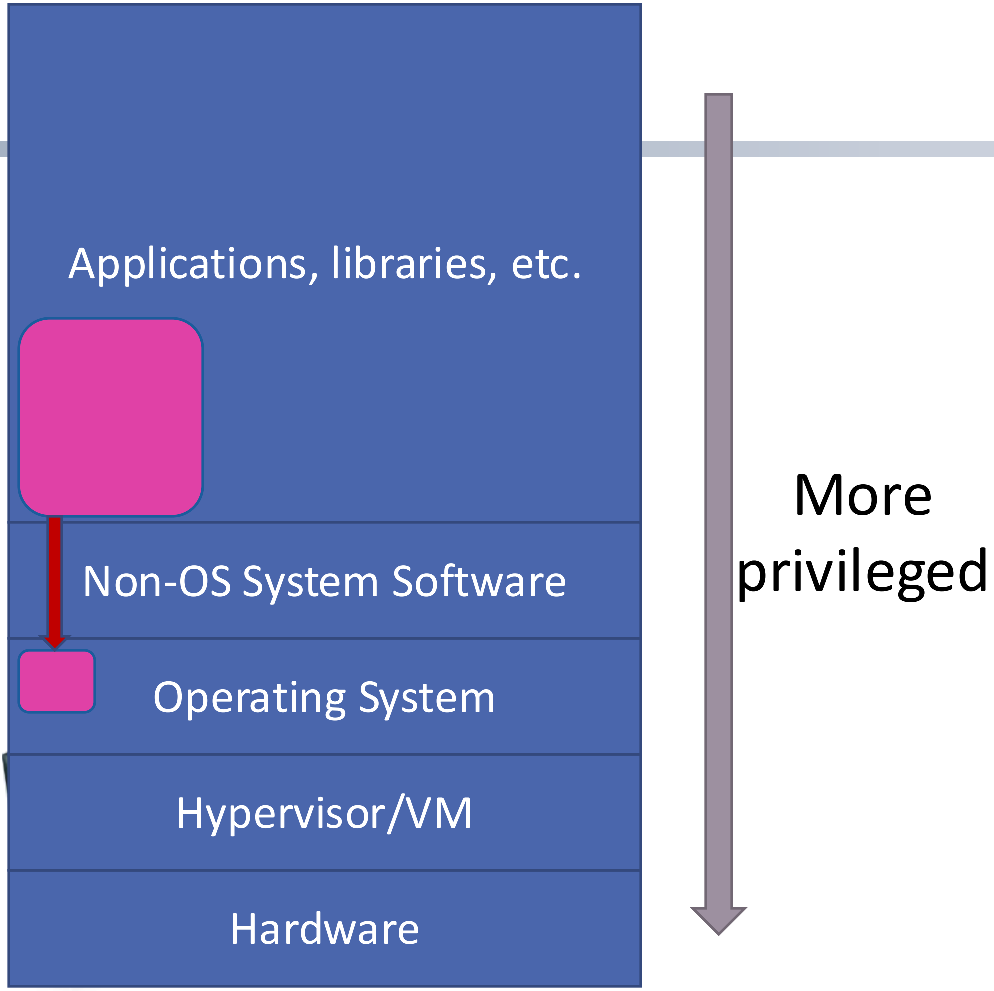


Typical Privilege Escalation

1. Compromise software running in user process

Malicious Input

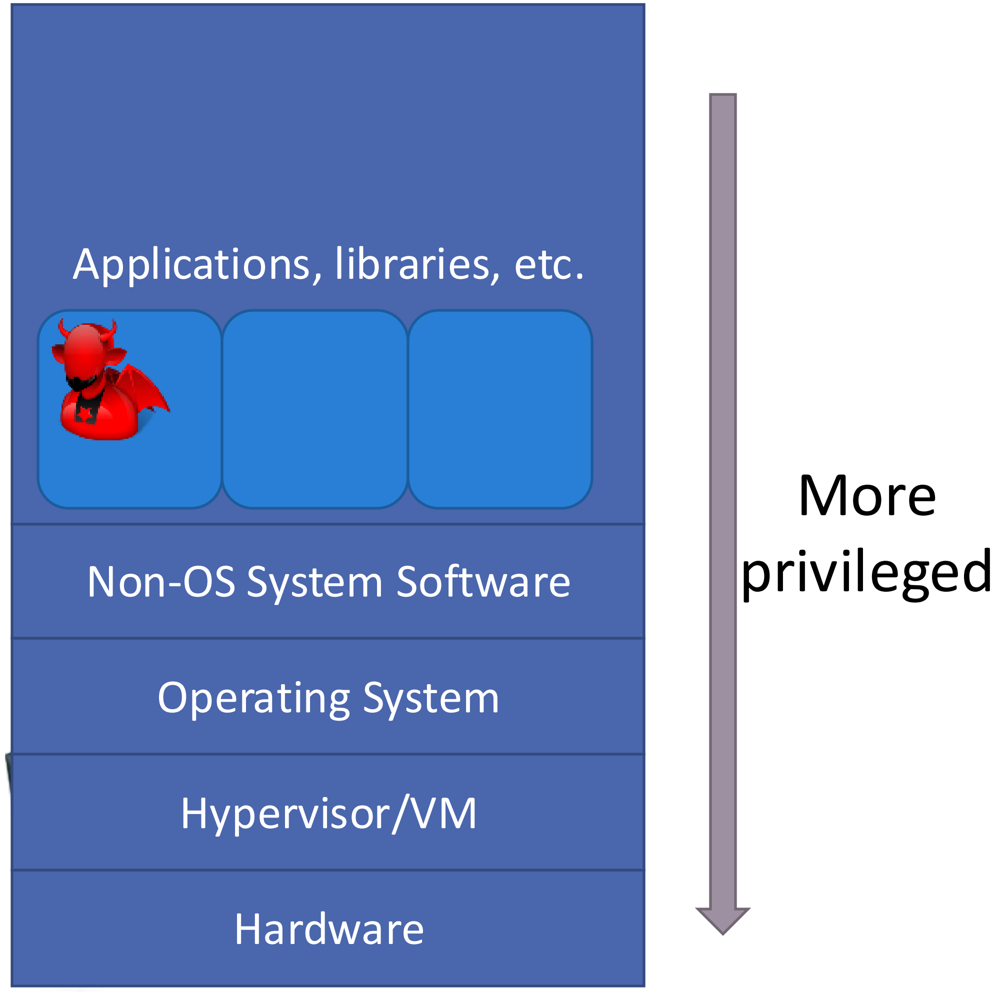
2. Compromise operating system



Vulnerable Target

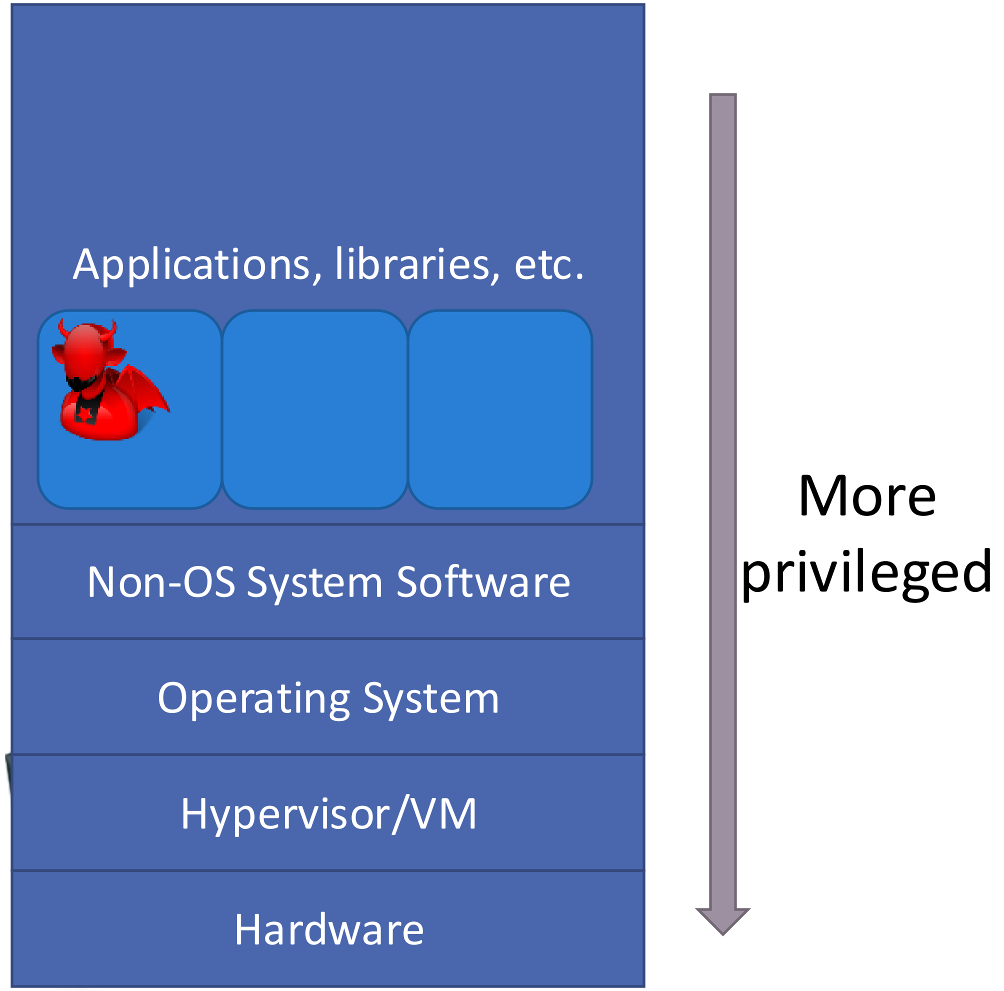
Types of Attack: Local

- The attacker already has access to the system but is limited by the access control in place
  - Example: non-admin user in a UNIX-based system



Types of Attack: Local

- The attacker already has access to the system but is limited by the access control in place
  - Example: non-admin user in a UNIX-based system
- Goal: elevate privileges
  - Exploit a vulnerability in software running in process with privileges
  - Exploit a vulnerability in the layers below
  - Special case: SUID binaries



(SUID) Set User ID on Execution

- Example: Enable a user to change their own password

user:user

$ passwd

File contains user meta data (name, shell, etc.)

-rw-r--r-- 1 root root … /etc/passwd

-rw-r----- 1 root shadow … /etc/shadow

File contains hashed password

-rwxr-xr-x 1 root root … /usr/bin/passwd

passwd program for changing password

(SUID) Set User ID on Execution

- Example: Enable a user to change their own password

File contains user meta data (name, shell, etc.)

-rw-r--r-- 1 root root … /etc/passwd

-rw-r----- 1 root shadow … /etc/shadow

File contains hashed password

user:user

$ passwd

passwd program

Process running as user:user

-rwxr-xr-x 1 root root … /usr/bin/passwd

passwd program for changing password

(SUID) Set User ID on Execution

- Example: Enable a user to change their own password
- **SUID programs run as the owner**

user:user

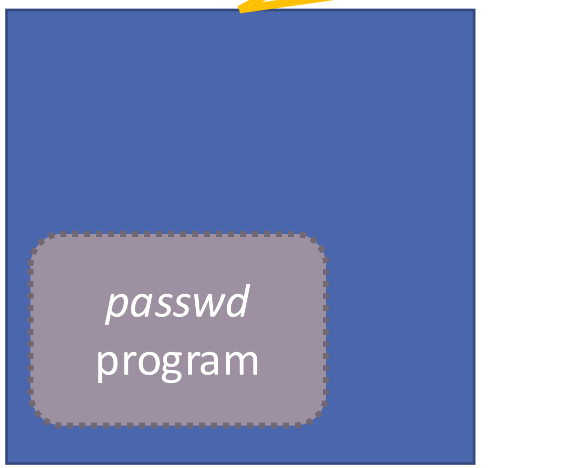
$ passwd

File contains user meta data (name, shell, etc.)

-rw-r--r-- 1 root root … /etc/passwd

-rw-r----- 1 root shadow … /etc/shadow

File contains hashed password



Process running as **root:**user

-rwx**s**-xr-x 1 root root … /usr/bin/passwd

passwd program for changing password

(SUID) Set User ID on Execution

- Example: Enable a user to change their own password
- **SUID programs run as the owner**
- Useful for performing actions that would otherwise require super-user privileges
- The program is **trusted** to perform only the actions advertised
- Also, SGID programs run as the group

- You can find SUID programs in your system using `find`

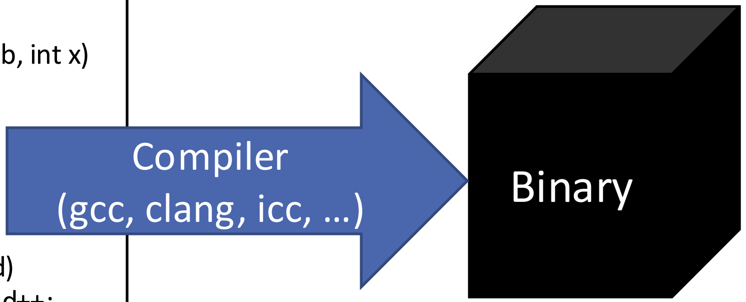
$ sudo find /usr/bin -perm -u=s

- **How can these programs be misused by attackers?**

Beyond Source Code

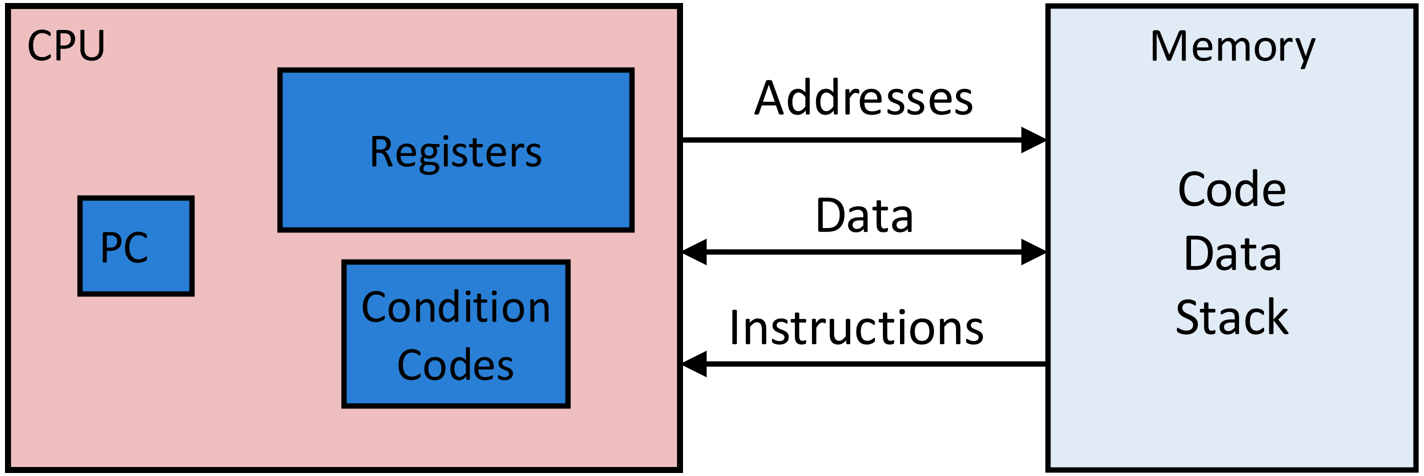
- Computer can only execute machine (binary) code
- C/C++ programs are compiled to binary code
- Binary (executable) programs are loaded to memory and executed by the operating system (OS)
  - Using a process

```
static int total;
static int total_over_threshold = 0;
static int arbitrary_theshold = 16700;

int simple_function(int a, int b, int x)
{
	int y;

	y = a*x*x + b*x;

	if (y > arbitrary_theshold)
		total_over_threshold++;

	total++;

	return total;
}
```



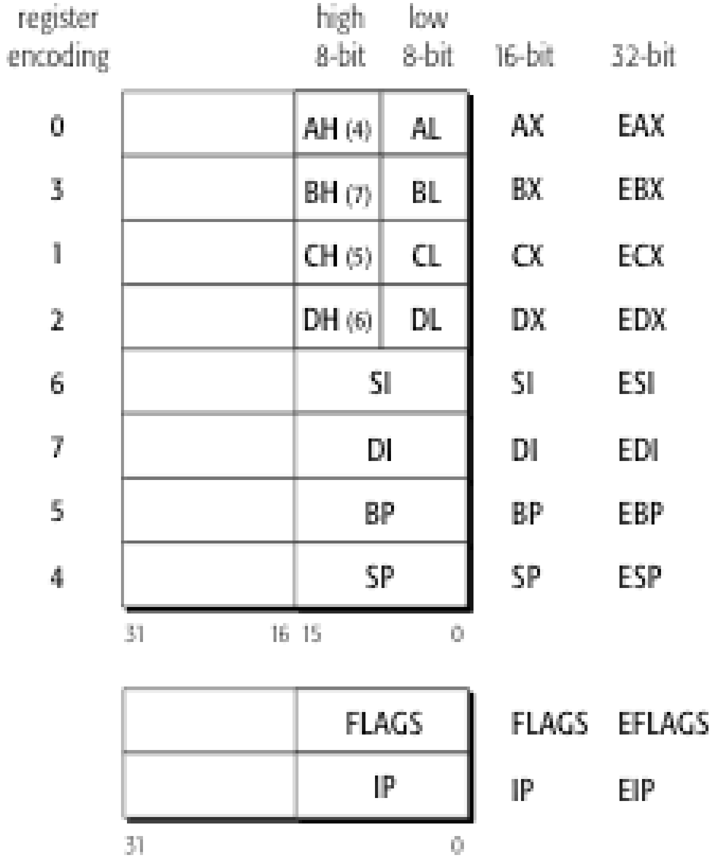
Computer Architecture Basics

- PC: Program counter
  - Address of next instruction
- Memory
  - Byte addressable array
  - Code and user data
  - Stack to support procedures
- Register file
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching



x86 Integer Registers

- General purpose registers
  - On 32-bit architectures EAX, EBX, ECX, EDX, EDI, ESI, ESP, EBP
- The instruction pointer (IP)
  - Also referred to as program counter (PC)
  - EIP on 32-bit
- FLAGS register
  - Used for control flow operations, etc.
  - EFLAGS



x86-64 Integer Registers

`%rax`	`%eax`	`%r8`	`%r8d`
`%rbx`	`%ebx`	`%r9`	`%r9d`
`%rcx`	`%ecx`	`%r10`	`%r10d`
`%rdx`	`%edx`	`%r11`	`%r11d`
`%rsi`	`%esi`	`%r12`	`%r12d`
`%rdi`	`%edi`	`%r13`	`%r13d`
`%rsp`	`%esp`	`%r14`	`%r14d`
`%rbp`	`%ebp`	`%r15`	`%r15d`

x86-64 Integer Registers

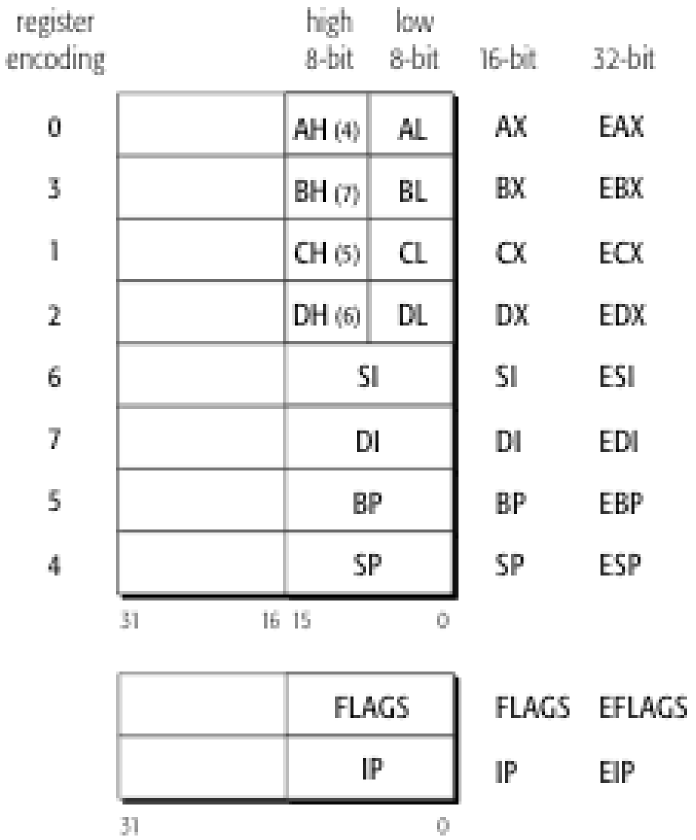
Can reference low-order bytes too

- d suffix for lower 32-bits (r8d)
- w suffix for lower 16-bits (r8w)
- b suffix for lower 8-bits (r8b)

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

Typical Register Uses

- EAX: accumulator
- EBX : Pointer to data
- ECX: Counter for string operations and loops
- EDX: I/O Operations
- EDI: Destination for string operations
- ESP: Stack pointer
- EBP: Frame pointer



Assembly Syntax

- Intel: OP dest, src

- AT&T: OP src, dest

- Unix systems prefer AT&T
  - We are going to use the same as the GNU assembler (gas syntax)

Assembly Instructions

- **pushq:** push quad word to stack
- **movq:** Move quad word
- **imull:** Signed multiply long
- **addl:** Add long

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -20(%rbp)
movl    %esi, -24(%rbp)
movl    %edx, -28(%rbp)
movl    -20(%rbp), %eax
imull   -28(%rbp), %eax
movl    %eax, %edx
movl    -24(%rbp), %eax
addl    %edx, %eax
imull   -28(%rbp), %eax
```

Operand Sizes

Instructions include a suffix that indicates the size of the operand(s)

Register operands are prefixed with a **%**

push$\boxed{\text{q}}$ %rax

- Intel syntax does not include a suffix, size depends on the size of the operand
- If omitted size depends on arch

Register operands must match size
For example,
- quad → rax
- long → eax
- word → ax
- byte → ah or al

Memory Operands

- Parentheses indicate a memory operand
- Each memory address can be defined as: **Base+Index*Scale+Disp**
  - In AT&T syntax: disp(base, index, scale)
    - disp, index, and scale are optional

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -20(%rbp)
movl    %esi, -24(%rbp)
movl    %edx, -28(%rbp)
movl    -20(%rbp), %eax
imull   -28(%rbp), %eax
movl    %eax, %edx
movl    -24(%rbp), %eax
addl    %edx, %eax
imull   -28(%rbp), %eax
```

Part 2. Smashing the Stack

Buffer Overflows

- Writing outside the boundaries of a buffer
  - Buffers are arrays of bytes, integers, structs, etc.
  - Spatial violation
- Common programmer errors that lead to it …
  - Insufficient input checks/wrong assumptions about input
  - Unchecked buffer size
  - Integer overflows

Example

```
char buf[16];

strcpy(buf, str);

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

return strlen(buf);
```

BO Variations

```c
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

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

        return 0;
}
```

```c
int mytest(char *str)
{
        char *buf = malloc(16);

        strcpy(buf, str);

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

        return 0;
}
```

```c
char buf[16];

int mytest(char *str)
{
        strcpy(buf, str);

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

        return 0;
}
```

BO Variations

Stack Buffer Overflow

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

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

        return 0;
}
```

Heap Buffer Overflow

```
int mytest(char *str)
{
        char *buf = malloc(16);

        strcpy(buf, str);

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

        return 0;
}
```

Global Buffer Overflow

```
char buf[16];

int mytest(char *str)
{
        strcpy(buf, str);

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

        return 0;
}
```

Buffer Overflows

- Can happen when calling common string and buffer functions
  - strcpy(), strcat(), memcpy(), memset(), memmove(), etc.
- But not limited to those functions
  - Can also happen with functions as read(), fread(), gets(), fgets(), etc.

Buffer Overflows

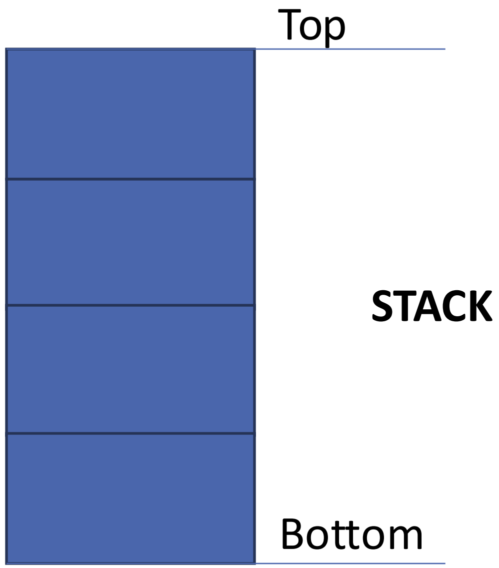
- Can happen when calling common string and buffer functions
  - strcpy(), strcat(), memcpy(), memset(), memmove(), etc.
- But not limited to those functions
  - Can also happen with functions as read(), fread(), gets(), fgets(), etc.
- Custom data copying code can also suffer

