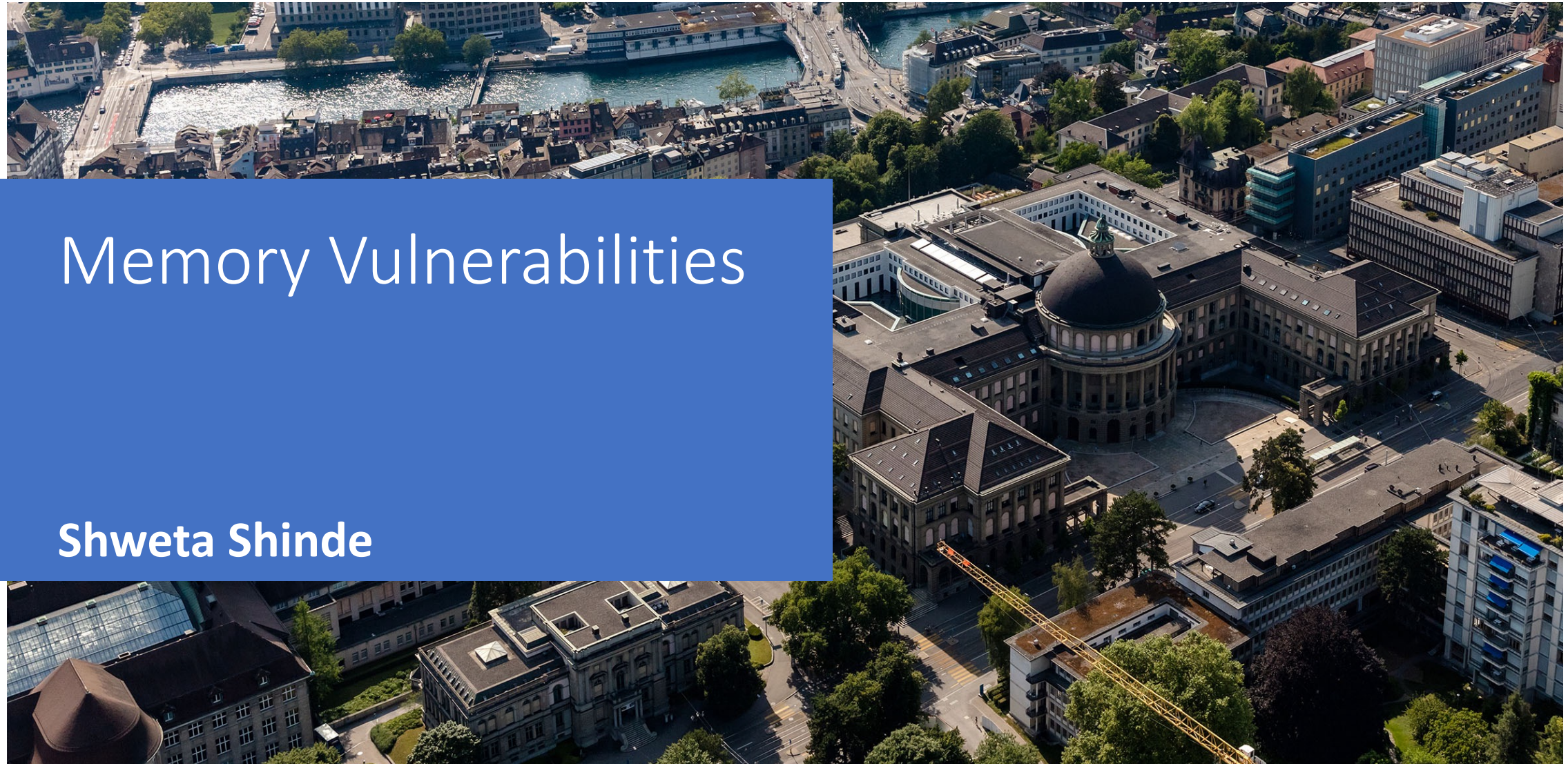
ETH zürich

