MONASH University

FIT5003: Standalone programs: C programming Language Vulnerabilities

Dr Xiaoning Du
Department of Software Systems and Cybersecurity
Faculty of Information Technology

Learning Outcomes of This Lecture

- Understand the importance of buffer overflow attacks
- Describe and analyze how buffer overflow works
- Know how to prevent and mitigate buffer overflows
- Show some more advanced types of code injection attacks

Why study it?

- The C program
  - Popular among

IEEE Spectrum's Top Programming Languages 2022

*https://spectrum.ieee.org/top-programming-languages/

MONASH University

Why study it?

- The C programming language is popular and important
  - Popular among all the programming languages
  - Heavily used in implementation of critical software
    - Most OS kernels and utilities, e.g., linux kernel, shell,…
    - Many high-performance servers, e.g., Apache httpd, MySQL, redis,…
    - Many embedded systems, e.g., industrial control systems, automobiles,…
    - Known open-source libraries, e.g., OpenSSL, OpenCV,…
- Buffer overflow vulnerabilities
  - Happen easily
  - Bad consequences

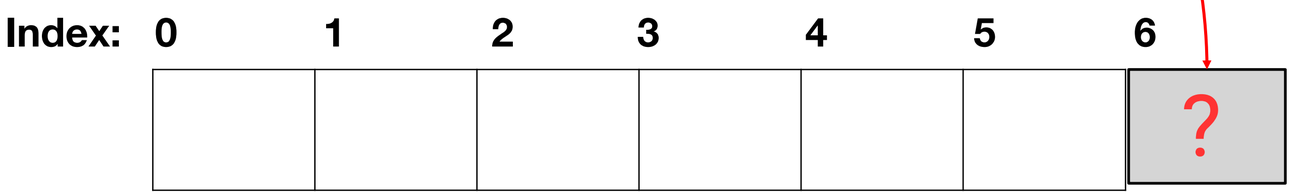
Terminology and Definition

- Buffer is a container for data
- A buffer overflow is a bug that affects software written in low-level programming languages, typically in C and C++, with significant security implications
- A buffer overflow is defined as any access of a buffer outside of its allotted bounds
  - Could be an over-read or an over-write
  - Could be during iteration “running of the end” or by direct access
  - Out-of-bounds access could be to addresses that precede or follow the buffer

An Example of Buffer Overflow

- Buffer are bytes in memory

```
int main()
{
    char buf[6];
    buf[6] = 'o';
}
```



https://flux.qa/EH2NNF

Buffer overflow Consequence

Nothing

- For performance reasons, the OS might allocate multiple bytes
- Overflowing **buf** by a few bytes might not result in an error

Segmentation Fault

- If the index is large, it will eventually reach a memory zone not allocated to the program
- The OS detects it and stops the program
- The overflow might also corrupt existing data

Custom Code Execution

- The overflowing bytes have to redirect the execution flow to the customized code

More examples of buffer overflow Caused by Insecure Coding

```
void DoSomething(char *BuffSrc, int BuffLen)
{
    char BuffDest[16];
    memcpy(BuffDest,BuffSrc,BuffLen);
}
```

Example 1

```
void main()
{
   char buf[32];
   gets(buf);
}
```

Example 2

```
void function1(const char* input)
{
   char buffer[10];
   strcpy(buffer, input);
}
```

Example 3

- In reality, inputs come from users in myriad ways:
  - Text input
  - Packets
  - Environment variables
  - File input
- Validating assumptions about user input is extremely important!

Long History of Buffer Overflow

- Morris worm 1988
  - One of the first computer worms; $10-100M loss
- Code Red worm 2001
  - It attacked Microsoft's IIS web server; 300K machines infected in 14h
- SQL Slammer 2003
  - It attacked Microsoft's SQL Server and Desktop Engine database products; 75K machines infected in 10m
- Stagefright against Android phones 2015
  - Allows an attacker to perform arbitrary operations on the victim's device through remote code execution and privilege escalation

The Common Weakness Enumeration (CWE™) Top 25 Most Dangerous Software Errors

Below is a brief listing of the weaknesses in the **2022** CWE Top 25, including the ov

Rank	ID	Name
1	CWE-787	Out-of-bounds Write
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
4	CWE-20	Improper Input Validation
5	CWE-125	Out-of-bounds Read
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'
7	CWE-416	Use After Free
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	CWE-352	Cross-Site Request Forgery (CSRF)
10	CWE-434	Unrestricted Upload of File with Dangerous Type

2023 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 Home | Share via: | View in table format | Key Insights

1. Out-of-bounds Write
   CWE-787 | CVEs in KEV: 70 | Rank Last Year: 1
2. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
   CWE-79 | CVEs in KEV: 4 | Rank Last Year: 2
3. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
   CWE-89 | CVEs in KEV: 6 | Rank Last Year: 3
4. Use After Free
   CWE-416 | CVEs in KEV: 44 | Rank Last Year: 7 (up 3) ▲
5. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
   CWE-78 | CVEs in KEV: 23 | Rank Last Year: 6 (up 1) ▲
6. Improper Input Validation
   CWE-20 | CVEs in KEV: 35 | Rank Last Year: 4 (down 2) ▼
7. Out-of-bounds Read
   CWE-125 | CVEs in KEV: 2 | Rank Last Year: 5 (down 2) ▼
8. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
   CWE-22 | CVEs in KEV: 16 | Rank Last Year: 8
9. Cross-Site Request Forgery (CSRF)
   CWE-352 | CVEs in KEV: 0 | Rank Last Year: 9
10. Unrestricted Upload of File with Dangerous Type
    CWE-434 | CVEs in KEV: 5 | Rank Last Year: 10

http://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html
https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

MONASH University

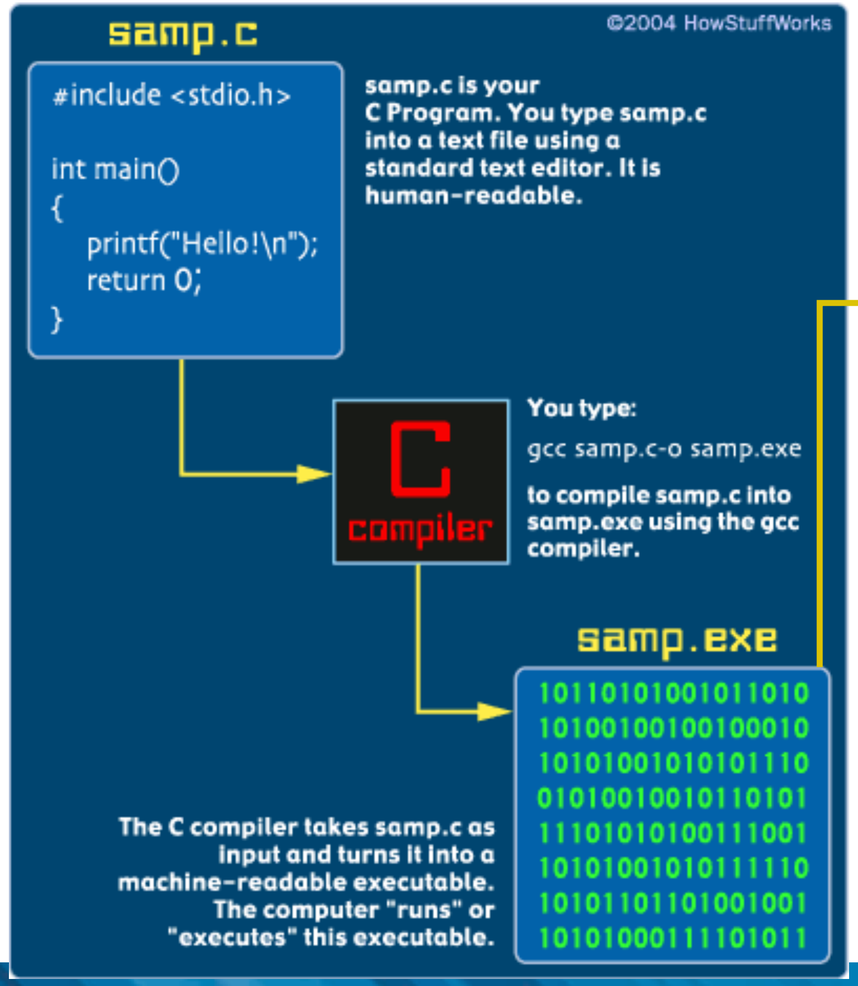
What we will study

- Understand how buffer overflow can be **exploited**, and how to defend against it
- Require knowledge on
  - The compiler
  - The OS
  - The architecture
  - A whole-systems view

Buffer Overflow Exploits

- Exploits depend on how software programs run “under the hood”
  - Compilation into binary machine code
  - How functions are called
  - Memory layout / stack frame structure
- Details depend on programming language, CPU hardware, and OS
  - Our examples: C language (gcc compiler), Intel x86 CPU and Linux Ubuntu

How a program is compiled and executed?




```
0:   8d 4c 24 04             lea    ecx,[esp+0x4]
4:   83 e4 f0                and    esp,0xfffffff0
7:   ff 71 fc                push   DWORD PTR [ecx-0x4]
a:   55                      push   ebp
b:   89 e5                   mov    ebp,esp
d:   51                      push   ecx
e:   83 ec 04                sub    esp,0x4
11:  83 ec 0c                sub    esp,0xc
14:  68 c0 84 04 08          push   0x80484c0
19:  e8 dc 82 04 08          call   80482fa <_main+0x80482fa>
1e:  83 c4 10                add    esp,0x10
21:  b8 00 00 00 00          mov    eax,0x0
26:  8b 4d fc                mov    ecx,DWORD PTR [ebp-0x4]
29:  c9                      leave
2a:  8d 61 fc                lea    esp,[ecx-0x4]
2d:  c3                      ret
```

```
0x804840b <main>:     0x8d  0x4c 0x24 0x04 0x83 0xe4 0xf0 0xff
0x8048413 <main+8>:     0x71 0xfc 0x55 0x89 0xe5 0x51 0x83 0xec
0x804841b <main+16>:    0x04 0x83 0xec 0x0c 0x68 0xc0 0x84 0x04
0x8048423 <main+24>:    0x08 0xe8 0xb7 0xfe 0xff 0xff 0x83 0xc4
0x804842b <main+32>:    0x10 0xb8 0x00 0x00 0x00 0x00 0x8b 0x4d
0x8048433 <main+40>:    0xfc 0xc9 0x8d 0x61 0xfc 0xc3 0x66 0x90
```

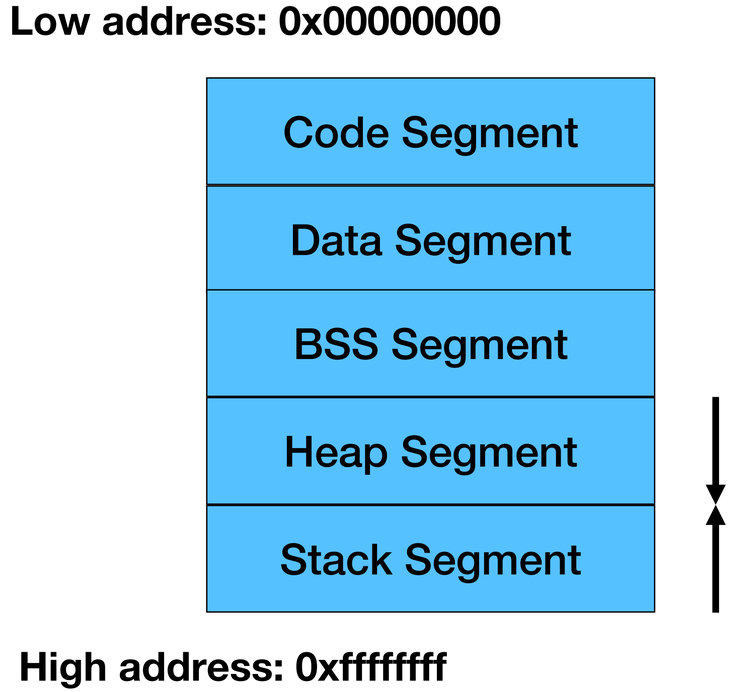
MONASH University

Assembly Code

- General Intel assembly syntax:
  - op <destination>, <source>
- Common assembly instructions
  - **Arithmetic**: add, sub, inc, mul, div; **Logical**: and, or, not, xor, nop;
  - **Control flow**
    - **Comparison**: cmp, test
    - **Jumps**: unconditional jump: jmp, and conditional jump: jle, je, jge
    - **Function call**: call, ret
  - **Copying data**: mov
  - **Stack manipulation**: push, pop
- x86 assembly guide: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Memory Layout

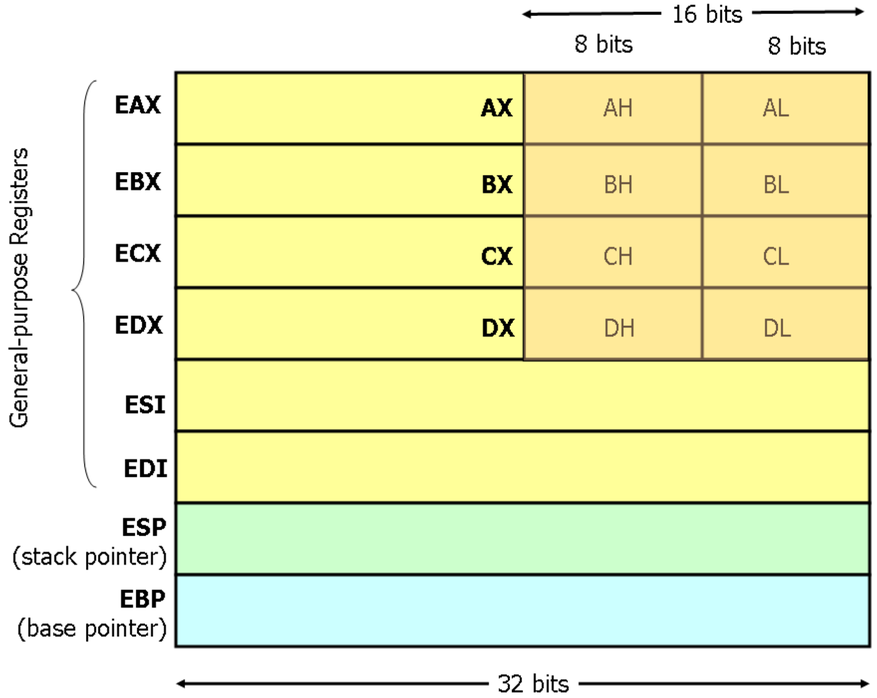
- Text (code) segment: program code and fixed program constants
- Data segment: initialised global and static variables
- BSS (Block Started by Symbol) segment: uninitialised global and static variables
- Heap segment: dynamic program data (e.g., malloc)
- Stack segment: function local variables, arguments, context (calling function return address/stack frame) – can also contain code!



Registers

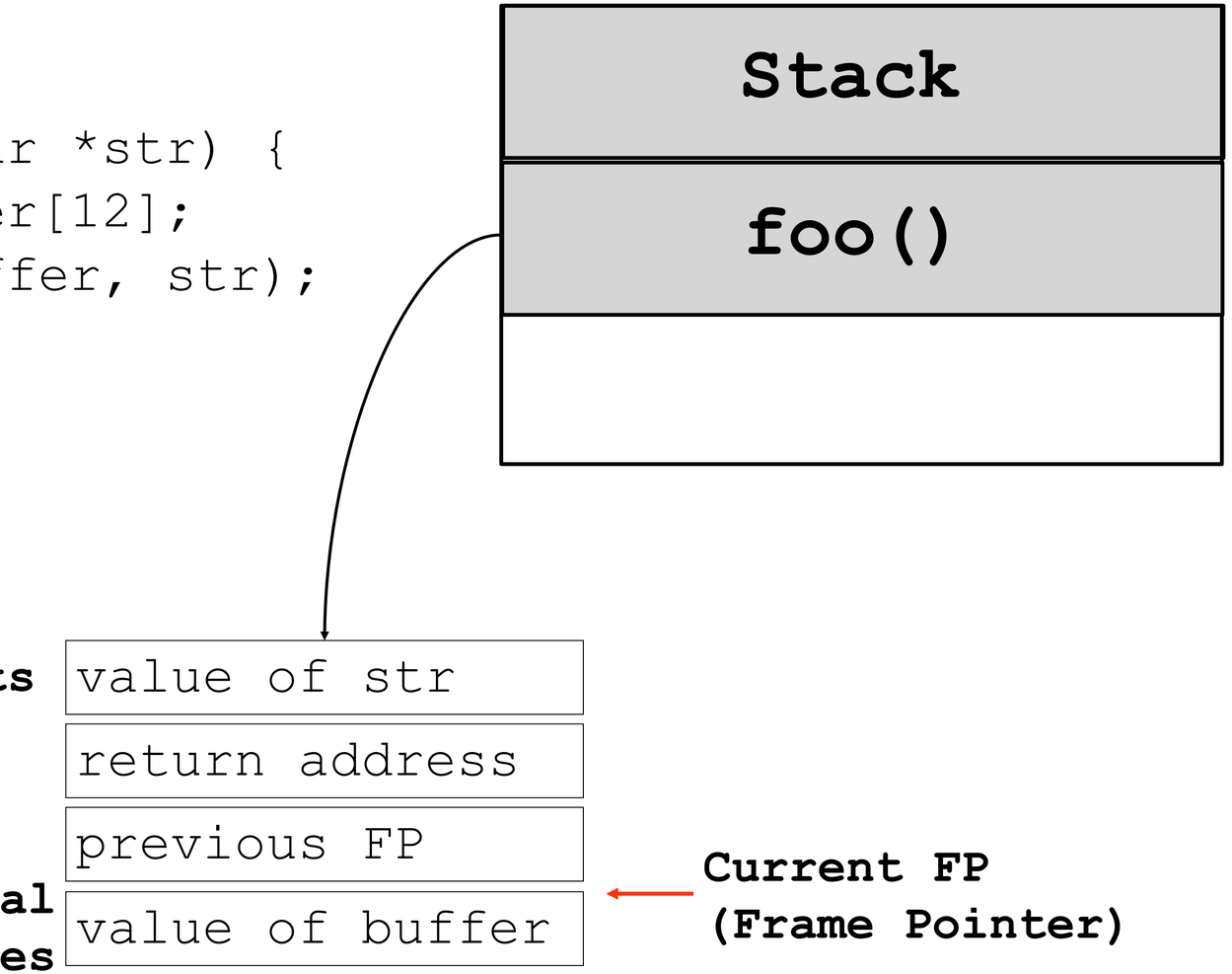
- A processor register is a quickly accessible location available to a computer's processor. Registers usually consist of a small amount of fast storage

- Some registers are for specific purposes
  - esp (stack pointer) will be updated when push or pop values to the stack; it always points to the frontier of the stack
  - ebp (base pointer or frame pointer)

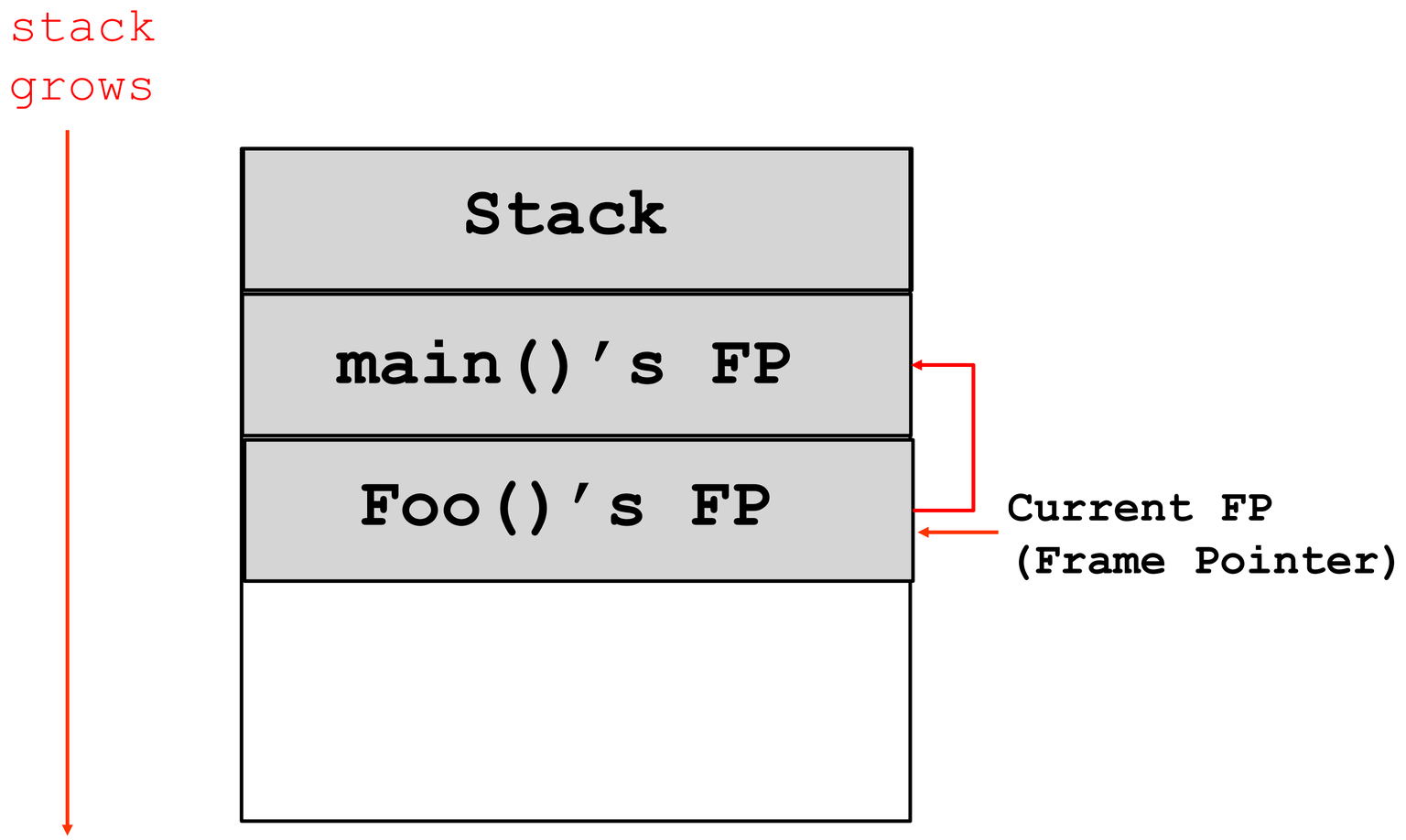


Function Frame

```
void foo(char *str) {
  char buffer[12];
  strcpy(buffer, str);
}
```



Link between frames



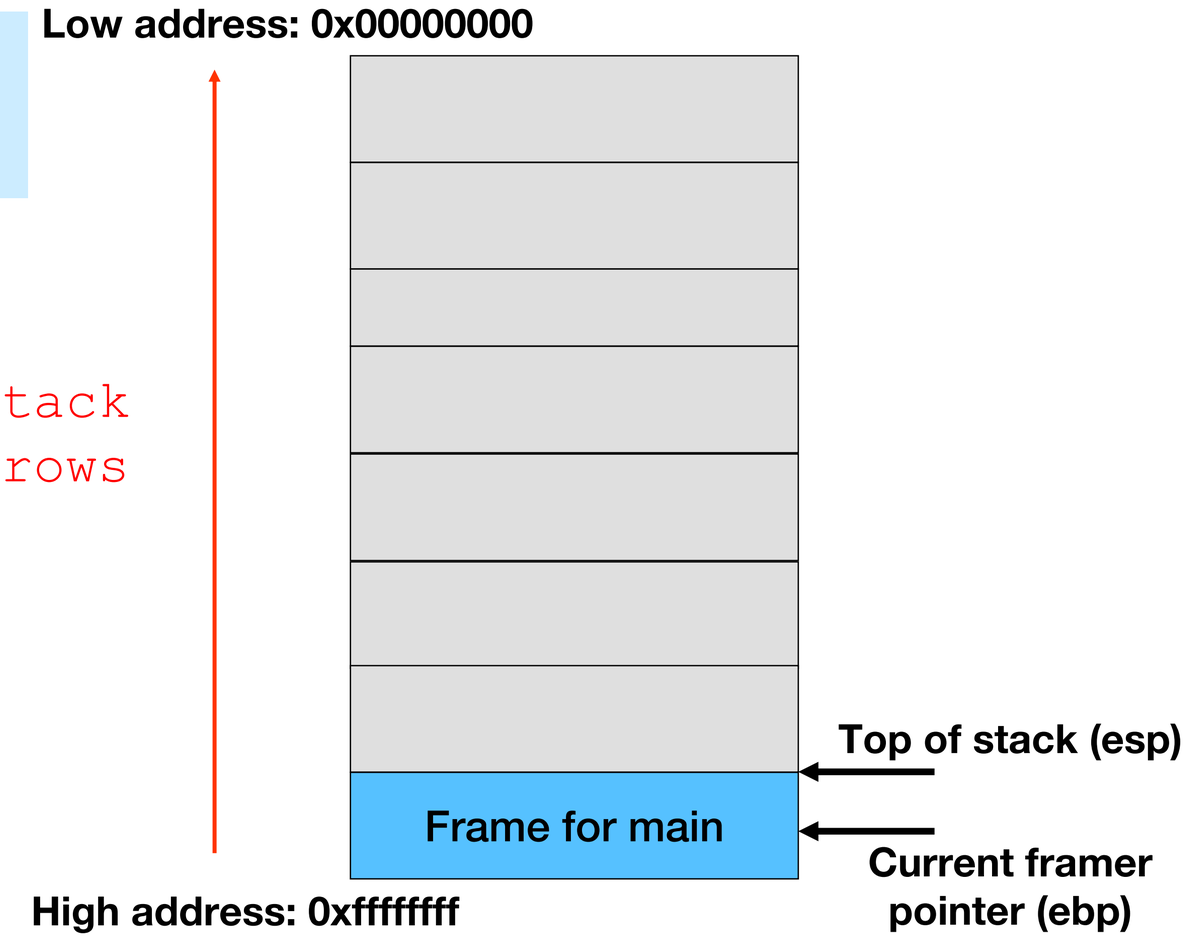

```
void foo(char *str) {
  char buffer[12];
  strcpy(buffer, str);
}

int main() {
  char *str = "test";
  foo(str);
}
```