```
static void my_copy(char *dst, const char *src)
{
        char *dp, *sp;
        for (sp = src, dp = dst; *sp != '\0'; sp++, dp++) {
                *dp = *sp;
        }
        *dp = '\0’;
}
```

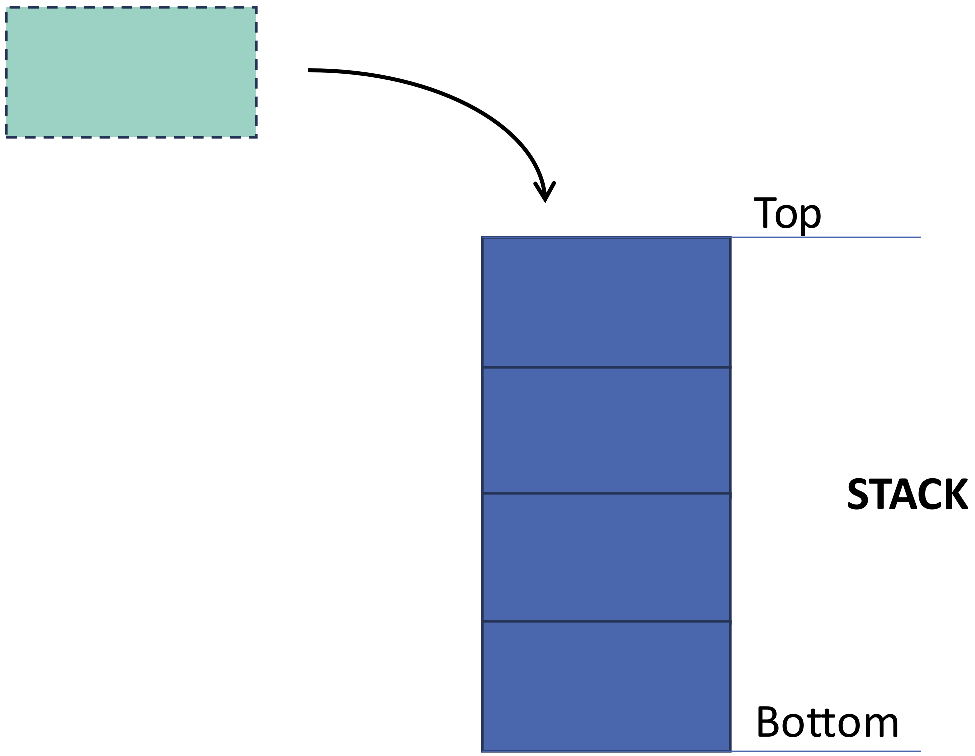
How Do Function Calls Work

Stack Data Structure

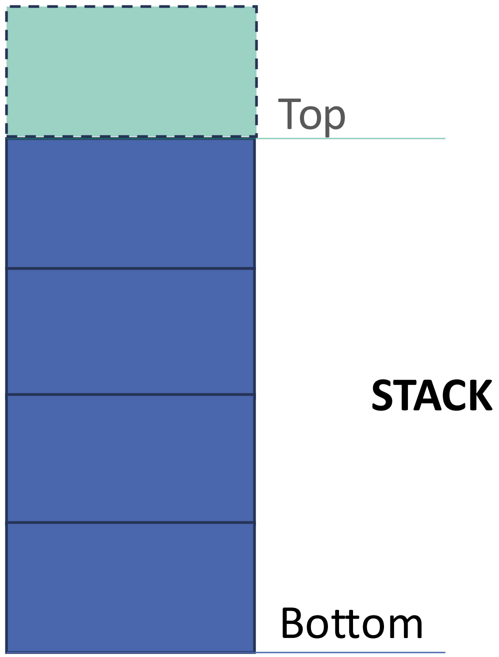
- Stack plays a crucial role in supporting functions
  - Follows last-in first-out semantic



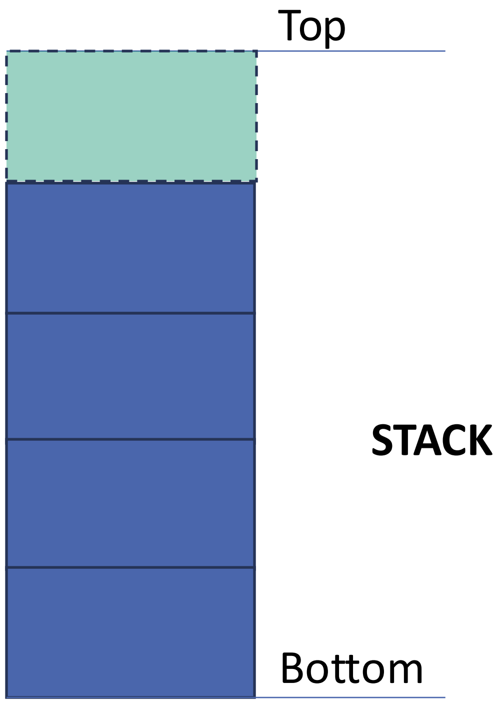
Stack Operation Push



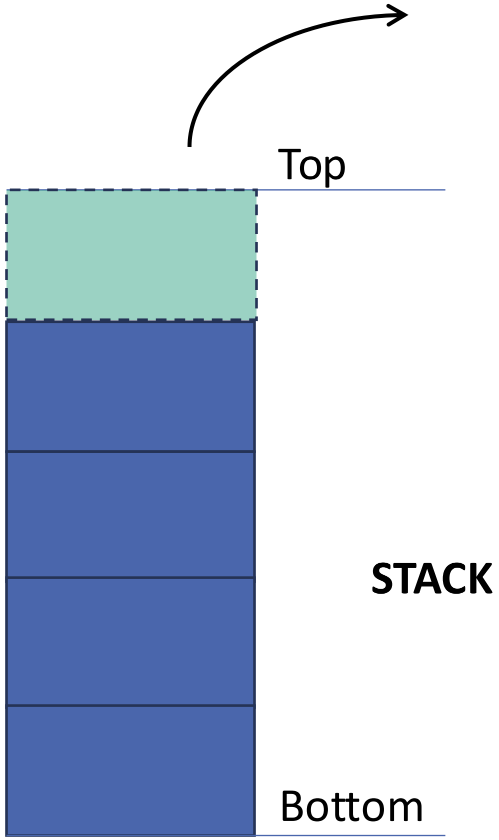
Stack Operation Push



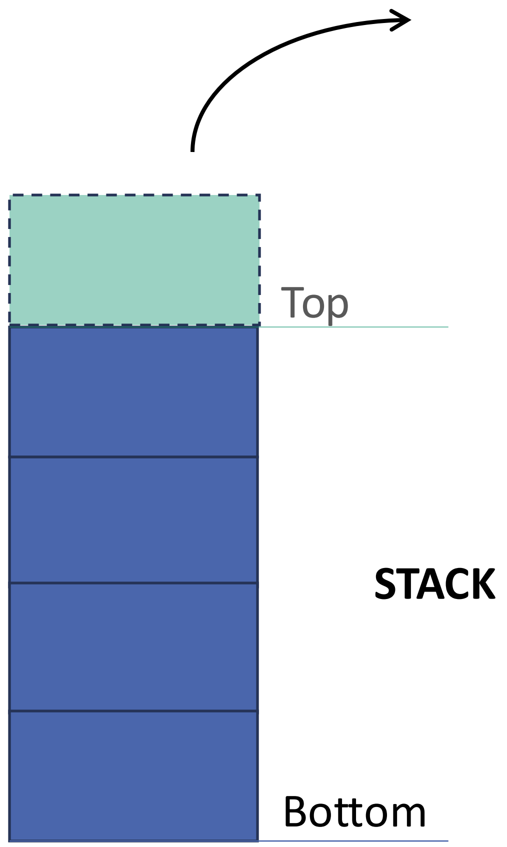
Stack Operation Push



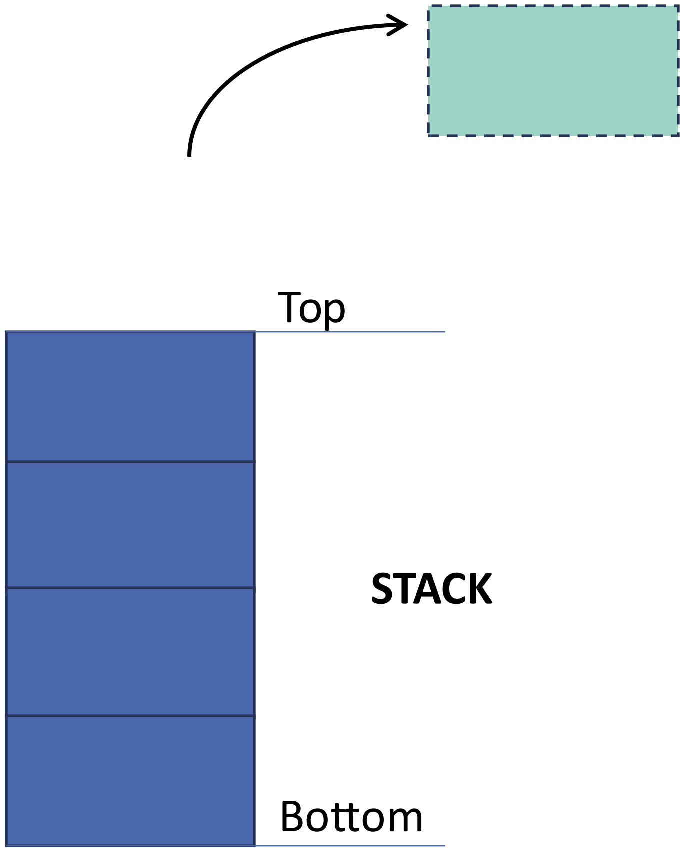
Stack Operation Pop



Stack Operation Pop

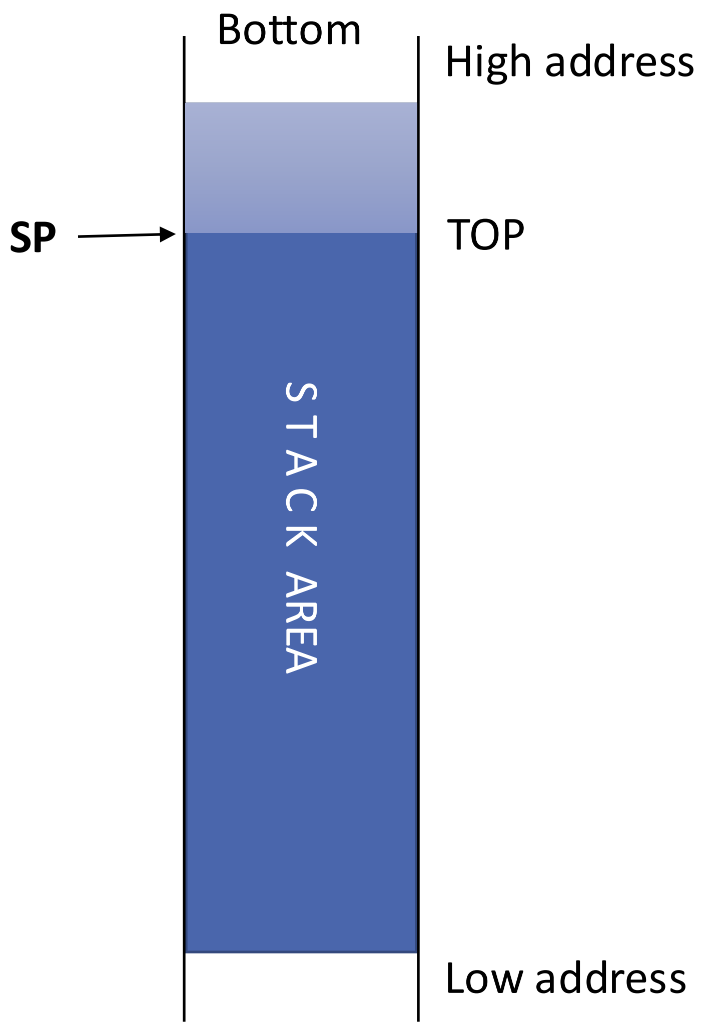


Stack Operation Pop



The Stack Pointer (SP)

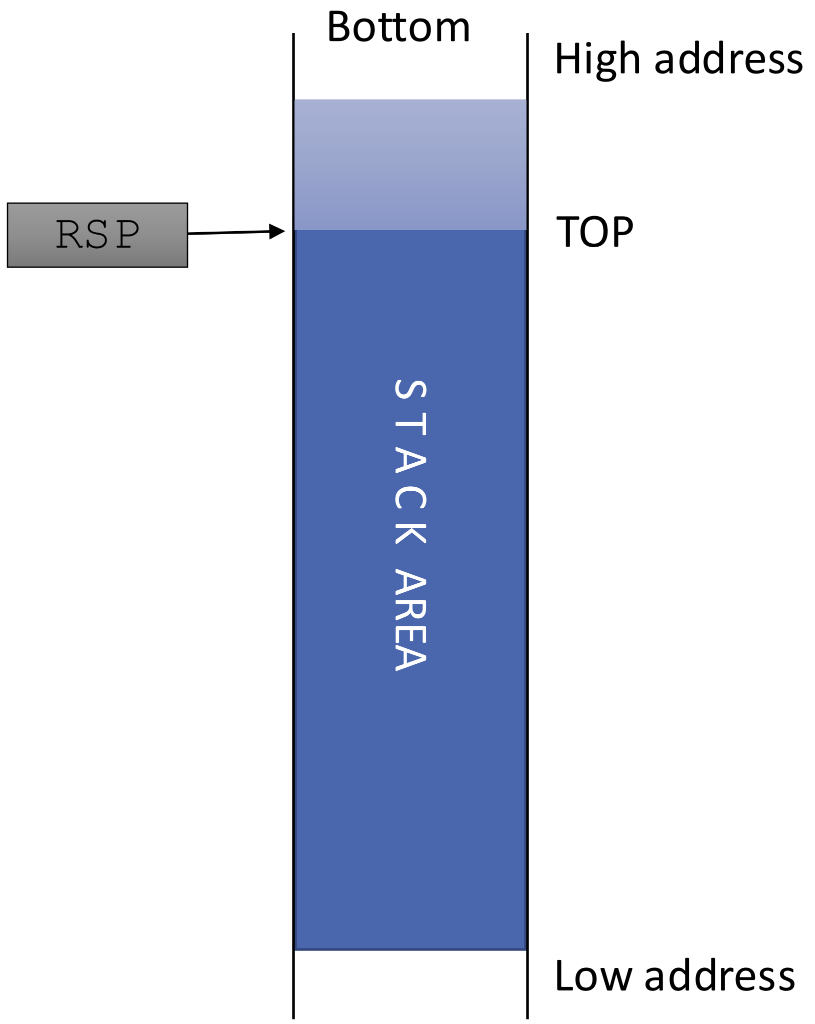
The stack pointer points to the first element in the stack (the top).



The Stack Pointer (SP)

The stack pointer points to the first element in the stack (the top).

Usually the RSP/ESP register is used to store the SP.

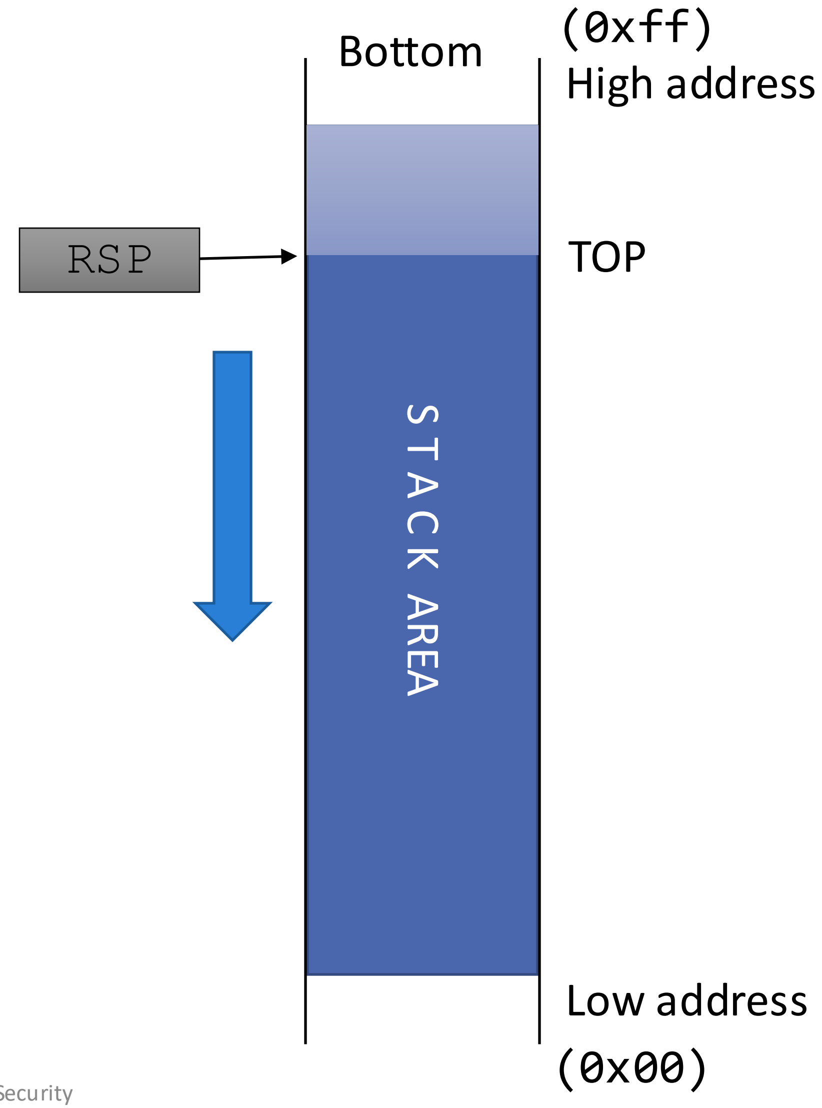


The Stack Pointer (SP)

The stack pointer points to the first element in the stack (the top).

Usually the RSP/ESP register is used to store the SP.

The stack grows towards lower addresses



Simple Function Call

C

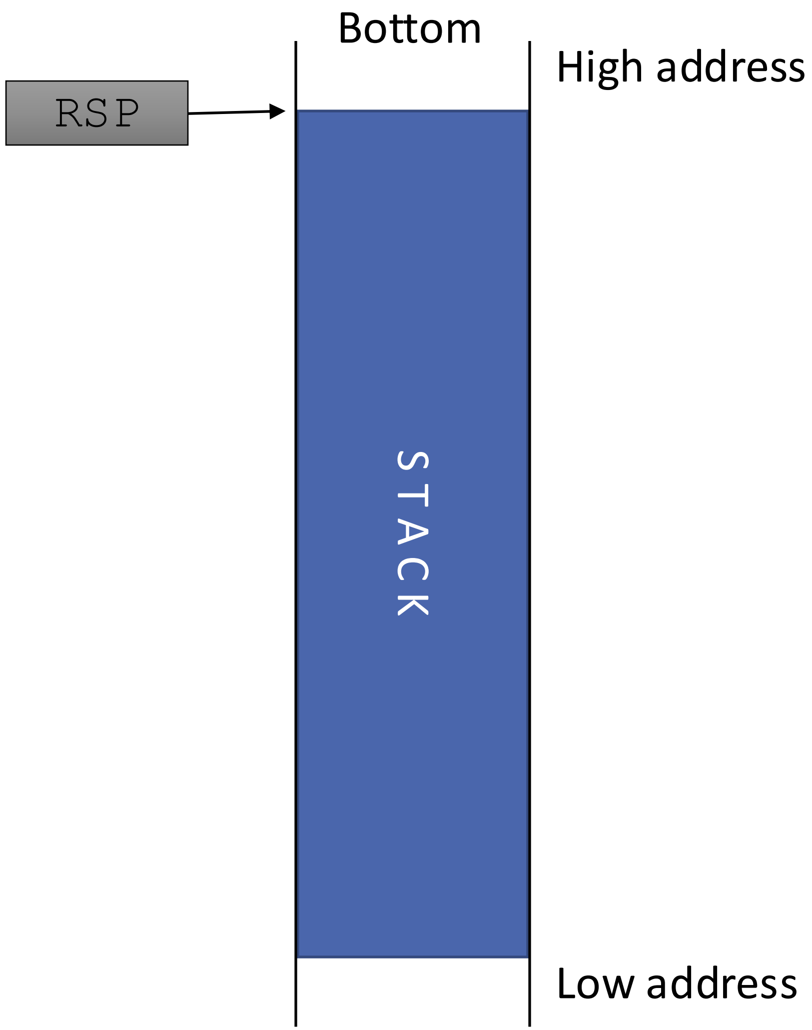
```
…
F1();
…
```