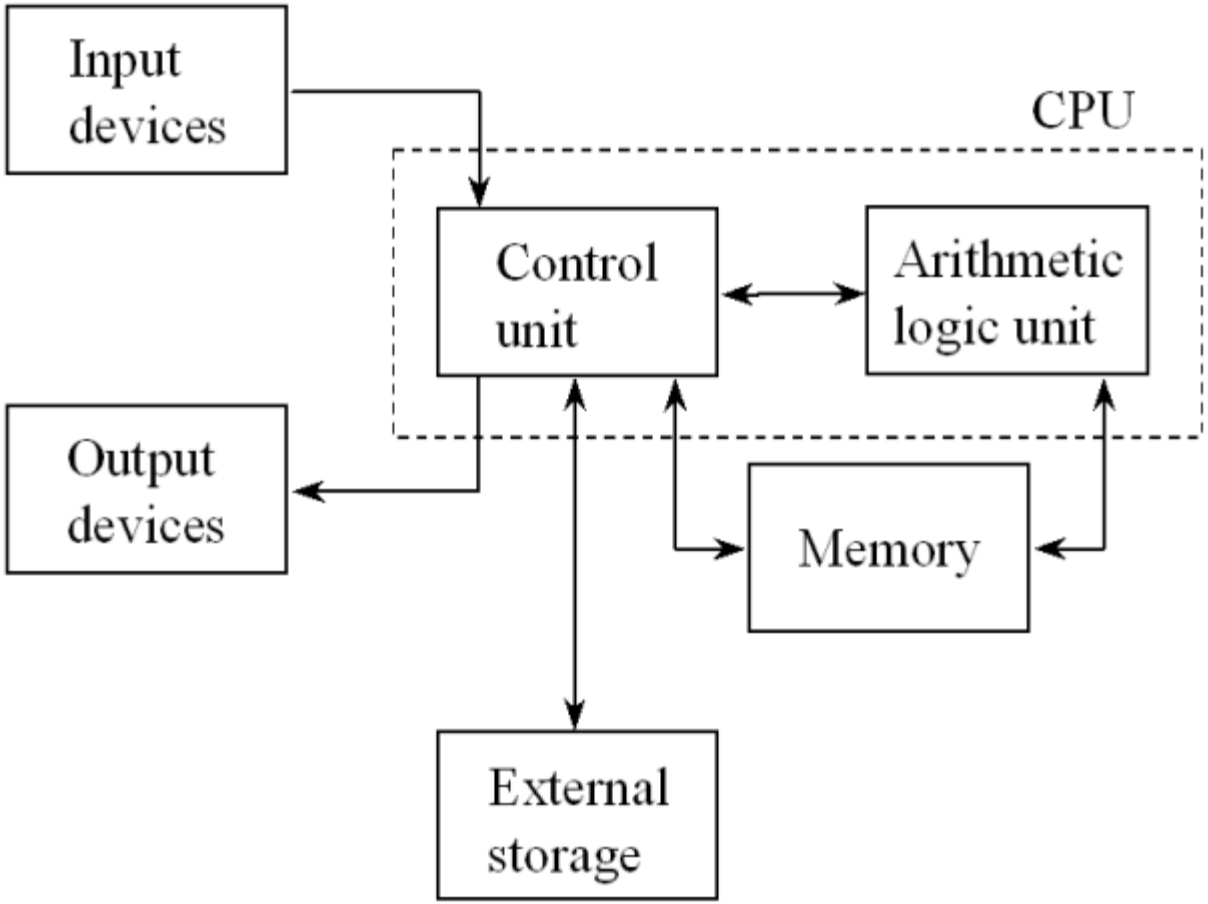
SECTRS
Secure & Trustworthy Systems Group