Stack and Function Calls

- ✓ **%esp always points to the frontier of the stack**
  - **Push inserts its operand on top of stack, decrements %esp**
  - **Pop removes a value from top of stack into its operand, increments %esp**
- ✓ **%ebp always points to the cell storing the old %ebp**

- Caller function:
  1. Push arguments onto the stack (in reverse)
  2. Push the return address, i.e., the address of the instruction you want run after control returns to you
  3. Jump to the function's address
- Called function:
  4. Push the old frame pointer onto the stack (%ebp)
  5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
  6. Push local variables onto the stack
- Returning:
  7. Reset the previous stack frame
  8. Jump back to return address

```
void test_funciton (int a, int b)
{
    int flag;
    char buffer[10];

    flag = 31337;
    buffer[0] = 'A';
}

int main()
{
    test_function(1, 2);
}
```

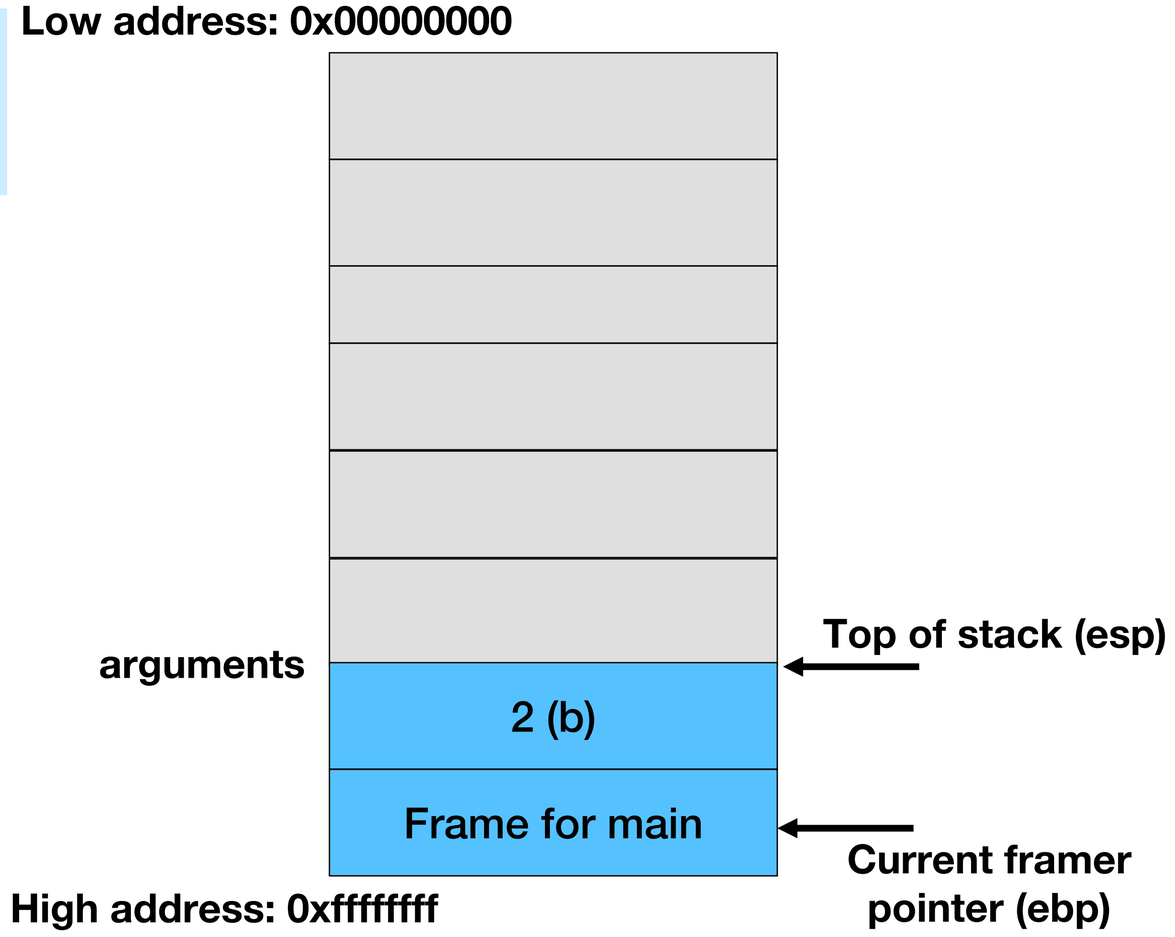


Stack and Function Calls

- ✓ **%esp always points to the frontier of the stack**
  - **Push inserts its operand on top of stack, decrements %esp**
  - **Pop removes a value from top of stack into its operand, increments %esp**
- ✓ **%ebp always points to the cell storing the old %ebp**

- Caller function:
  1. Push arguments onto the stack (in reverse)
  2. Push the return address, i.e., the address of the instruction you want run after control returns to you
  3. Jump to the function's address
- Called function:
  4. Push the old frame pointer onto the stack (%ebp)
  5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
  6. Push local variables onto the stack
- Returning:
  7. Reset the previous stack frame
  8. Jump back to return address

```
void test_funciton (int a, int b)
{
    int flag;
    char buffer[10];

    flag = 31337;
    buffer[0] = 'A';
}

int main()
{
    test_function(1, 2);
}
```



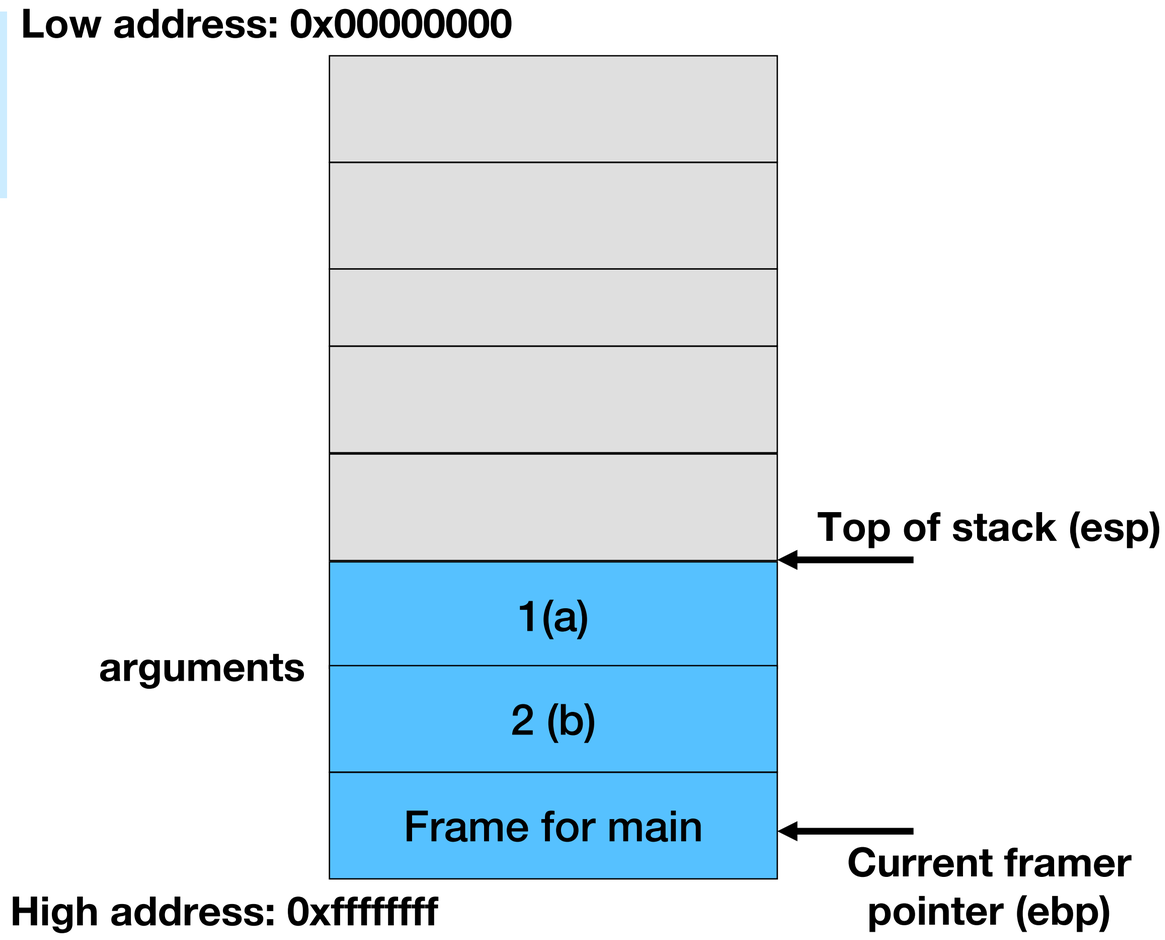
Stack and Function Calls

✓ **%esp always points to the frontier of the stack**
- **Push inserts its operand on top of stack, decrements %esp**
- **Pop removes a value from top of stack into its operand, increments %esp**

✓ **%ebp always points to the cell storing the old %ebp**

- Caller function:
1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction you want run after control returns to you
3. Jump to the function's address
- Called function:
4. Push the old frame pointer onto the stack (%ebp)
5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
6. Push local variables onto the stack
- Returning:
7. Reset the previous stack frame
8. Jump back to return address

```c
void test_funciton (int a, int b)
{
    int flag;
    char buffer[10];

    flag = 31337;
    buffer[0] = 'A';
}

int main()
{
    test_function(1, 2);
}
```



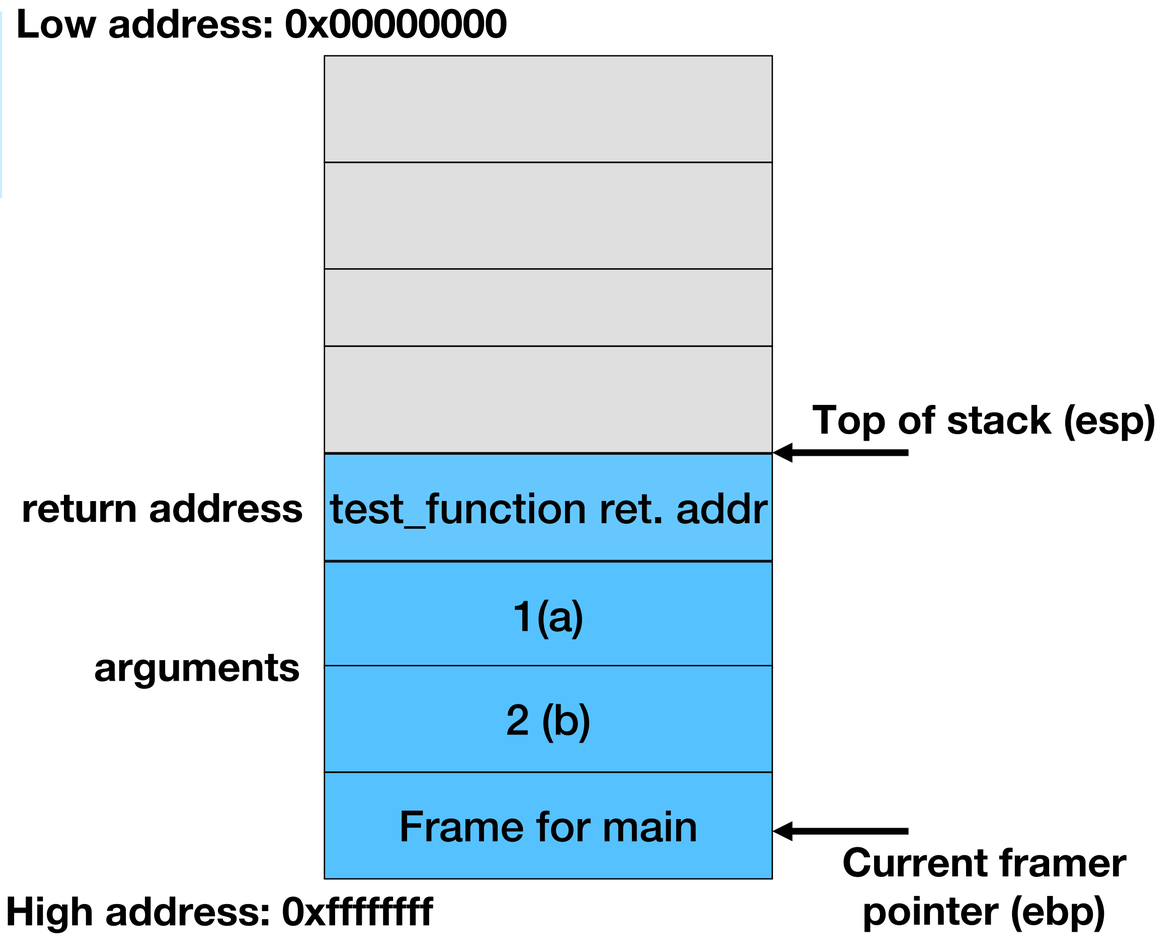
Stack and Function Calls

✓ **%esp always points to the frontier of the stack**
- **Push inserts its operand on top of stack, decrements %esp**
- **Pop removes a value from top of stack into its operand, increments %esp**

✓ **%ebp always points to the cell storing the old %ebp**

- Caller function:
  1. Push arguments onto the stack (in reverse)
  2. Push the return address, i.e., the address of the instruction you want run after control returns to you
  3. Jump to the function's address
- Called function:
  4. Push the old frame pointer onto the stack (%ebp)
  5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
  6. Push local variables onto the stack
- Returning:
  7. Reset the previous stack frame
  8. Jump back to return address

```c
void test_funciton (int a, int b)
{
    int flag;
    char buffer[10];

    flag = 31337;
    buffer[0] = 'A';
}

int main()
{
    test_function(1, 2);
}
```



MONASH University

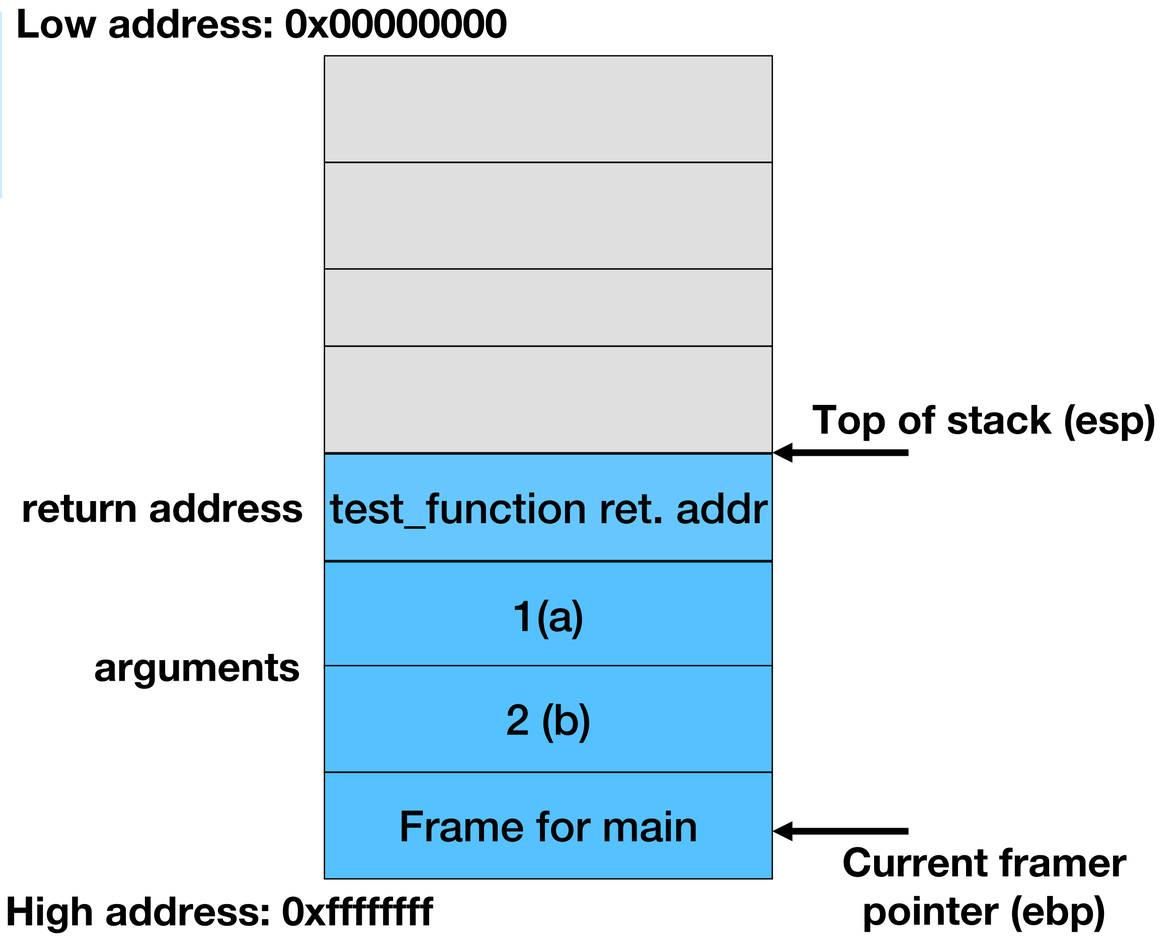
Stack and Function Calls

✓ **%esp always points to the frontier of the stack**
- **Push inserts its operand on top of stack, decrements %esp**
- **Pop removes a value from top of stack into its operand, increments %esp**

✓ **%ebp always points to the cell storing the old %ebp**

- Caller function:
  1. Push arguments onto the stack (in reverse)
  2. Push the return address, i.e., the address of the instruction you want run after control returns to you
  3. Jump to the function's address
- Called function:
  4. Push the old frame pointer onto the stack (%ebp)
  5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
  6. Push local variables onto the stack
- Returning:
  7. Reset the previous stack frame
  8. Jump back to return address

```c
void test_funciton (int a, int b)
{
    int flag;
    char buffer[10];

    flag = 31337;
    buffer[0] = 'A';
}

int main()
{
    test_function(1, 2);
}
```



MONASH University

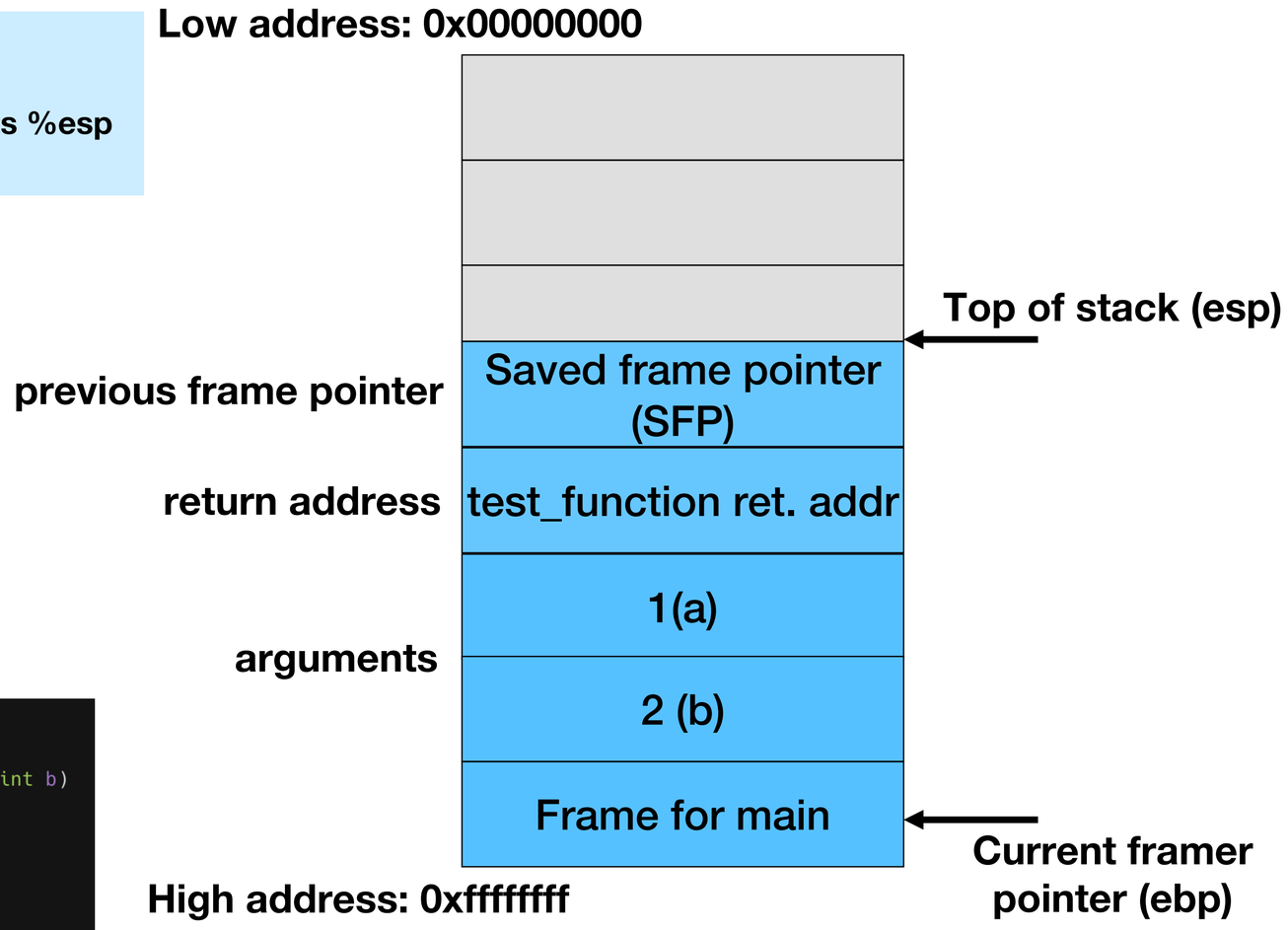
Stack and Function Calls

✓ **%esp always points to the frontier of the stack**
- **Push inserts its operand on top of stack, decrements %esp**
- **Pop removes a value from top of stack into its operand, increments %esp**

✓ **%ebp always points to the cell storing the old %ebp**

- Caller function:
  1. Push arguments onto the stack (in reverse)
  2. Push the return address, i.e., the address of the instruction you want run after control returns to you
  3. Jump to the function's address
- Called function:
  4. Push the old frame pointer onto the stack (%ebp)
  5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
  6. Push local variables onto the stack
- Returning:
  7. Reset the previous stack frame
  8. Jump back to return address

```c
void test_funciton (int a, int b)
{
    int flag;
    char buffer[10];

    flag = 31337;
    buffer[0] = 'A';
}

int main()
{
    test_function(1, 2);
}
```