```
void F1()
{
      ….
}
```

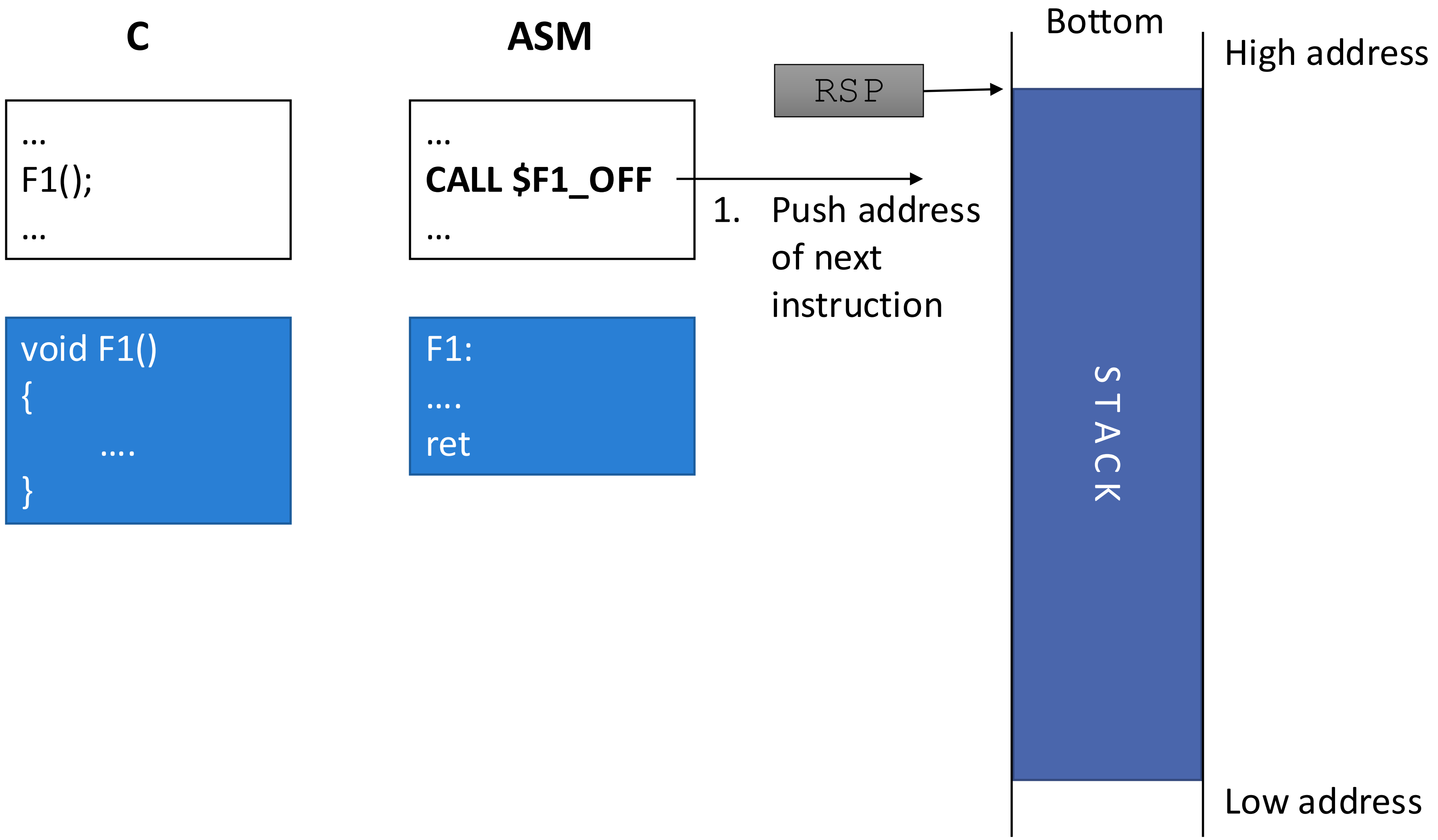
ASM

```
…
CALL $F1_OFF
…
```

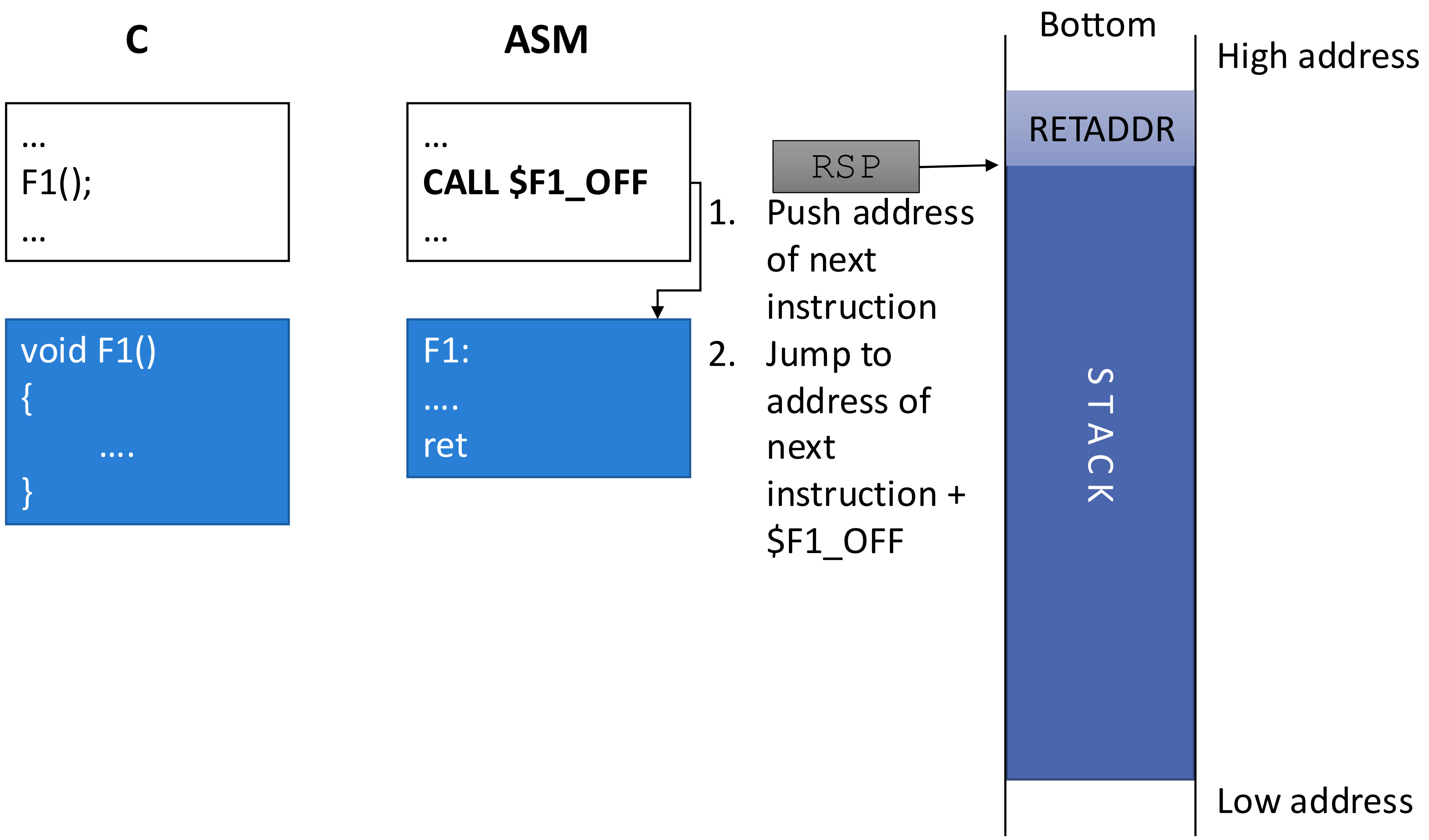
```
F1:
….
ret
```



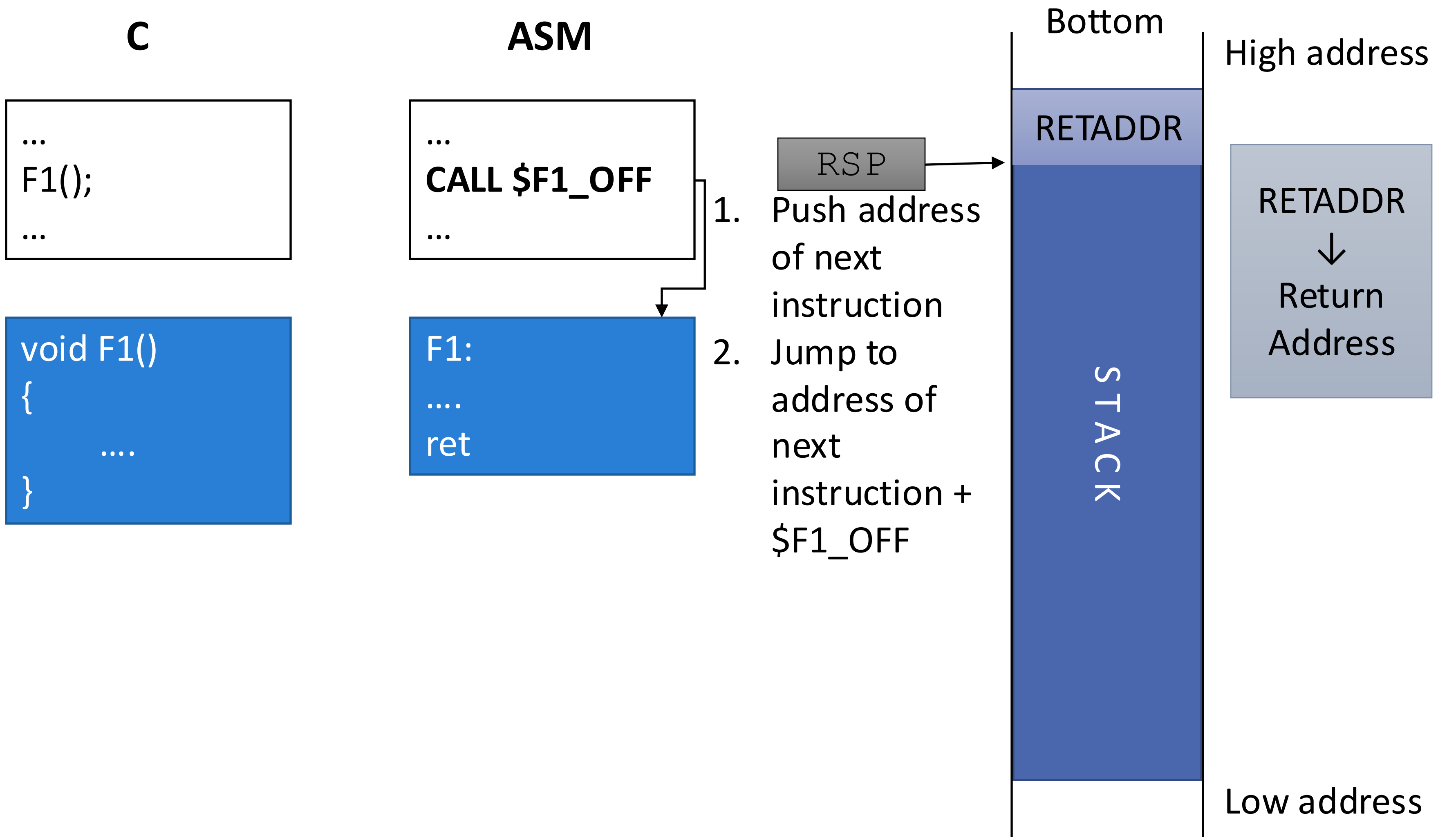
Simple Function Call



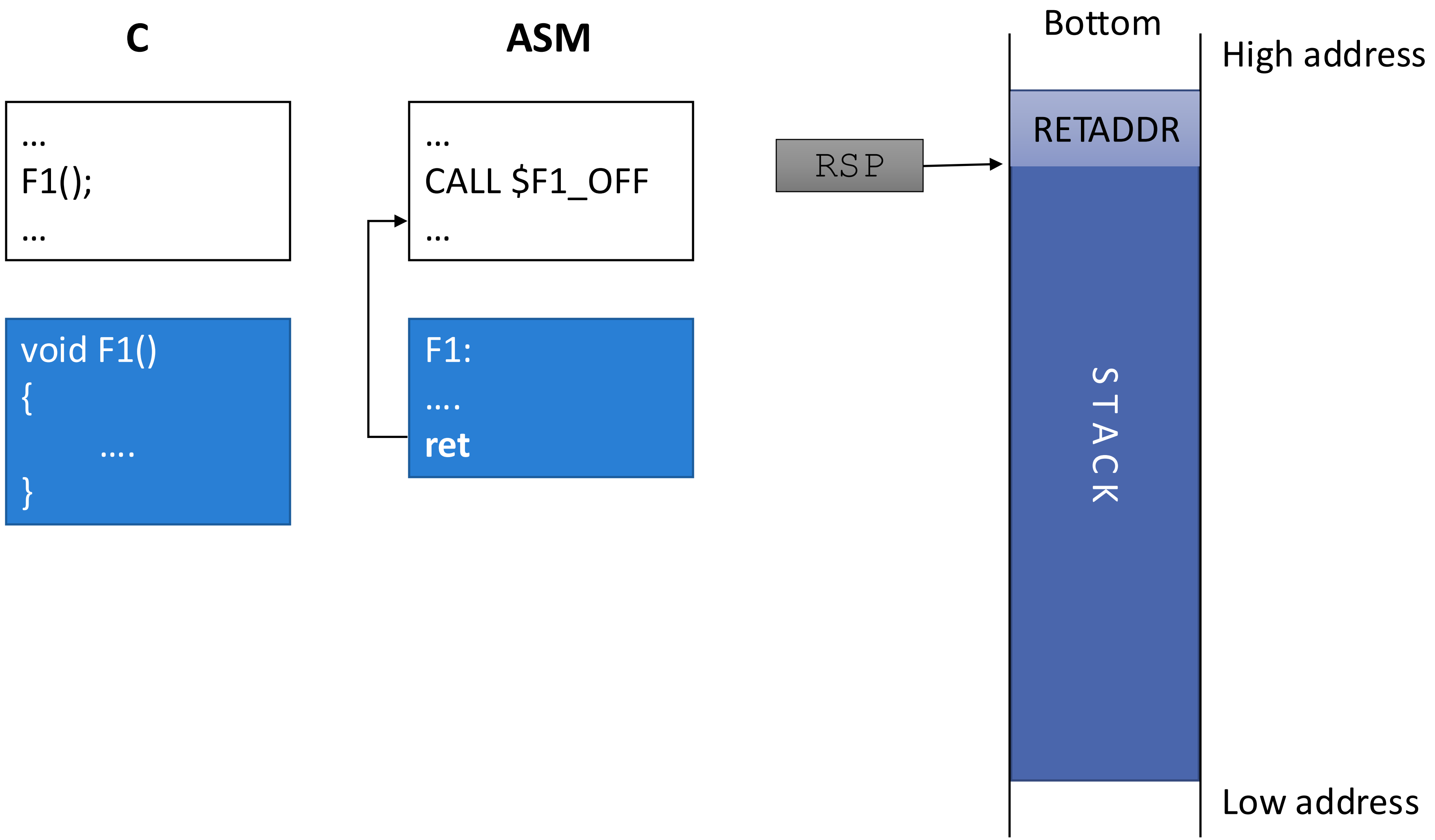
Simple Function Call



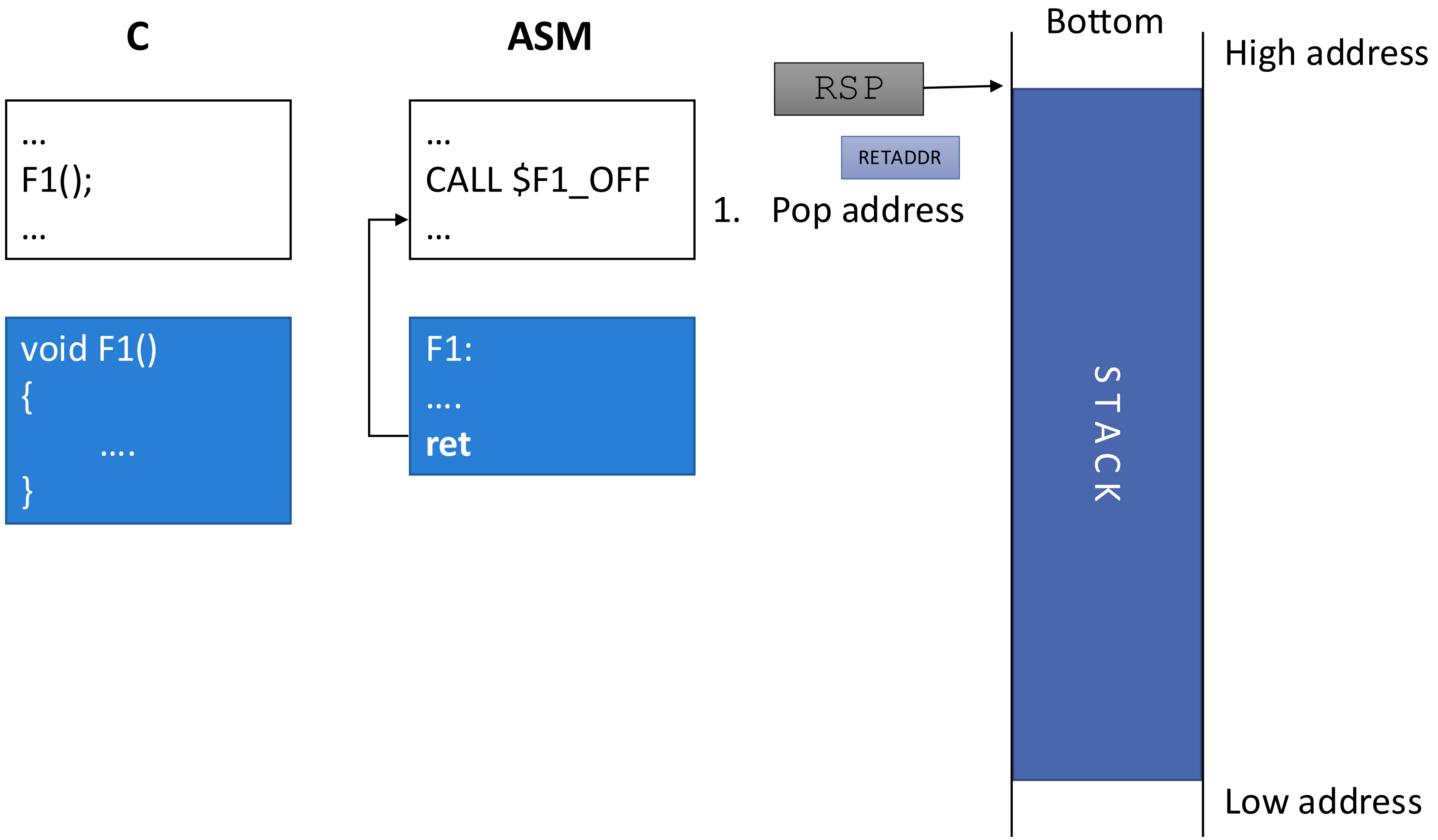
Simple Function Call



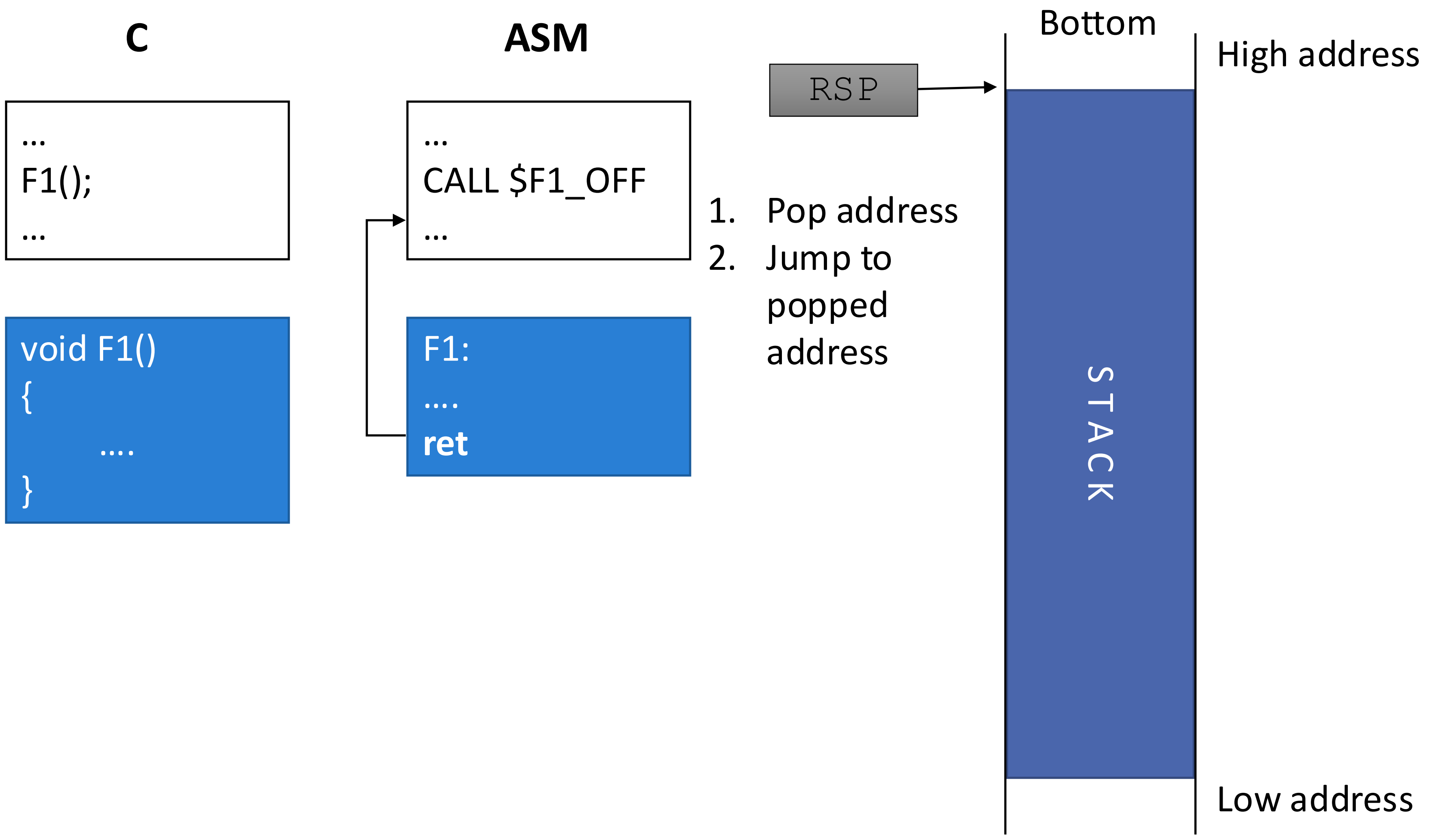
Simple Function Return



Simple Function Return



Simple Function Return



1. Pop address
2. Jump to popped address

Function Calls and Returns

Calling a function (the callee)

- CALL instruction
  - Pushes next_ins_addr on stack and transfers control to address described by operand
- Most common syntax: CALL OFFSET
  - Target is next_ins_addr + OFFSET

Returning to caller

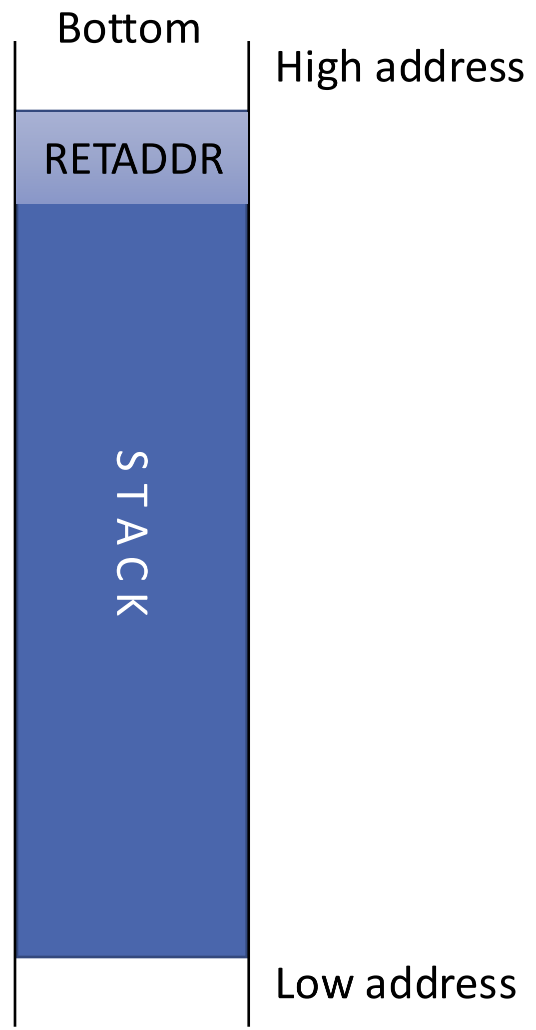
- RET instruction
  - Pop return address from stack and transfers control to it

```
CALL tgt → push next_ins_addr; jmp tgt
RET → pop retaddr; jmp retaddr
```

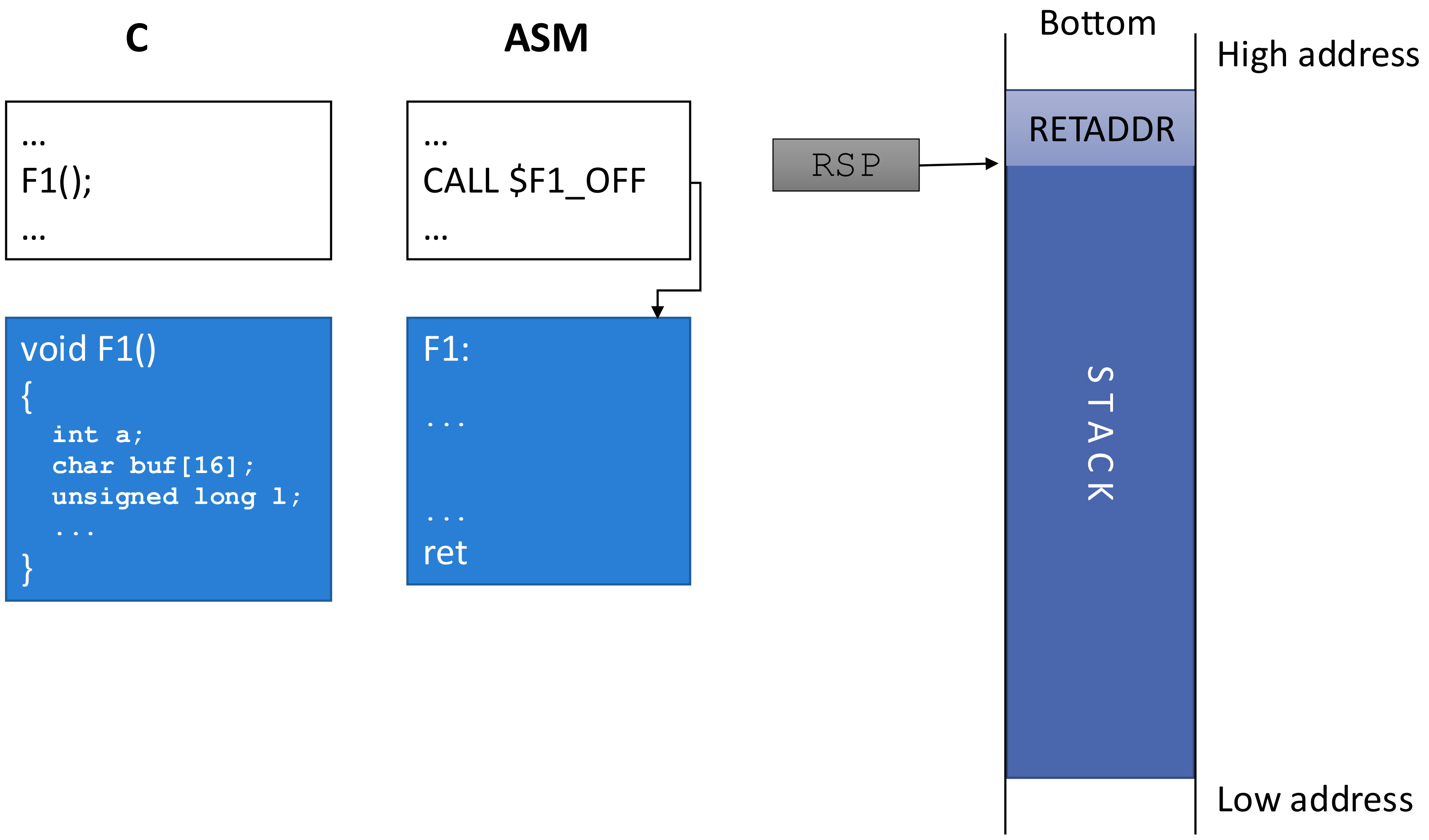
`call` and `ret` implicitly use the SP

The Stack Is Used…

- …to store the return address of caller functions
  - Code pointers!
- …to store local variables
  - Aka stack variables



Local Variables



Local Variables

C

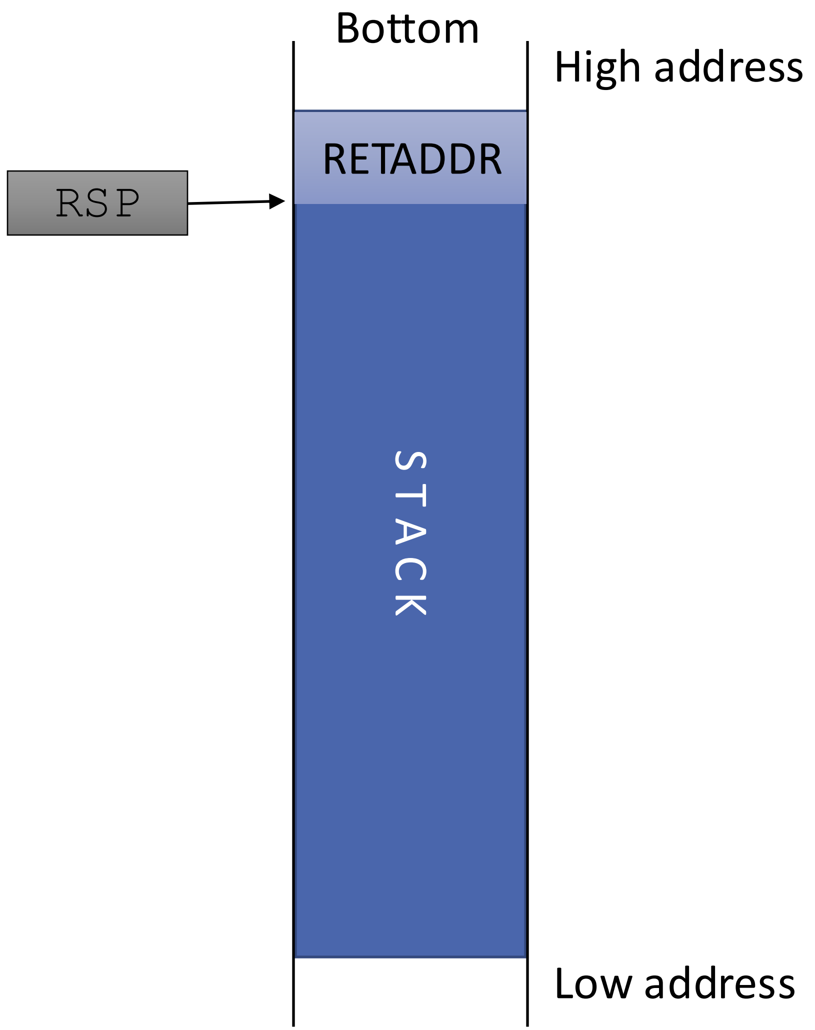
```
…
F1();
…
```

```
void F1()
{
  int a;
  char buf[16];
  unsigned long l;
  ...
}
```

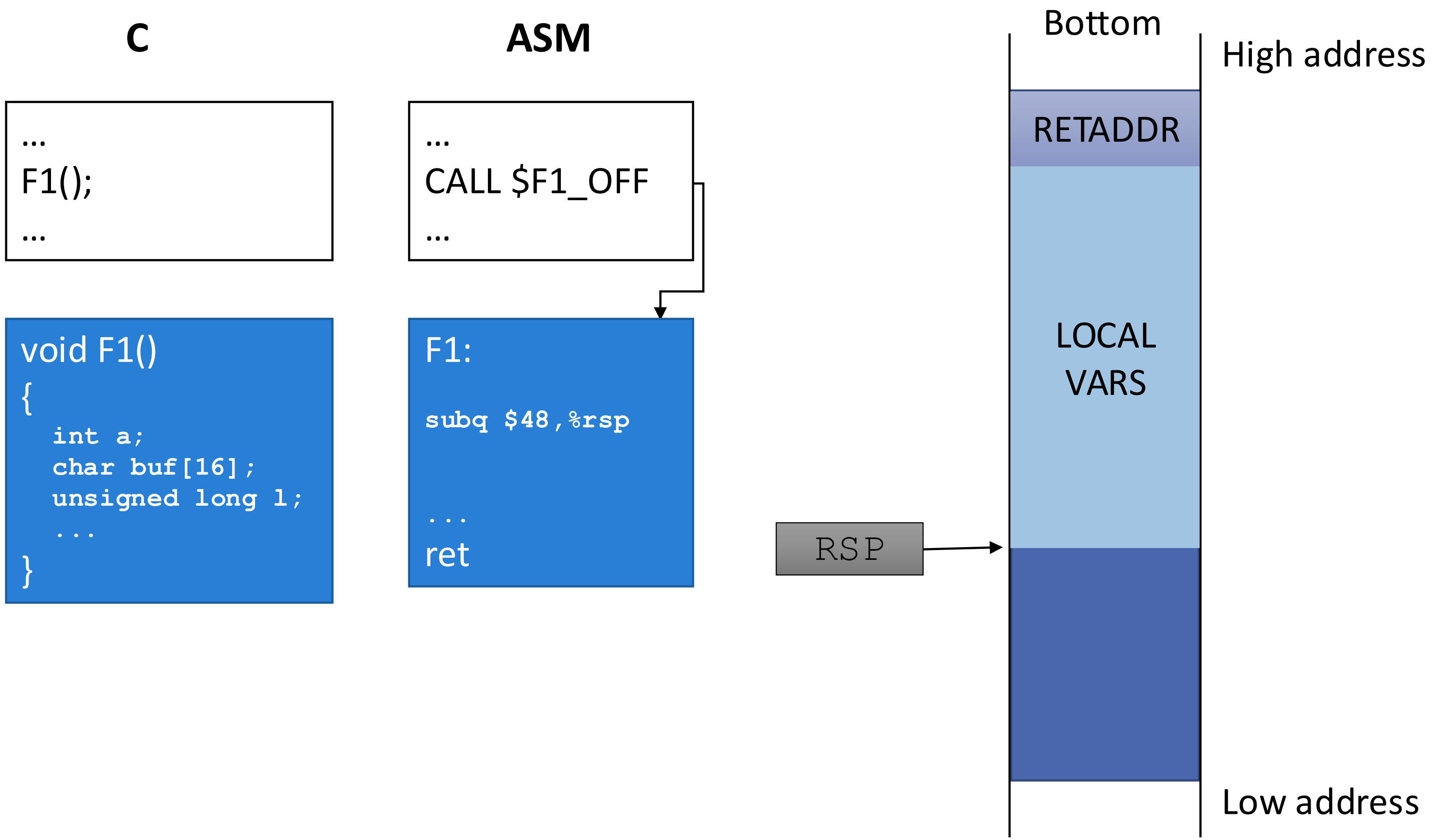
ASM

```
…
CALL $F1_OFF
…
```

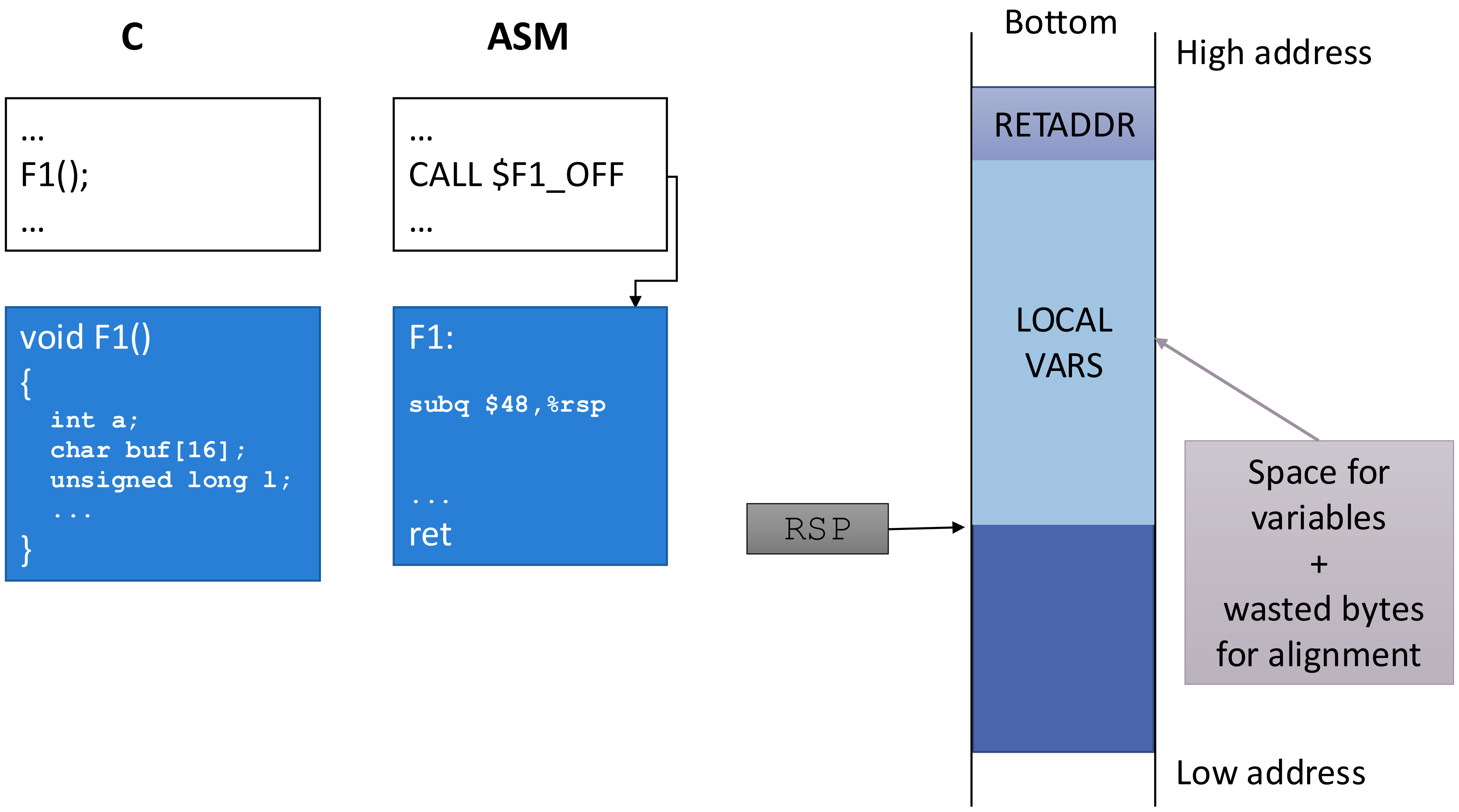
```
F1:

subq $48,%rsp

...
ret
```



Local Variables

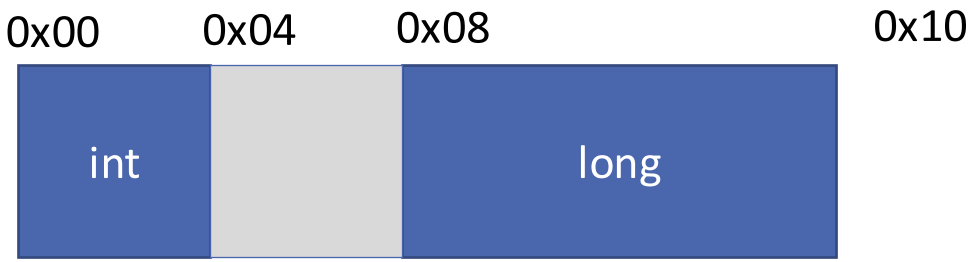


Local Variables



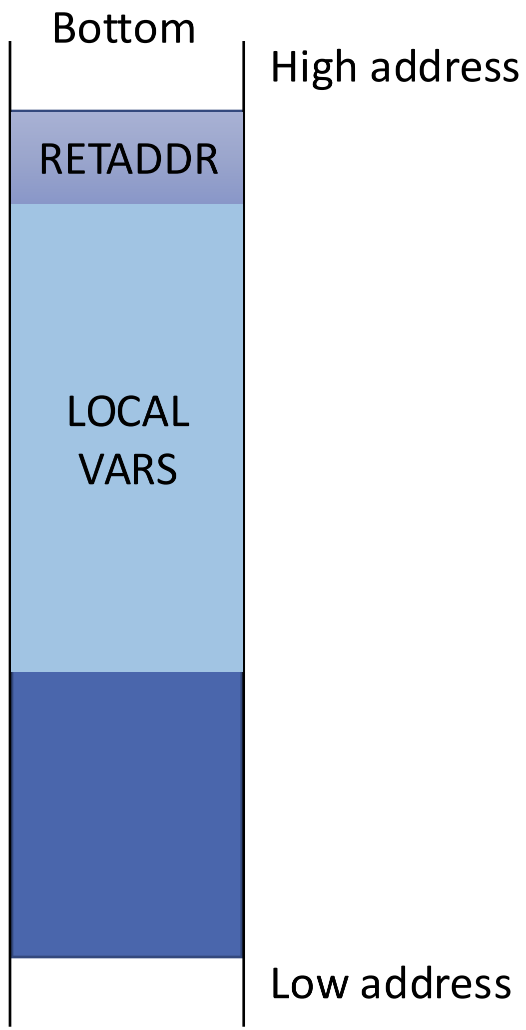
Alignment

- CPUs like aligned data
  - Better performance
- Compilers try to align data



The Stack Is Used…

- …to store the return address of caller functions
  - Code pointers!
- …to store local variables
  - Aka stack variables
- …to pass function arguments
  - Mostly on 32-bit architectures



Calling Conventions

- Defines the standard for passing arguments
- Caller and callee need to agree on the standard
- Enforced by compiler
- Important when using 3rd party libraries
  - Hence, also referred to as the Application Binary Interface (ABI)
- Different styles ↔ different advantages

```
…
F1(arg1, arg2, arg3);
…
```

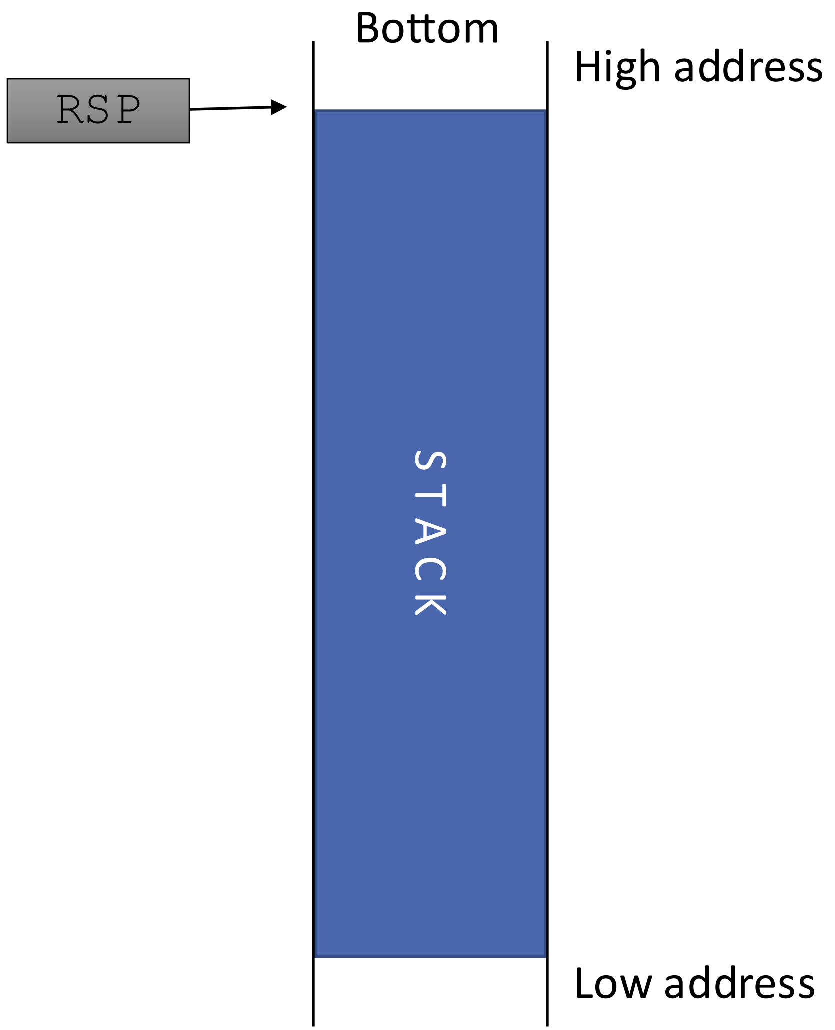
??