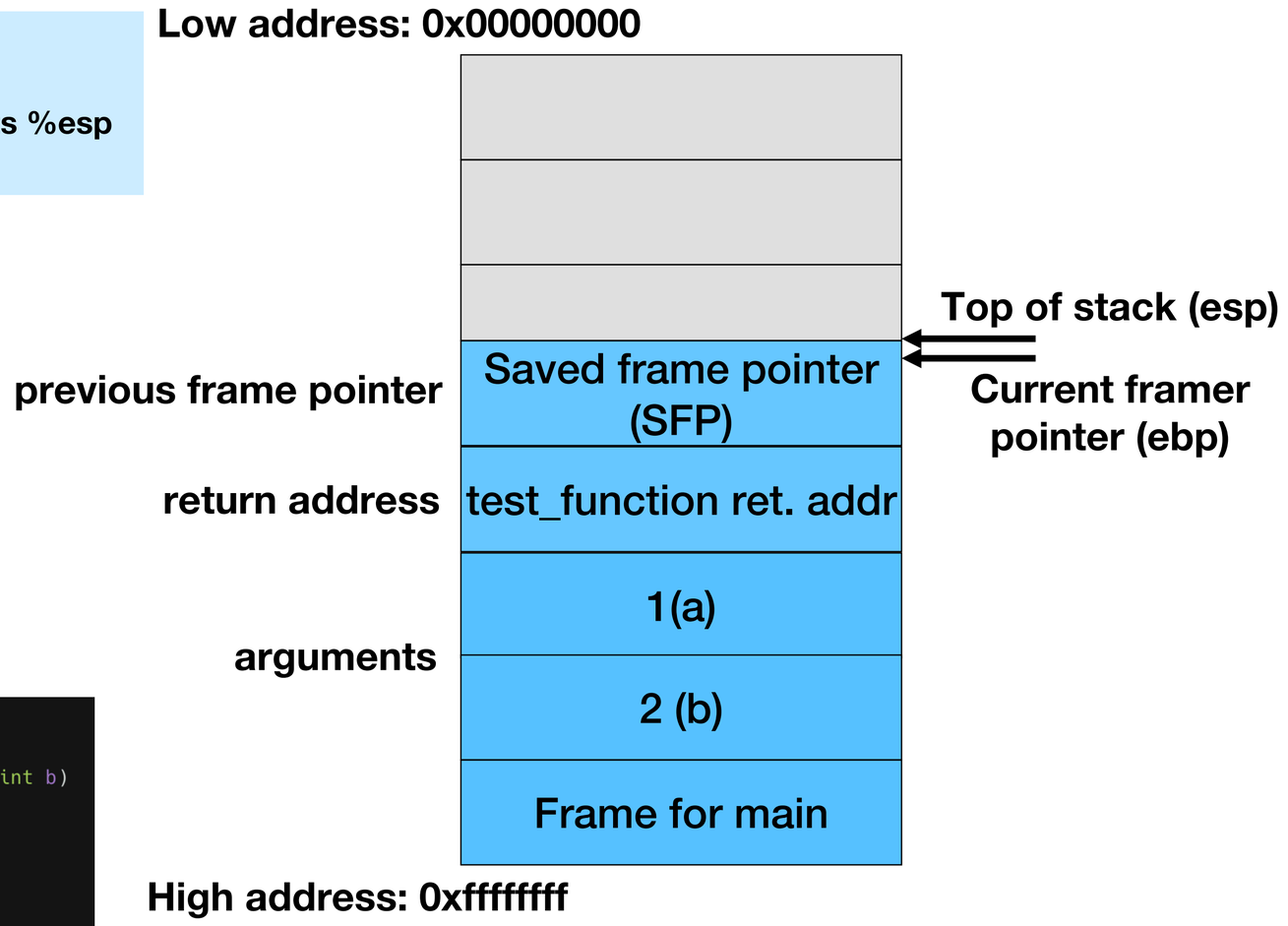
Stack and Function Calls

✓ **%esp always points to the frontier of the stack**
- **Push inserts its operand on top of stack, decrements %esp**
- **Pop removes a value from top of stack into its operand, increments %esp**

✓ **%ebp always points to the cell storing the old %ebp**

- Caller function:
  1. Push arguments onto the stack (in reverse)
  2. Push the return address, i.e., the address of the instruction you want run after control returns to you
  3. Jump to the function's address
- Called function:
  4. Push the old frame pointer onto the stack (%ebp)
  5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
  6. Push local variables onto the stack
- Returning:
  7. Reset the previous stack frame
  8. Jump back to return address

```c
void test_funciton (int a, int b)
{
    int flag;
    char buffer[10];

    flag = 31337;
    buffer[0] = 'A';
}

int main()
{
    test_function(1, 2);
}
```

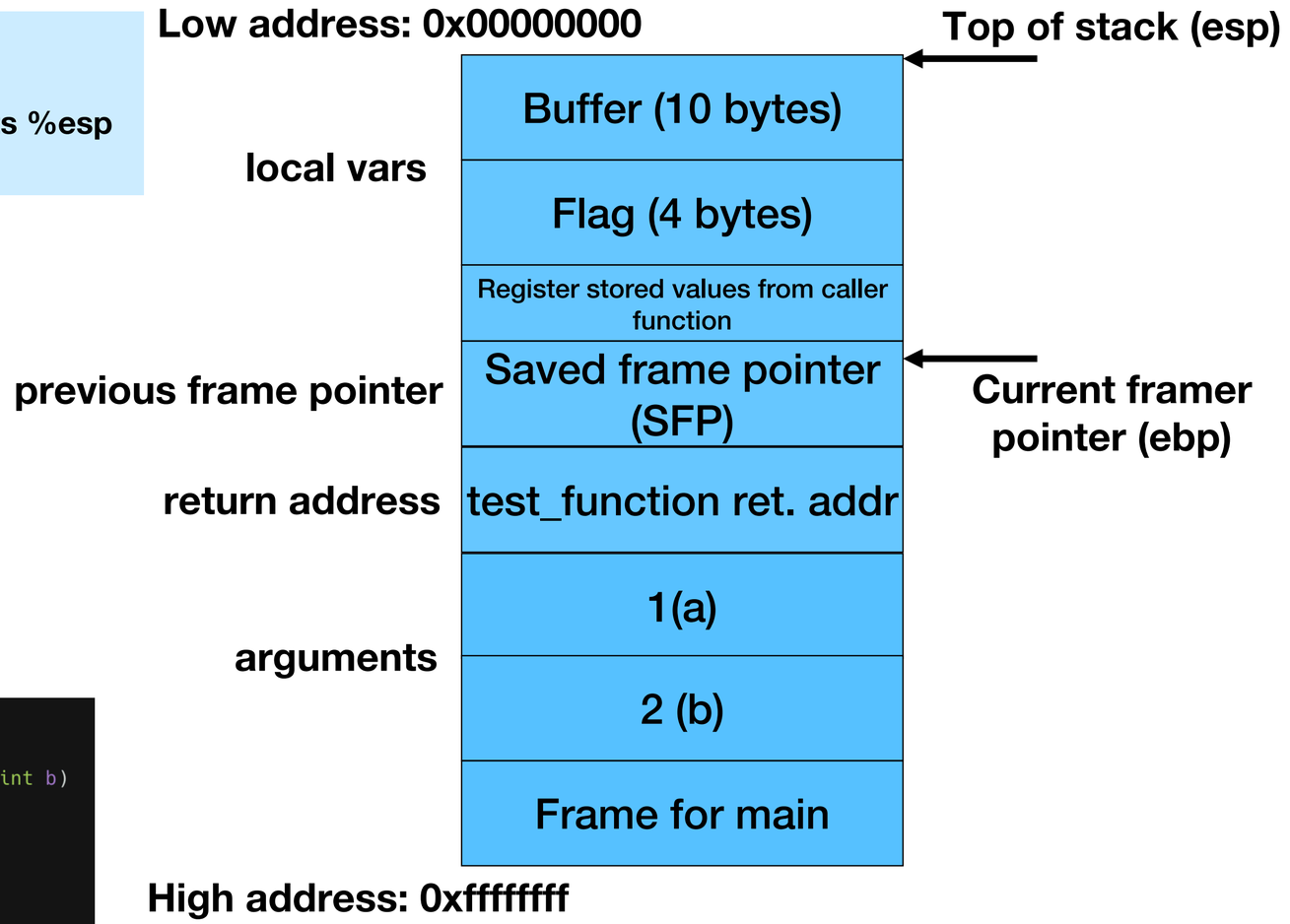


Stack and Function Calls

- ✓ **%esp always points to the frontier of the stack**
  - **Push inserts its operand on top of stack, decrements %esp**
  - **Pop removes a value from top of stack into its operand, increments %esp**
- ✓ **%ebp always points to the cell storing the old %ebp**

- Caller function:
  1. Push arguments onto the stack (in reverse)
  2. Push the return address, i.e., the address of the instruction you want run after control returns to you
  3. Jump to the function's address
- Called function:
  4. Push the old frame pointer onto the stack (%ebp)
  5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
  6. Push local variables onto the stack
- Returning:
  7. Reset the previous stack frame
  8. Jump back to return address

```c
void test_funciton (int a, int b)
{
    int flag;
    char buffer[10];

    flag = 31337;
    buffer[0] = 'A';
}

int main()
{
    test_function(1, 2);
}
```



MONASH University

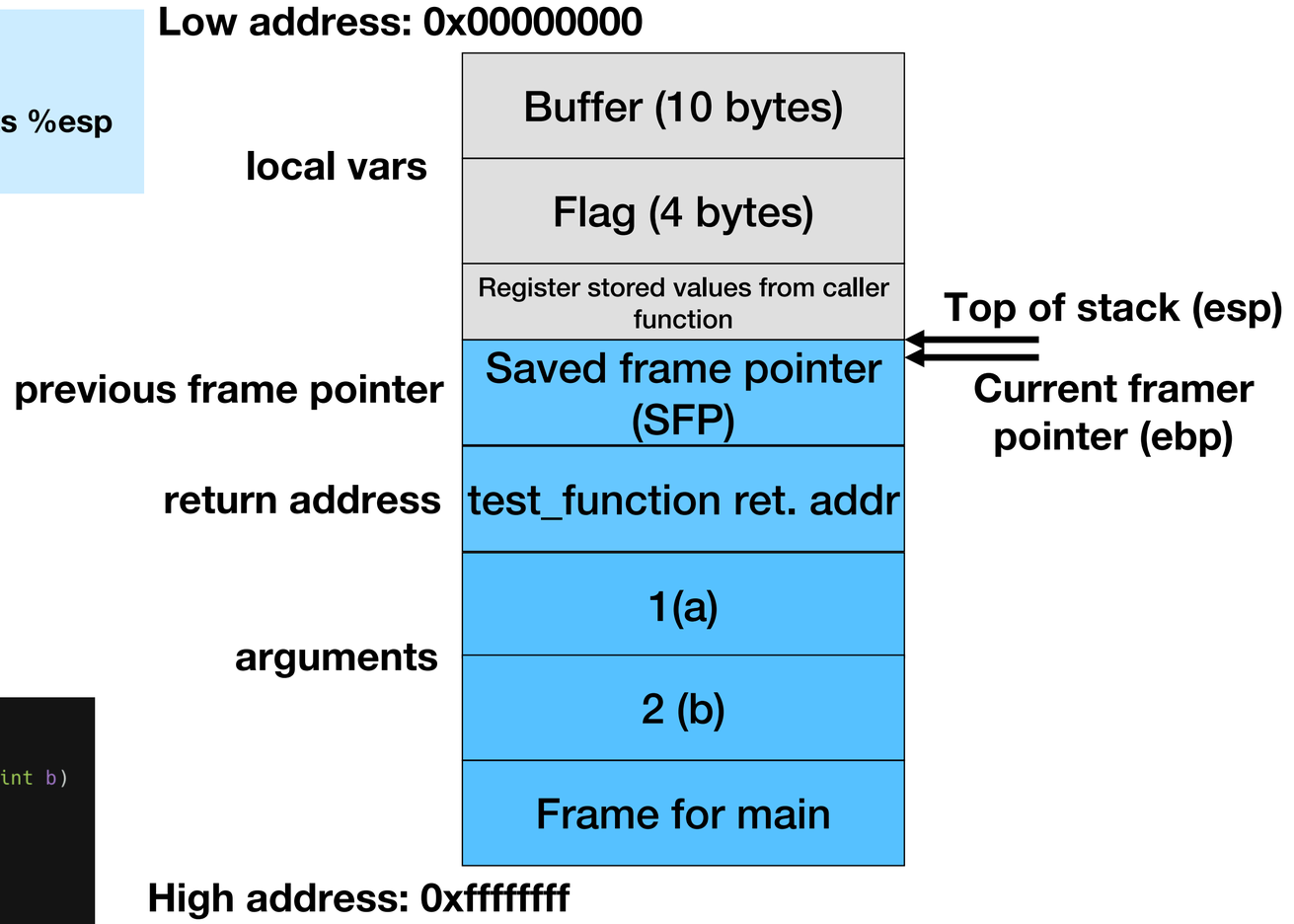
Stack and Function Calls

✓ **%esp always points to the frontier of the stack**
- **Push inserts its operand on top of stack, decrements %esp**
- **Pop removes a value from top of stack into its operand, increments %esp**

✓ **%ebp always points to the cell storing the old %ebp**

- Caller function:
  1. Push arguments onto the stack (in reverse)
  2. Push the return address, i.e., the address of the instruction you want run after control returns to you
  3. Jump to the function's address
- Called function:
  4. Push the old frame pointer onto the stack (%ebp)
  5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
  6. Push local variables onto the stack
- Returning:
  7. Reset the previous stack frame
  8. Jump back to return address

```c
void test_funciton (int a, int b)
{
    int flag;
    char buffer[10];

    flag = 31337;
    buffer[0] = 'A';
}

int main()
{
    test_function(1, 2);
}
```

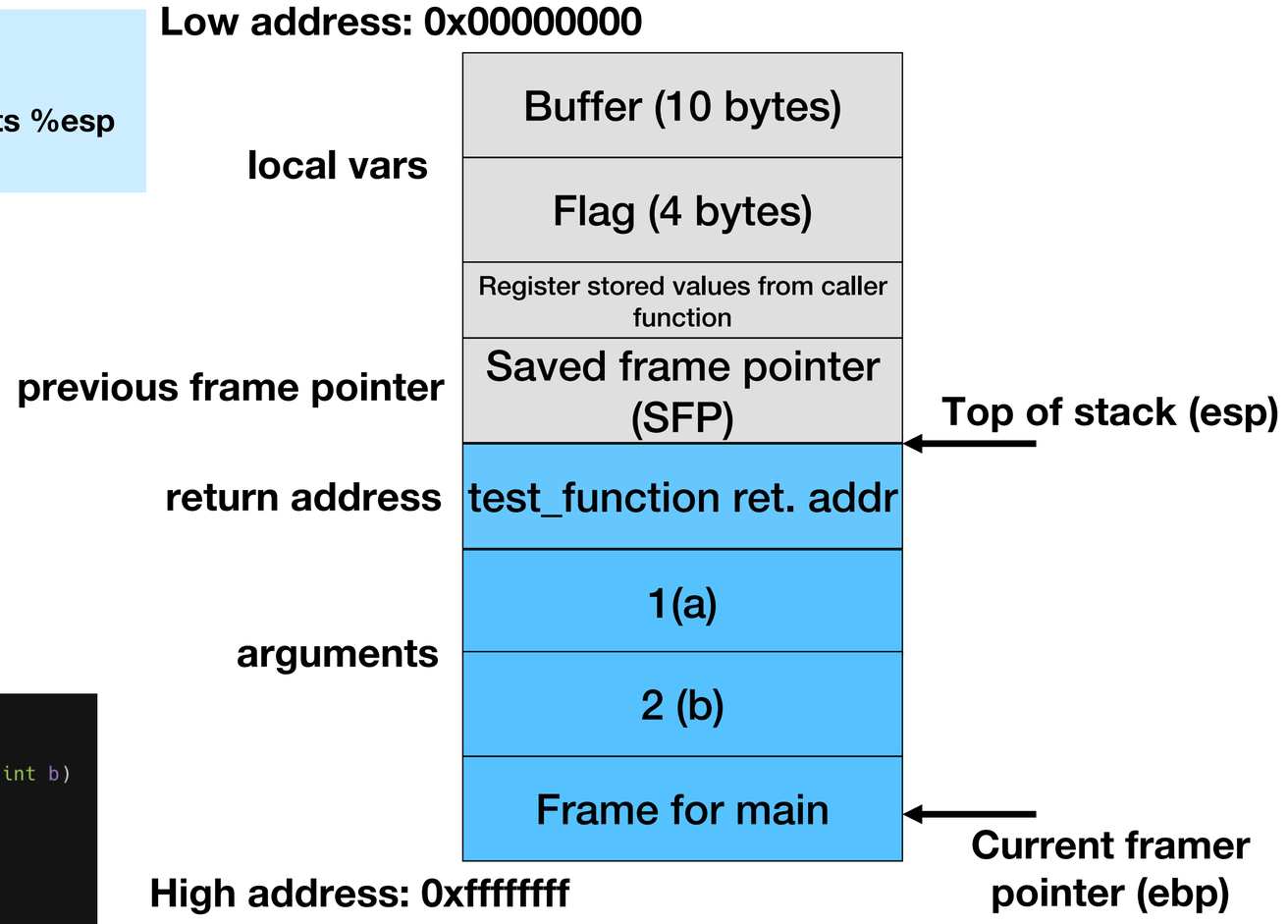


Stack and Function Calls

- ✓ **%esp always points to the frontier of the stack**
  - **Push inserts its operand on top of stack, decrements %esp**
  - **Pop removes a value from top of stack into its operand, increments %esp**
- ✓ **%ebp always points to the cell storing the old %ebp**

- Caller function:
  1. Push arguments onto the stack (in reverse)
  2. Push the return address, i.e., the address of the instruction you want run after control returns to you
  3. Jump to the function's address
- Called function:
  4. Push the old frame pointer onto the stack (%ebp)
  5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
  6. Push local variables onto the stack
- Returning:
  7. Reset the previous stack frame
  8. Jump back to return address

```c
void test_funciton (int a, int b)
{
    int flag;
    char buffer[10];

    flag = 31337;
    buffer[0] = 'A';
}

int main()
{
    test_function(1, 2);
}
```

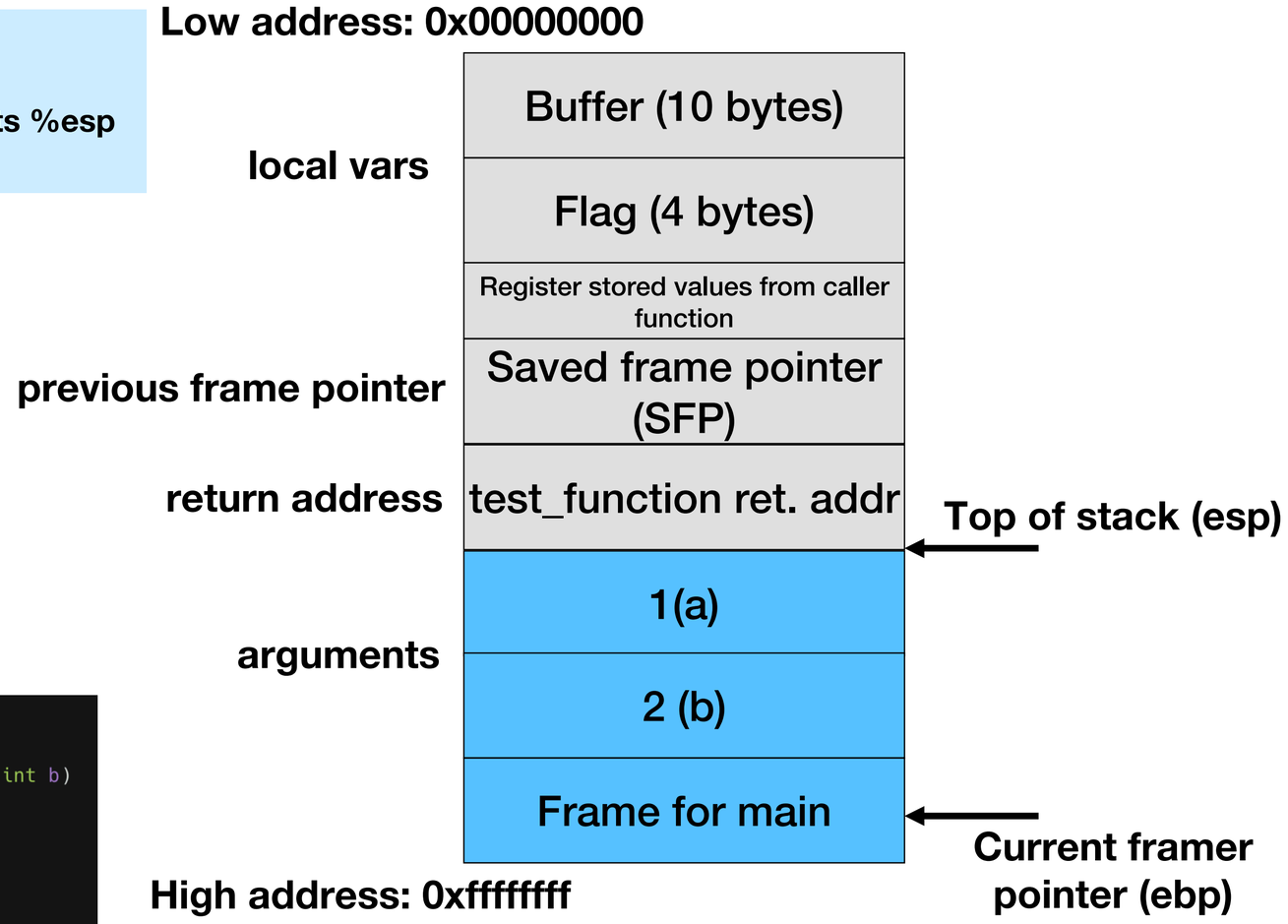


Stack and Function Calls

- ✓ **%esp always points to the frontier of the stack**
  - **Push inserts its operand on top of stack, decrements %esp**
  - **Pop removes a value from top of stack into its operand, increments %esp**
- ✓ **%ebp always points to the cell storing the old %ebp**

- Caller function:
  1. Push arguments onto the stack (in reverse)
  2. Push the return address, i.e., the address of the instruction you want run after control returns to you
  3. Jump to the function's address
- Called function:
  4. Push the old frame pointer onto the stack (%ebp)
  5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
  6. Push local variables onto the stack
- Returning:
  7. Reset the previous stack frame
  8. Jump back to return address

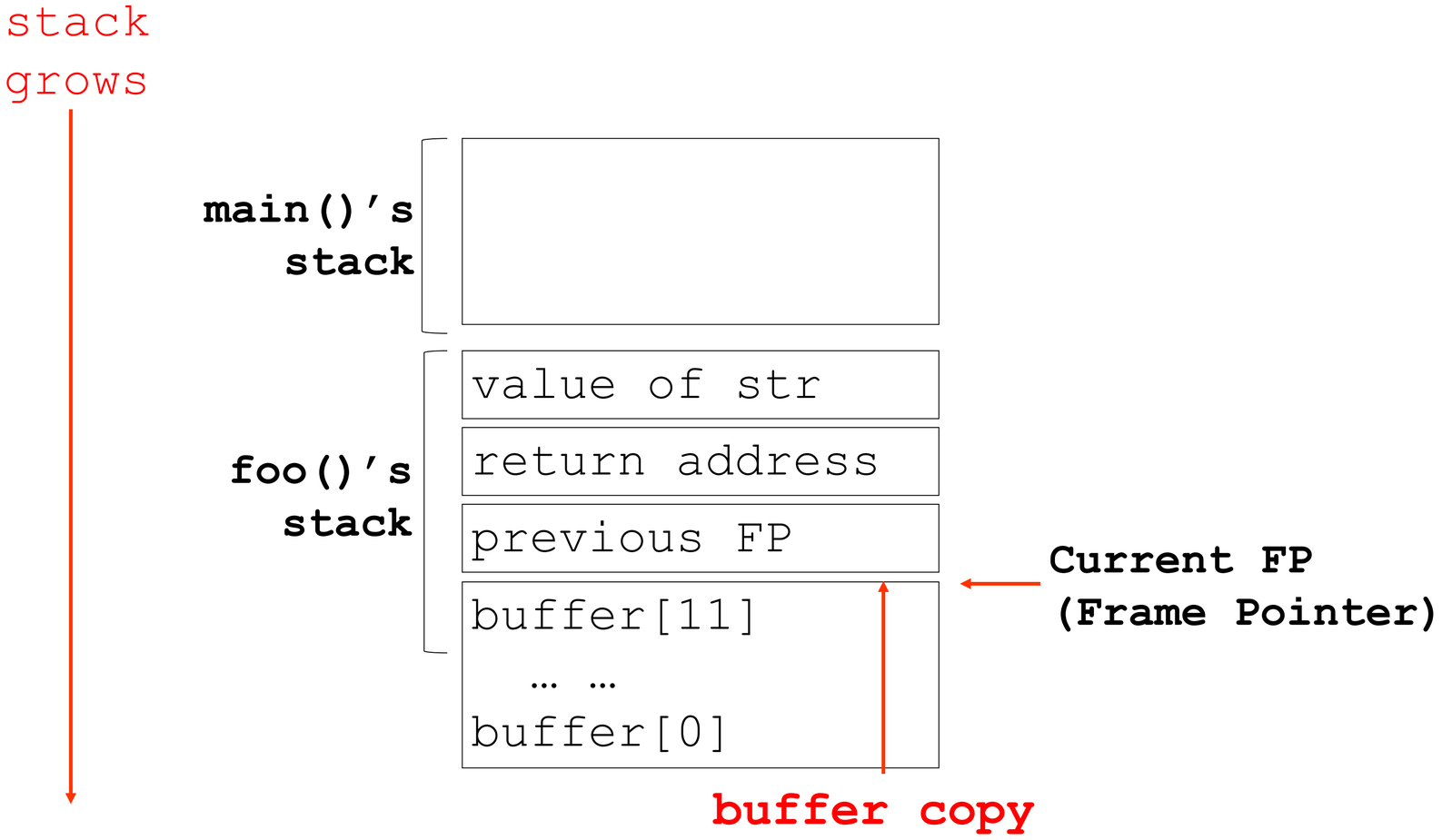
```c
void test_funciton (int a, int b)
{
    int flag;
    char buffer[10];

    flag = 31337;
    buffer[0] = 'A';
}

int main()
{
    test_function(1, 2);
}
```



When to exploit?

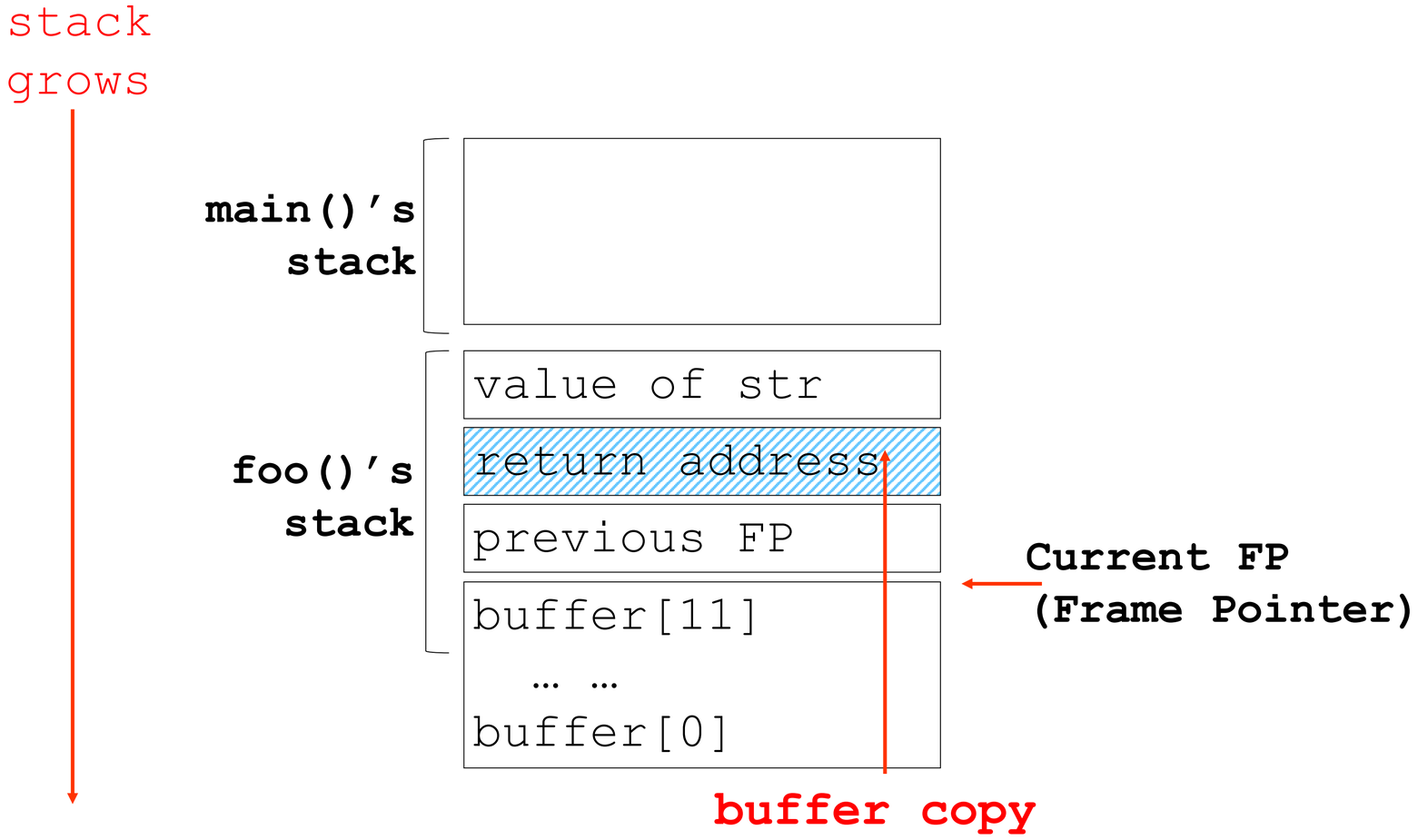
- What’s the best chance to hack the program execution?

How to exploit the vulnerability?



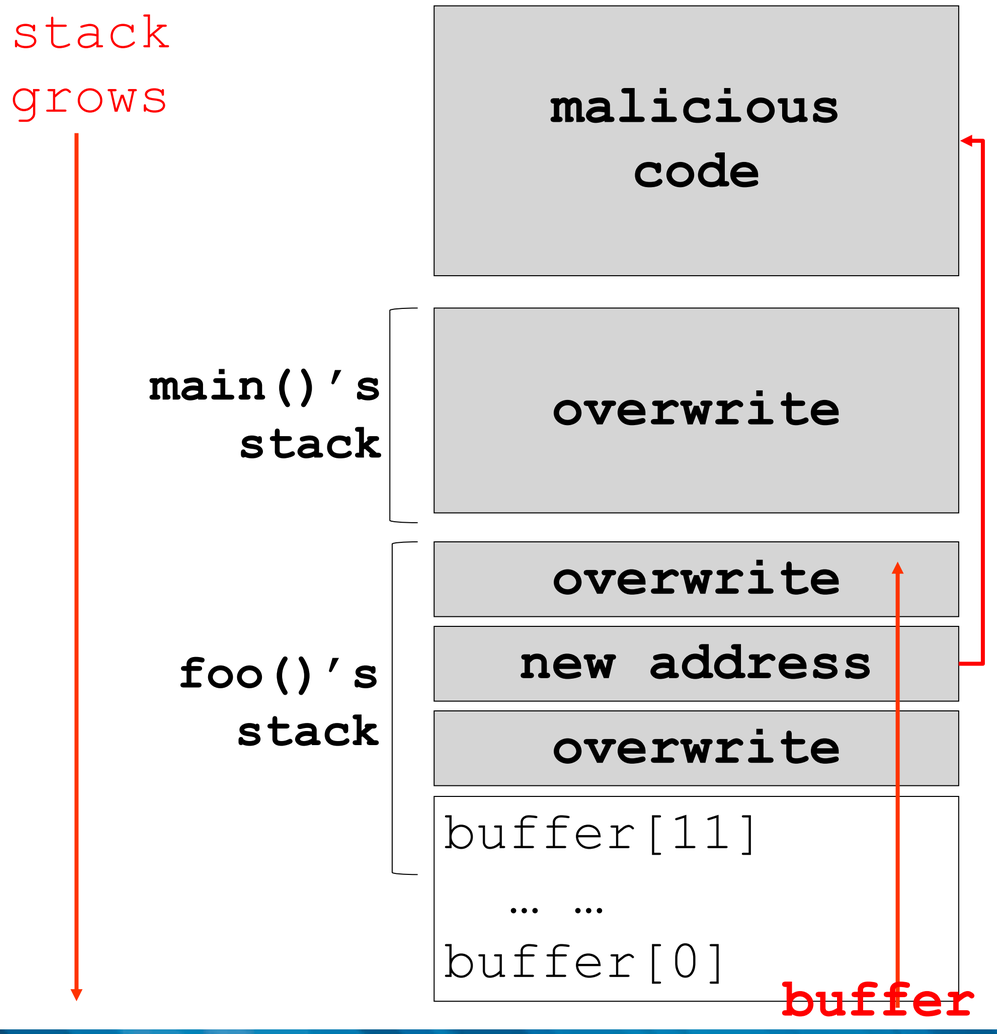

```
void foo(char *str) {
  char buffer[12];
  strcpy(buffer, str);
}

int main() {
  char *str = "test";
  foo(str);
}
```

How to exploit the vulnerability?




```
void foo(char *str) {
  char buffer[12];
  strcpy(buffer, str);
}

int main() {
  char *str = "This is
  definitely longer than 12";
  foo(str);
}
```

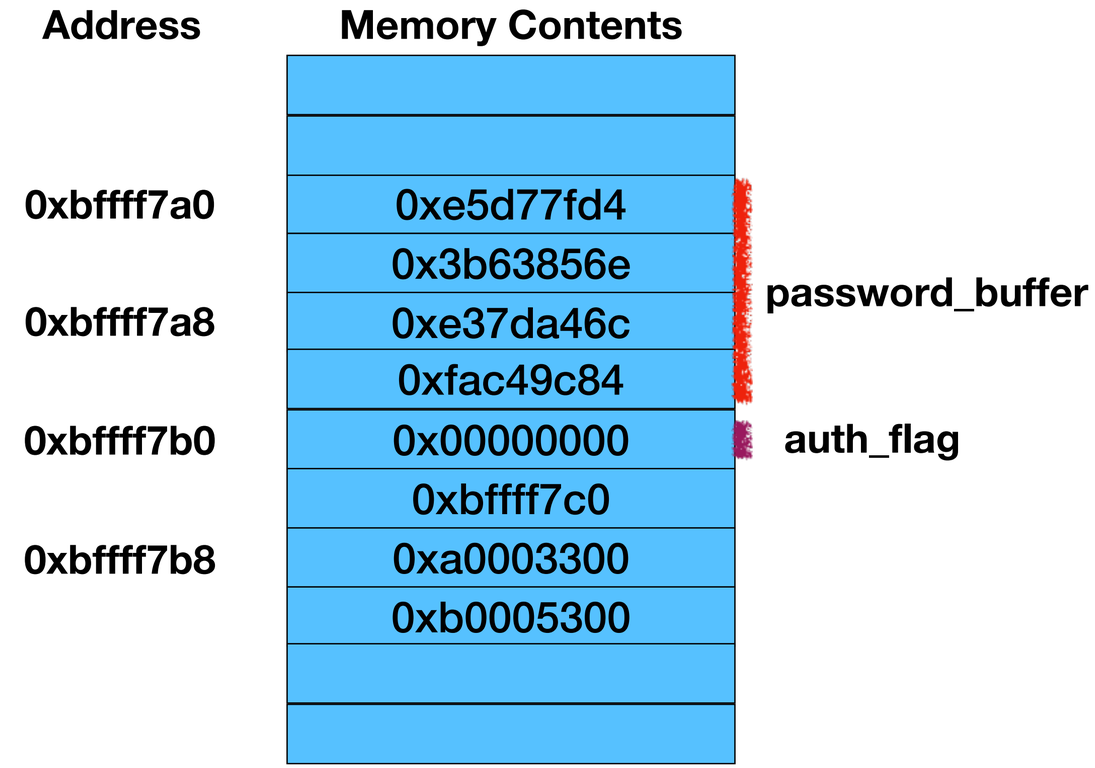
How to exploit the vulnerability?




```
void foo(char *str) {
  char buffer[12];
  strcpy(buffer, str);
}

int main() {
  char *str = "some text to
overflow buf [new address]
some other text [malicious
code]";
  foo(str);
}
```

Buffer Overflow: Example 1

```c
int check_authentication(char *password)
{
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "brillig") == 0)
    auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if(check_authentication(argv[1]))
    {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
     }
    else
    {
        printf("\nAccess Denied.\n");
    }
}
```



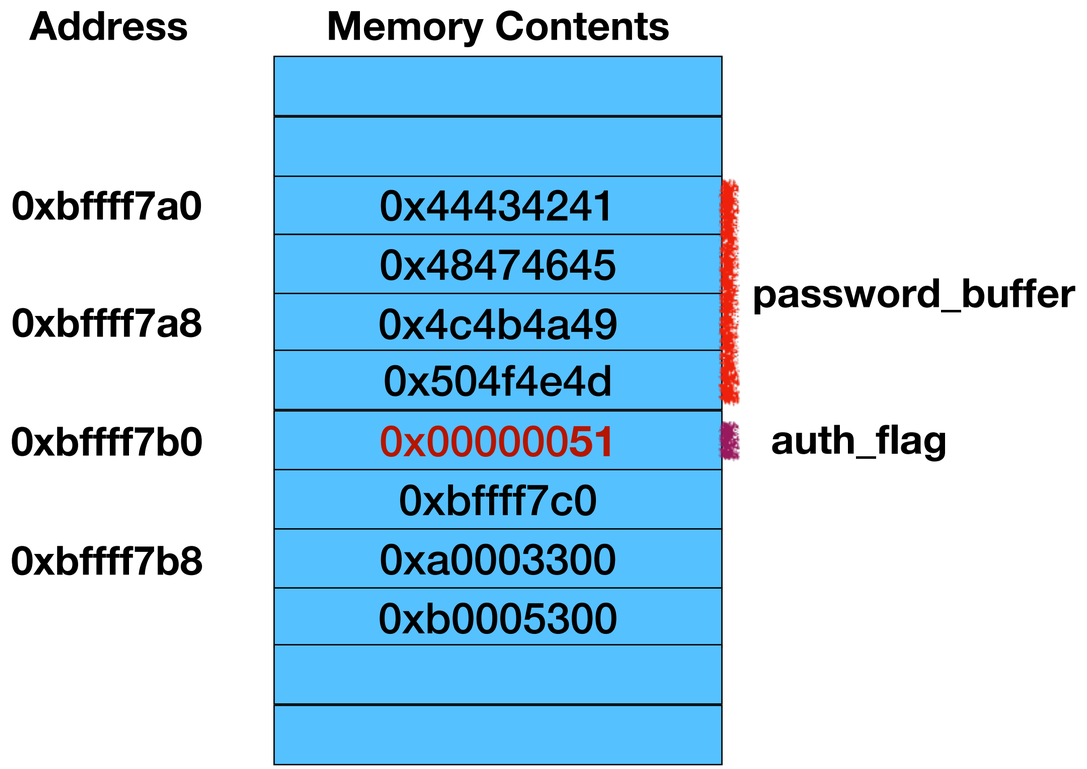
How to bypass the authentication check without knowing the password?

https://flux.qa/EH2NNF

MONASH University

Buffer Overflow: Example 1

```c
int check_authentication(char *password)
{
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "brillig") == 0)
    auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if(check_authentication(argv[1]))
    {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
     }
    else
    {
        printf("\nAccess Denied.\n");
    }
}
```



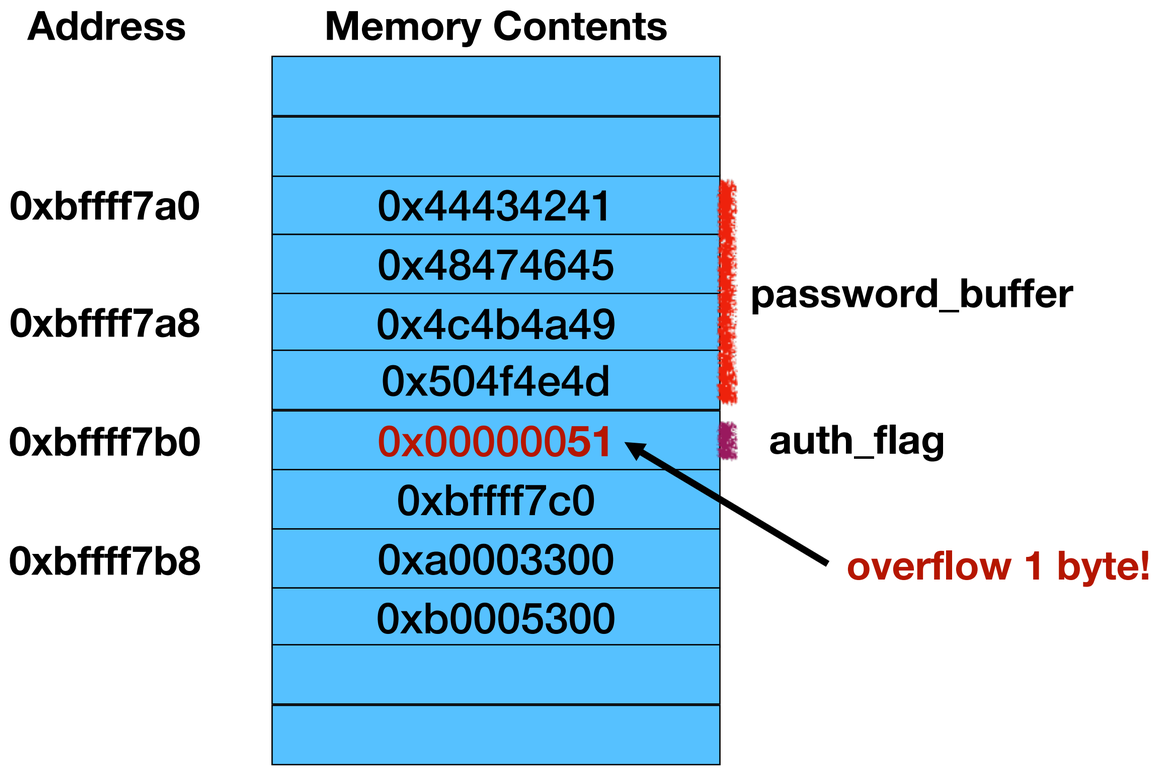
Memory contents after strcpy of password

“ABCDEFGHIJKLMNOPQ” from user (attacker) input

MONASH University

Buffer Overflow: Example 1

```c
int check_authentication(char *password)
{
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "brillig") == 0)
    auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if(check_authentication(argv[1]))
    {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
     }
    else
    {
        printf("\nAccess Denied.\n");
    }
}
```



Buffer allocated for passwords up to 16 chars

(bytes): longer passwords overflow into auth_flag

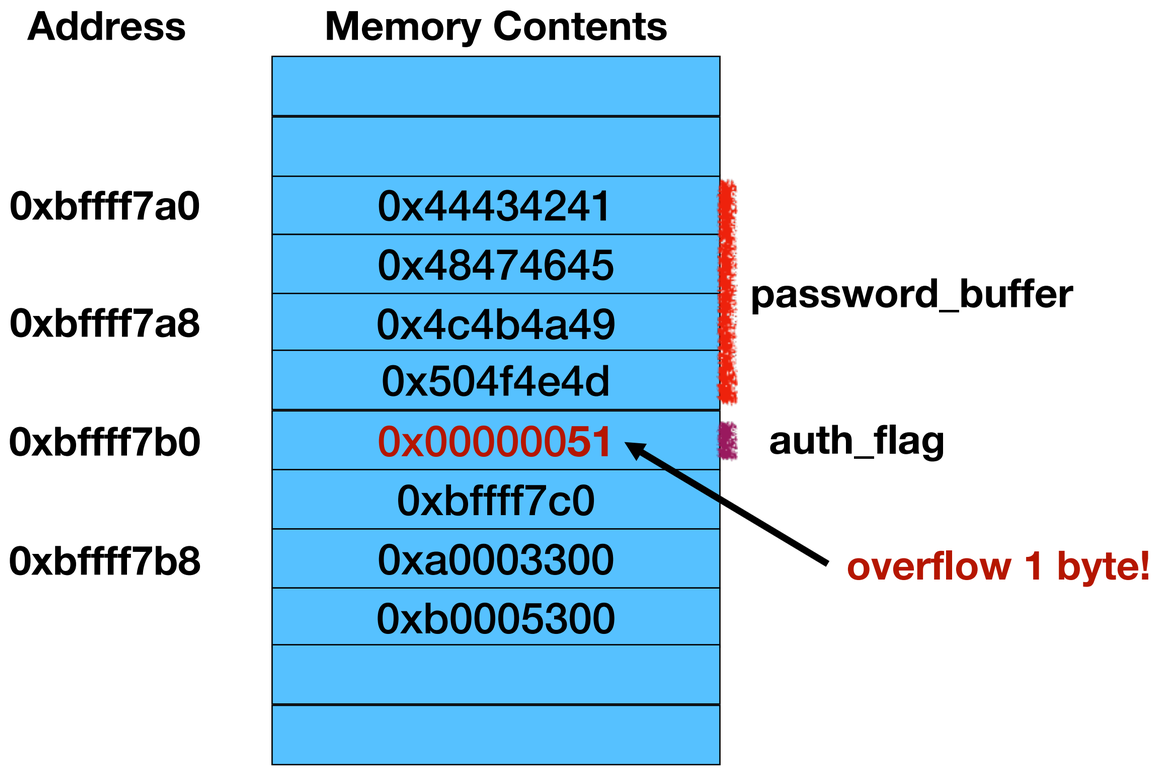
MONASH University

Corrupting data

- Attackers can overflow data:
  - *Modify state variables* to bypass authorization check (our example)
  - *Modify a secret key* to be one known to the attacker, to be able to decrypt future intercepted messages
  - *Modify interpreted strings* used as part of commands, e.g., to facilitate SQL injection

Buffer Overflow: Example 1

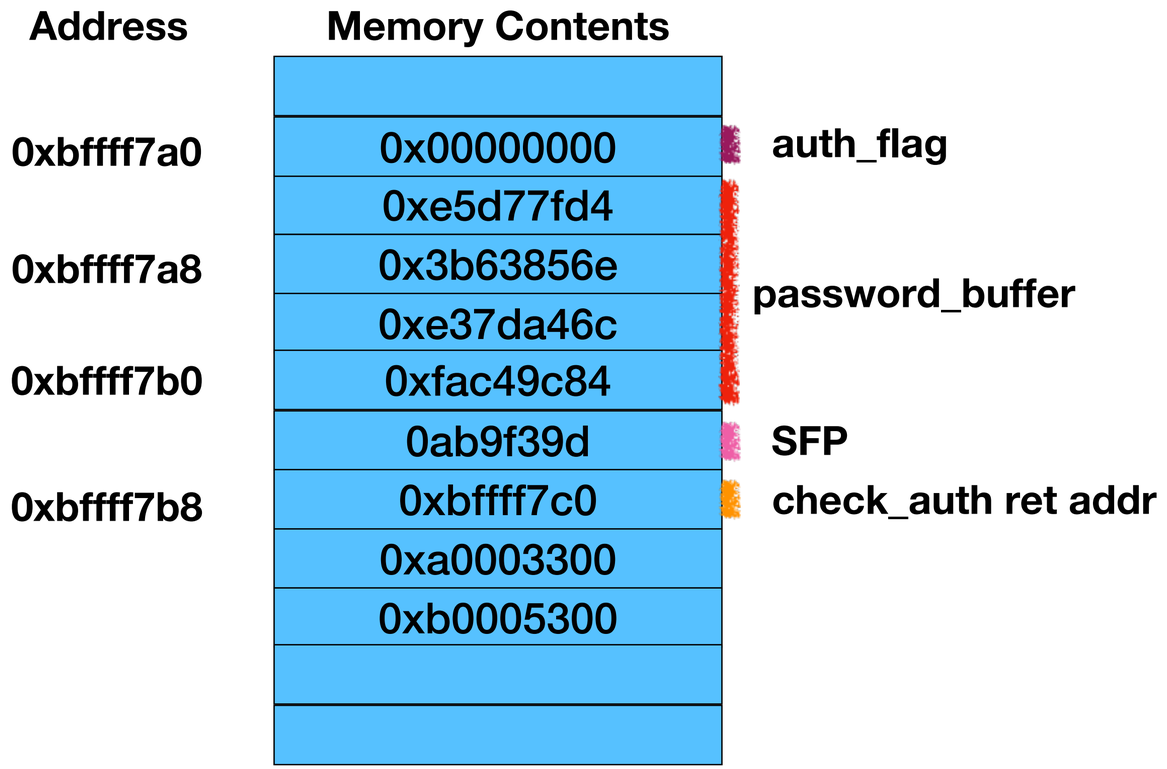
```c
int check_authentication(char *password)
{
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "brillig") == 0)
    auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if(check_authentication(argv[1]))
    {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
     }
    else
    {
        printf("\nAccess Denied.\n");
    }
}
```



How to prevent such exploit?