```
void F1(short a1, long a2, char *a3)
{
  int a;
  char buf[16];
  unsigned long l;
  ...
}
```

Popular conventions:
- cdecl (32-bit)
- System V AMD64 ABI

System V AMD64 ABI

- Arguments are passed using registers
  - First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9

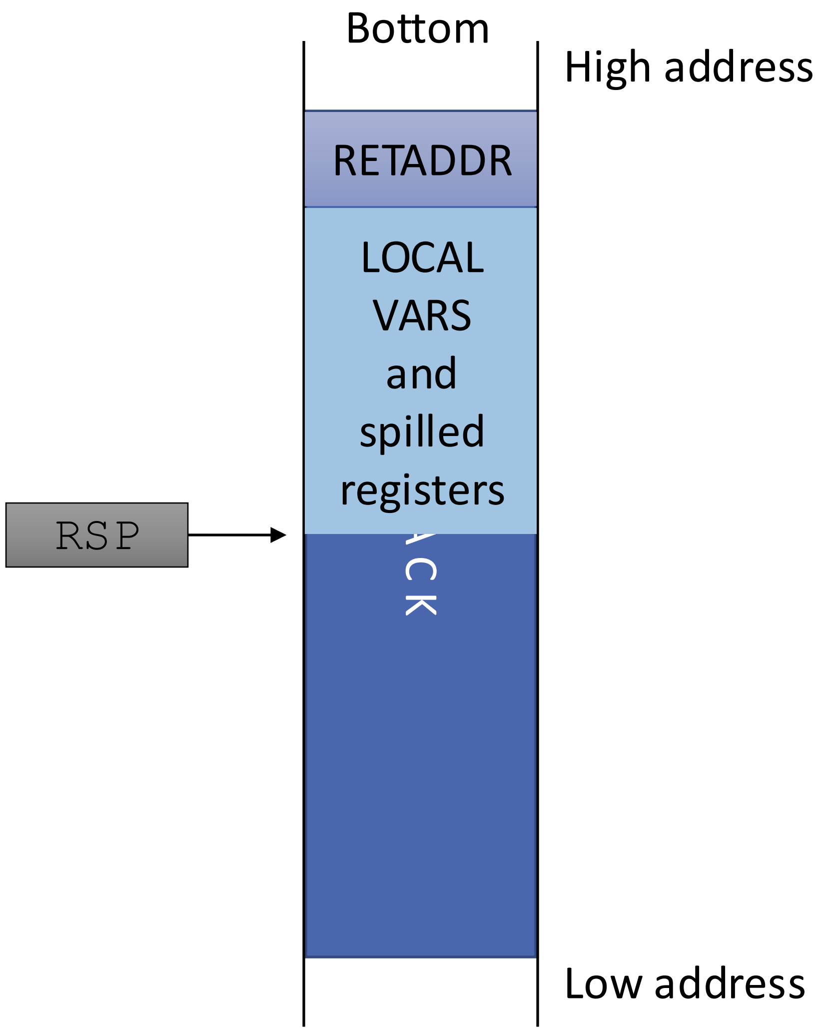



```
…
F1(0xff, UINT_MAX, argv[0]);
…
```

```
...
movq    (%rsi), %rdx
movl    $4294967295, %esi
movl    $255, %edi
call     F1
...
```

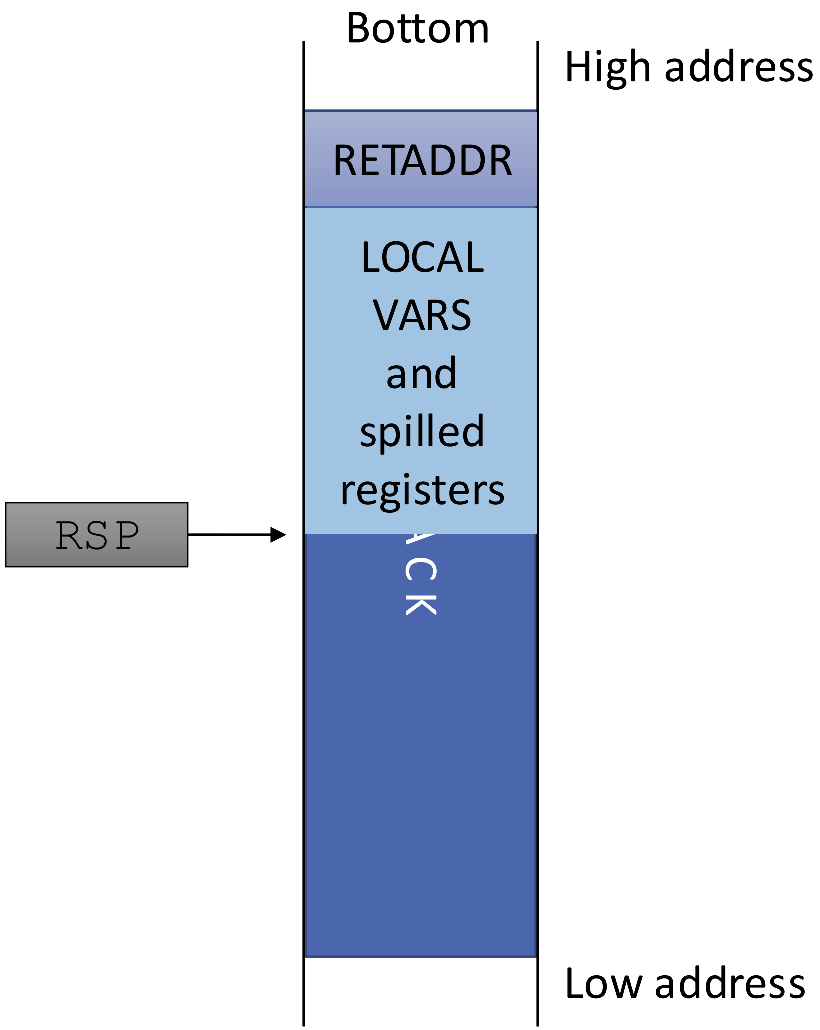
System V AMD64 ABI

- Arguments are passed using registers
  - First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9
- RBP, RBX, and R12–R15 are callee saved



System V AMD64 ABI

- Arguments are passed using registers
  - First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9
- RBP, RBX, and R12–R15 are callee saved
- RAX used for function return



Popular Conventions Summary

cdecl (mostly 32-bit)

- Arguments are passed on the stack
  - Pushed right to left
- eax, edx, ecx are caller saved
  - callee can overwrite without saving
- ebx, esi, edi are callee saved
  - callee must ensure they have same value on return
- eax used for function return value

System V AMD64 ABI

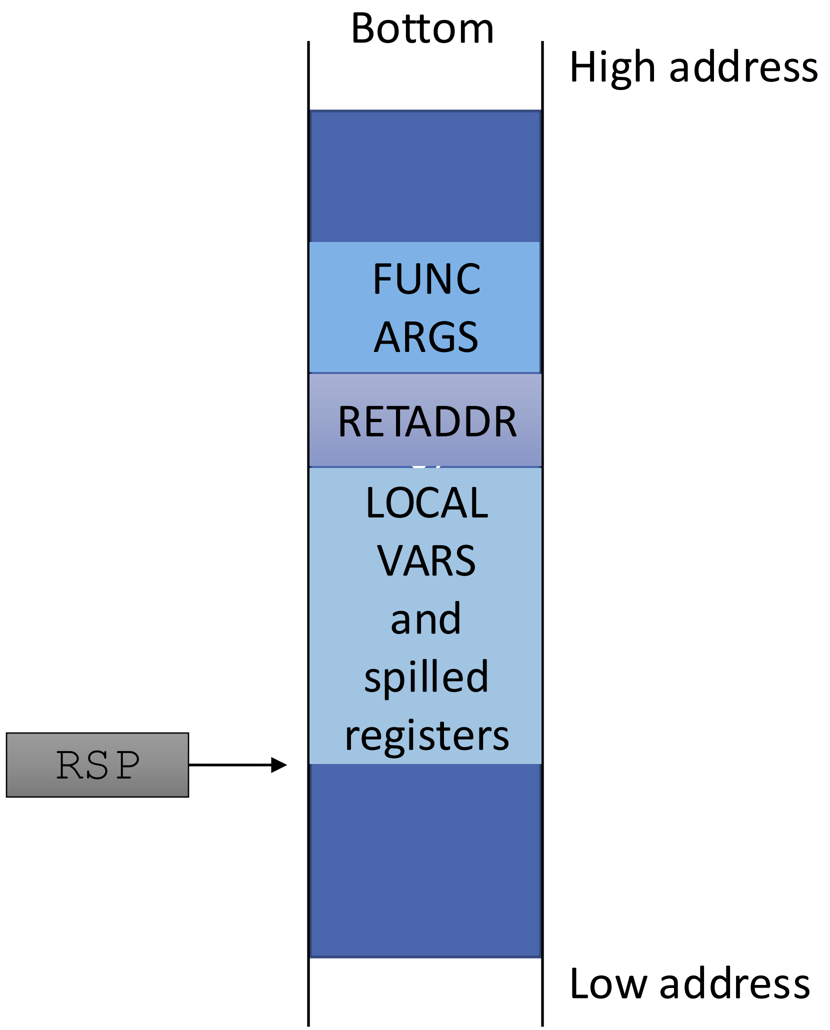
- Arguments are passed using registers
  - First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9
- RBP, RBX, and R12–R15 are callee saved
- RAX used for function return

Conventions include additional information, consult reading material for thorough description

- Example: handling of floating point regs

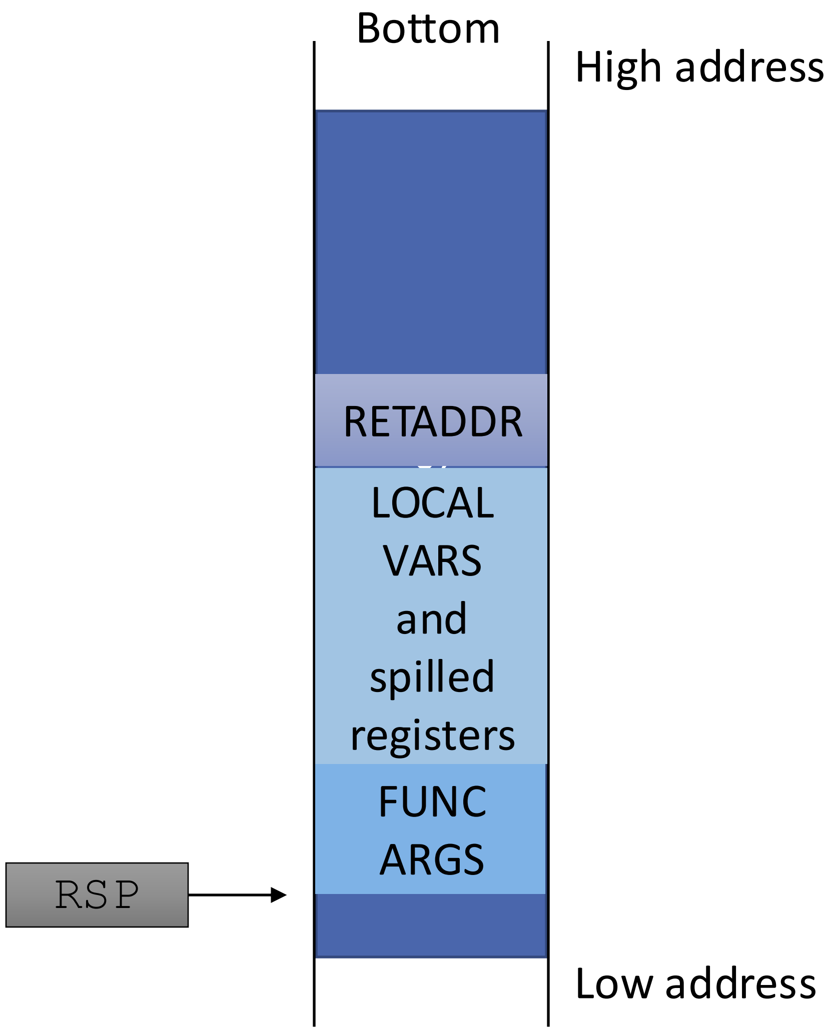
The Stack Is Used…

- …to store the return address of caller functions
  - Code pointers!
- …to store local variables
  - Aka stack variables
- …to pass function arguments
- …to temporarily store register values
- …to store the frame pointer



Stack Frame

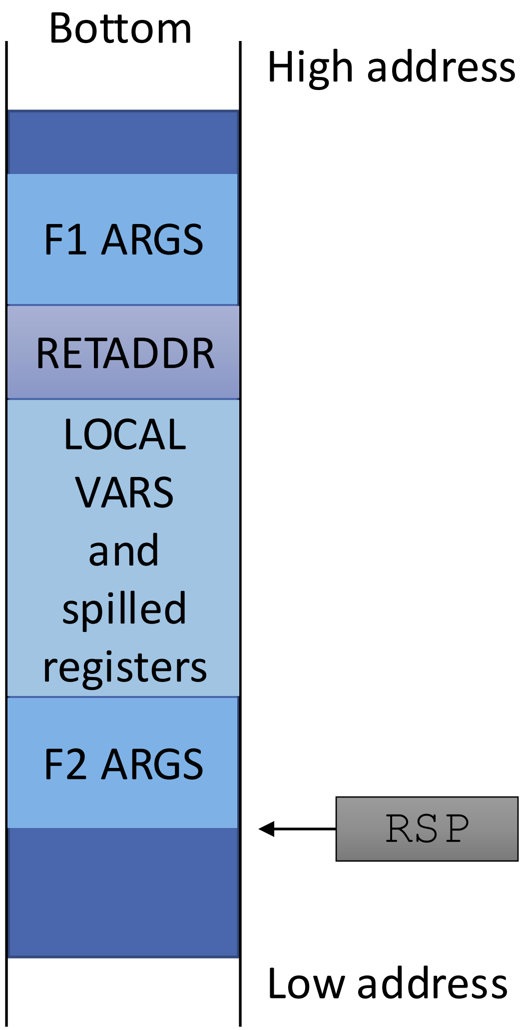
- A stack frame includes all function-local data
  - Local variables
  - Spilled registers
  - Function arguments pushed to the stack to make calls
- More of a logical entity
- Can grow as function executes



Stack Frame Boundaries

- Start below return address
- Stop at stack pointer

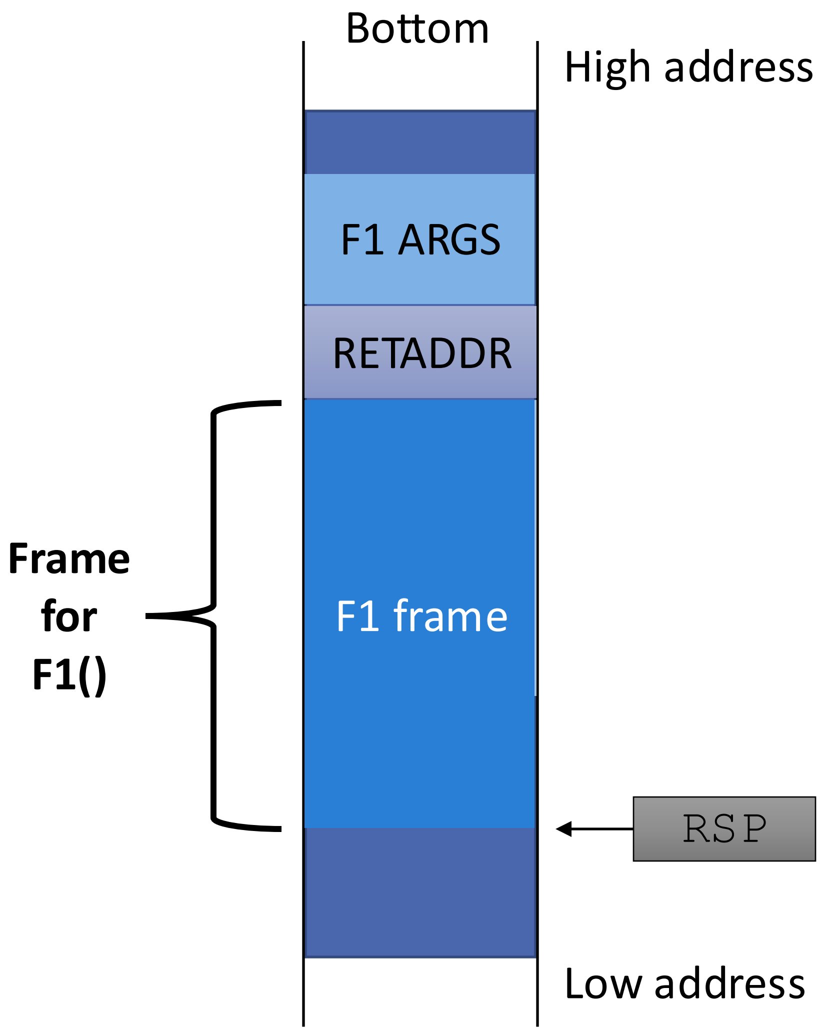
```
void F1(short a1, long a2, char *a3)
{
  int a;
  char buf[16];
  unsigned long l;
  ...
  long l2 = F2(a);
  ...
}
```



Stack Frame Boundaries

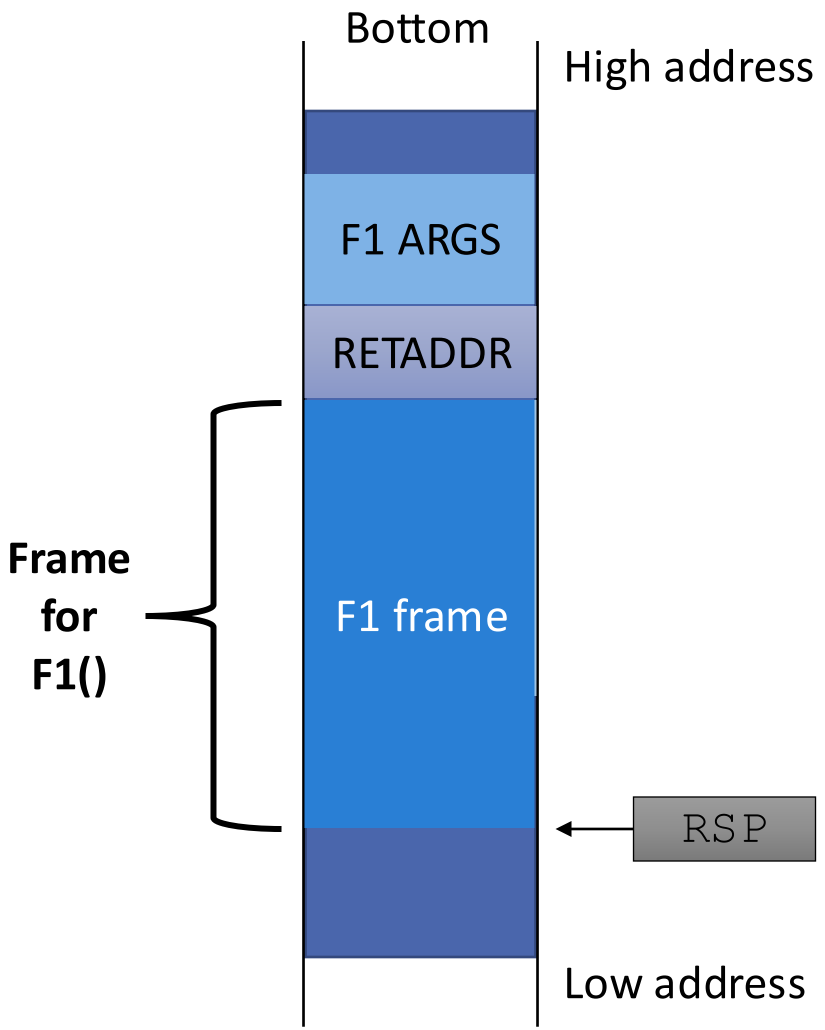
- Start below return address
- Stop at stack pointer

```
void F1(short a1, long a2, char *a3)
{
  int a;
  char buf[16];
  unsigned long l;
  ...
  long l2 = F2(a);
  ...
}
```



RETADDR and Stack Frames

- The return address may also be considered part of the frame
- We will not consider it for simplicity

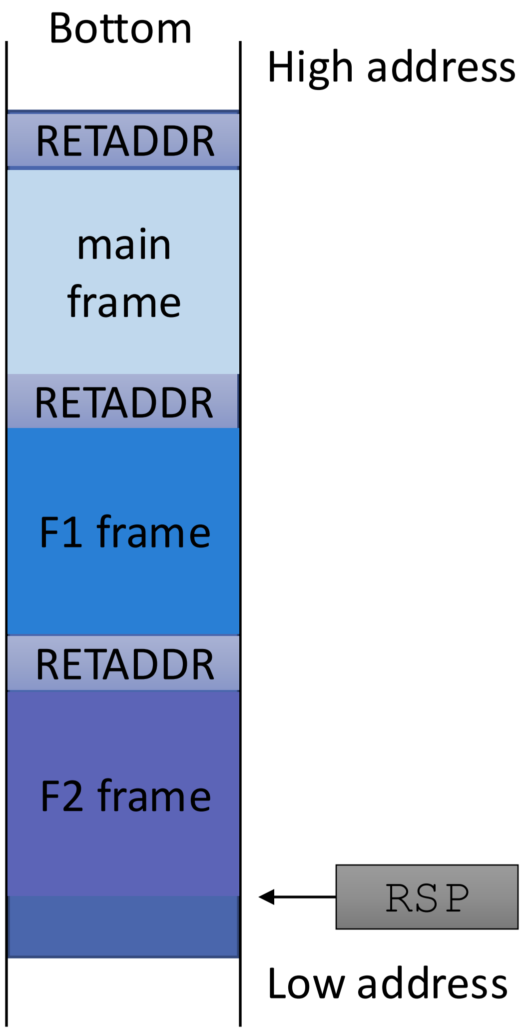


Stack Frames Example

```
int main(int argc, char **argv)
{
  ...
  F1(0xFF, UINT_MAX, argv[0]);
  ...
}
```

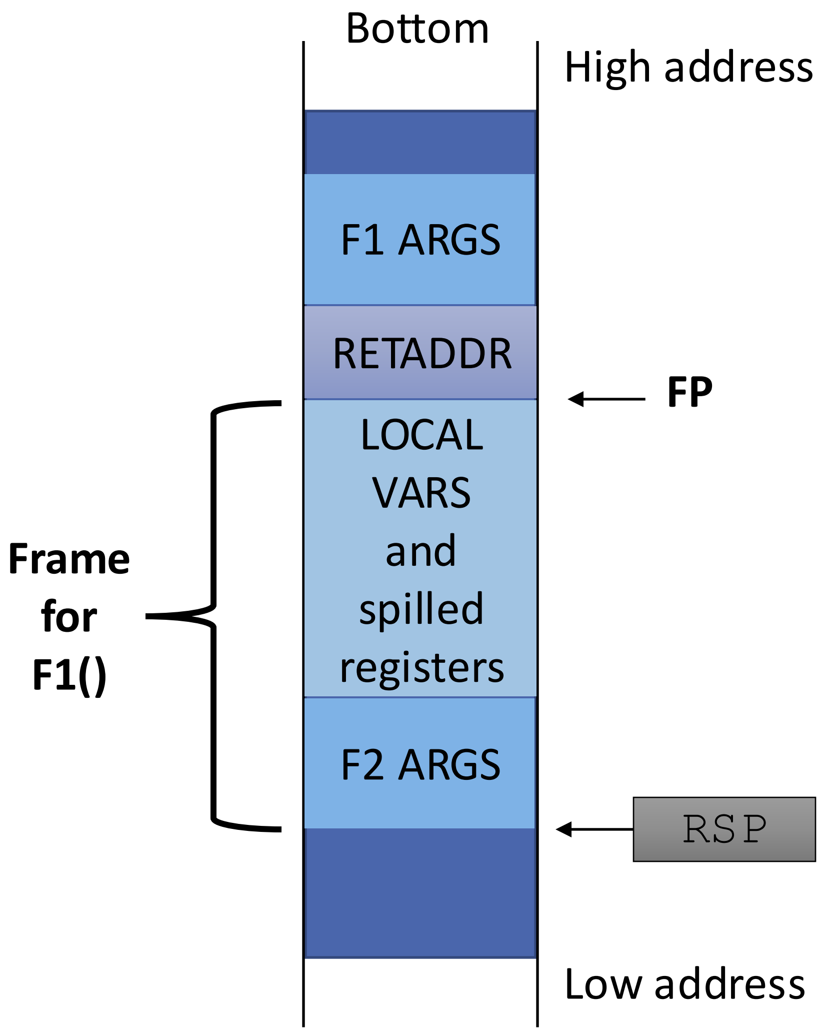
```
void F1(short a1, long a2, char *a3)
{
   int a;
   char buf[16];
   unsigned long l;
   ...
   long l2 = F2(a);
   ...
}
```

```
long F2(long a1)
{
  ...
}
```



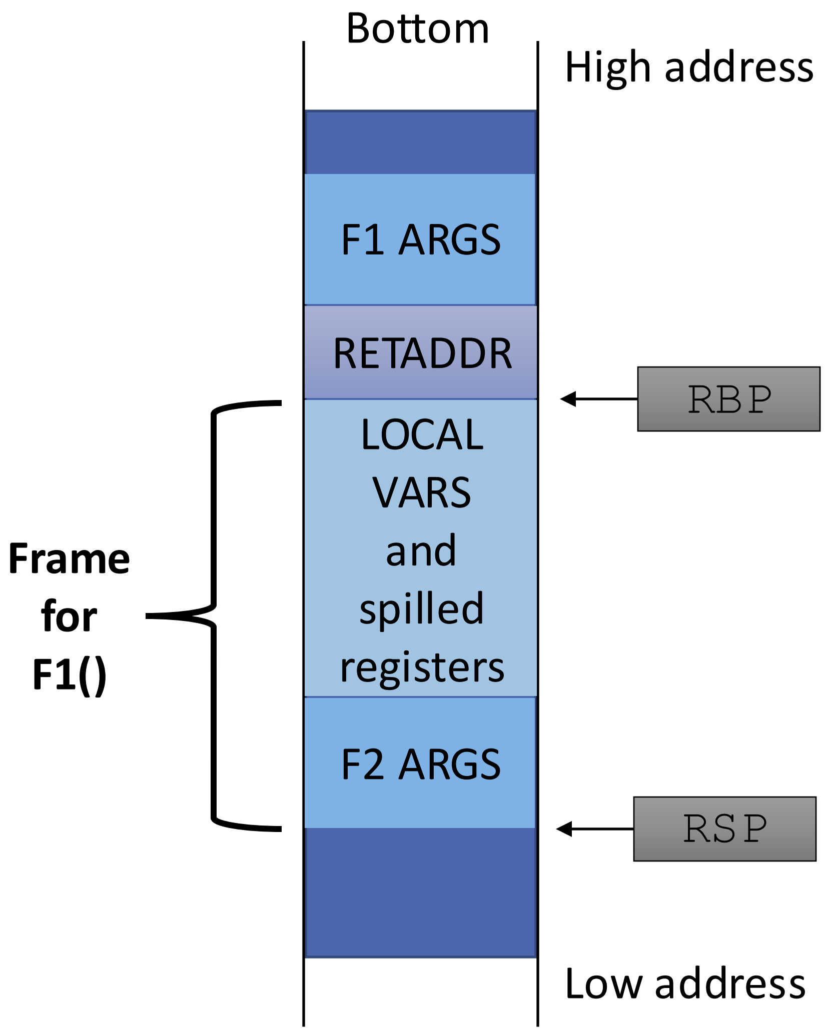
Frame Pointer (FP)

- Marks the highest address in the frame
  - Bottom of the frame
- Aka Base Pointer

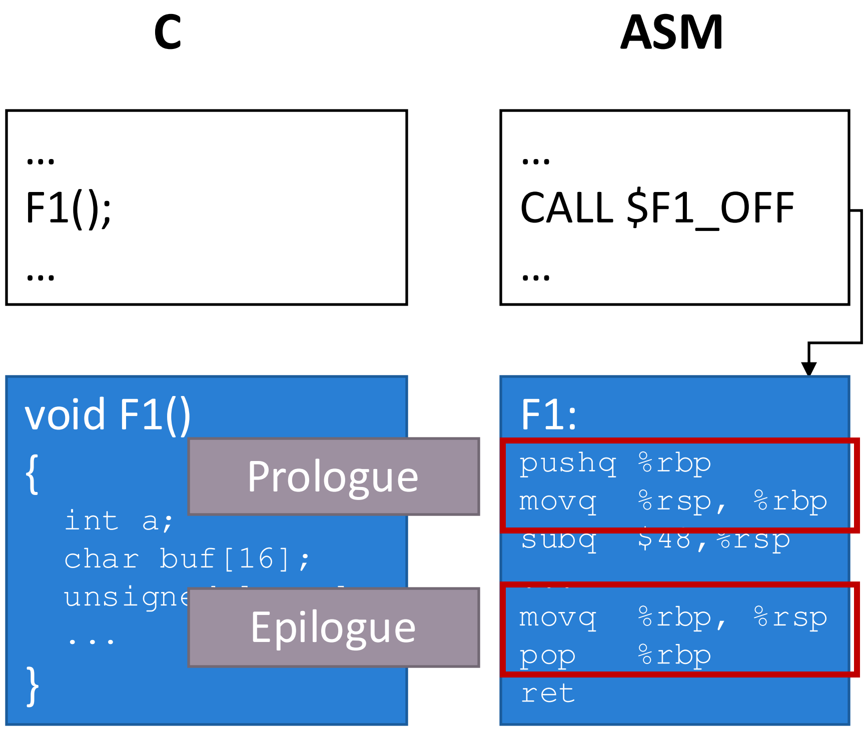


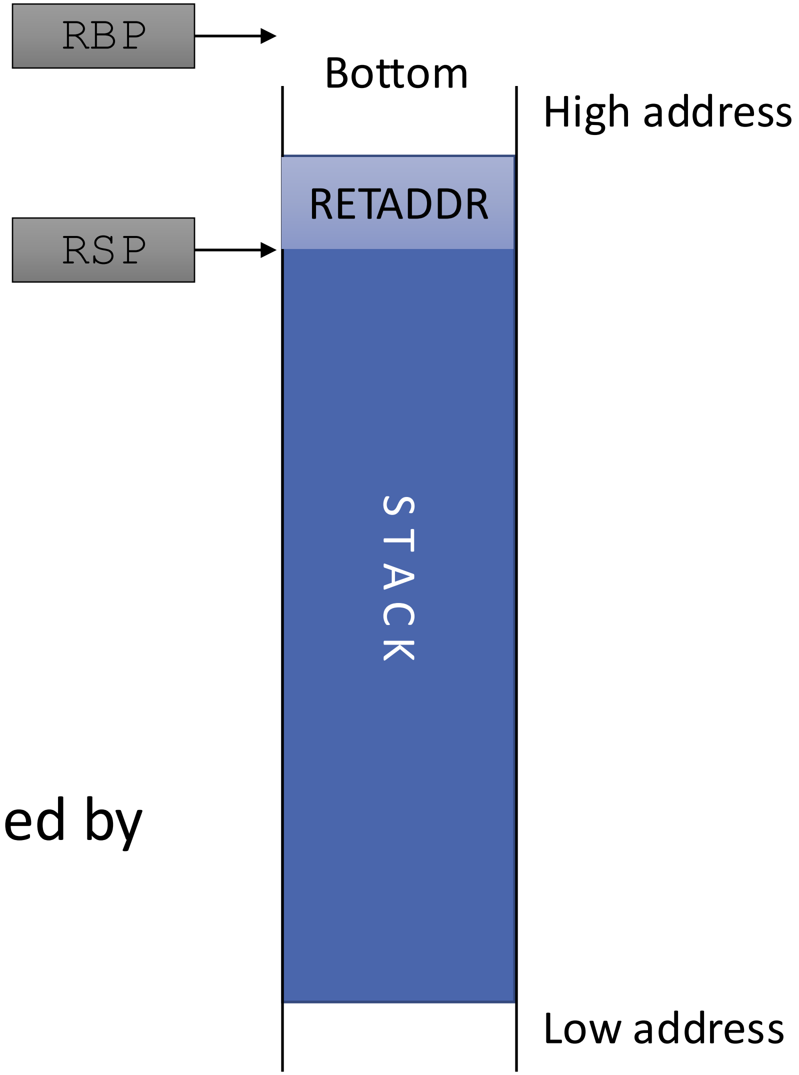
Frame Pointer (FP)

- Marks the highest address in the frame
  - Bottom of the frame
- Aka Base Pointer
- The RBP/EBP register commonly contains the FP
- RBP needs to updated upon entry/exit of function



Maintaining the FP





- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP

C

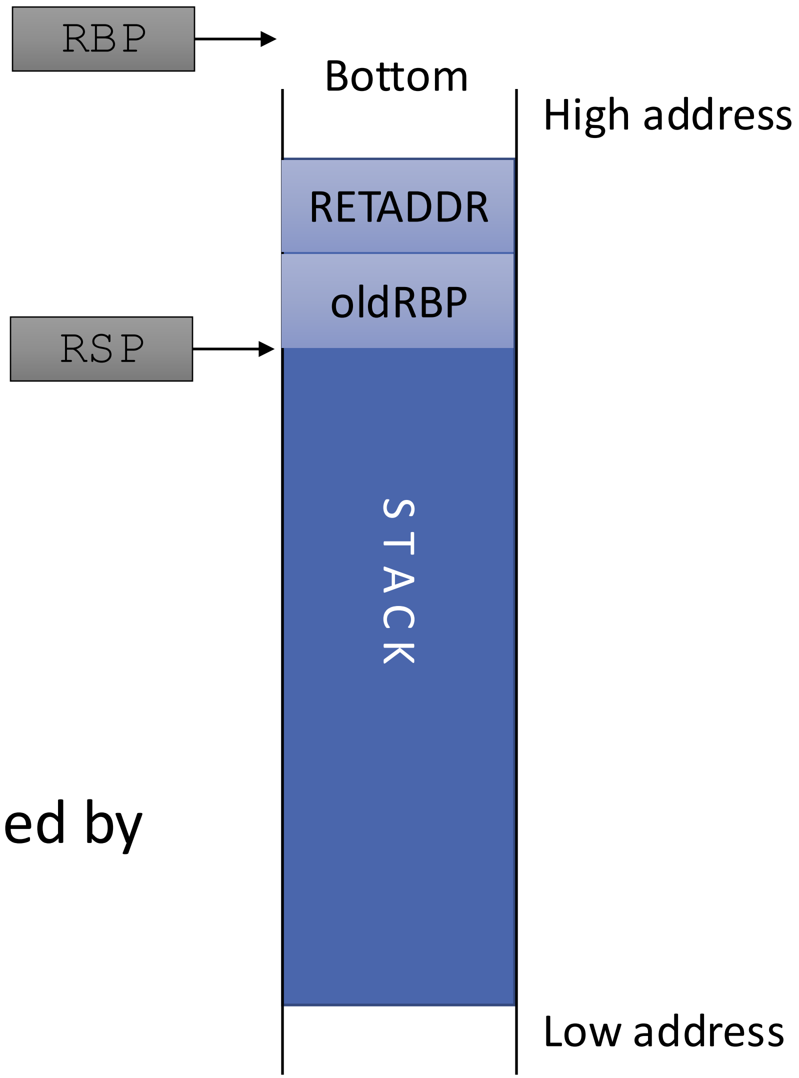
```
…
F1();
…
```

```
void F1()
{
  int a;
  char buf[16];
  unsigned long l;
  ...
}
```

ASM

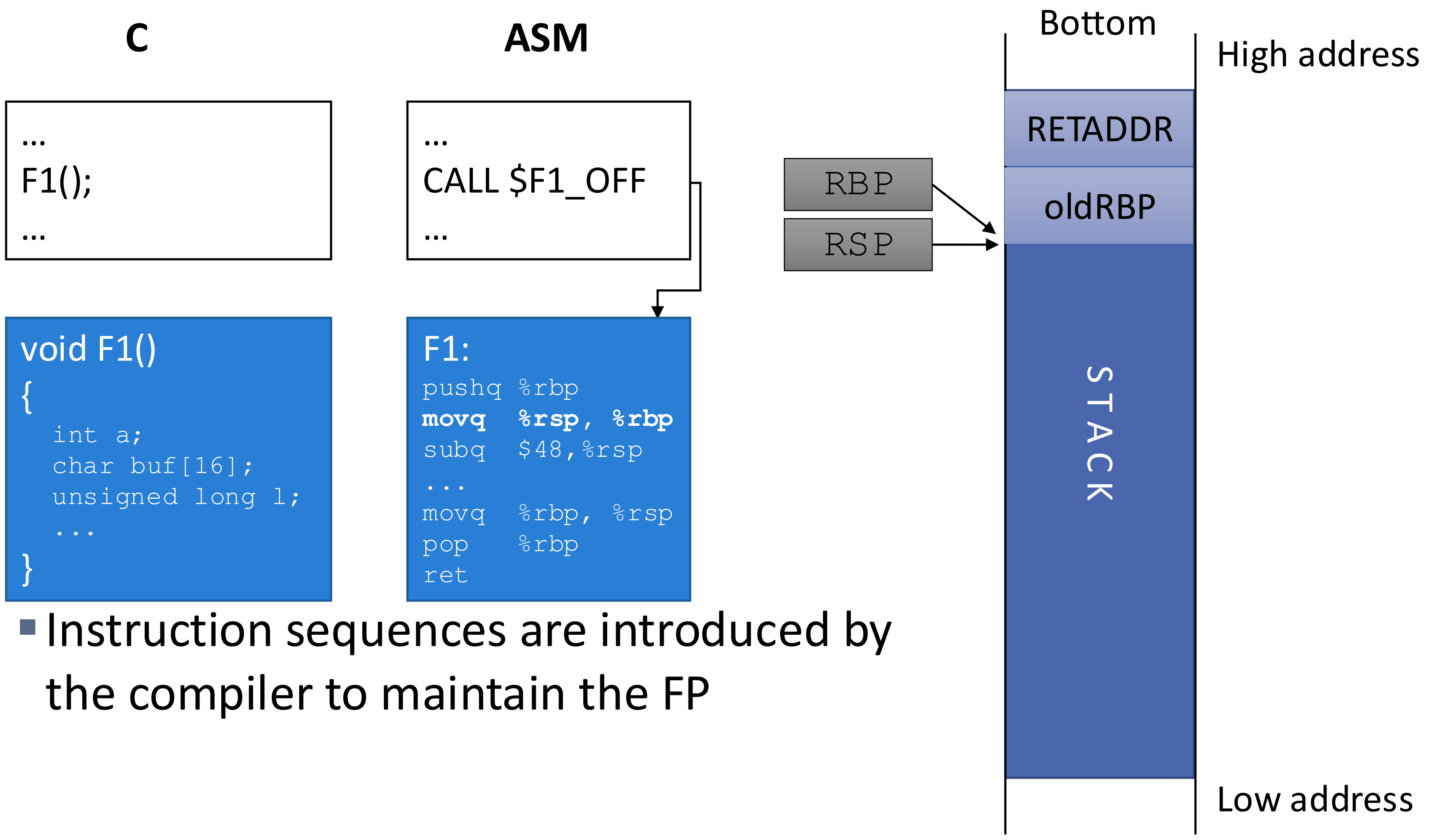
```
…
CALL $F1_OFF
…
```

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48,%rsp
...
movq  %rbp, %rsp
pop   %rbp
ret
```



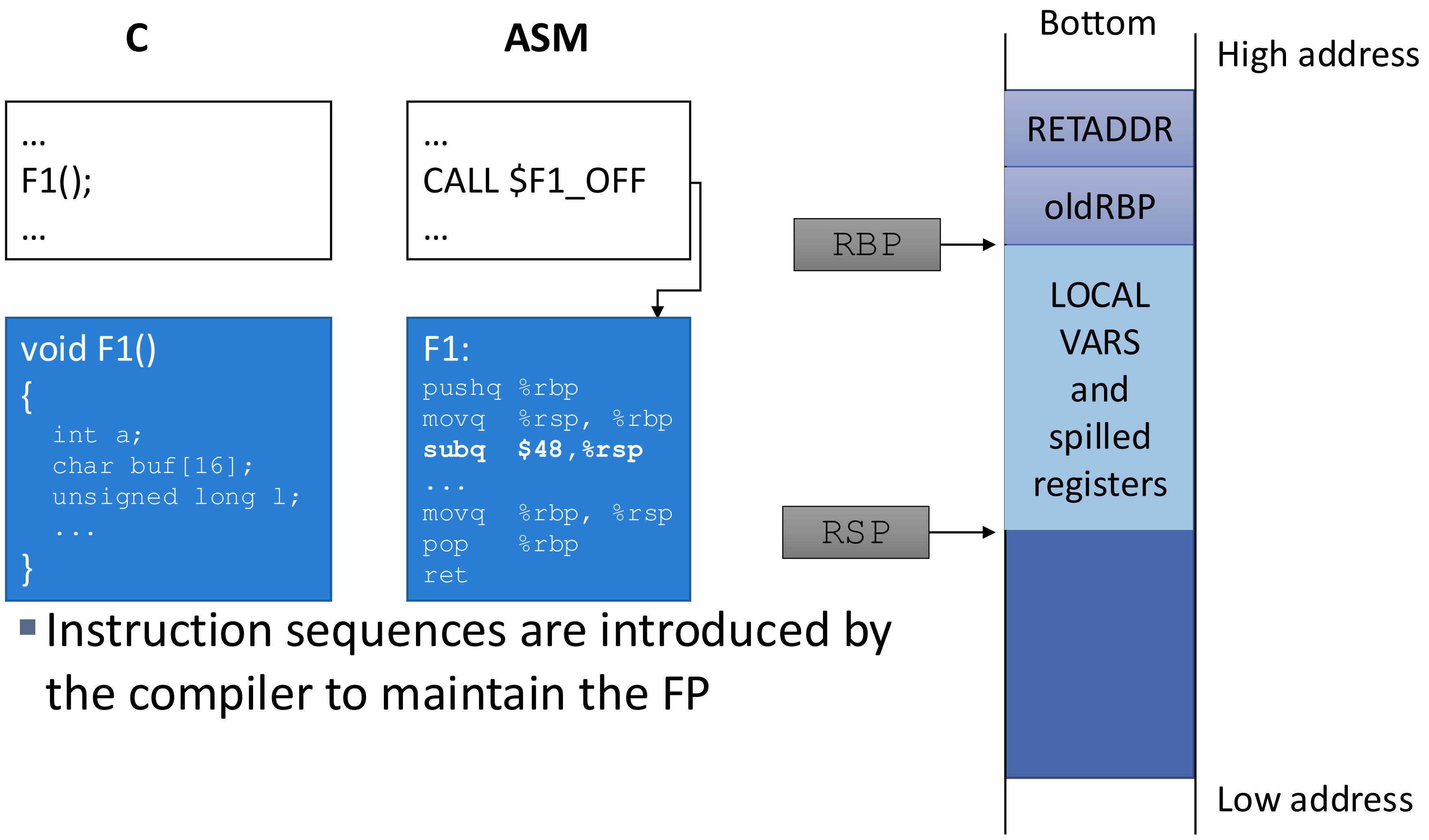
- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP

C

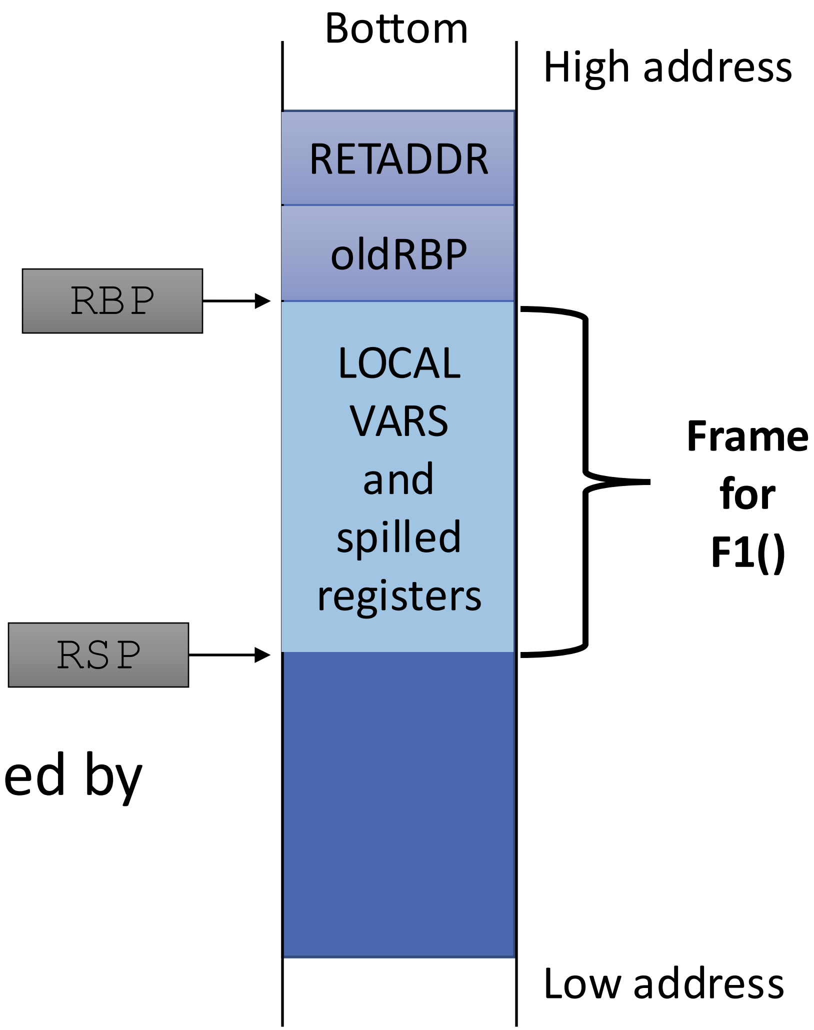
```
…
F1();
…
```

```
void F1()
{
   int a;
   char buf[16];
   unsigned long l;
   ...
}
```

ASM

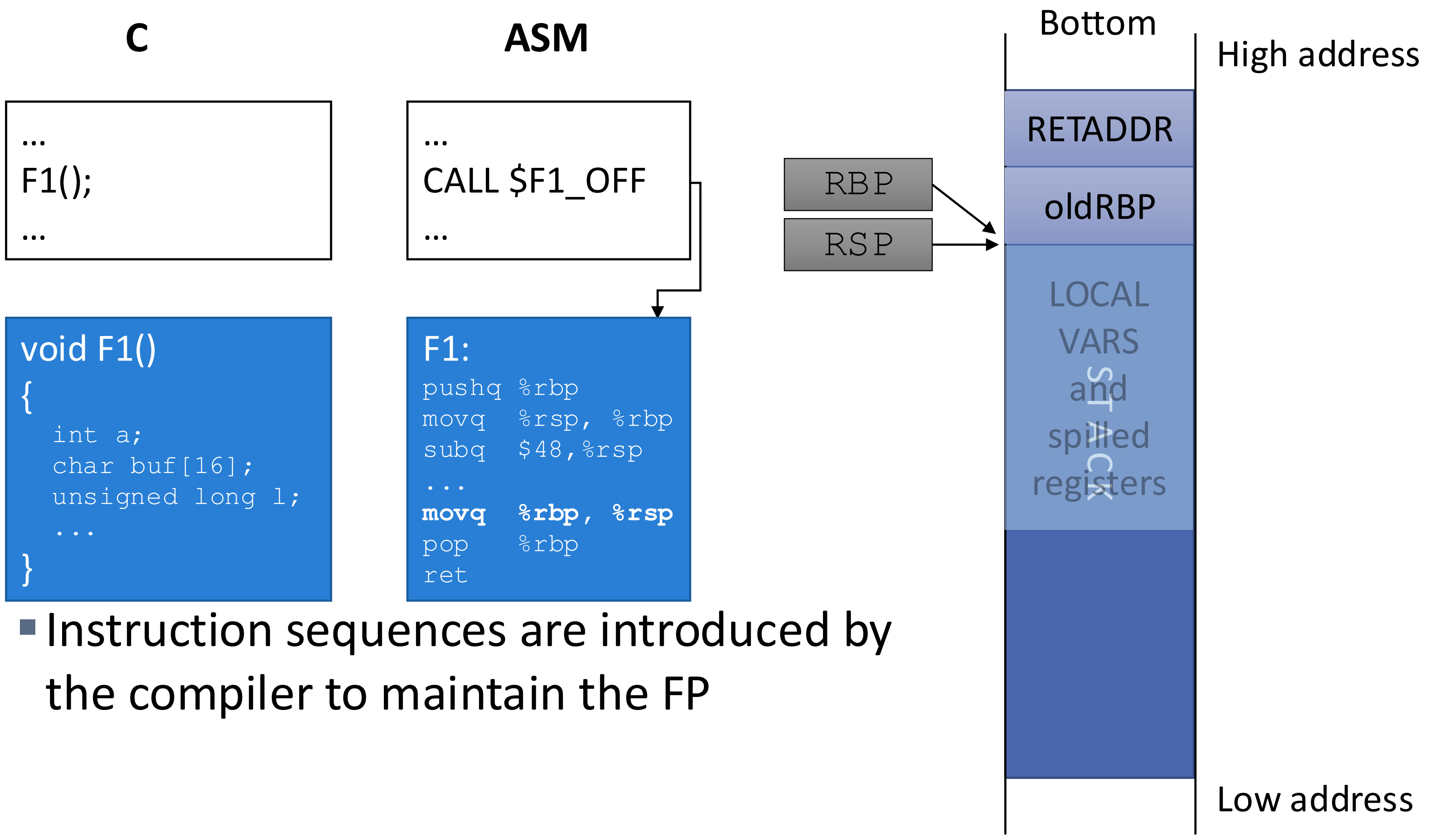
```
…
CALL $F1_OFF
…
```

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48,%rsp
...
movq  %rbp, %rsp
pop   %rbp
ret
```



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP

C

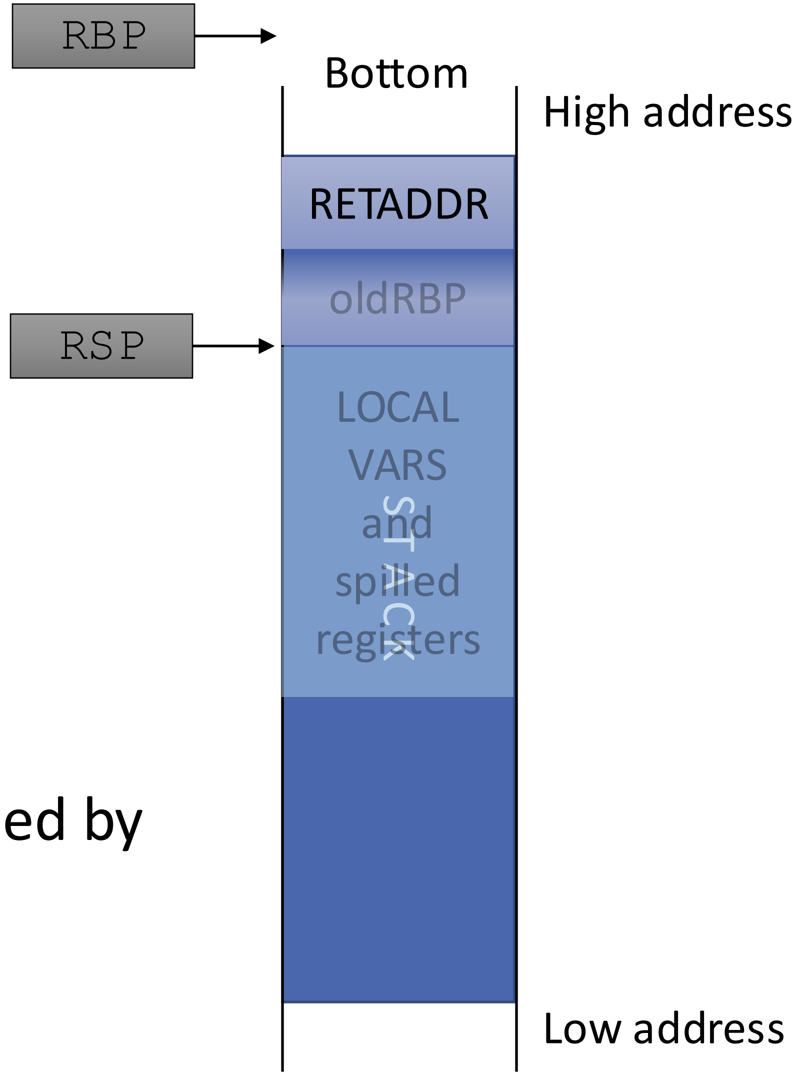
```
...
F1();
...
```

```
void F1()
{
  int a;
  char buf[16];
  unsigned long l;
  ...
}
```

ASM

```
...
CALL $F1_OFF
...
```

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48,%rsp
...
movq  %rbp, %rsp
pop   %rbp
ret
```



- Instruction sequences are introduced by the compiler to maintain the FP

Maintaining the FP

C

```
…
F1();
…
```

```
void F1()
{
  int a;
  char bu...
  unsigne...
  ...
}
```