Buffer Overflow: Example 2

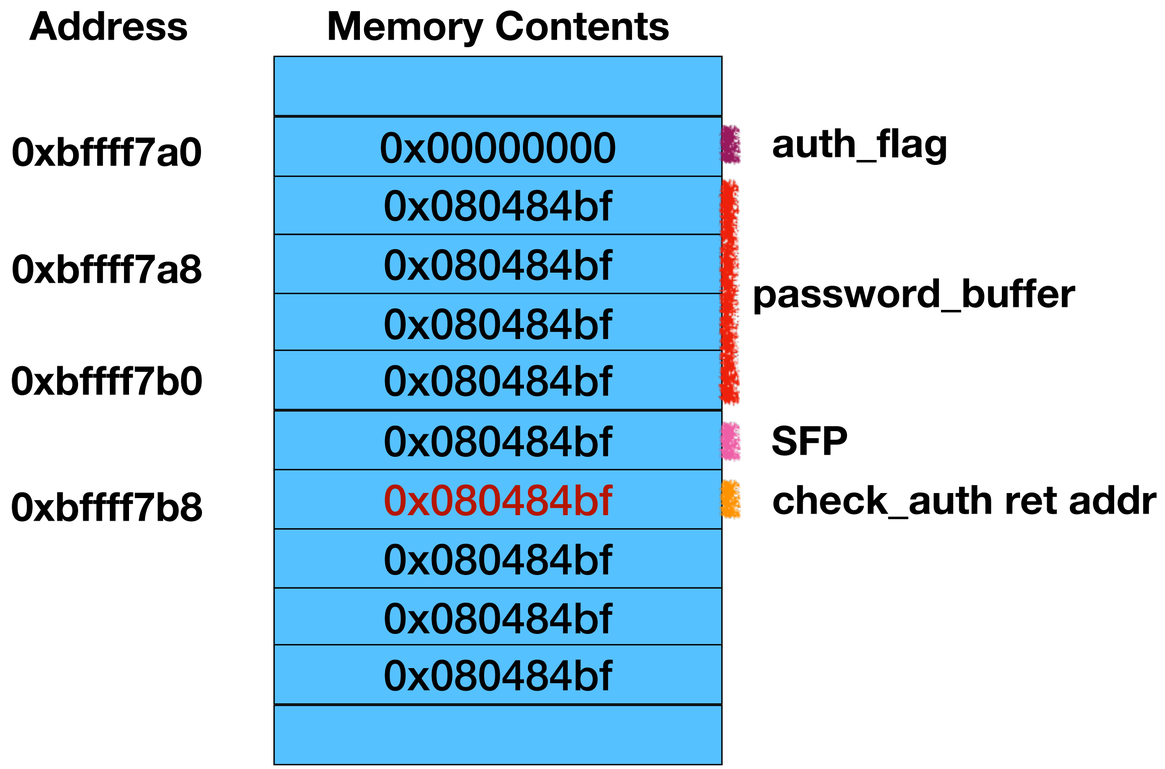
```c
int check_authentication(char *password)
{
    char password_buffer[16];
    int auth_flag = 0;

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "brillig") == 0)
    auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if(check_authentication(argv[1]))
    {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("       Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
     }
    else
    {
        printf("\nAccess Denied.\n");
    }
}
```



password_buffer is stored after auth_flag: buffer overflow cannot overwrite auth_flag

Buffer Overflow: Example 2

```c
int check_authentication(char *password)
{
    char password_buffer[16];
    int auth_flag = 0;

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "brillig") == 0)
    auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if(check_authentication(argv[1]))
    {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("       Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
     }
    else
    {
        printf("\nAccess Denied.\n");
    }
}
```



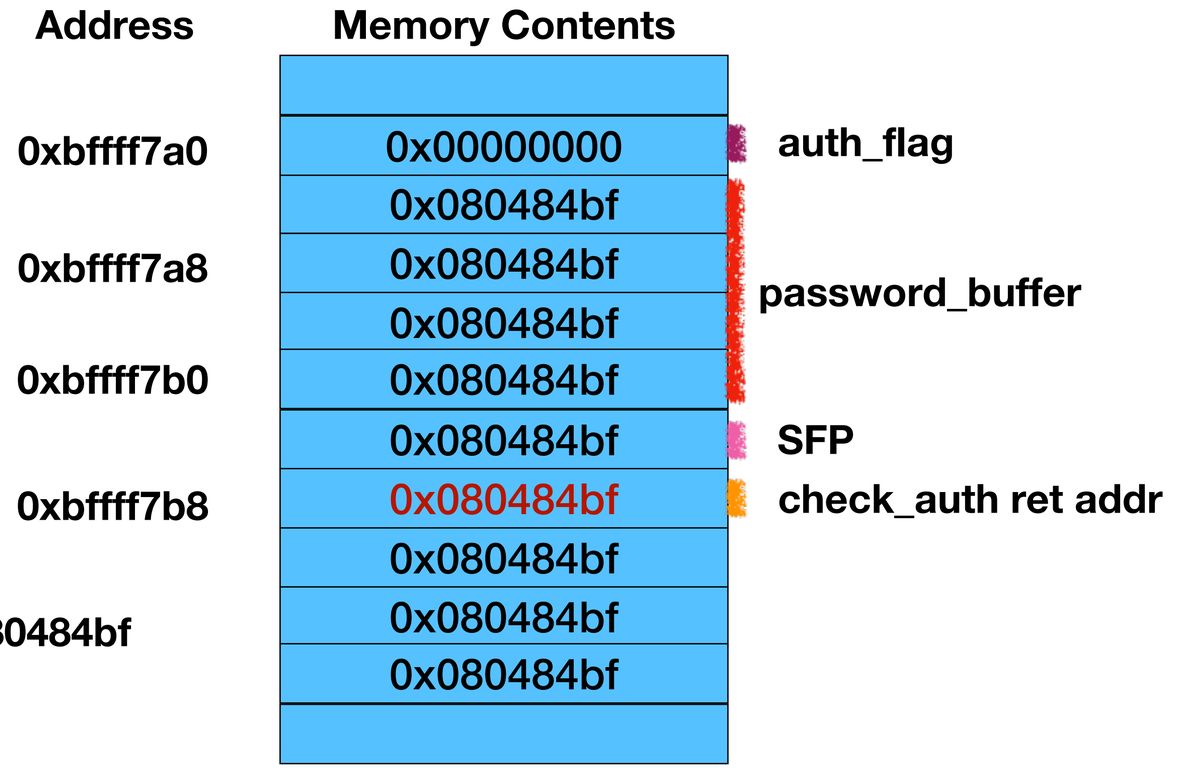
Memory Contents after strcpy of password

“\xbf\x84\x04\x08” x 10 (attacker input)

Buffer Overflow: Example 2

```c
int check_authentication(char *password)
{
    char password_buffer[16];
    int auth_flag = 0;

    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "brillig") == 0)
    auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if(check_authentication(argv[1]))
    {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
     }
    else
    {
        printf("\nAccess Denied.\n");
    }
}
```

jump to 0x080484bf

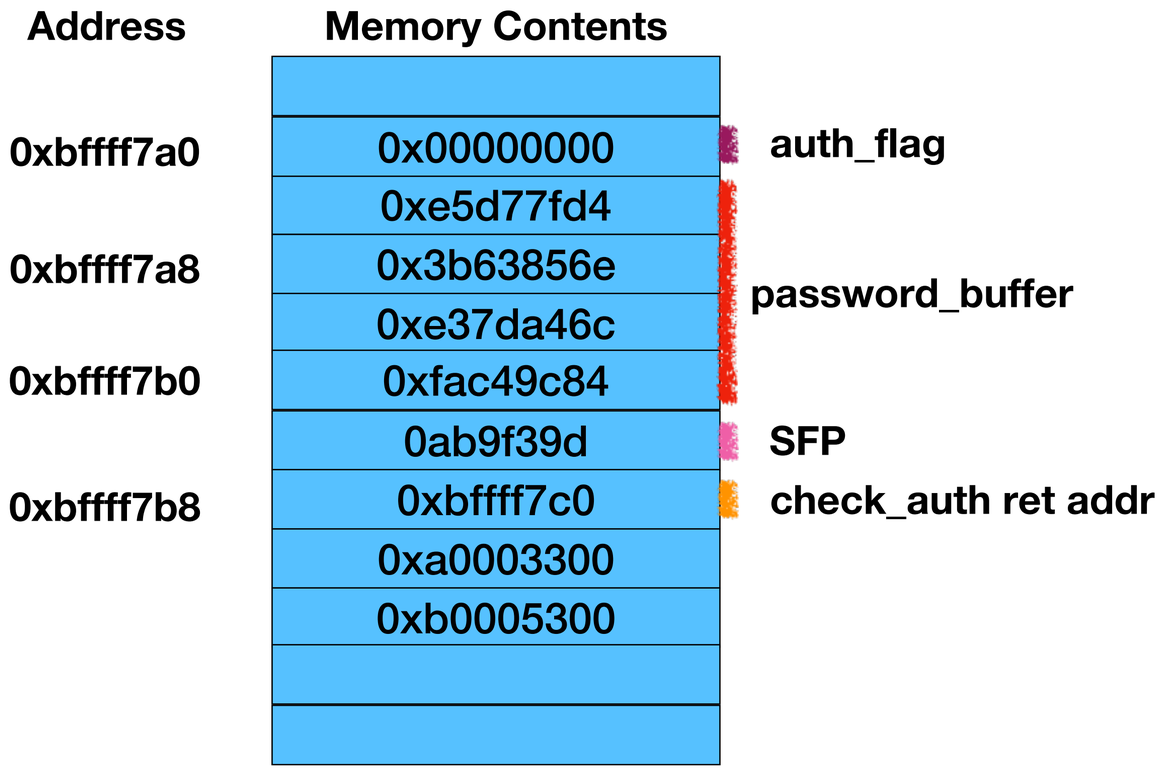


Attacker uses repeated return addresses to modify check_auth ret addr even she does not know the exact location of it.

Buffer Overflows: Example 3

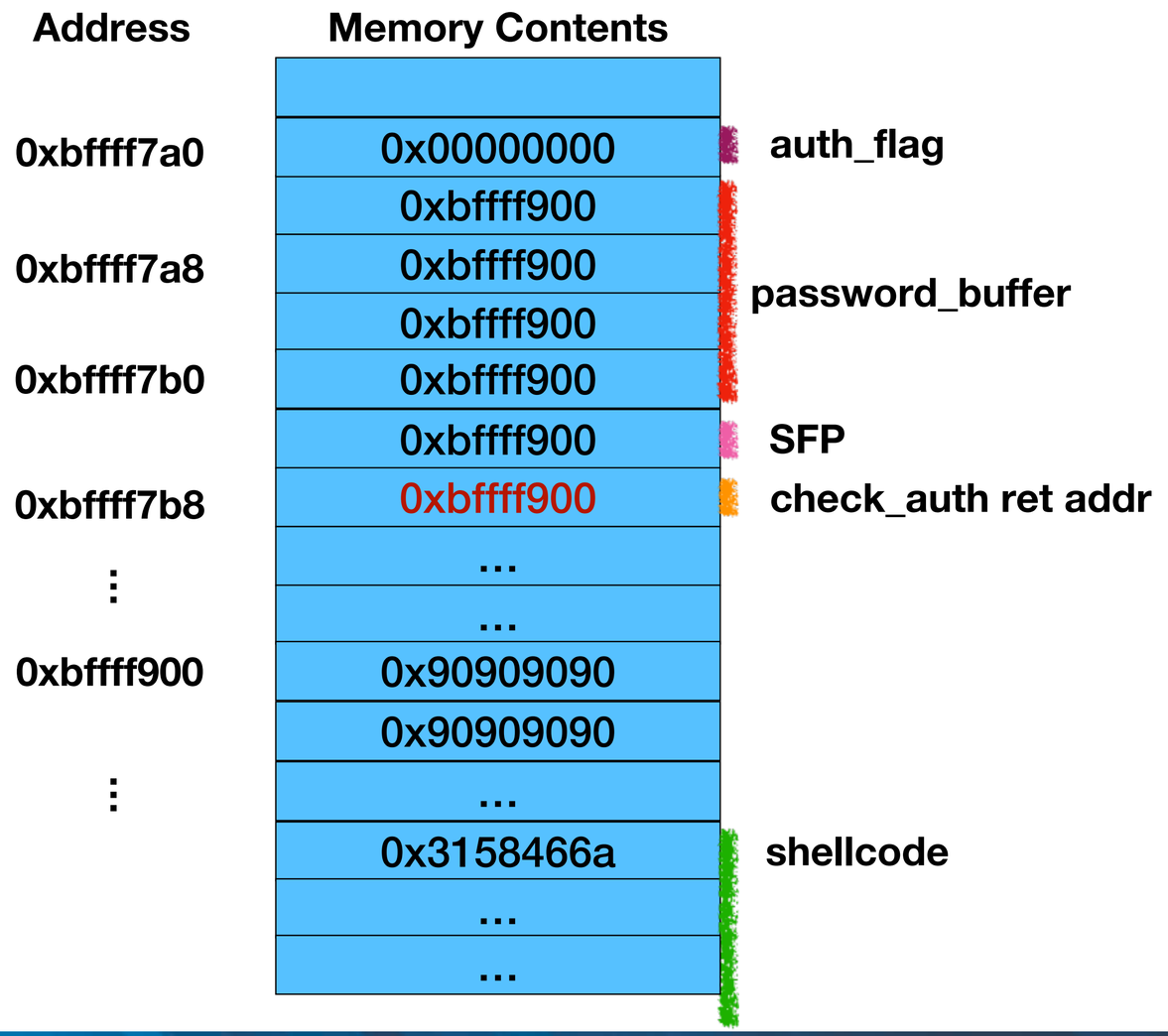
Q: what if the attacker wants to jump to her own code (e.g., shellcode with root user access)?

- How can the attacker know the address of the injected code?



Buffer Overflows: Example 3

- Attacker injects the shellcode (machine code) in the overflowing input string, followed by repeated ret. address
- Leverage “NOP” : insert a sequence of consecutive NOP (do nothing) instructions (opcode = 0x90) before the actual shellcode.
- The program will return to the address of 0xbffff900, execute a number of NOP instructions, and eventually execute the shellcode.



Tips to Prevent Buffer Overflow

- Ban unsafe function calls (e.g. read, gets)
- Use safer functions, such as fgets, strncat … from <strsafe.h>, <libsafe>, … libraries.
  - Take length as explicit parameter, do not rely on null termination.
- Always do bounds checking (some modern compliers do this for you)
- Use through testing and code reviews
  - Use source code scanning tools, e.g., RATS, Flawfinder, ITS4, PurifyPlus Software Suite…

Secure Coding Examples

```
void main() {
   char buf[32];

    if (fgets(buf, sizeof(buf), stdin) == NULL)
    return 1;
}
```

read at most 31 chars into buf and write a null char at end.

```
void DoSomething(char *BuffSrc, int BuffLen)
{
    char BuffDest[16];
    memcpy_s(BuffDest,16,BuffSrc,BuffLen);
}
```

memory copy 16 bytes

```
void function1(const char* input)
{
   char buffer[10];
   strncpy(buffer,input,sizeof(buffer));
}
```

read at most 10 bytes from input string into buffer

System-Level Mitigations

- **OS approaches:**

- Address randomisation: randomize the start location of a stack, and other types of memory, such as heap, libraries, etc. [widely used technique: Address space layout randomization]

    - All areas of process are located at random places which may cause compatibility issues;

    - Sometimes a reduced range of the addresses are available for randomization;

Principle of ASLR

To randomize the start location of the stack that is every time the code is loaded in the memory, the stack address changes.

Difficult to guess the stack address in the memory.

Difficult to guess %ebp address and address of the malicious code

Example: Address Space Layout Randomization

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
   char x[12];
   char *y = malloc(sizeof(char)*12);

   printf("Address of buffer x (on stack): 0x%x\n", x);
   printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

Address Space Layout Randomization : Working

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

1

```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

2

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

3

System-Level Mitigations

- **OS approaches:**

- Address randomisation: randomize the start location of a stack, and other types of memory, such as heap, libraries, etc. [widely used technique: Address space layout randomization]

    - All areas of process are located at random places which may cause compatibility issues;

    - Sometimes a reduced range of the addresses are available for randomization;

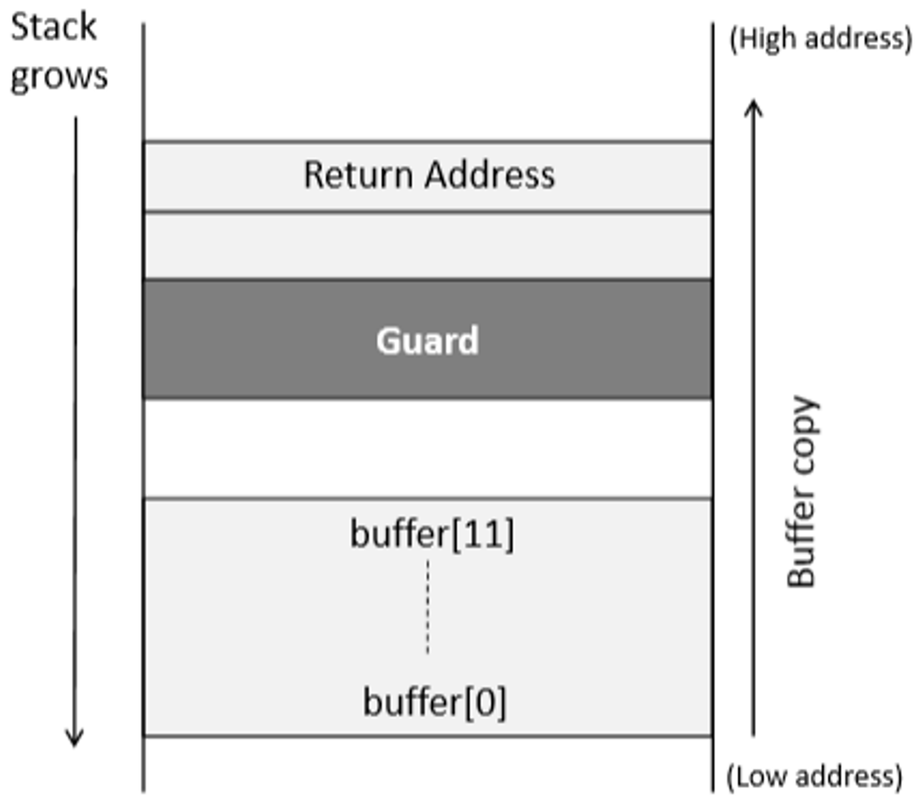
Compiler approaches:

- Stackshield: copy return address at some other place to check whether it is modified.

- StackGuard: place a guard between the return address and the buffer to detect modification of return address.

Stack guard

```
void foo (char *str)
{
    int guard;
    guard = secret;

    char buffer[12];
    strcpy (buffer, str);

    if (guard == secret)
        return;
    else
        exit(1);
}
```



Execution with StackGuard

```
seed@ubuntu:~$ gcc -o prog prog.c
seed@ubuntu:~$ ./prog hello
Returned Properly

seed@ubuntu:~$ ./prog hello00000000000
*** stack smashing detected ***:   ./prog terminated
```

Canary check done by compiler.

```
foo:
.LFB0:
    .cfi_startproc
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl     %esp, %ebp
    .cfi_def_cfa_register 5
    subl     $56, %esp
    movl     8(%ebp), %eax
    movl     %eax, -28(%ebp)
    // Canary Set Start
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    // Canary Set End
    movl     -28(%ebp), %eax
    movl     %eax, 4(%esp)
    leal     -24(%ebp), %eax
    movl     %eax, (%esp)
    call     strcpy
    // Canary Check Start
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L2
    call __stack_chk_fail
    // Canary Check End
```

System-Level Mitigations

- **OS approaches:**

- Address randomisation: randomize the start location of a stack, and other types of memory, such as heap, libraries, etc. [widely used technique: Address space layout randomization]

    - All areas of process are located at random places which may cause compatibility issues;

    - Sometimes a reduced range of the addresses are available for randomization;

Compiler approaches:

- Stackshield: copy return address at some other place to check whether it is modified.

- StackGuard: place a guard between the return address and the buffer to detect modification of return address.

Hardware approaches:

- Hardware protection: modern CPU supports NX (No-eXecute) bit. OS can mark memory region as non-executable.

Non-executable stack

- NX bit, standing for No-eXecute feature in CPU separates code from data which marks certain areas of the memory as non-executable.

- This countermeasure can be defeated using a different technique called **Return-to-libc** attack

Return Oriented Programming Attacks

- Main Idea: Chain gadgets to execute malicious code
- A gadget is a suite of instructions which ends with the return instruction
- Objective: Use gadgets instead of classical shellcode
- Why?
  - Gadgets are mainly located in segments without ASLR and on pages marked as executables
  - It can bypass the ASLR
  - It can bypass the NX bit

Function Prologue and Epilogue example

```
void foo(int x) {
    int a;
    a = x;
}
```

1 **Function prologue**

2 **Function epilogue**

```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
    1   pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
    2   movl  %ebp, %esp
        popl  %ebp
        ret
```

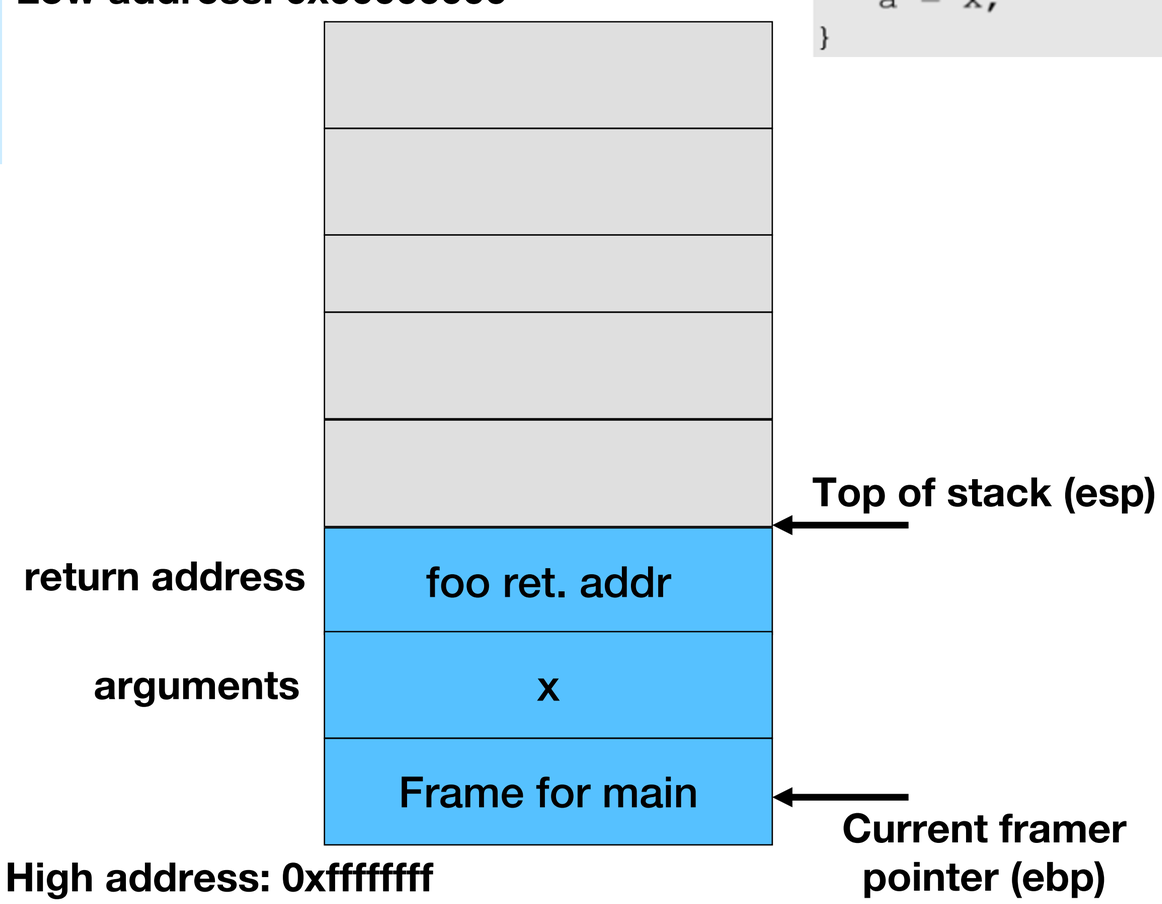
- From here, mov <src>, <dst> copies data at address <src> to address <dst> (using the default AT&T syntax, instead of Intel syntax)

Function Prologue and Epilogue example

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
        movl     8(%ebp), %eax
        movl     %eax, -4(%ebp)
        movl  %ebp, %esp
        popl  %ebp
        ret
```

```
void foo(int x) {
    int a;
    a = x;
}
```

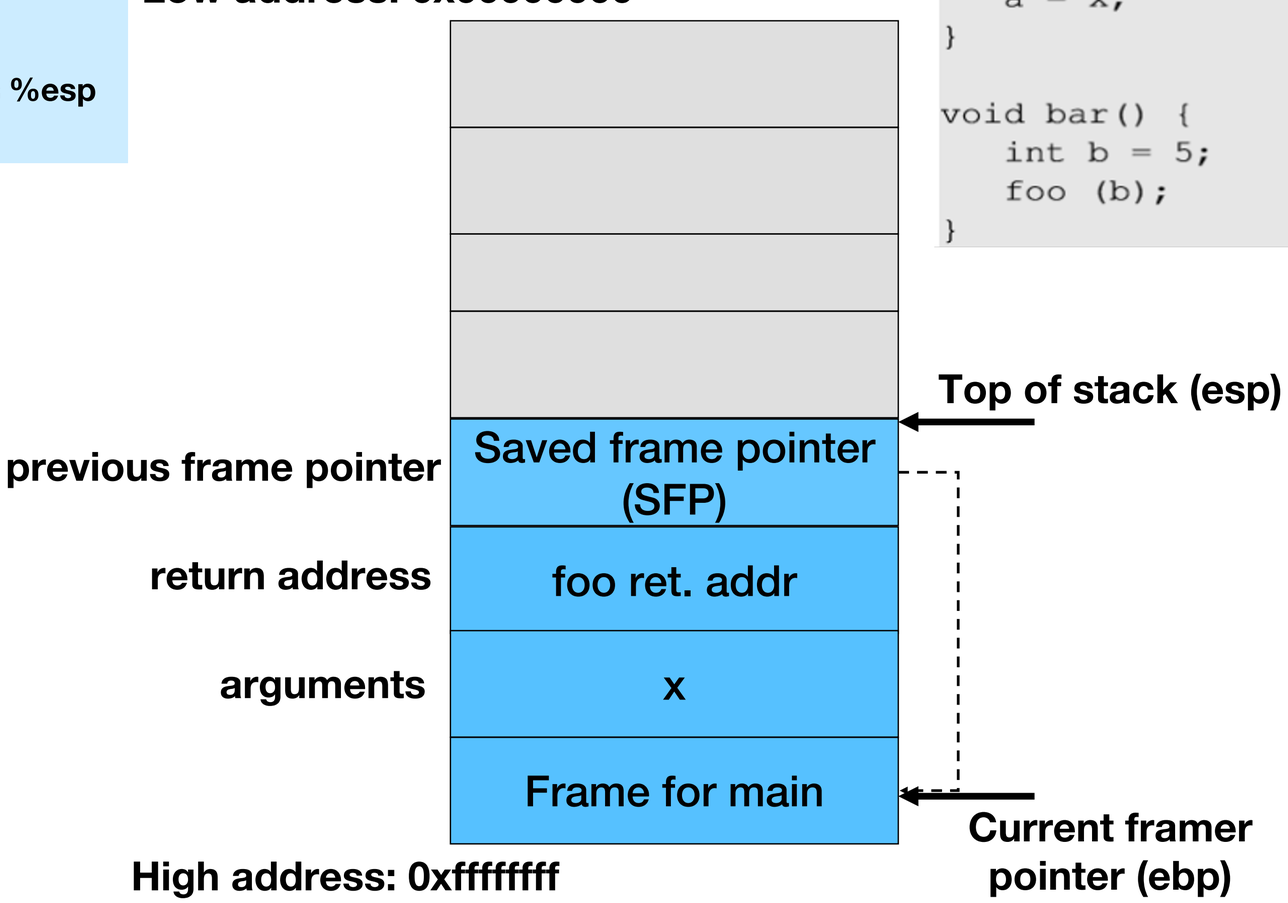


Function Prologue and Epilogue example

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
        movl     8(%ebp), %eax
        movl     %eax, -4(%ebp)
        movl  %ebp, %esp
        popl  %ebp
        ret
```

```
void foo(int x) {
    int a;
    a = x;
}

void bar() {
    int b = 5;
    foo (b);
}
```

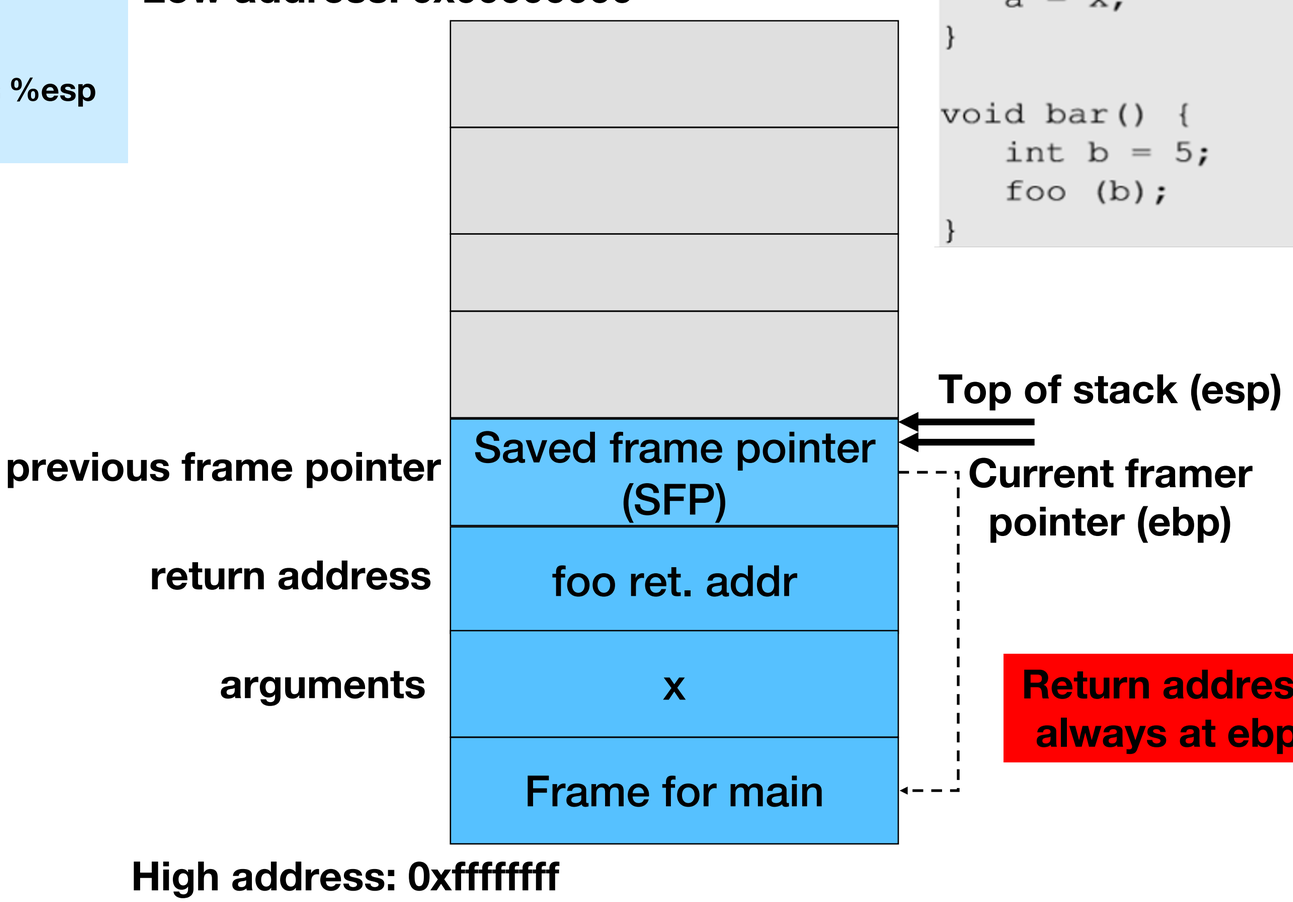


Function Prologue and Epilogue example

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
        movl     8(%ebp), %eax
        movl     %eax, -4(%ebp)
        movl  %ebp, %esp
        popl  %ebp
        ret
```

```
void foo(int x) {
    int a;
    a = x;
}

void bar() {
    int b = 5;
    foo (b);
}
```

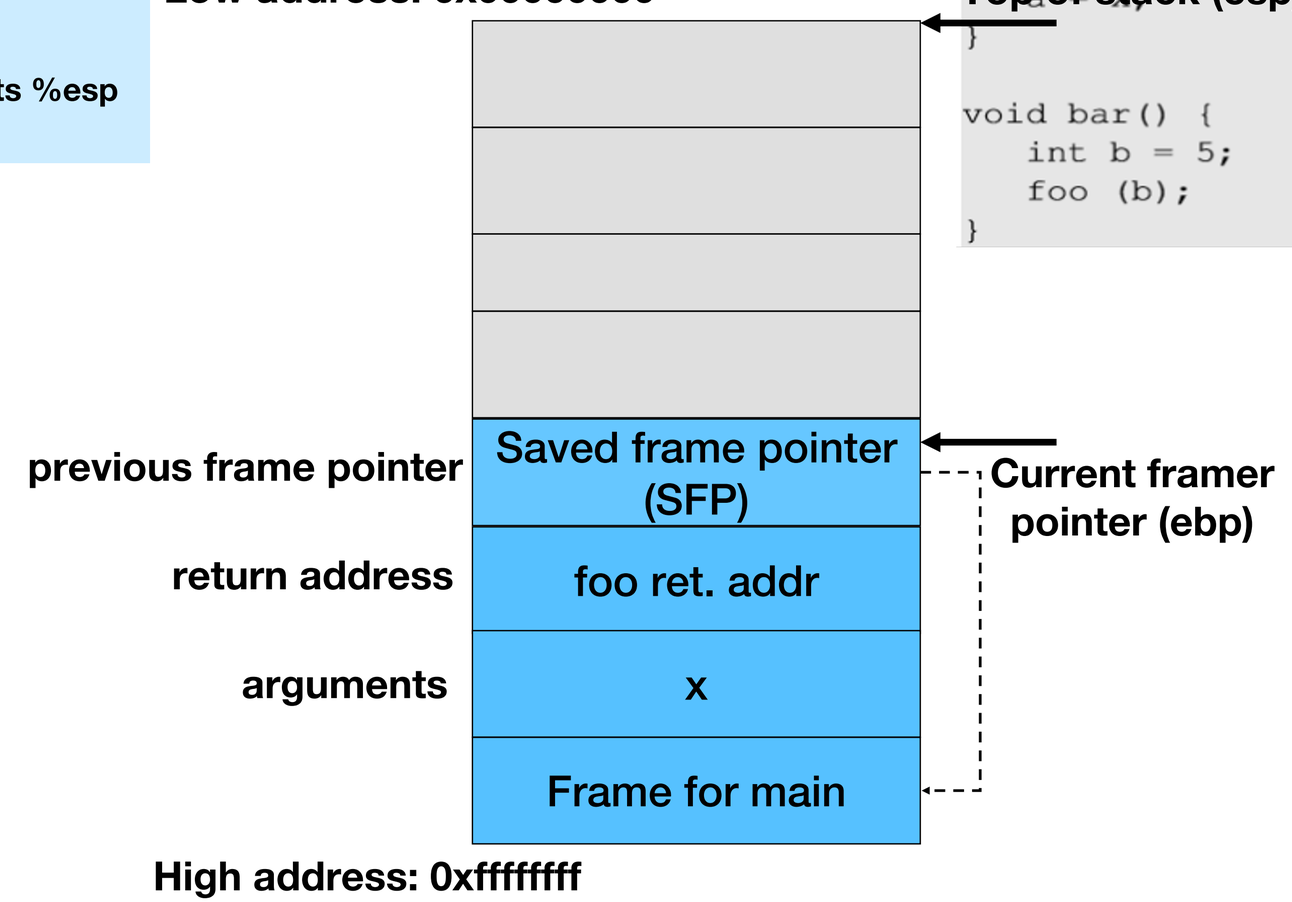


Function Prologue and Epilogue example

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
        movl     8(%ebp), %eax
        movl     %eax, -4(%ebp)
        movl  %ebp, %esp
        popl  %ebp
        ret
```

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

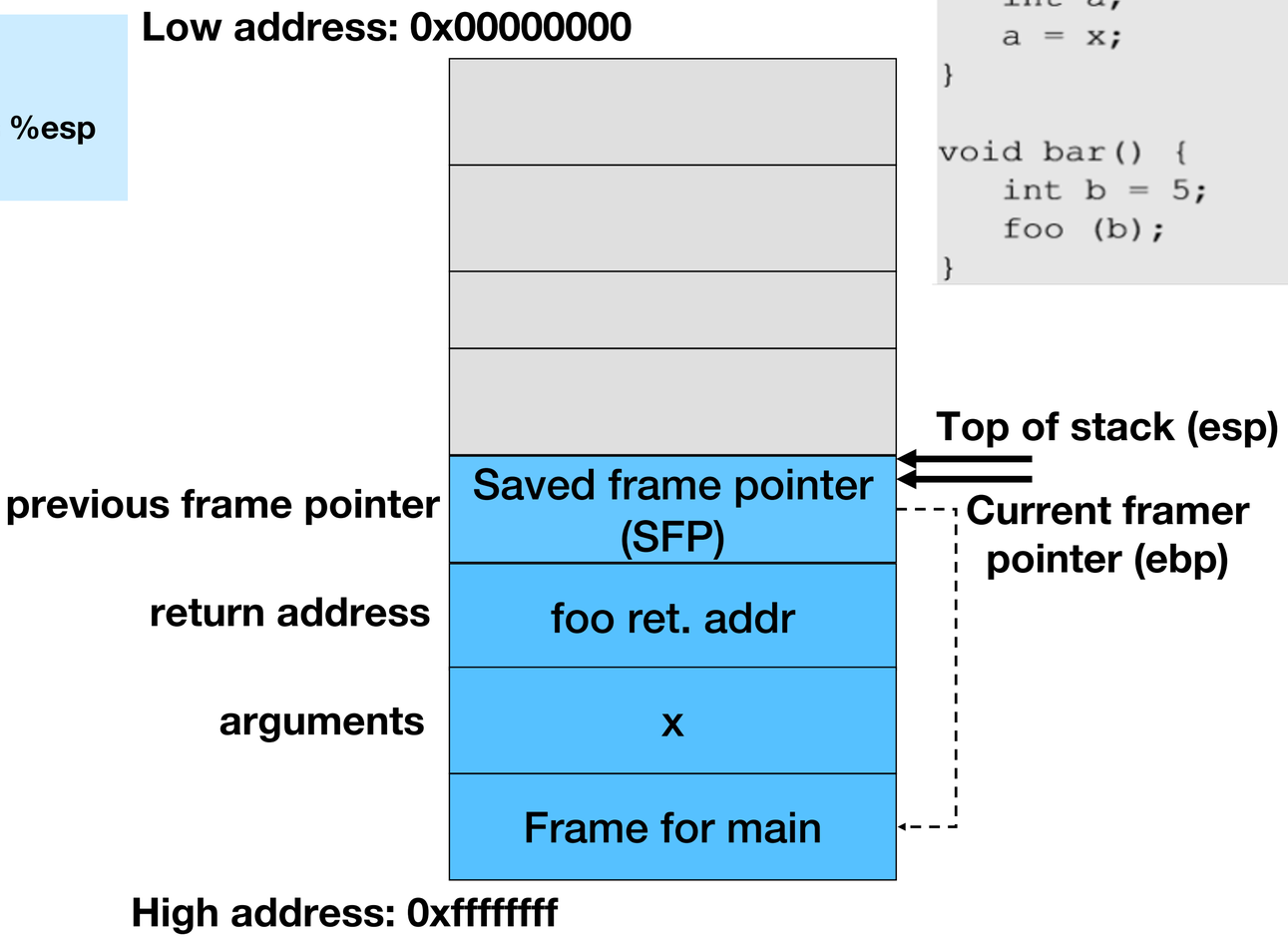


Function Prologue and Epilogue example

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
        movl     8(%ebp), %eax
        movl     %eax, -4(%ebp)
        movl  %ebp, %esp
        popl  %ebp
        ret
```

```
void foo(int x) {
    int a;
    a = x;
}

void bar() {
    int b = 5;
    foo (b);
}
```

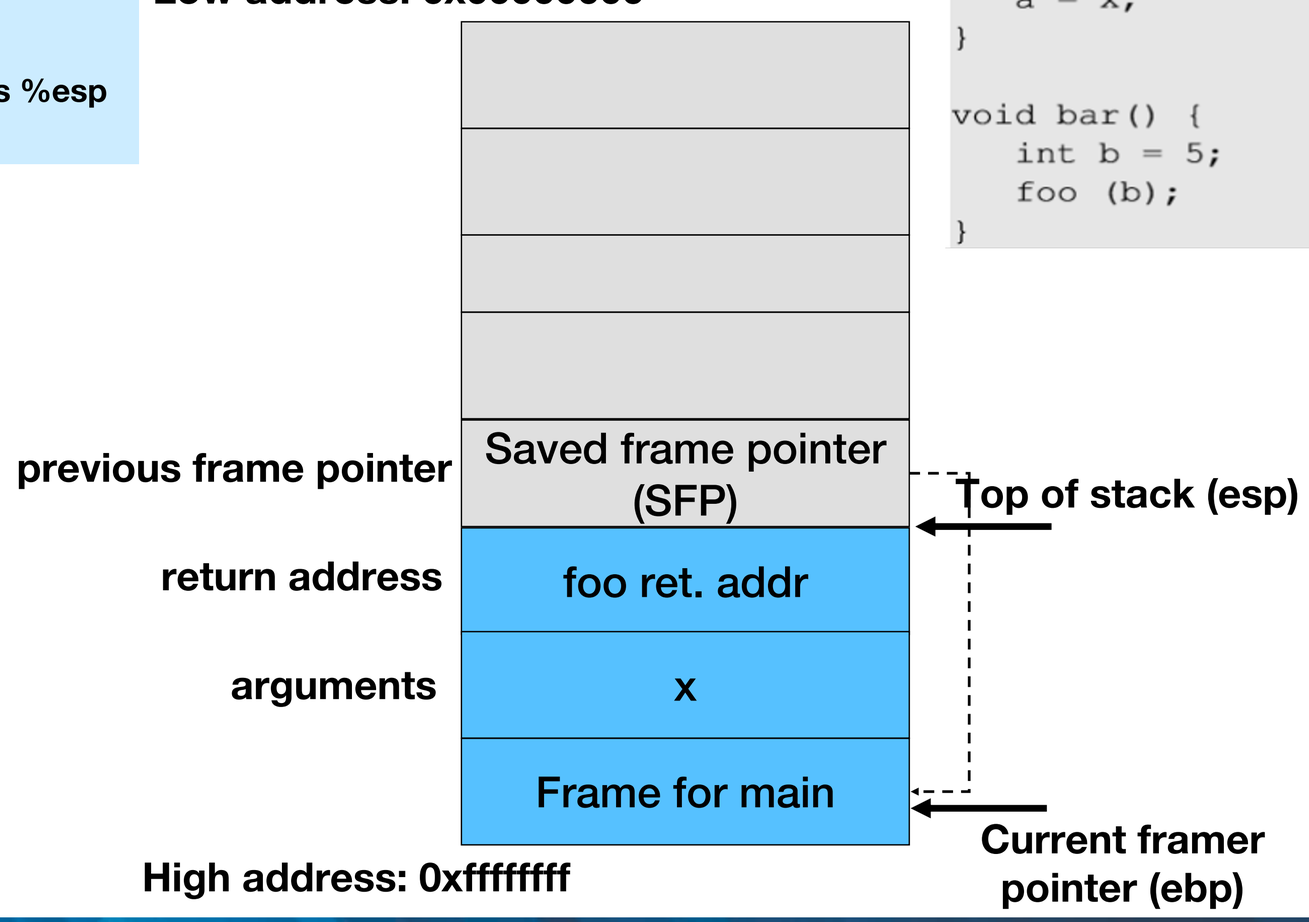


Function Prologue and Epilogue example

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
        movl     8(%ebp), %eax
        movl     %eax, -4(%ebp)
        movl  %ebp, %esp
        popl  %ebp
        ret
```

```
void foo(int x) {
    int a;
    a = x;
}

void bar() {
    int b = 5;
    foo (b);
}
```

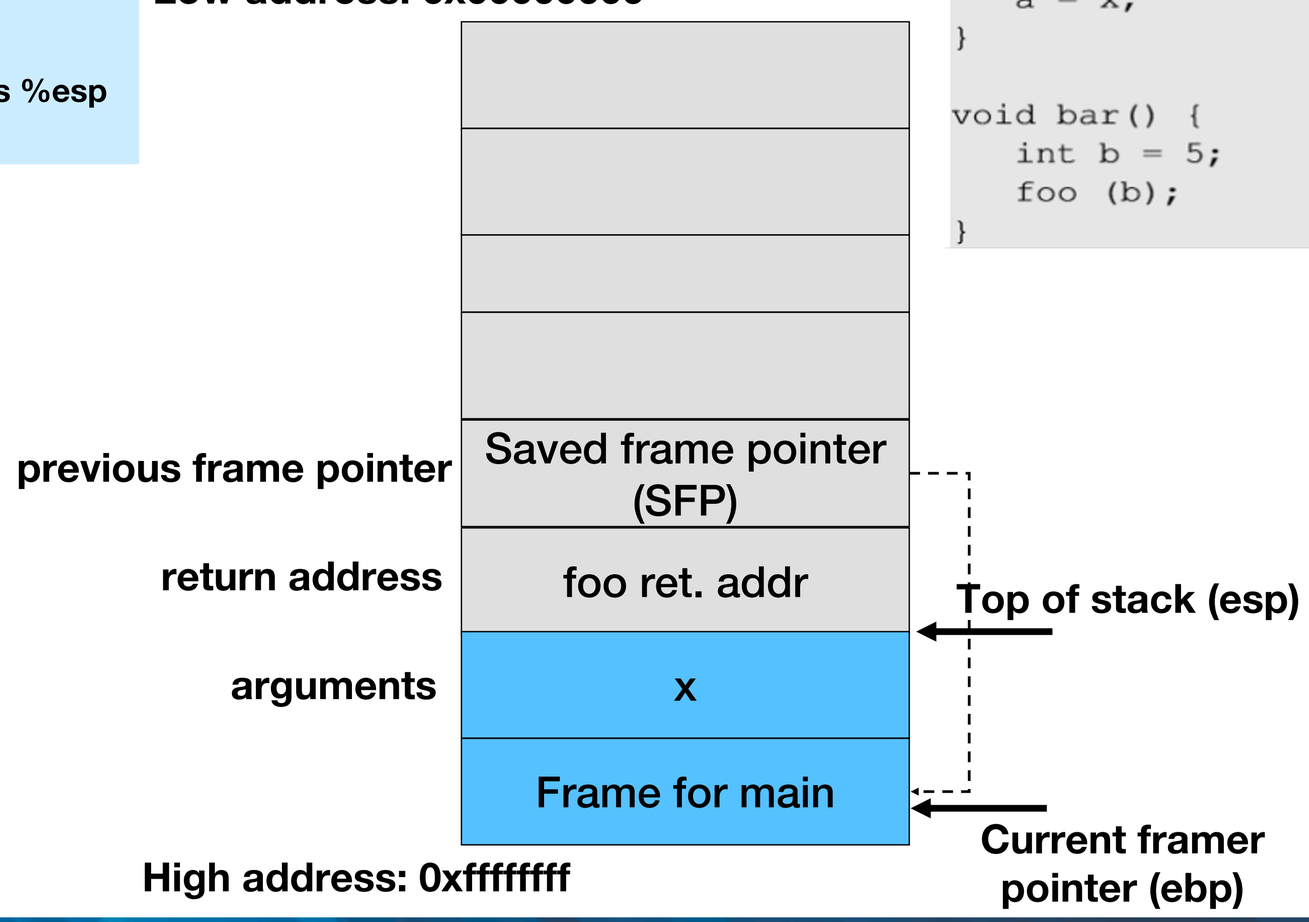


Function Prologue and Epilogue example

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

```
$ gcc -S prog.c
$ cat prog.s
// some instructions omitted
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
        movl     8(%ebp), %eax
        movl     %eax, -4(%ebp)
        movl  %ebp, %esp
        popl  %ebp
        ret
```