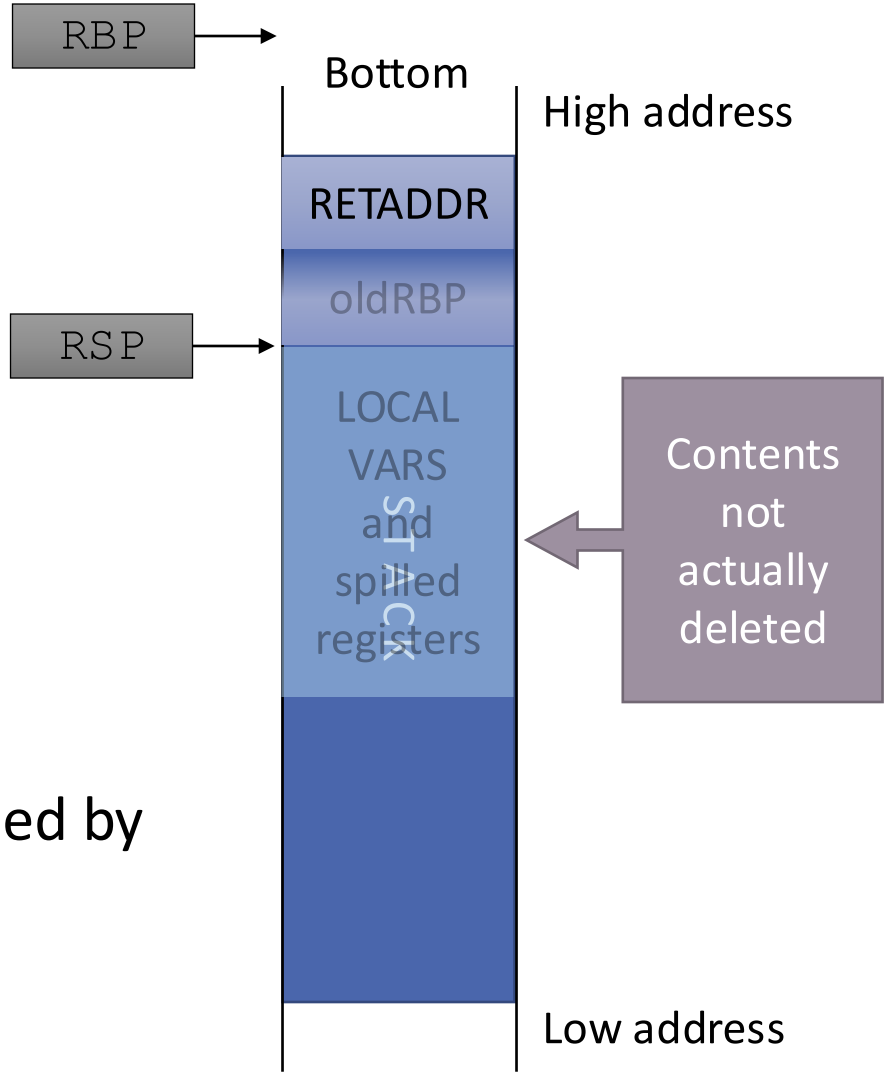
Equivalent Instruction: `enter`

Equivalent Instruction: `leave`

ASM

```
…
CALL $F1_OFF
…
```

```
F1:
pushq %rbp
movq  %rsp, %rbp
subq  $48, %rsp
...
movq  %rbp, %rsp
pop   %rbp
ret
```



- Instruction sequences are introduced by the compiler to maintain the FP

Putting It All Together

Function Call

- Prepare function call arguments
- Make the call
- Function prologue
  - Save RBP/EBP
  - Setup new RBP/EBP
- Callee saves registers that need to be preserved
- Callee allocates stack space

Function Return

- Function epilogue
  - Release stack space
  - Restore BP
- Return

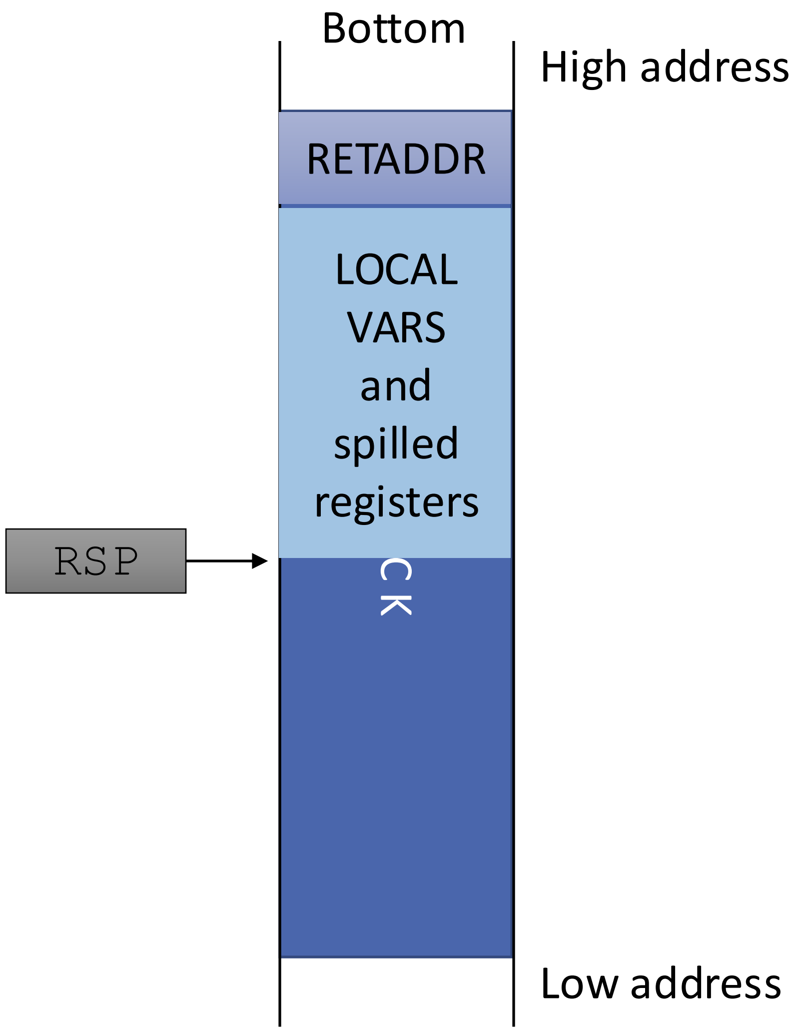
Accessing Stack Variables (no FP)

C

```
void F1()
{
  int a;
  char buf[16];
  long l;
  ...
  l = F2(a);
  ...
}
```

ASM

```
F1:

...
movl   76(%rsp), %eax
cltq
movq   %rax, %rdi
call   F2
movq   %rax, 40(%rsp)
...
ret
```



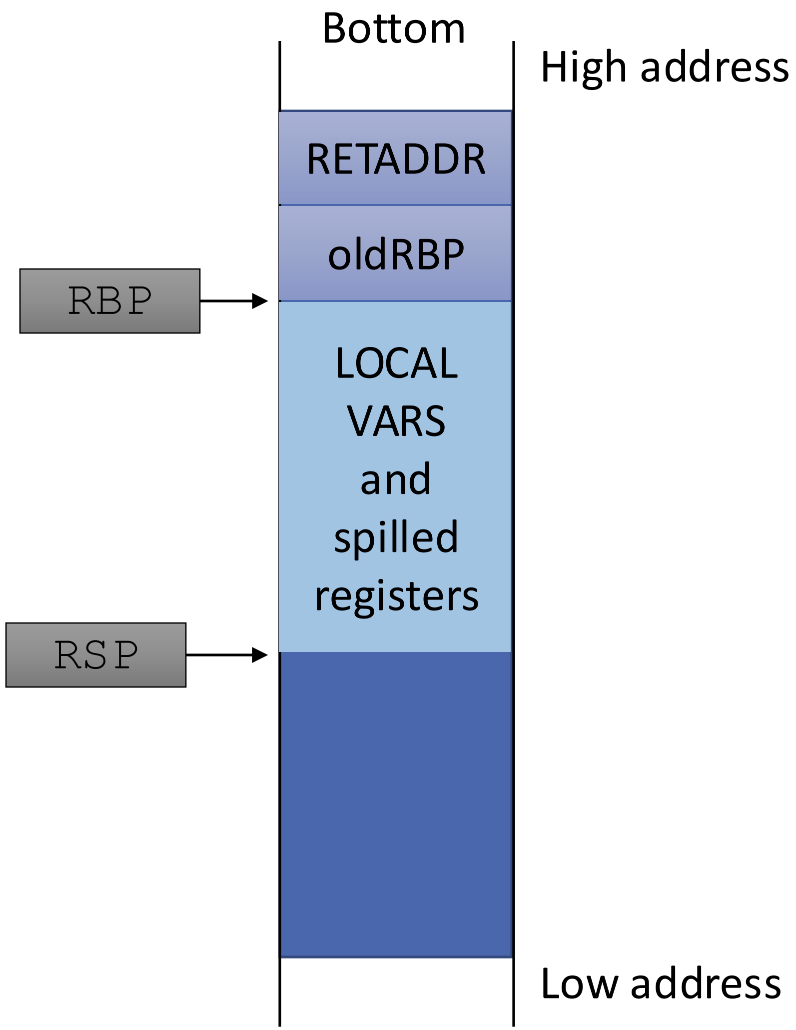
Accessing Stack Variables (with FP)

C

```
void F1()
{
   int a;
   char buf[16];
   long l;
   ...
   l = F2(a);
   ...
}
```

ASM

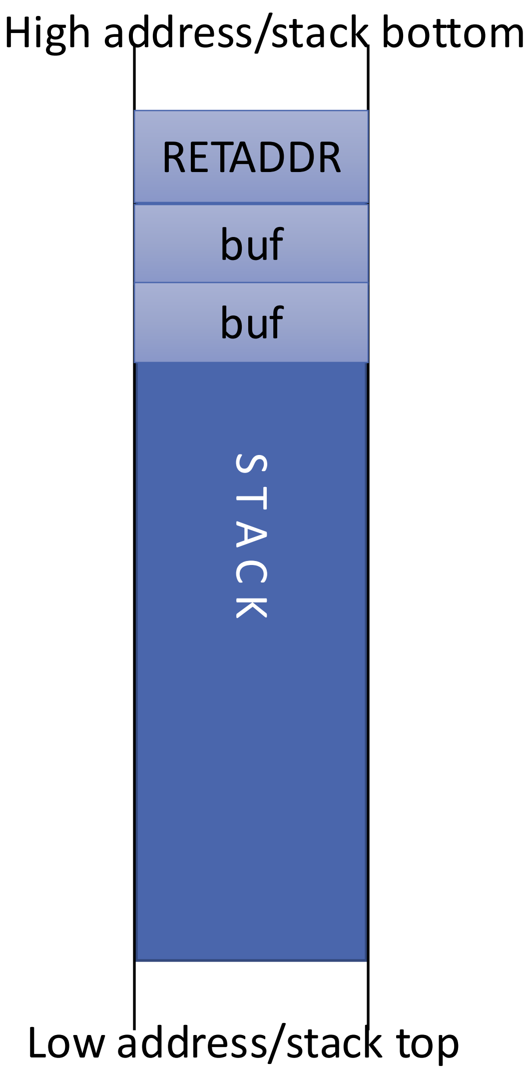
```
F1:

...
movl   -4(%rbp), %eax
cltq
movq   %rax, %rdi
call   F2
movq   %rax, -40(%rbp)
...
ret
```



Stack Smashing Attacks

Stack Overflow Example

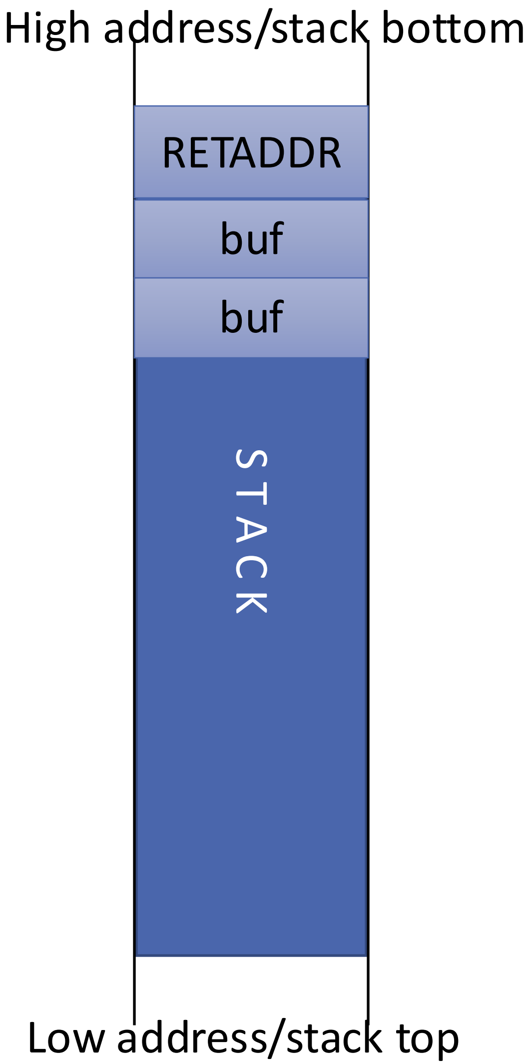
```
void copy(const char *str)
{
        char buf[16];

        strcpy(buf, str);

        puts(buf);
}
```



Stack Overflow Example

```
void copy(const char *str)
{
        char buf[16];

        strcpy(buf, str);

        puts(buf);
}
```

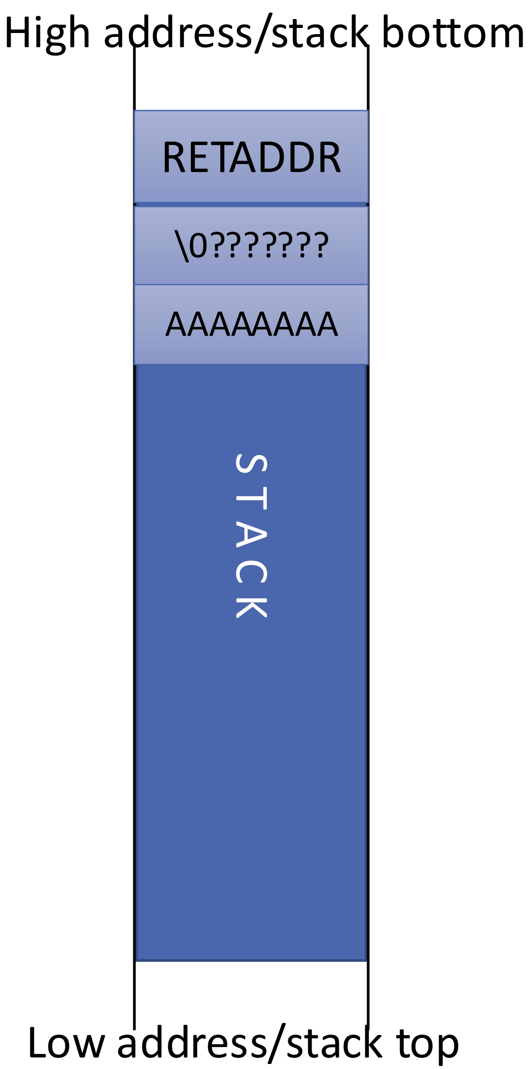
```
./copy AAAAAAAA
```



Stack Overflow Example

```
void copy(const char *str)
{
        char buf[16];

        strcpy(buf, str);

        puts(buf);
}
```

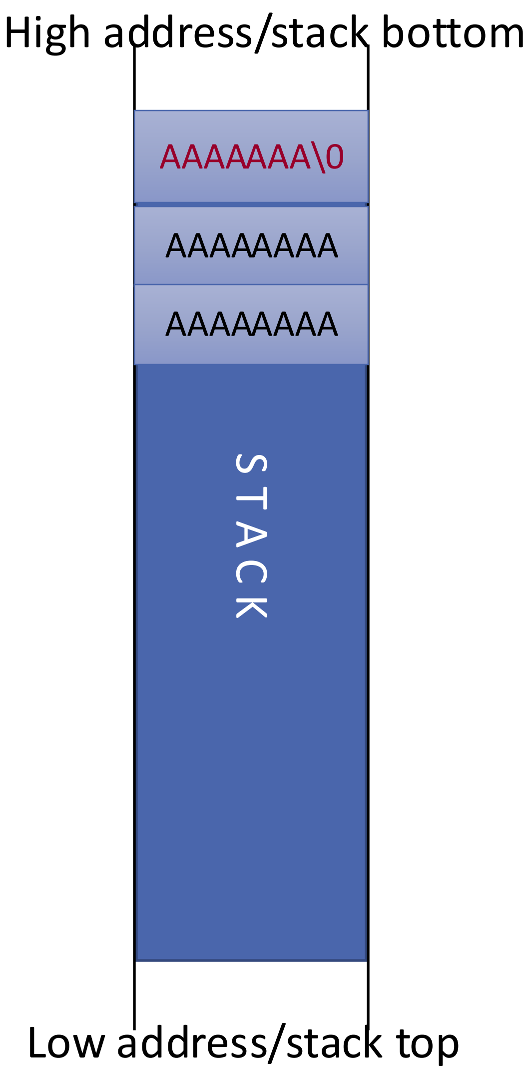
```
./copy AAAAAAAA
```



Stack Overflow Example

```
void copy(const char *str)
{
        char buf[16];

        strcpy(buf, str);

        puts(buf);
}
```

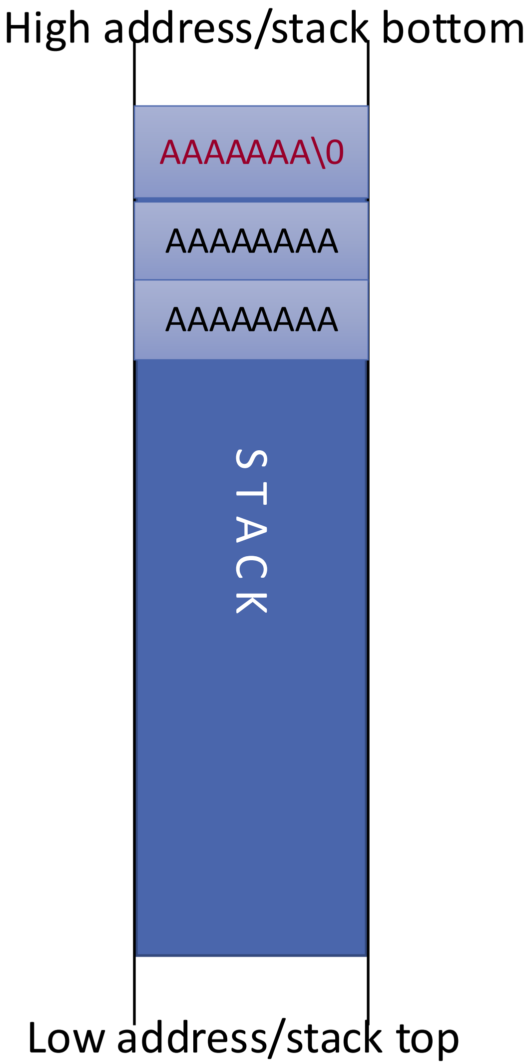
./copy AAAAAAAAAAAAAAAAAAAAAAAAA



Stack Overflow Example

```
void copy(const char *str)
{
        char buf[16];

        strcpy(buf, str);

        puts(buf);
}
```

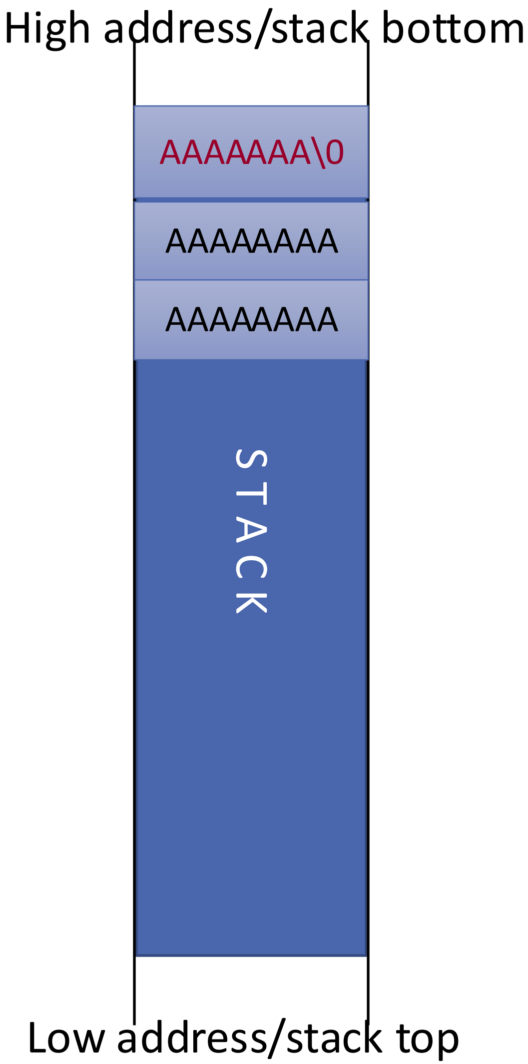
```
subq    $40, %rsp
movq    %rdi, 8(%rsp)
movq    8(%rsp), %rdx
leaq    16(%rsp), %rax
movq    %rdx, %rsi
movq    %rax, %rdi
call    strcpy@PLT
leaq    16(%rsp), %rax
movq    %rax, %rdi
call    puts@PLT
nop
addq    $40, %rsp
ret
```



Stack Overflow Example

- This stack overflow allows a to control the return address stored in the stack
- When ret executes, the control-flow of the program will be redirected to an arbitrary address → control-flow hijacking

```
subq     $40, %rsp
movq     %rdi, 8(%rsp)
movq     8(%rsp), %rdx
leaq     16(%rsp), %rax
movq     %rdx, %rsi
movq     %rax, %rdi
call     strcpy@PLT
leaq     16(%rsp), %rax
movq     %rax, %rdi
call     puts@PLT
nop
addq     $40, %rsp
ret
```



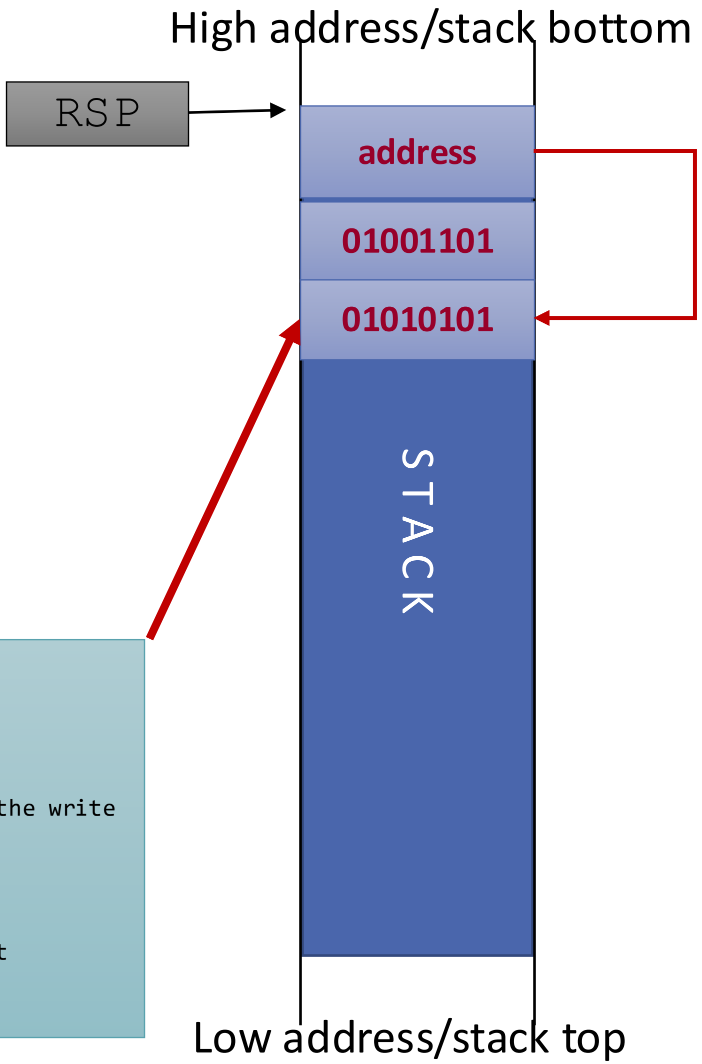
Control-Flow Hijacking Attacks

- Untrusted inputs that lead to corruption of a code pointer, which will be later dereferenced, lead to **control-flow hijacking attacks**

Original Stack Smashing Attack

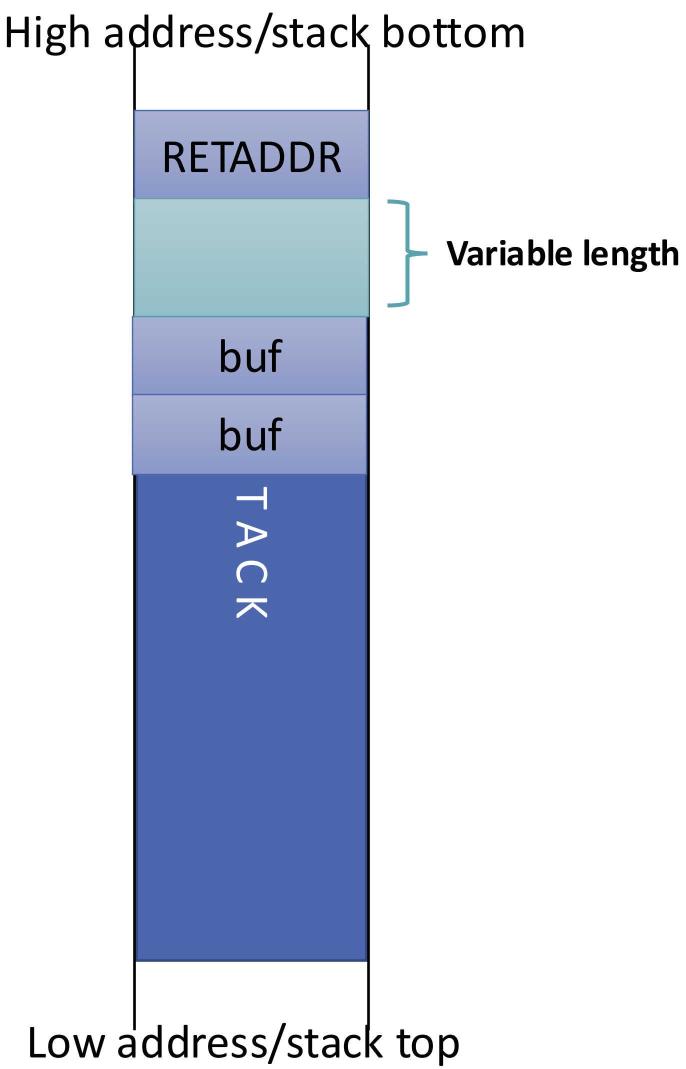
- Appeared at Phrack magazine
  http://phrack.org/issues/49/14.html#article
- Exploits the fact that stack used to be executable
  - Stores binary code in the controlled buffer
    - **Any** executable, controlled buffer will do!
  - Redirect program to inject code
- Performs arbitrary **code injection**!

```
        # write(1, message, 13)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi
        mov     $13, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello, world\n"
```



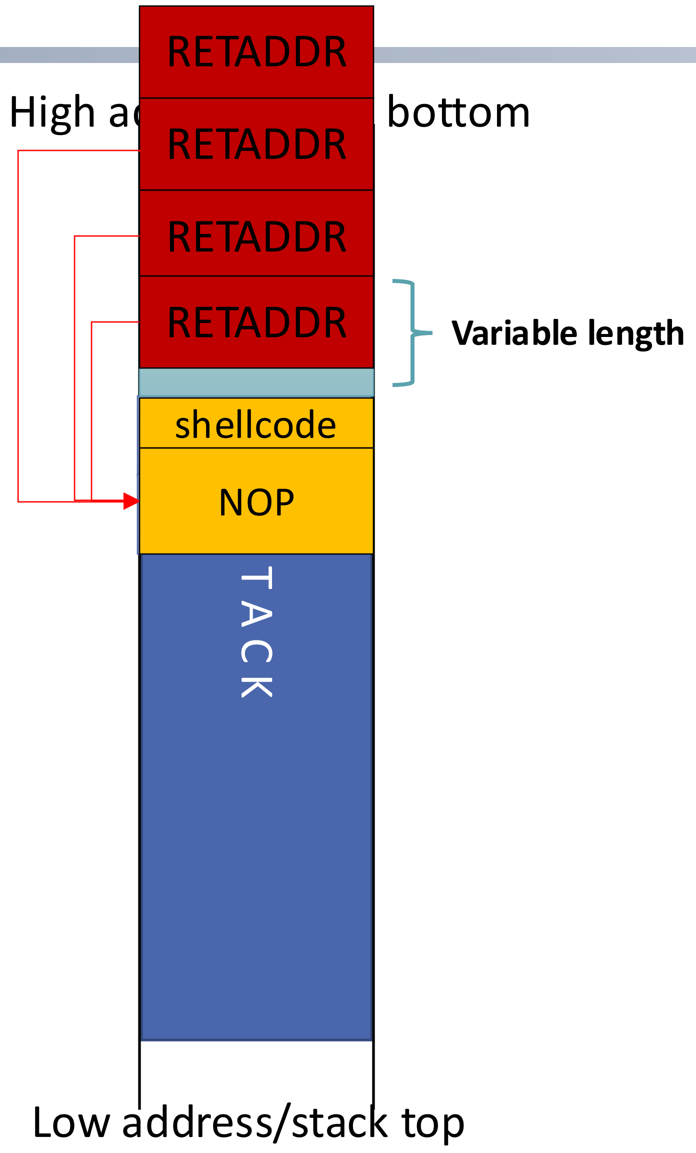
Making Exploits More Robust

- Observation: Different compiler may use different alignment, spill different register, etc.
- Problems:
  - Exact distance of return address may be different between binaries
  - Exact address of buffer may be different