```
void foo(int x) {
    int a;
    a = x;
}

void bar() {
    int b = 5;
    foo (b);
}
```



Chaining Function Calls

Assumption:

- foo and bar are compiled and loaded to the code area in the memory.
- the stack is overflowed before foo returns

Objective: to repletely execute bar

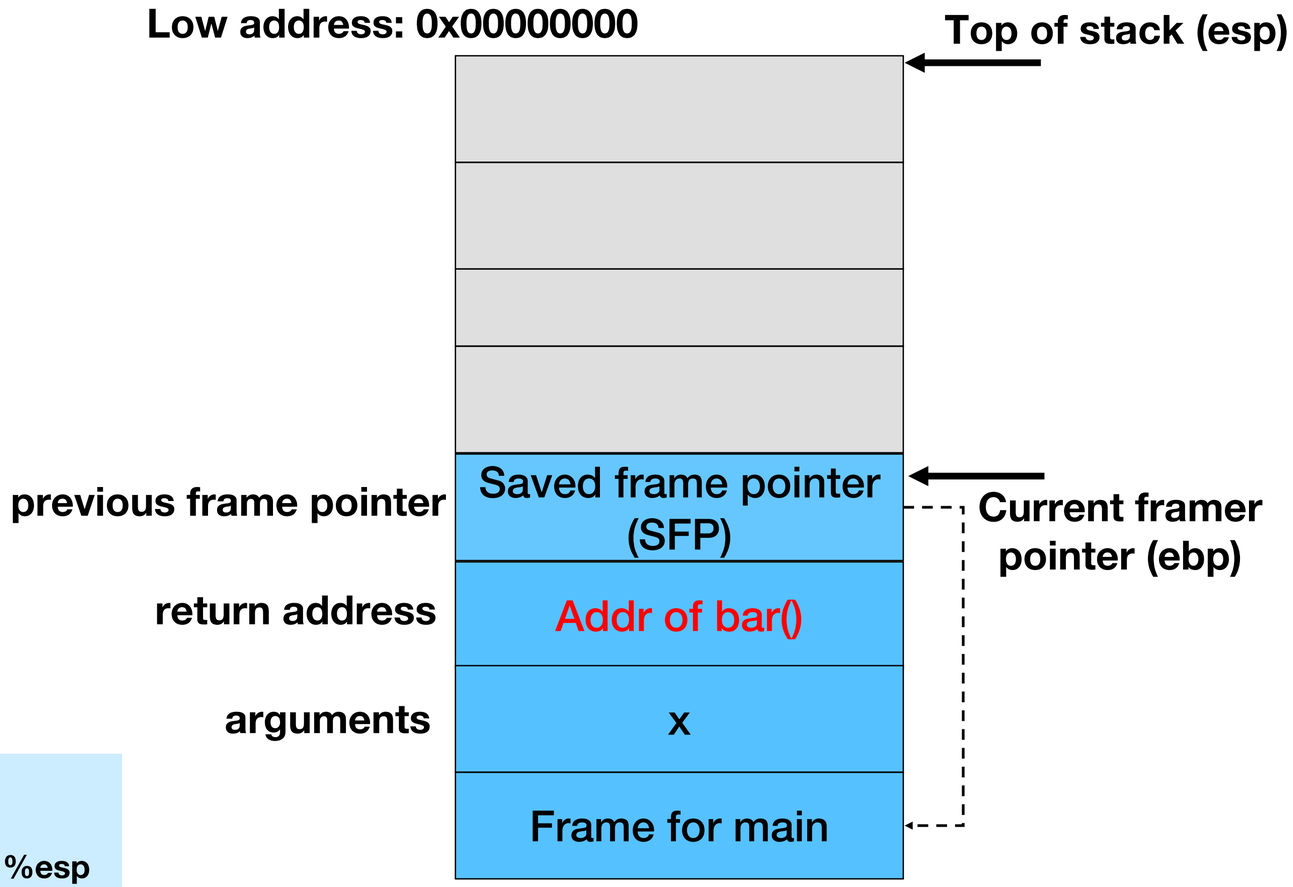
```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```



- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

Chaining Function Calls

Assumption:

- foo and bar are compiled and loaded to the code area in the memory.
- the stack is overflowed before foo returns

Objective: to repletely execute bar

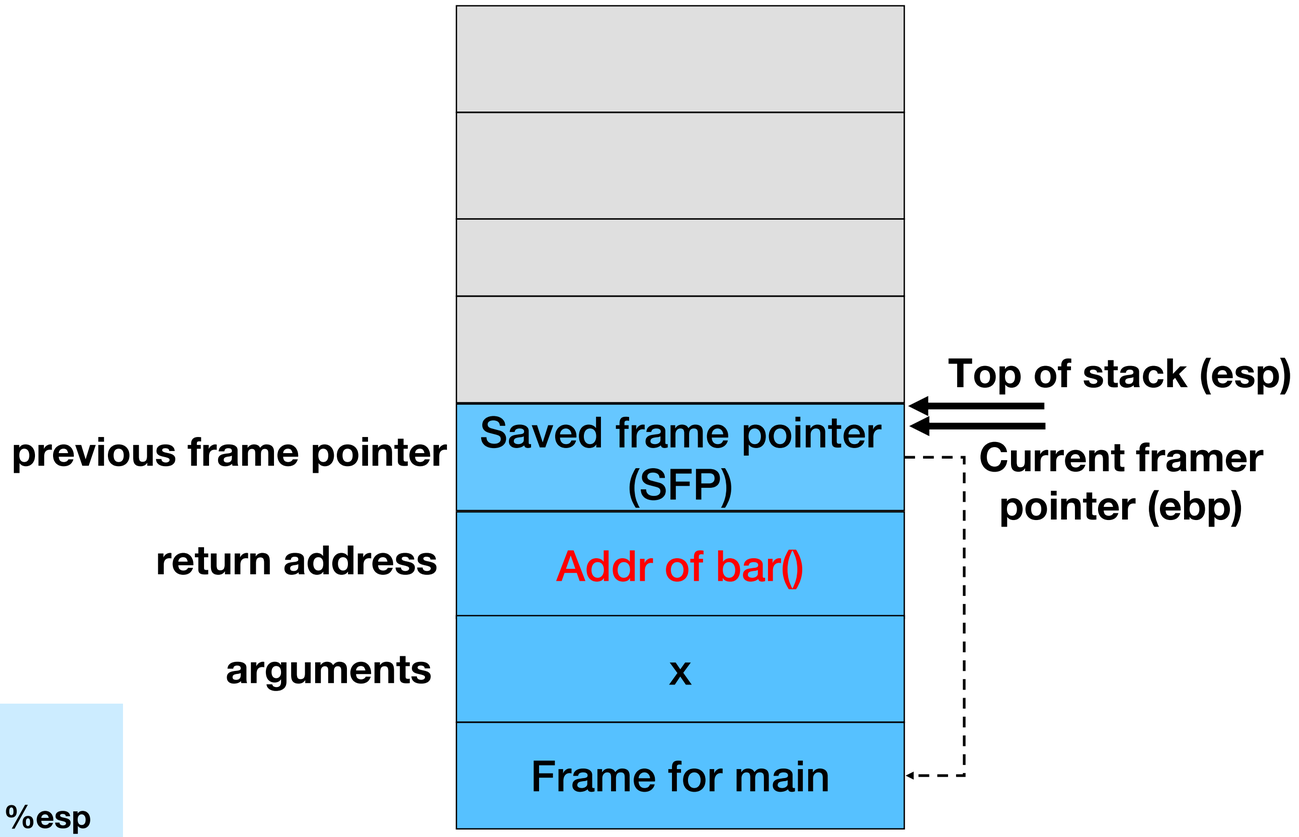
```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```



- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

Chaining Function Calls

Assumption:

- foo and bar are compiled and loaded to the code area in the memory.
- the stack is overflowed before foo returns

Objective: to repletely execute bar

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

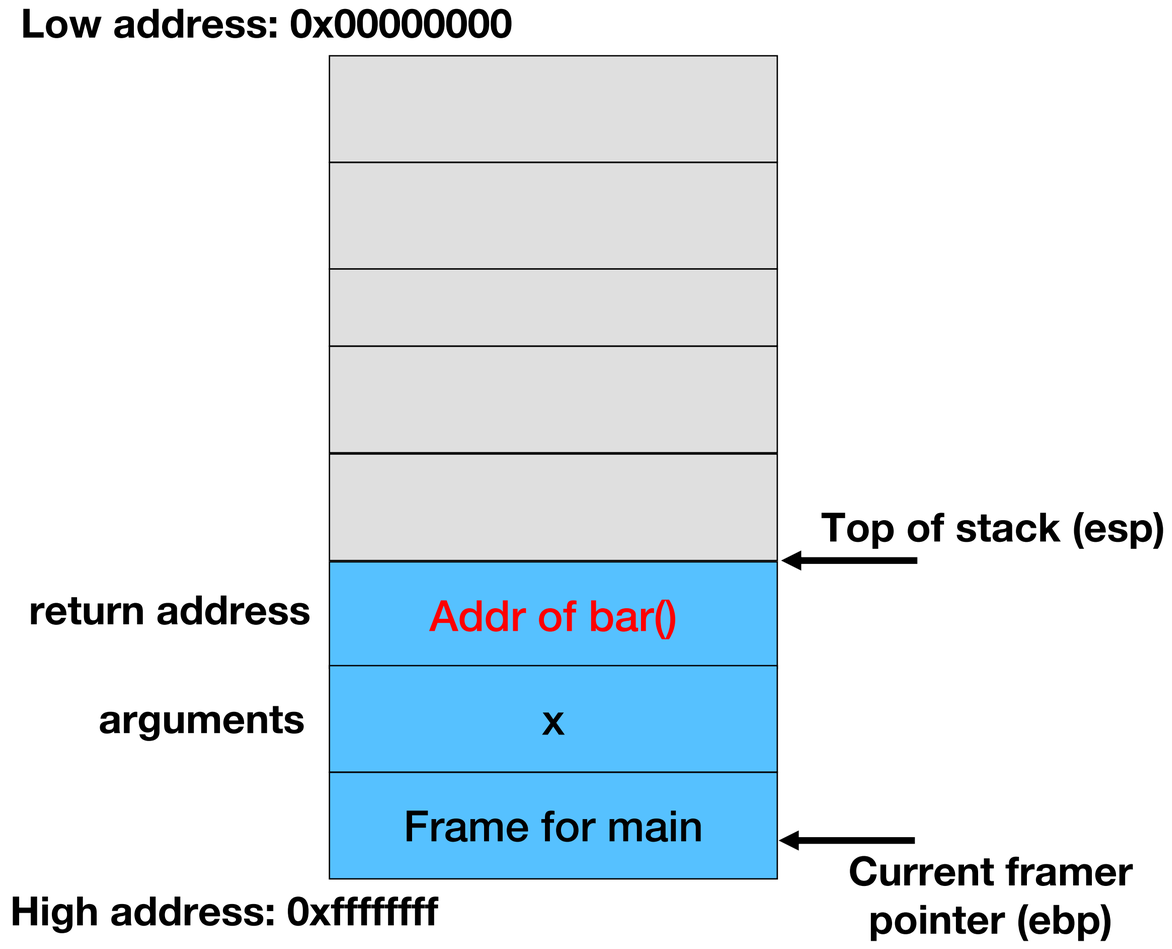
Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp



Chaining Function Calls

Assumption:

- foo and bar are compiled and loaded to the code area in the memory.
- the stack is overflowed before foo returns

Objective: to repletely execute bar

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

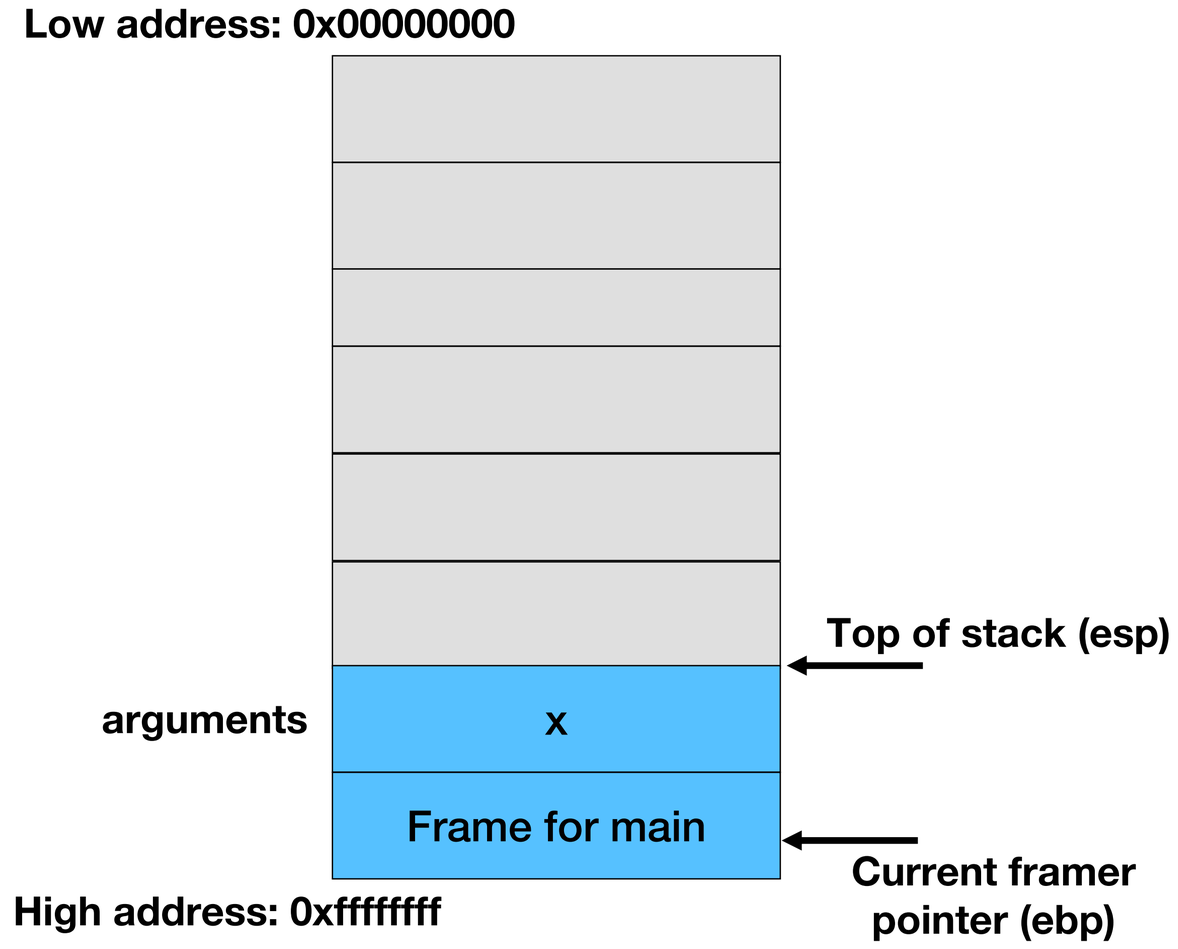
Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp



Chaining Function Calls

Assumption:

- foo and bar are compiled and loaded to the code area in the memory.
- the stack is overflowed before foo returns

Objective: to repletely execute bar

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

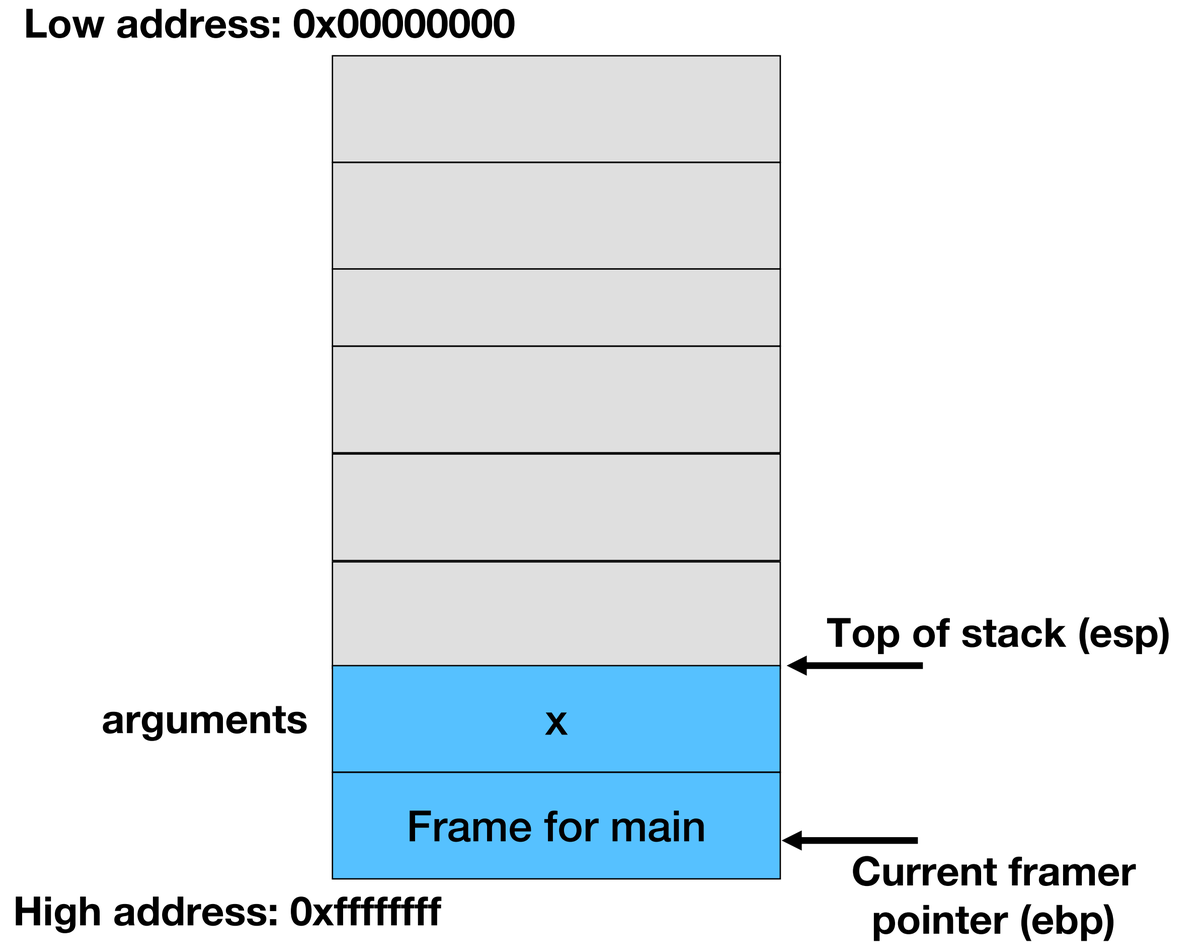
Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp



Chaining Function Calls

Assumption:

- foo and bar are compiled and loaded to the code area in the memory.
- the stack is overflowed before foo returns

Objective: to repletely execute bar

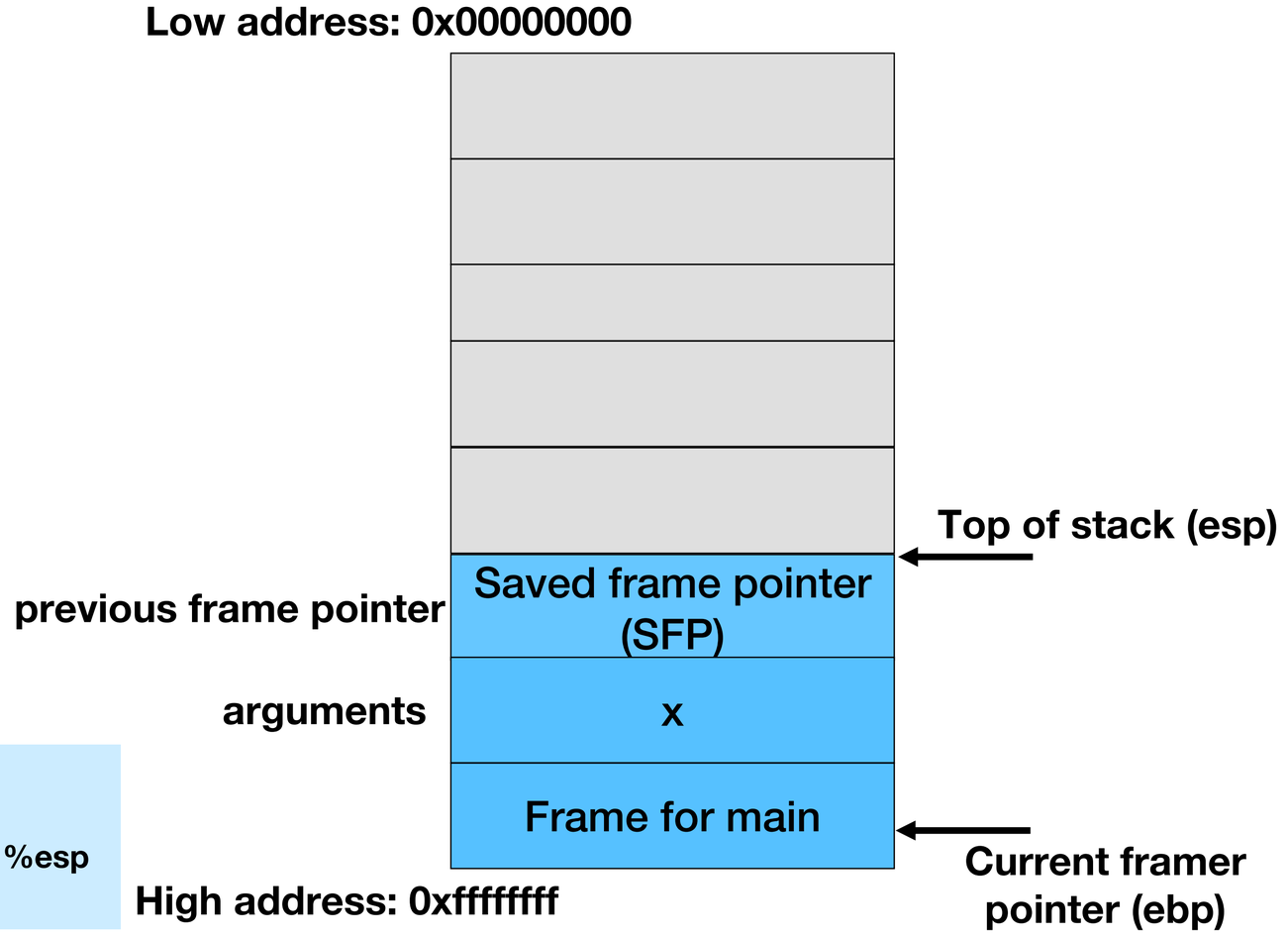
```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```



- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

Chaining Function Calls

Assumption:

- foo and bar are compiled and loaded to the code area in the memory.
- the stack is overflowed before foo returns