Making Exploits More Robust

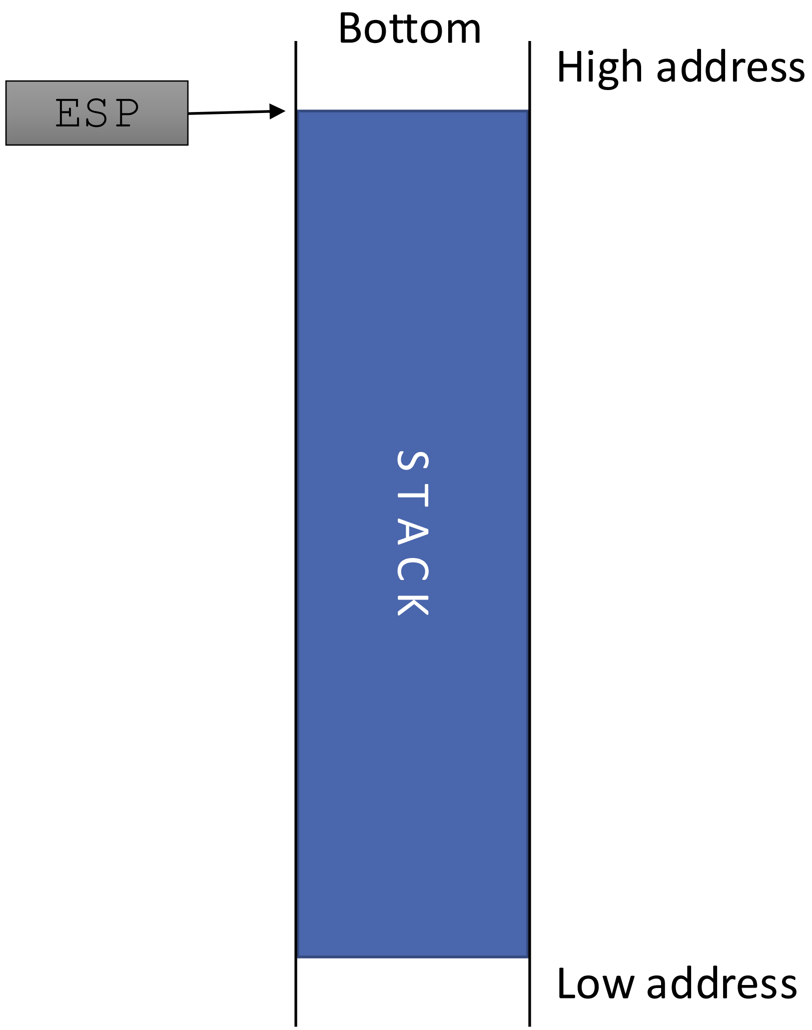
- Observation: Different compiler may use different alignment, spill different register, etc.
- Problems:
  - Exact distance of return address may be different between binaries
  - Exact address of buffer may be different
- Solutions:
  - Use multiple copies of the target address
  - Prepend a **NOP sled** to shellcode
    - NOPs → No operations are special one byte instructions to do nothing
  - Aim for target address pointing into NOP sled
    - Execution will slide into shellcode



Appendix: cdecl Calling Convetion (Optional)

cdecl

- Arguments are passed on the stack
  - Pushed right to left




```
…
F1(0xff, UINT_MAX, argv[0]);
…
```

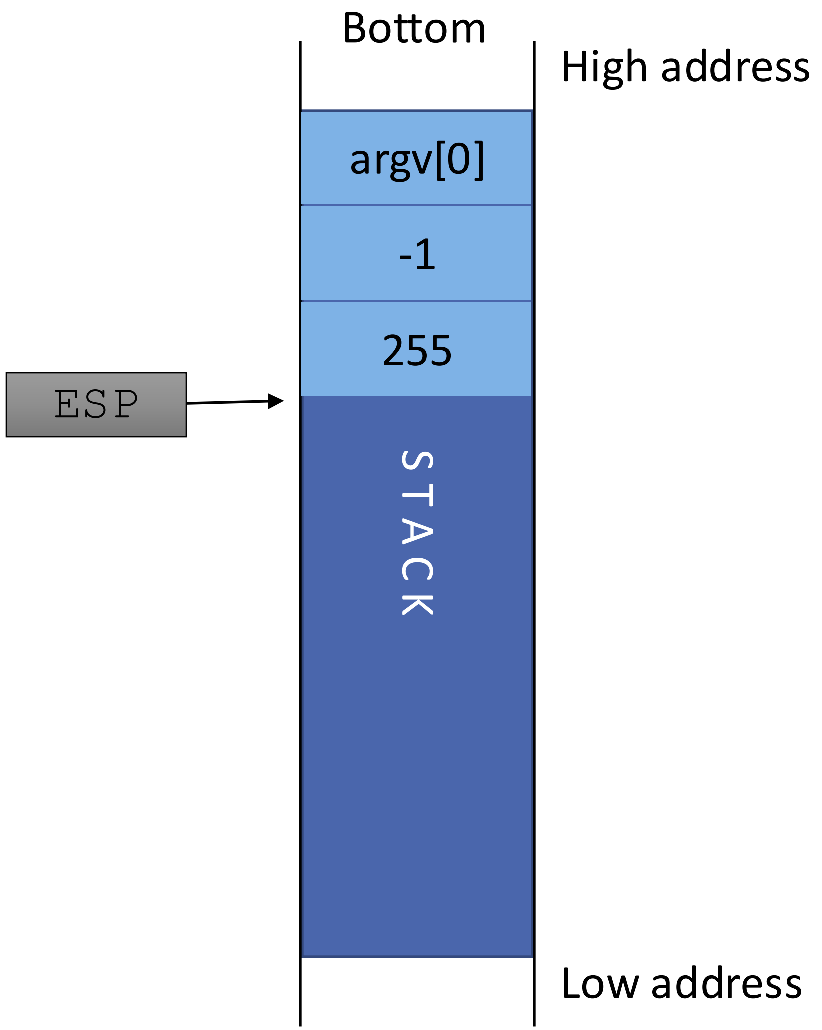
```
...
pushl   (%eax)
pushl   $-1
pushl   $255
call    F1
...
```

cdecl

- Arguments are passed on the stack
  - Pushed right to left

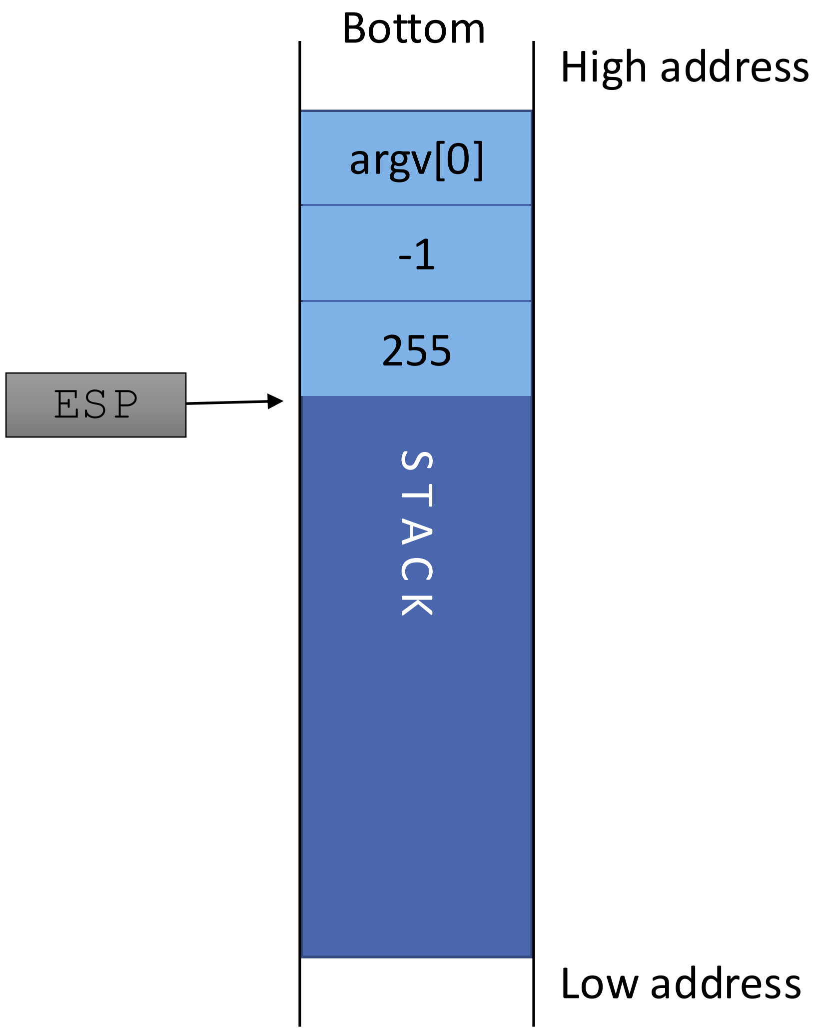
```
…
F1(0xff, UINT_MAX, argv[0]);
…
```

```
...
pushl   (%eax)
pushl   $-1
pushl   $255
call    F1
...
```



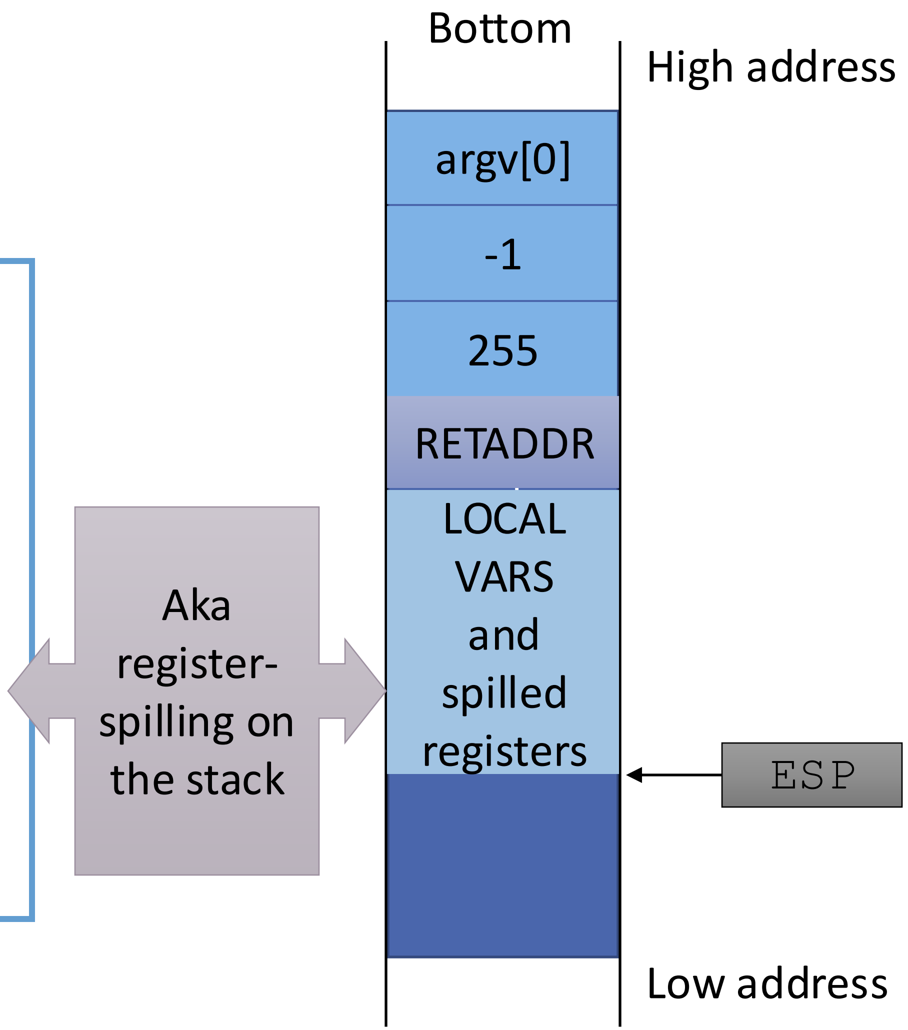
cdecl

- Arguments are passed on the stack
  - Pushed right to left
- eax, edx, ecx are caller saved
  - callee can overwrite without saving
- ebx, esi, edi are callee saved
  - callee must ensure they have same value on return



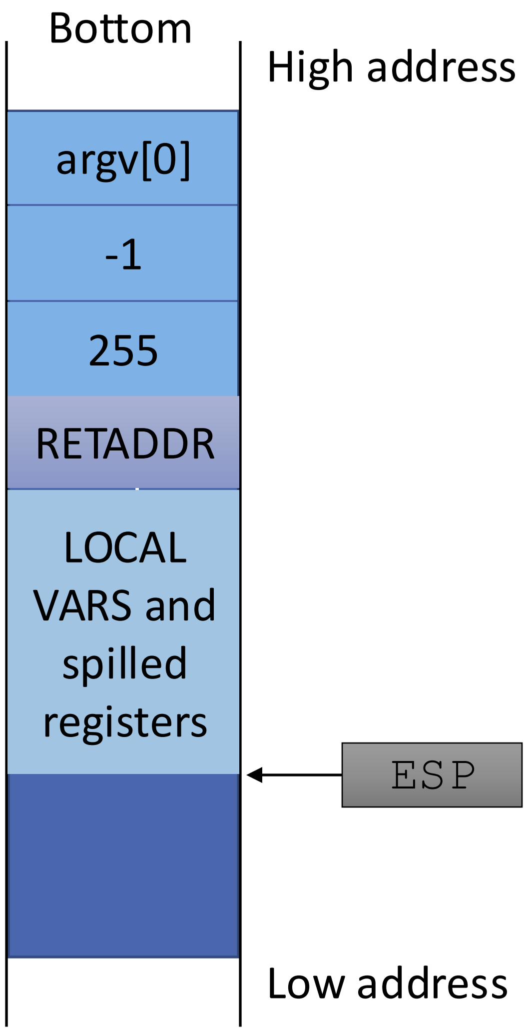
cdecl

- Arguments are passed on the stack
  - Pushed right to left
- eax, edx, ecx are caller saved
  - callee can overwrite without saving
- ebx, esi, edi are callee saved
  - callee must ensure they have same value on return



cdecl

- Arguments are passed on the stack
  - Pushed right to left
- eax, edx, ecx are caller saved
  - callee can overwrite without saving
- ebx, esi, edi are callee saved
  - callee must ensure they have same value on return
- eax used for function return value



Part 3. Data Execution Prevention

Writable and Executable Memory

- Code injection is possible because there is a memory area that is both writable and executable

Writable and Executable Memory

- Code injection is possible because there is a memory area that is both writable and executable
- Can be eliminated if we introduce the following policy

W^X Policy

The Write XOR Execute (W^X) policy mandates that in a program there are no memory pages that are both writable and executable

Writable and Executable Memory

- Code injection is possible because there is a memory area that is both writable and executable
- Can be eliminated if we introduce the following policy

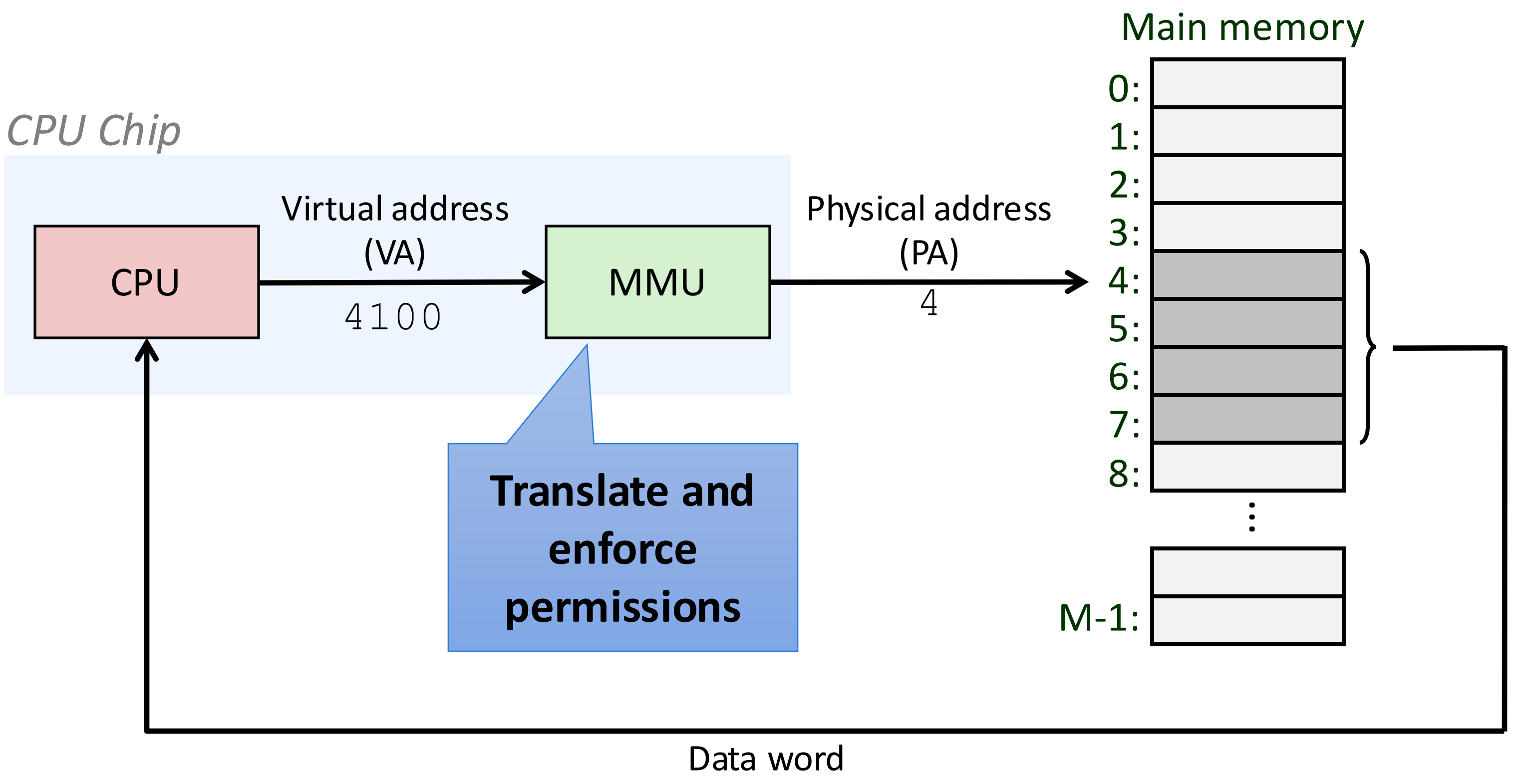
W^X Policy

How?

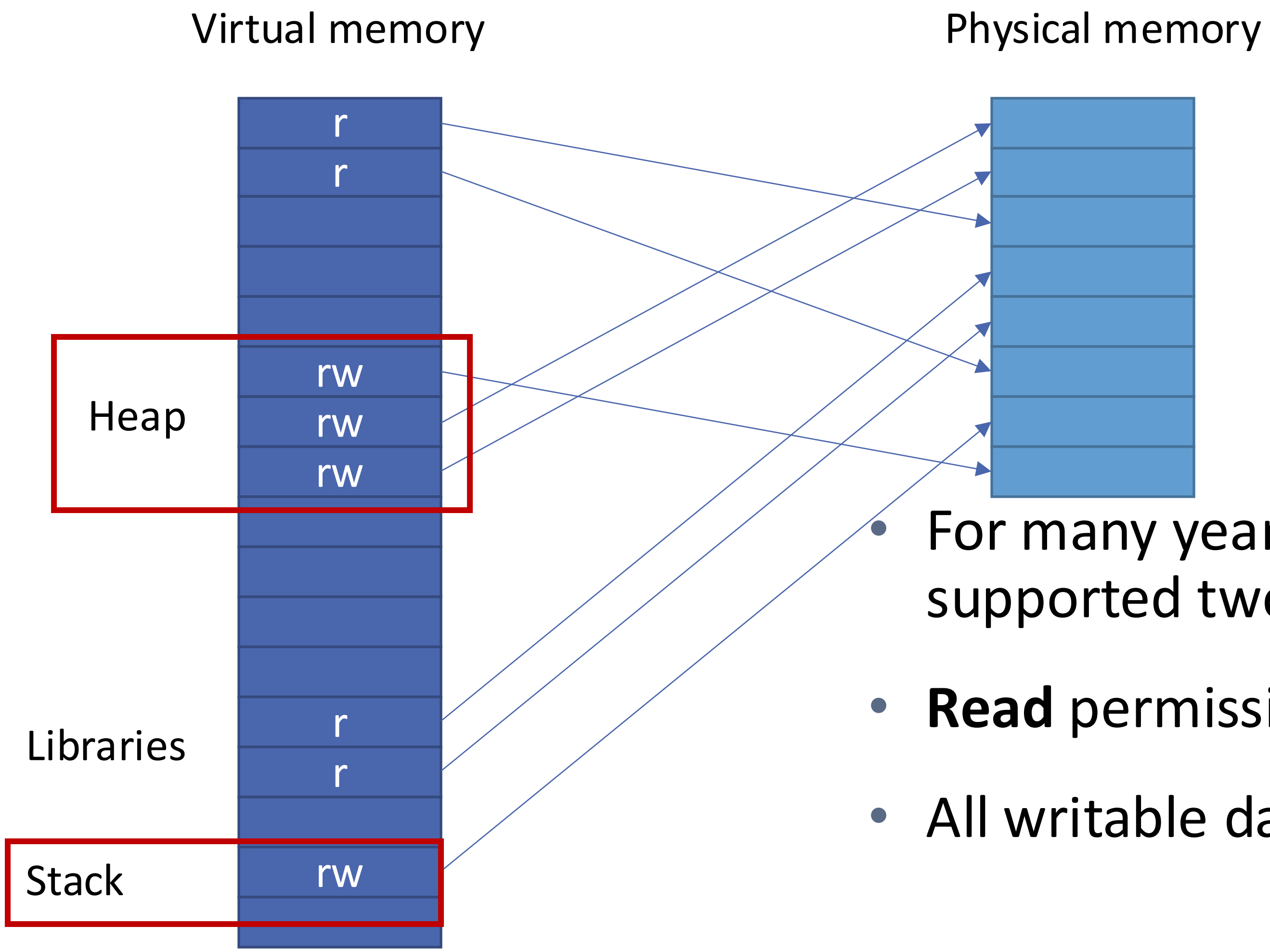
The Write XOR Execute (W^X) policy mandates that in a program there are no memory pages that are both writable and executable

The Memory Management Unit (MMU) - Paging

- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science



Page Permissions



- For many years (<2000), processors supported two permissions (bits)
- **Read** permission implied **Execute**
- All writable data segments violated W^X

Hardware Support: NX-bit

- Processor manufacturers introduced a new bit in page permissions to prevents code injections
- Coined **N**o-e**X**ecute or **E**xecute **N**ever
- The NX-bit (No-eXecute) was introduced first by AMD to resolve such issues in 2001
  - Asserting NX, makes a readable page non-executable
  - Frequently referred to as Data Execution Prevention (DEP) on Windows
- **Marketed as antivirus technology**

Virus Bulletin :: Enhance

virusbulletin.com/conference/vb2005/abstracts/enhanced-virus-protection

virus BULLETIN Covering the global threat landscape

Blog Bulletin VB

Enhanced virus protection

Costin Raiu *Kaspersky Lab*

download slides *(PDF)*

AMD Athlon 64 CPU Feature:

1. HyperTransport technology
2. Cool'n'Quiet technology
3. Enhanced Virus Protection for Microsoft Windows XP SP2

The AMD64 architecture is an affordable way of getting the power of 64-bit processing into a desktop computer. Interesting enough, AMD has not only designed an improved CPU core and longer registers, but they have also included a feature designed to significantly increase the security of modern operating systems.

The idea of hardware protection isn't new – every contemporary CPU includes at least a basic hardware mechanism for enforcing a security scheme, for instance, those from the Intel x86 family, based on

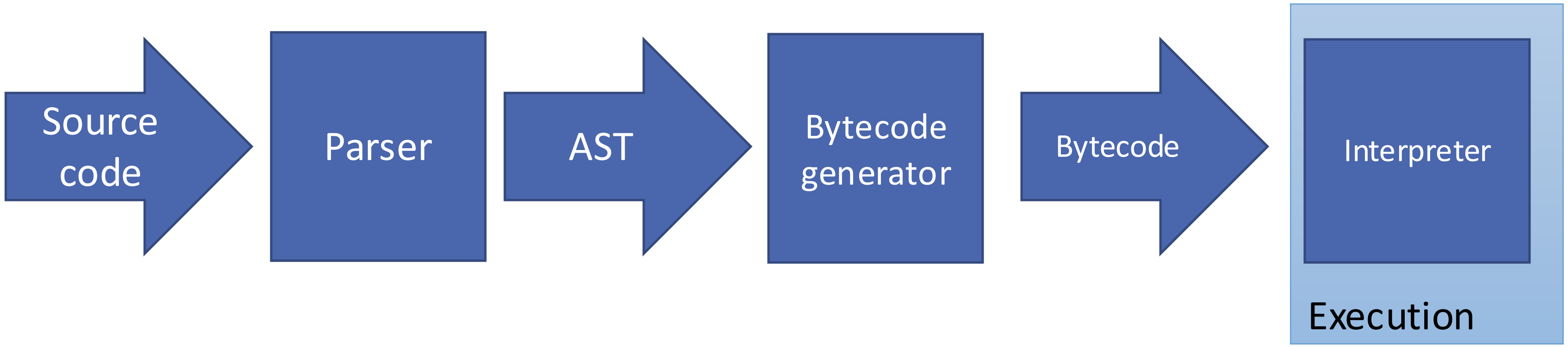
Adoption

- A non-executable stack was not immediately adopted
- The OS occasionally needed to place code in the stack
  - For example, trampoline code for handling UNIX signals
- Widely adopted today

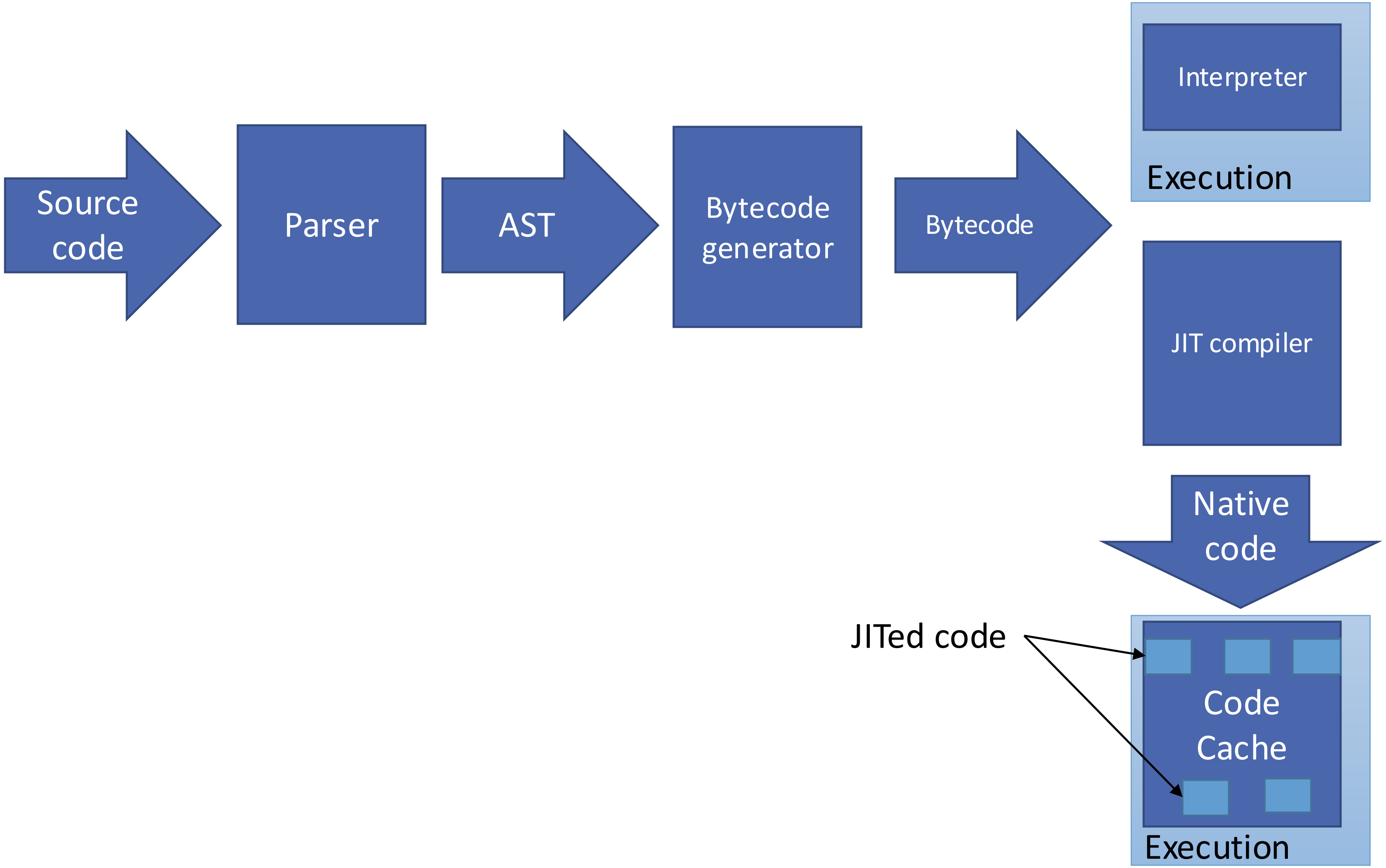
Unless You Are a Browser…

- Very popular software
  - Probably installed on every client device
- Large and complex software
- Execute JavaScript