Objective: to repletely execute bar

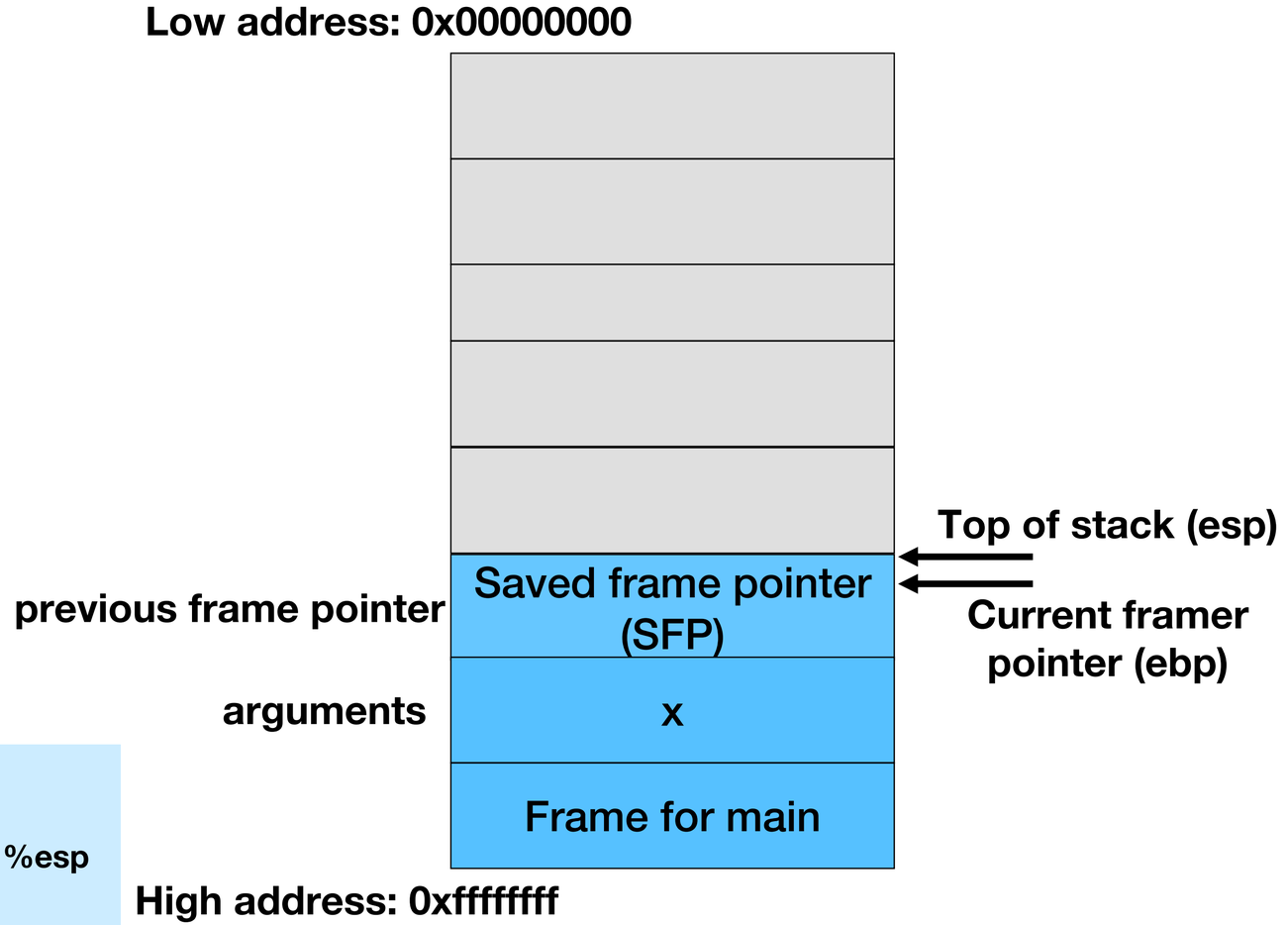
```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```



- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

Chaining Function Calls

Assumption:

- foo and bar are compiled and loaded to the code area in the memory.
- the stack is overflowed before foo returns

Objective: to repletely execute bar

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

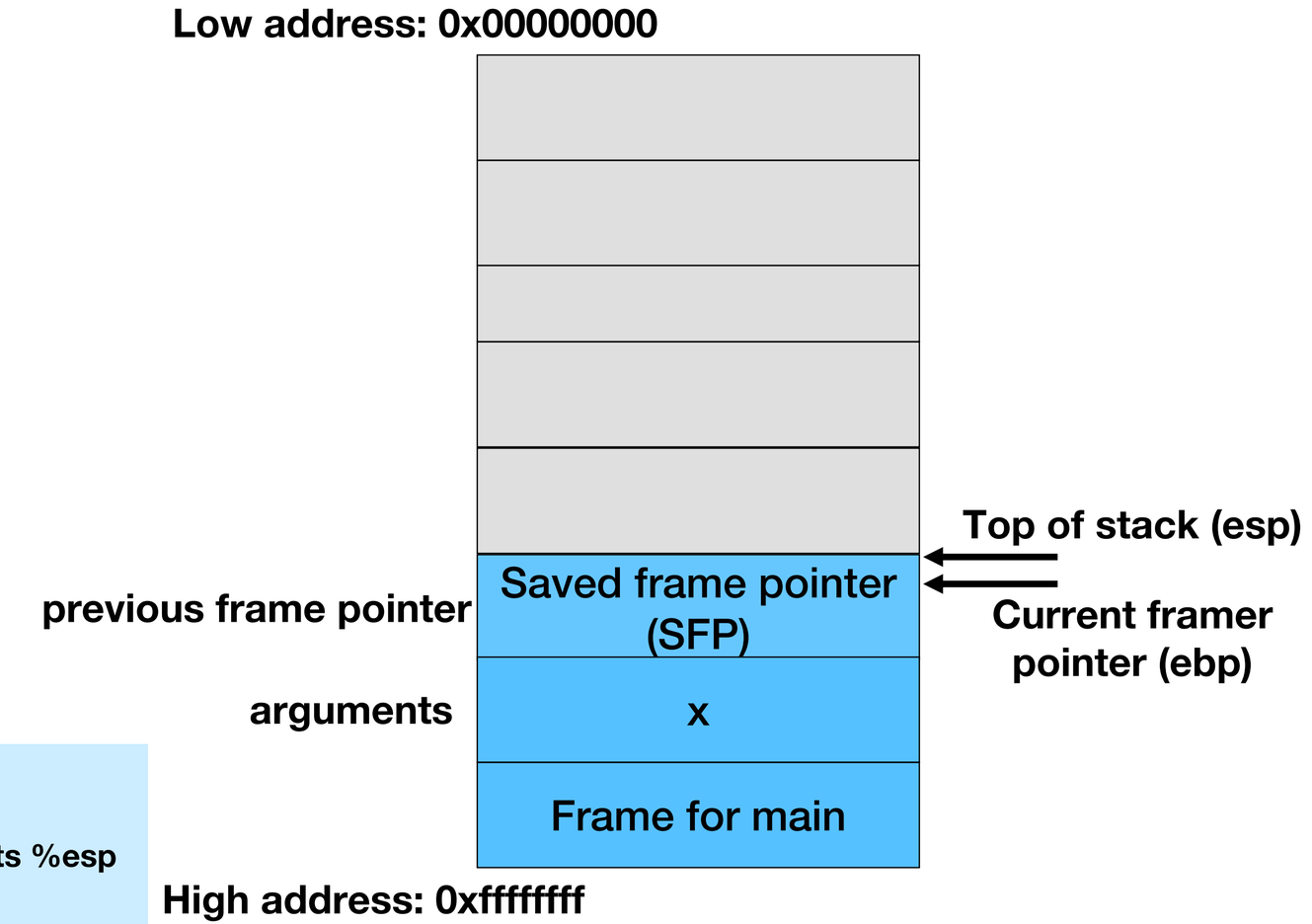
Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp



Chaining Function Calls

Assumption:

- foo and bar are compiled and loaded to the code area in the memory.
- the stack is overflowed before foo returns

Objective: to repletely execute bar

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

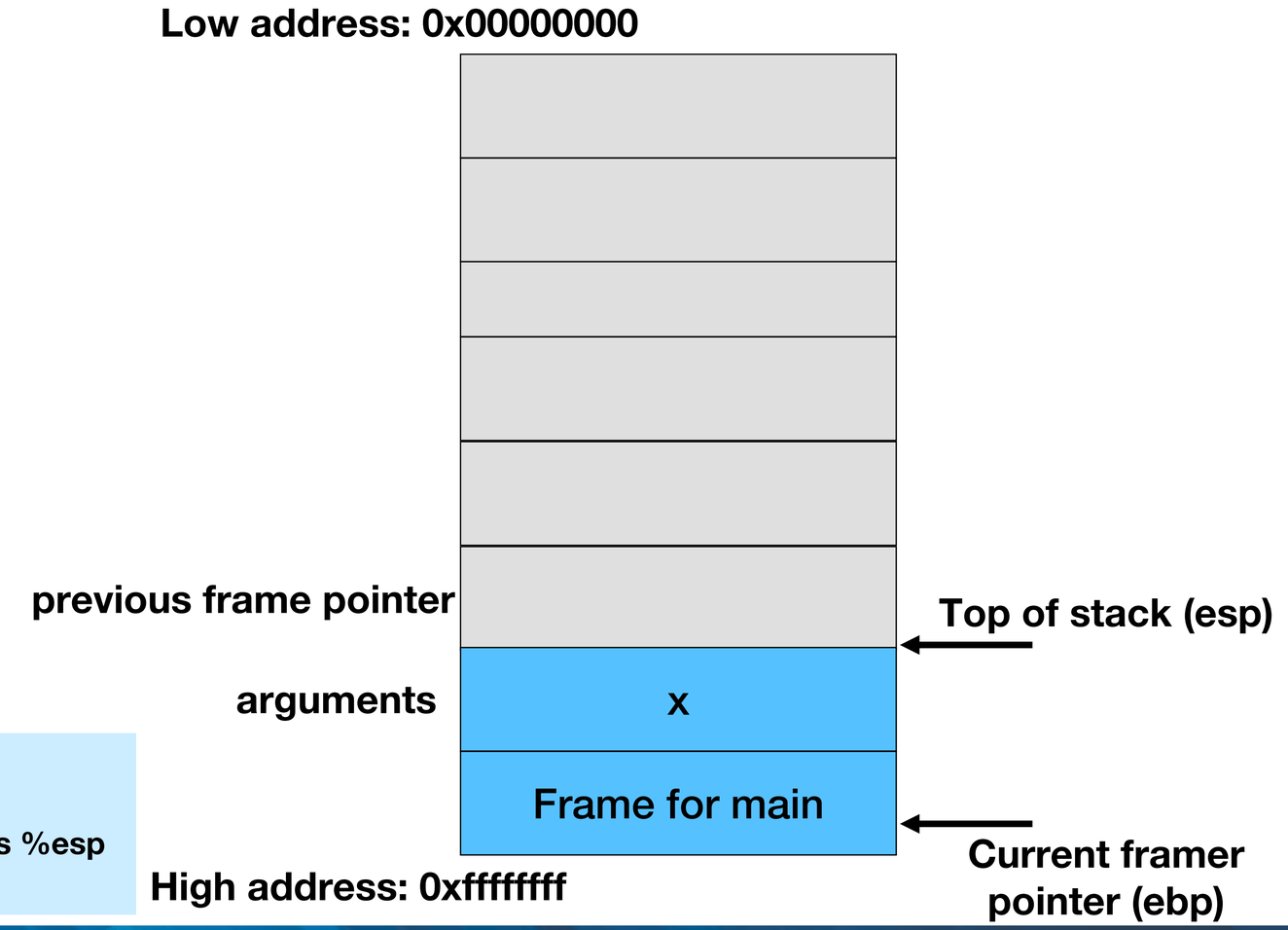
Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp



Chaining Function Calls

Assumption:

- **foo and bar are compiled and loaded to the code area in the memory.**
- **the stack is overflowed before foo returns**

Objective: to repletely execute bar

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

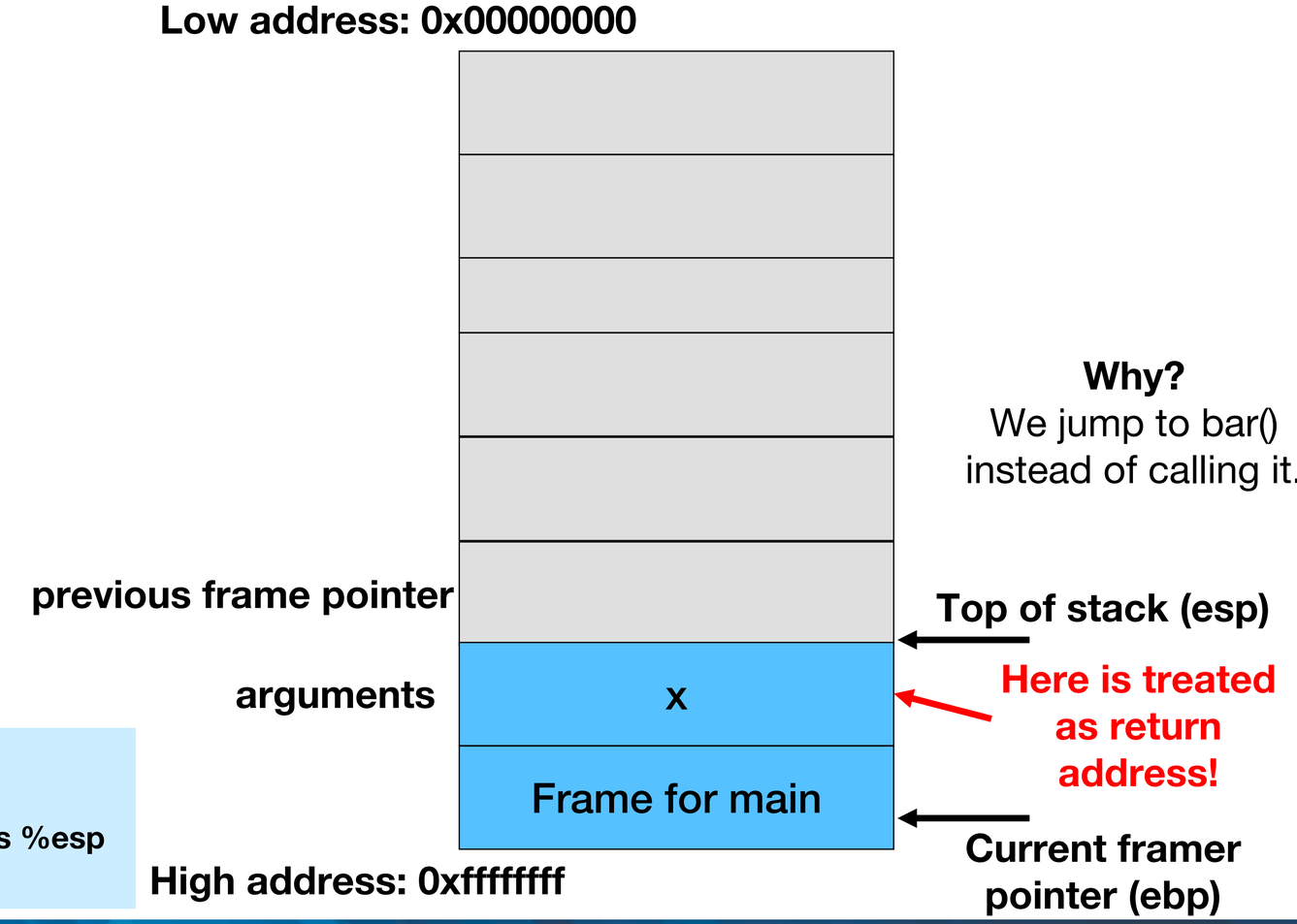
Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```

- ✓ **%esp always points to the frontier of the stack**
  - **Push inserts its operand on top of stack, decrements %esp**
  - **Pop removes a value from top of stack into its operand, increments %esp**
- ✓ **%ebp always points to the cell storing the old %ebp**



Why?
We jump to bar() instead of calling it.

Chaining Function Calls

Assumption:

- foo and bar are compiled and loaded to the code area in the memory.
- the stack is overflowed before foo returns

Objective: to repletely execute bar

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

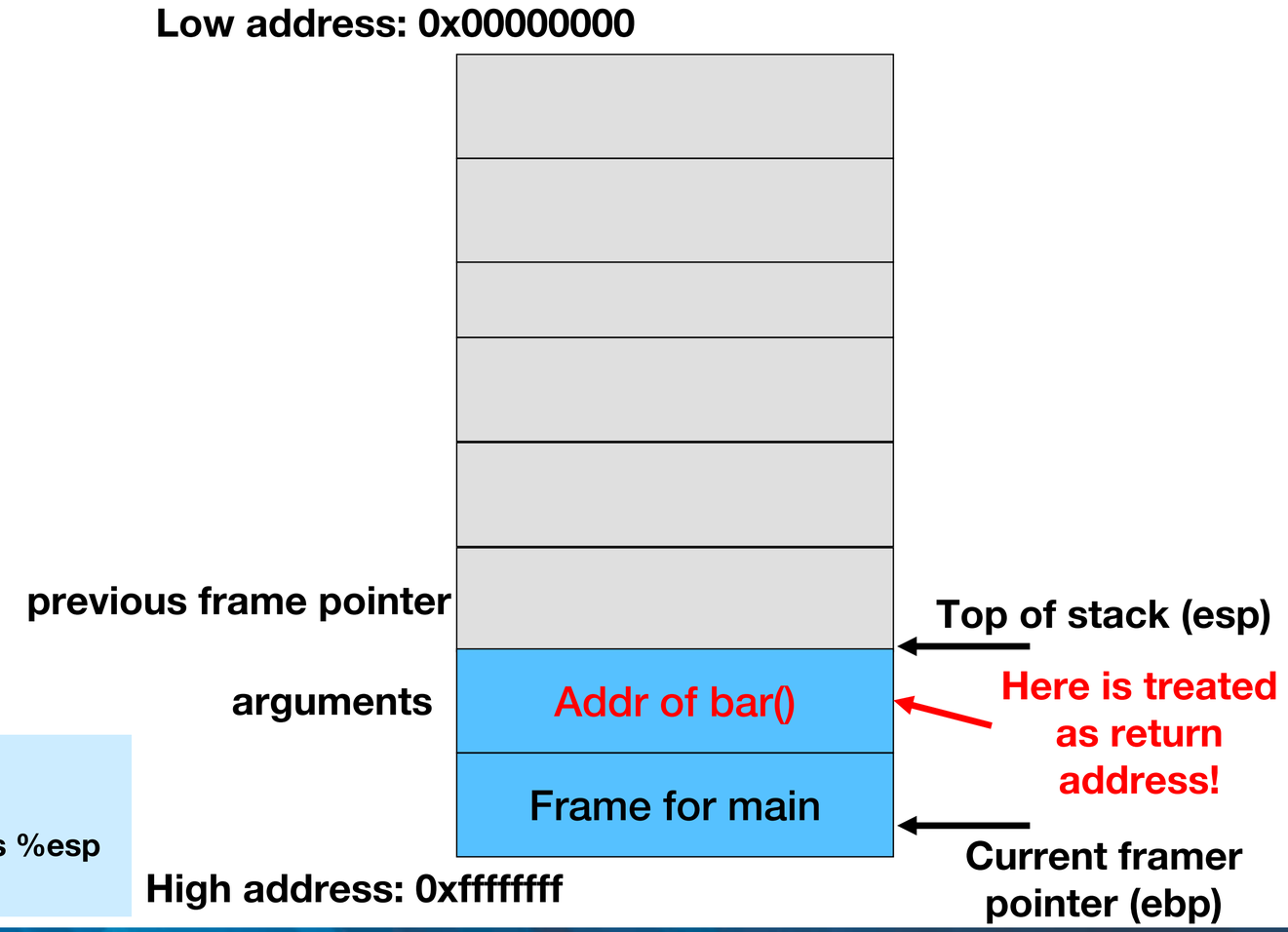
Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```

- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp



Chaining Function Calls

Assumption:

- foo and bar are compiled and loaded to the code area in the memory.
- the stack is overflowed before foo returns

Objective: to repletely execute bar

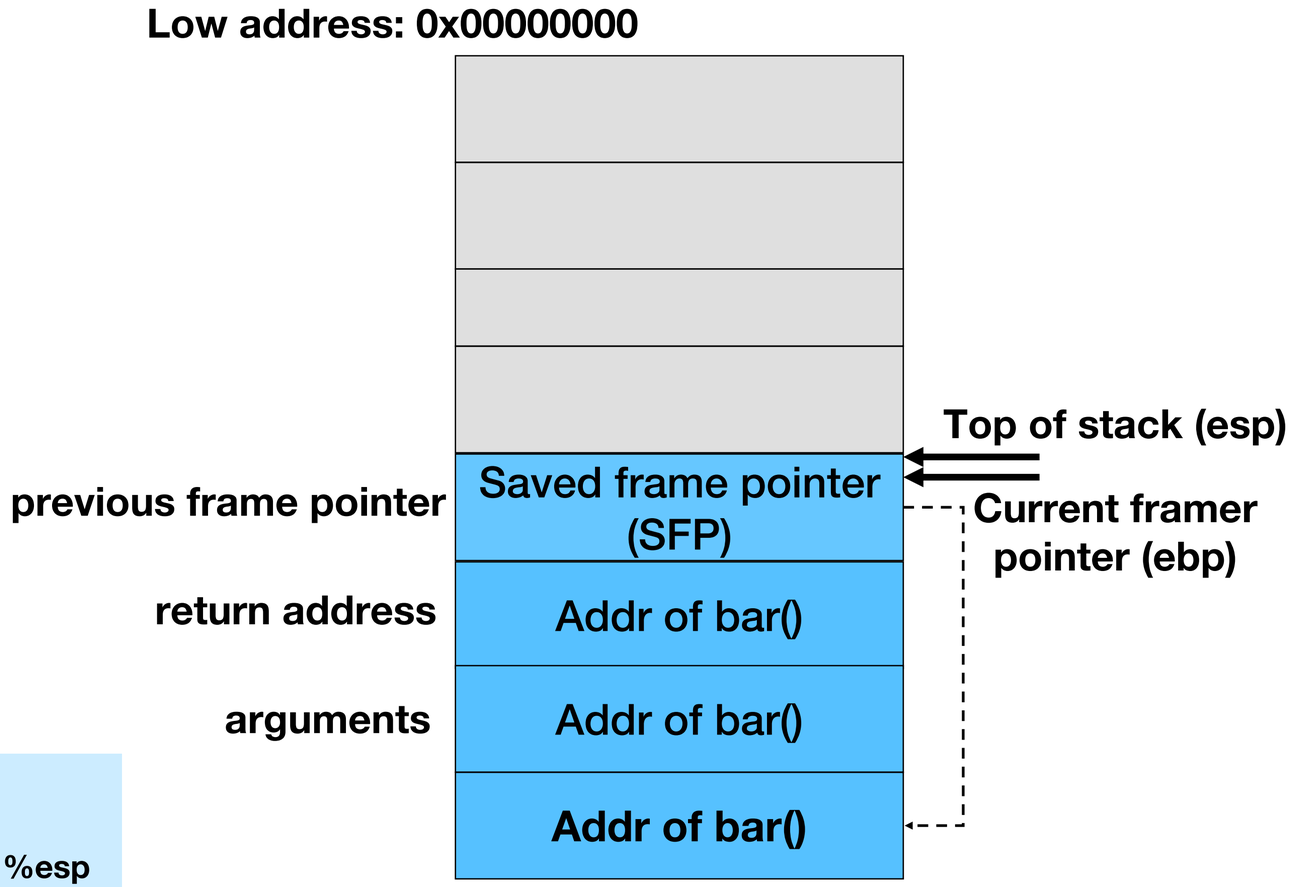
```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo (b);
}
```

Prologue

```
pushl %ebp
movl %esp, %ebp
```

Epilogue

```
movl  %ebp, %esp
popl  %ebp
ret
```

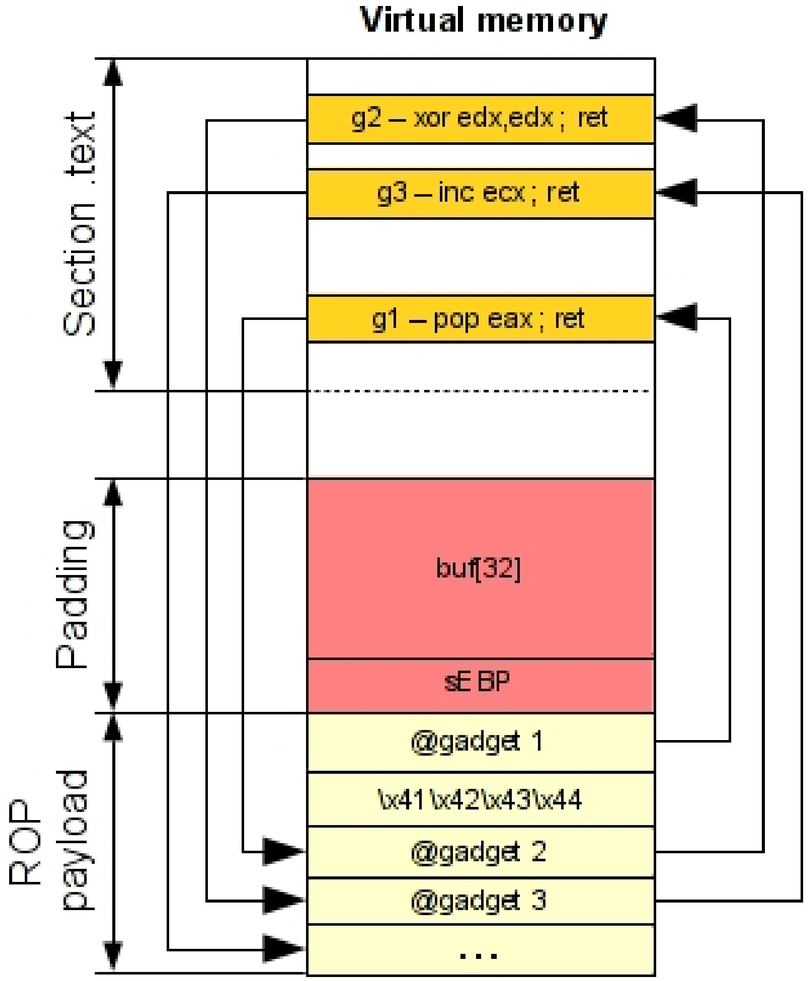


- ✓ %esp always points to the frontier of the stack
  - Push inserts its operand on top of stack, decrements %esp
  - Pop removes a value from top of stack into its operand, increments %esp
- ✓ %ebp always points to the cell storing the old %ebp

ROP at work

Other ROP attacks:

- To chain various functions
- To chain functions with arguments
- To chain functions with special arguments (e.g., 0)
- Chain instruction fragments



W Du, Computer Security - A Hands-on Approach, 2017

Summary

- Definition of buffer overflow
- How buffer overflow can be exploited
- Mitigations to prevent such exploitation
- Some advanced exploitation, e.g., ROP