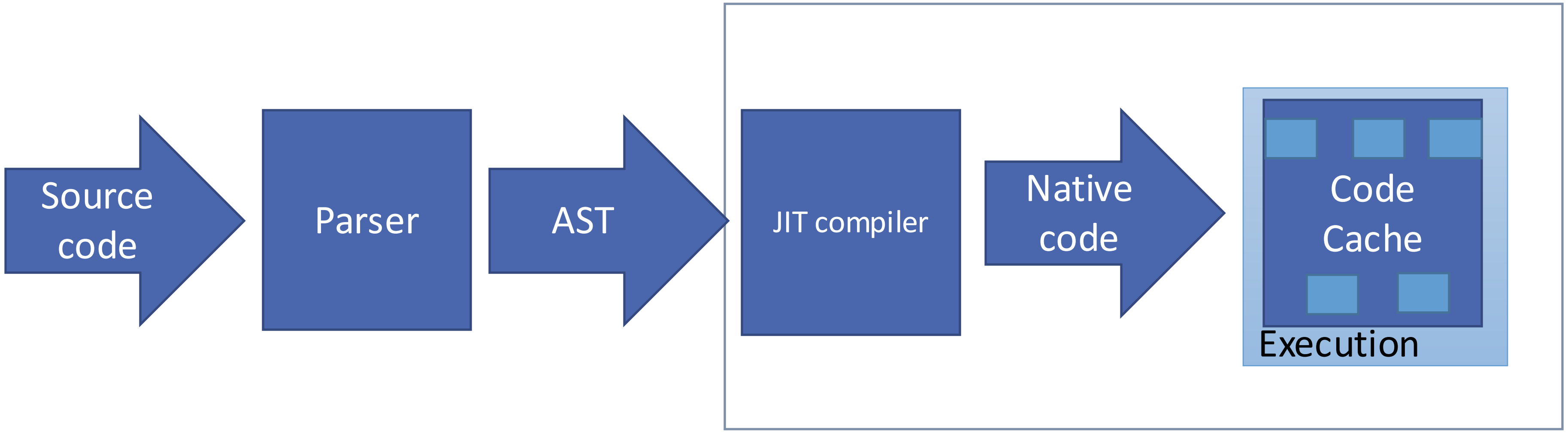
How Does JavaScript Run



How Does JavaScript Run



How Does JavaScript Run

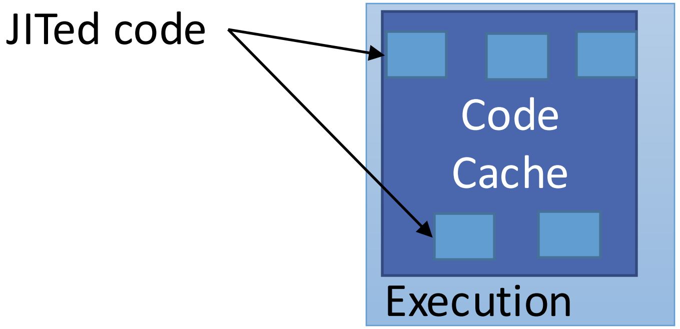


- Google V8 designed specifically to execute at speed.
- Bytecode generation skipped
- Directly emit native code
- Overall JavaScript execution improved by 150%

Code Cache

- JITed code and code cache have interesting properties from the perspective of the attacker
  - Code is continuously generated
  - Code needs to be executable

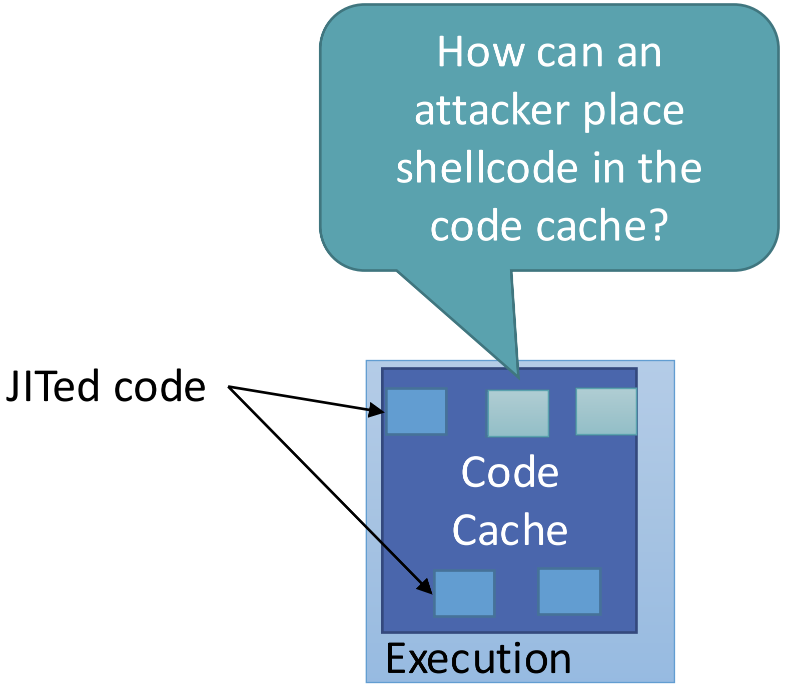
- **Violates the W^X policy**



Code Cache

- JITed code and code cache have interesting properties from the perspective of the attacker
  - Code is continuously generated
  - Code needs to be executable

- **Violates the W^X policy**



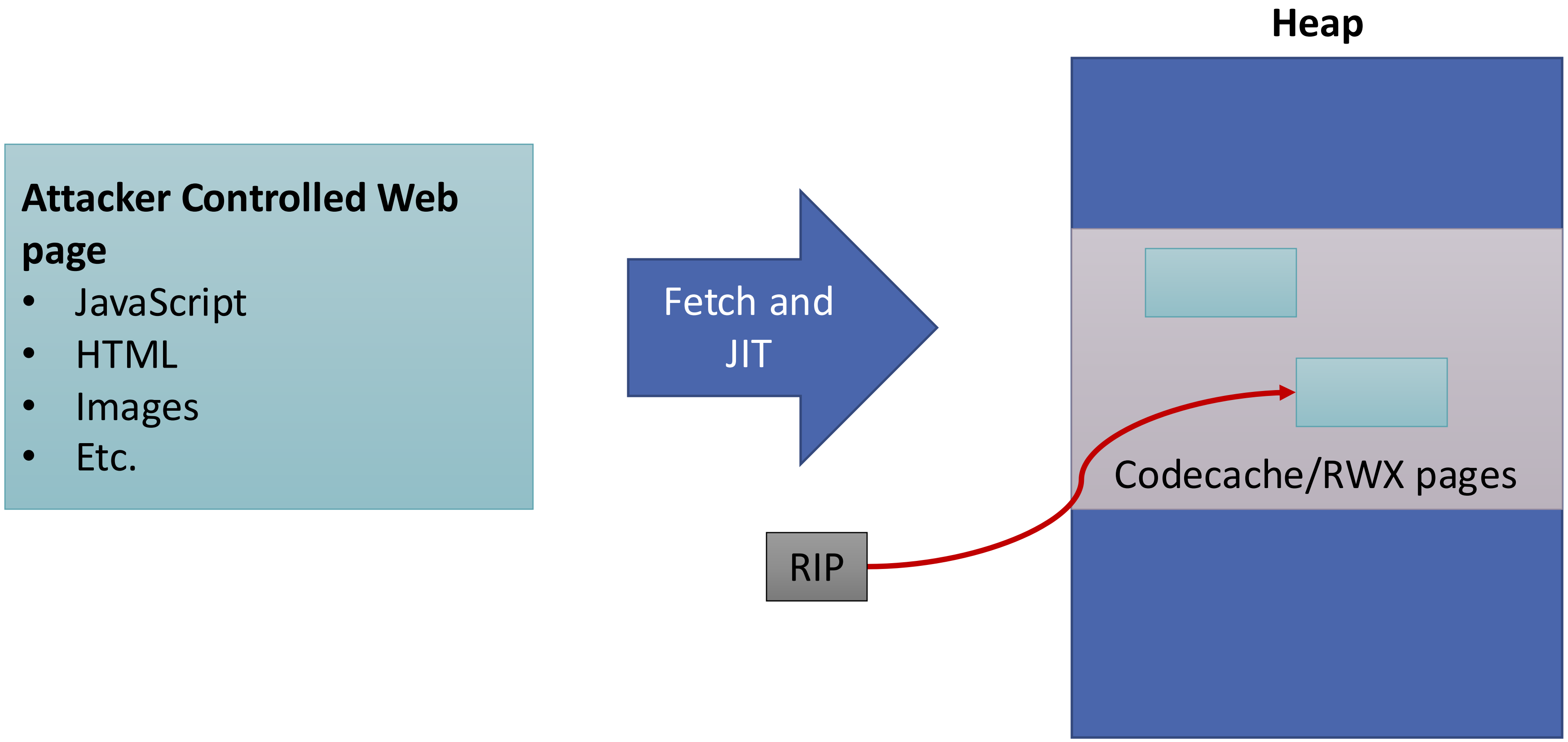
From JS to Code Cache

- JS code is JITed and placed in the code cache
- Some JS engines do not separate data and code

```
<html>
<body>
<script language='javascript'>

var myvar = unescape('%u\4F43%u\4552'); // CORE
myvar += unescape('%u\414C%u\214E'); // LAN!
alert("allocation done");

</script>
</body>
</html>
```

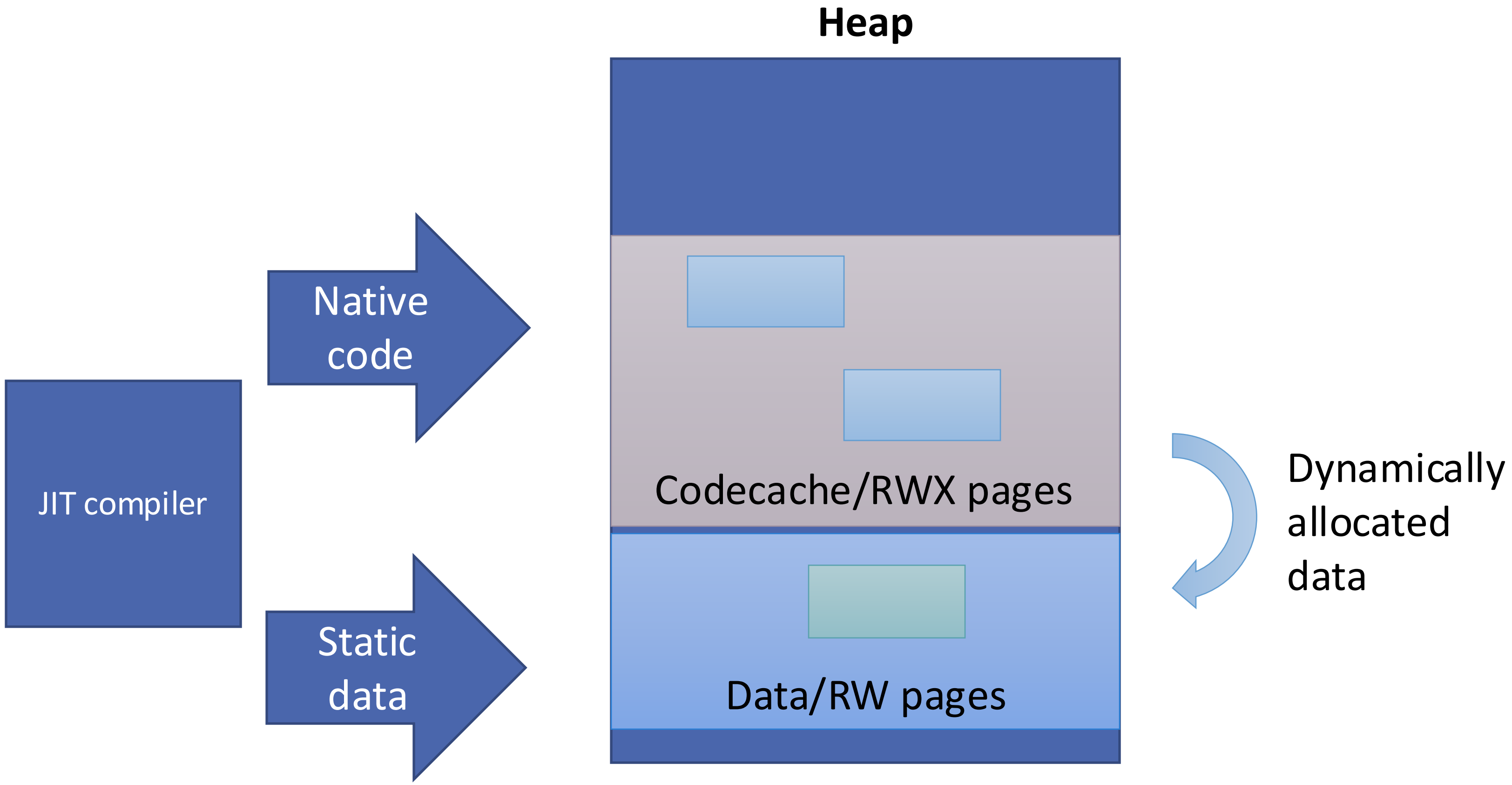
Code-Injection Attacks Against Browsers

- Return to code injected in the codecache

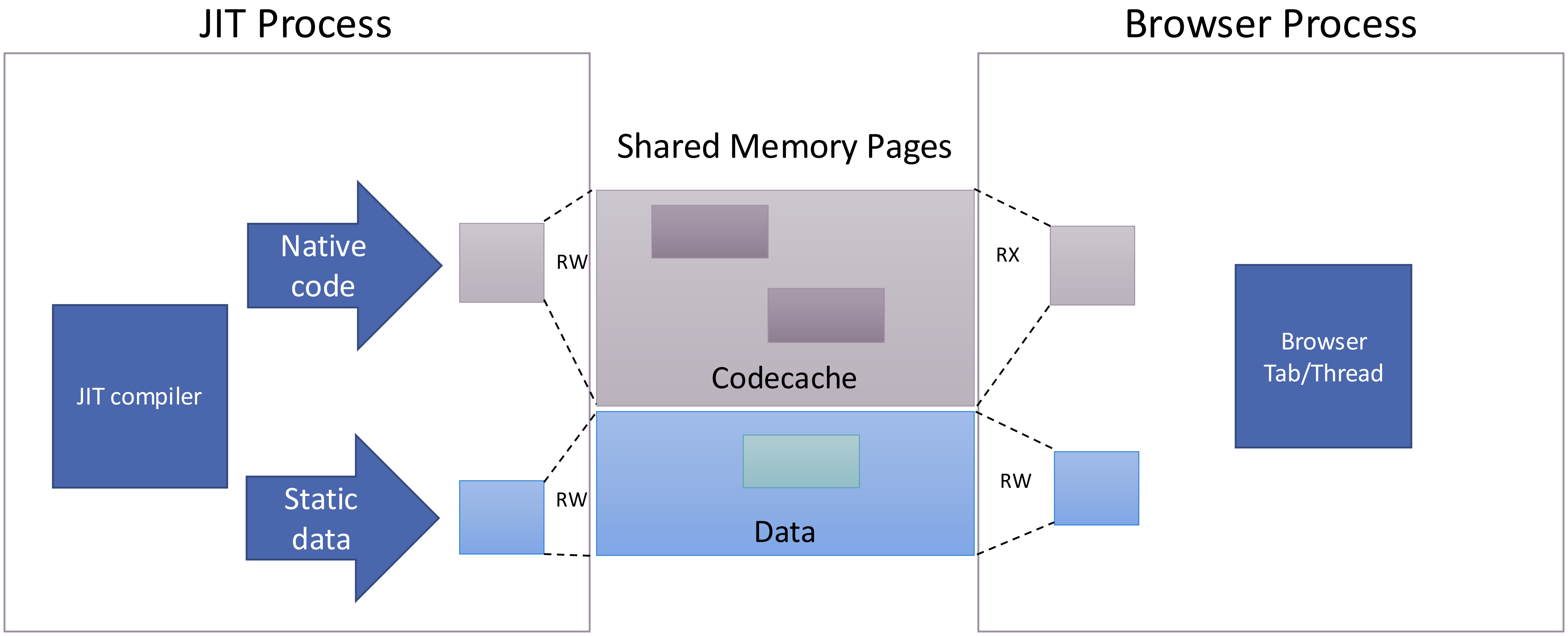


Avoiding Code Injection in Browsers

- Separate code and data into separate memory areas
- Still violates W^X



W^X Semantics in Browser Processes



Additional Reading (Optional)

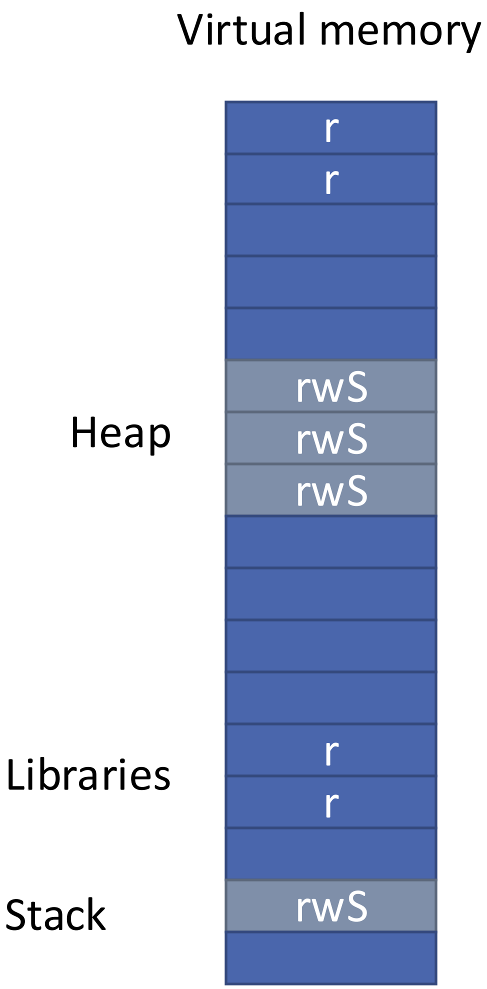
- The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines
  - https://www.portokalidis.net/files/devilinconstants_ndss15.pdf
- libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)
  - https://www.usenix.org/system/files/atc19-park-soyeon.pdf

Appendix: Emulating NX-bit (Optional)

Early Approaches: PAGEEXEC

- A Linux kernel patch emulating non-executable memory
- Introduced in 2000 by the PaX team
- PAGEEXEC refused code execution on selected writable pages
  - Heap and stack

Emulating Non-Executable Memory



- Mark writable pages so that access causes a page fault
  - Not present → a page-fault will be raised on every access
  - With supervisor bit (S) → Access only allowed from the kernel
- Custom page-fault handler intercepts and checks accesses:
  - Fault caused by other instruction → data access → OK
  - Faulting address is being executed → code execution → Violation

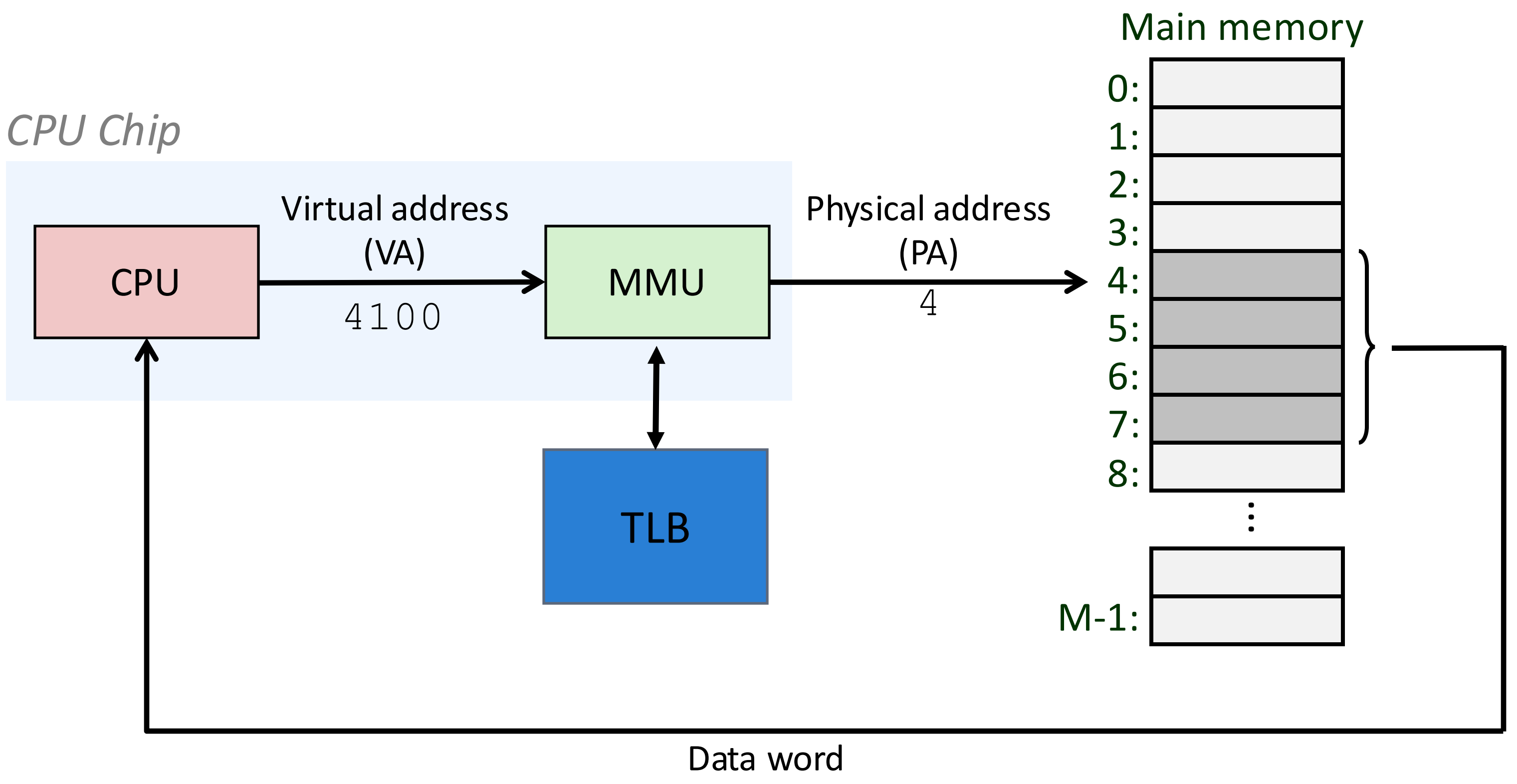
Emulating Non-Executable Memory

Should be very expensive….

- Mark writable pages so that access causes a page fault
  - Not present → a page-fault will be raised on every access
  - With supervisor bit (S) → Access only allowed from the kernel
- Custom page-fault handler intercepts and checks accesses:
  - Fault caused by other instruction → data access → OK
  - Faulting address is being executed → code execution → Violation

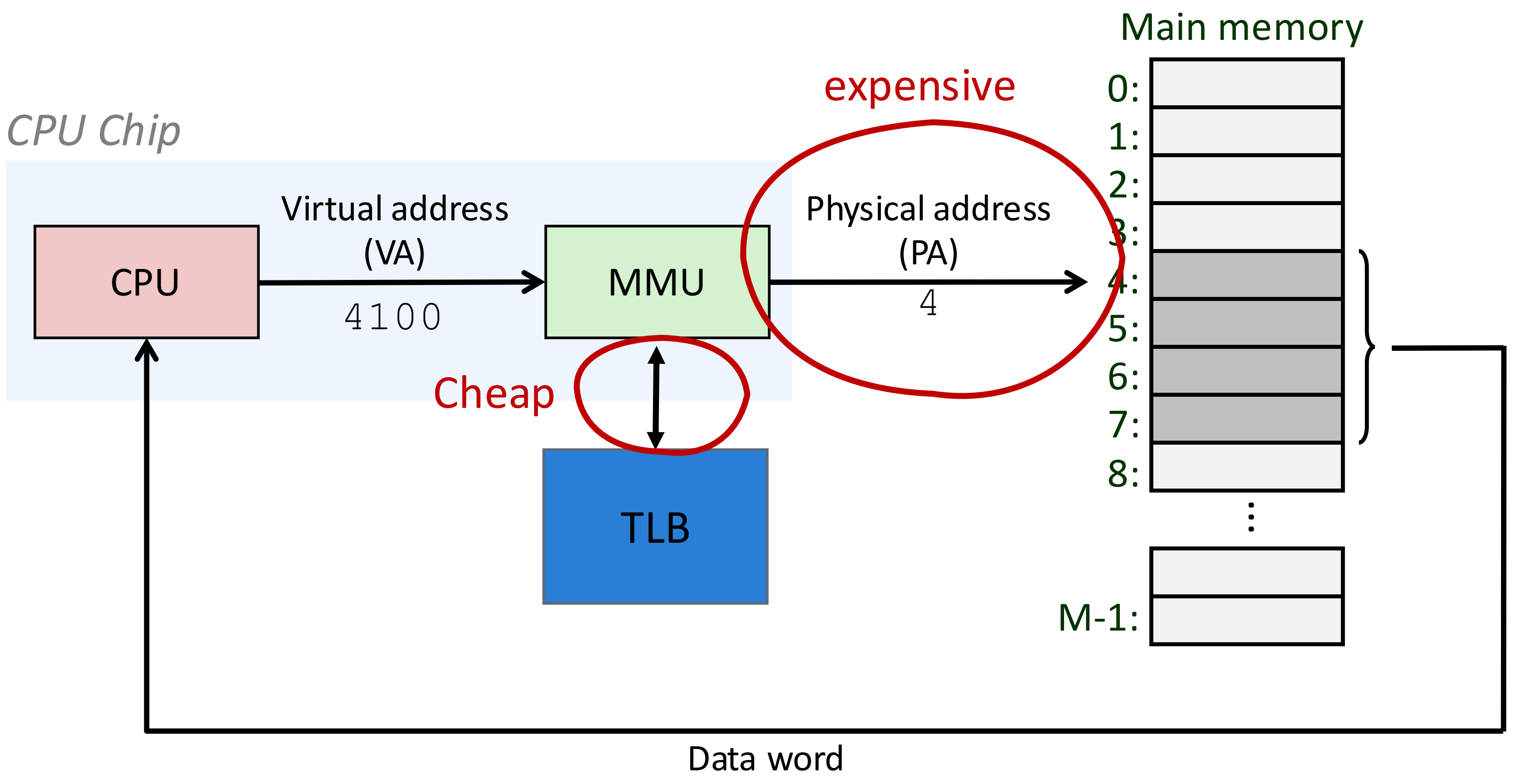
Translation Lookaside Buffer (TLB)

- A cache for storing the translations for the most frequently accessed pages



Translation Lookaside Buffer (TLB)

- A cache for storing the translations for the most frequently accessed pages



Split TLBs

- Instruction TLB (ITLB) used when fetching bytes to be decoded and executed as an instruction
  - PC → memory `addr`
  - `addr` → ITLB
- Data TLB (DTLB) used when reading/write bytes required by the executing instruction
  - `(addr)` -> memory `addr`
    - Example: mov (addr), reg
  - `addr` -> DTLB

Split TLBs and PAGEEXEC

- Fault caused because PC points within data area → Violation
- Fault caused by other access
  - Remove supervisor bit from page
  - Complete load which will be added to the DTLB
  - Add supervisor bit to page
  - Subsequent accesses to address will be served by the DTLB
    - Until it is flushed or the entry for the address evicted