Memory Vulnerabilities

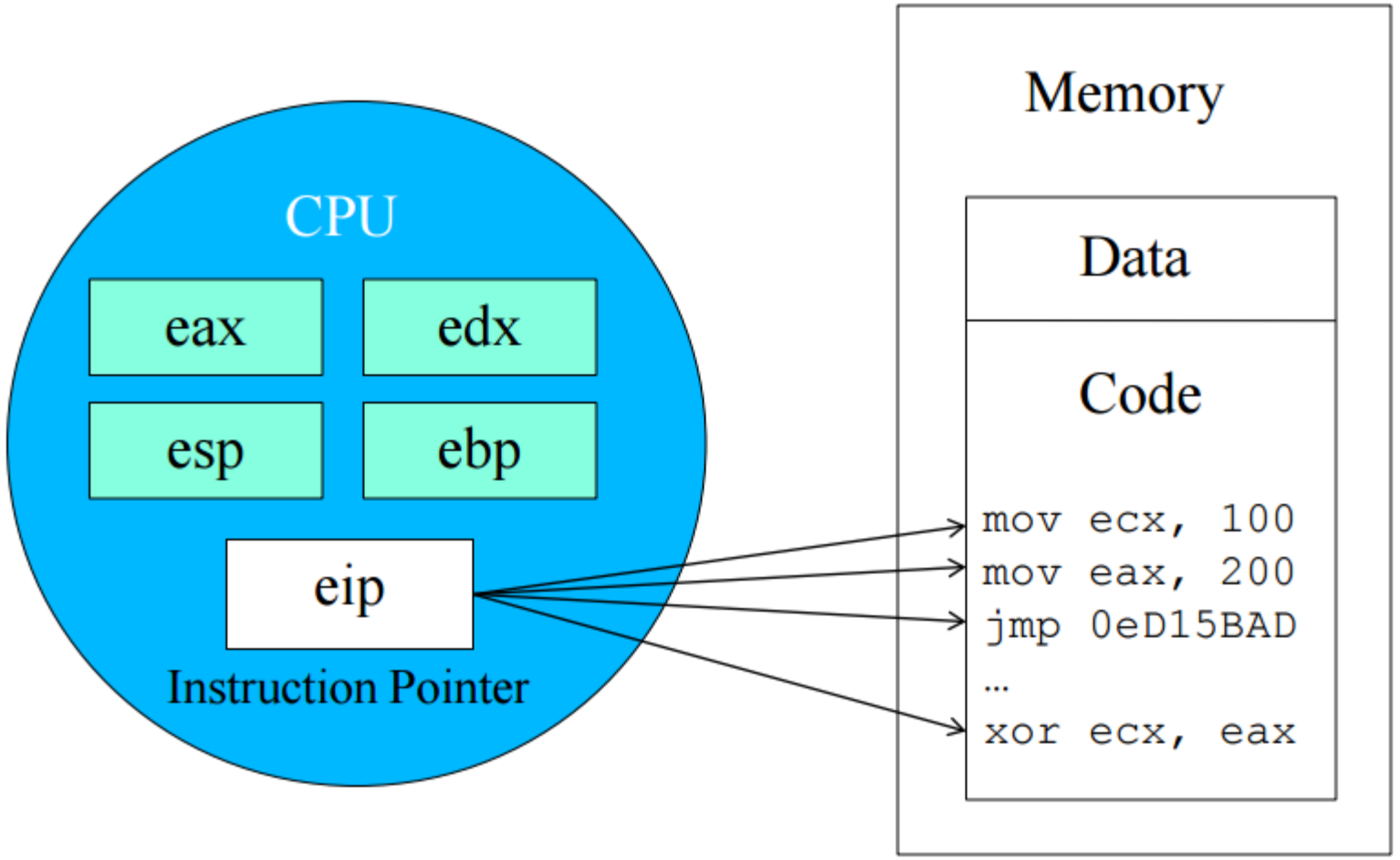
Shweta Shinde

Basics:
The x86 Machine Model



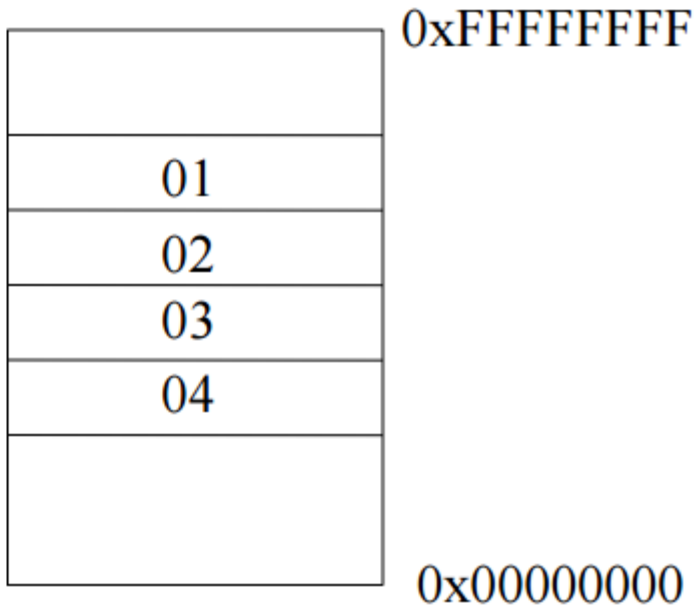
Von Neumann Architecture

Basics:
The x86 Machine Model

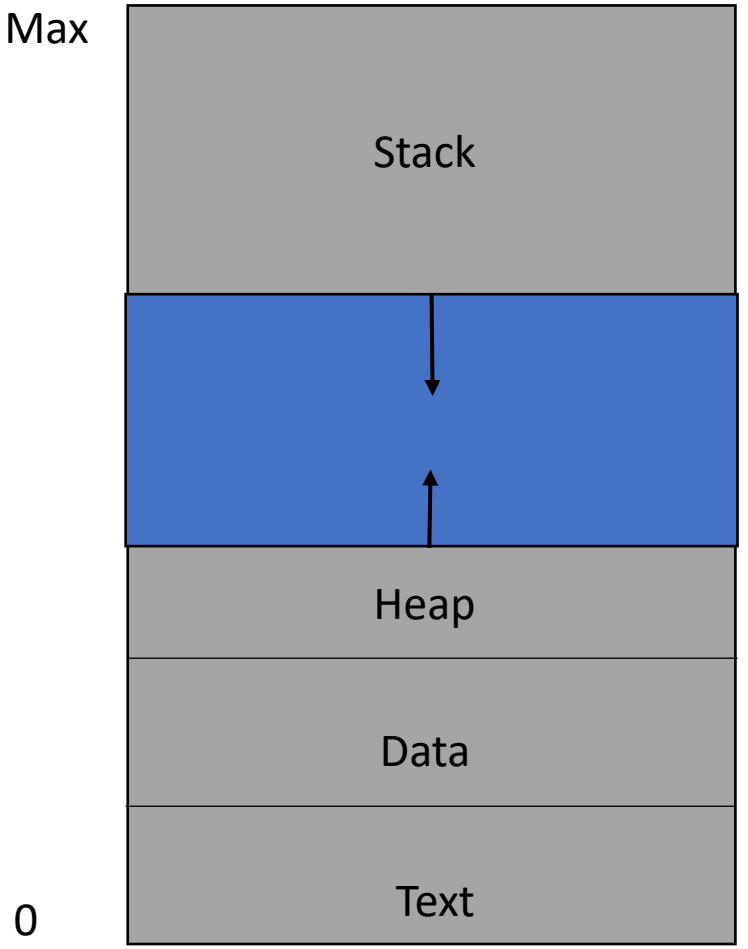


Basics:
The x86 Machine Model

- Both code and data are represented as numbers
- Code
  - `lea ecx, [esp+4]` represented as `0x8d 0x4c 0x24 0x04`
- Data
  - On Intel CPUs, least significant bytes is put at lower addresses
  - It is called little endian
  - For example, `0x01020304`

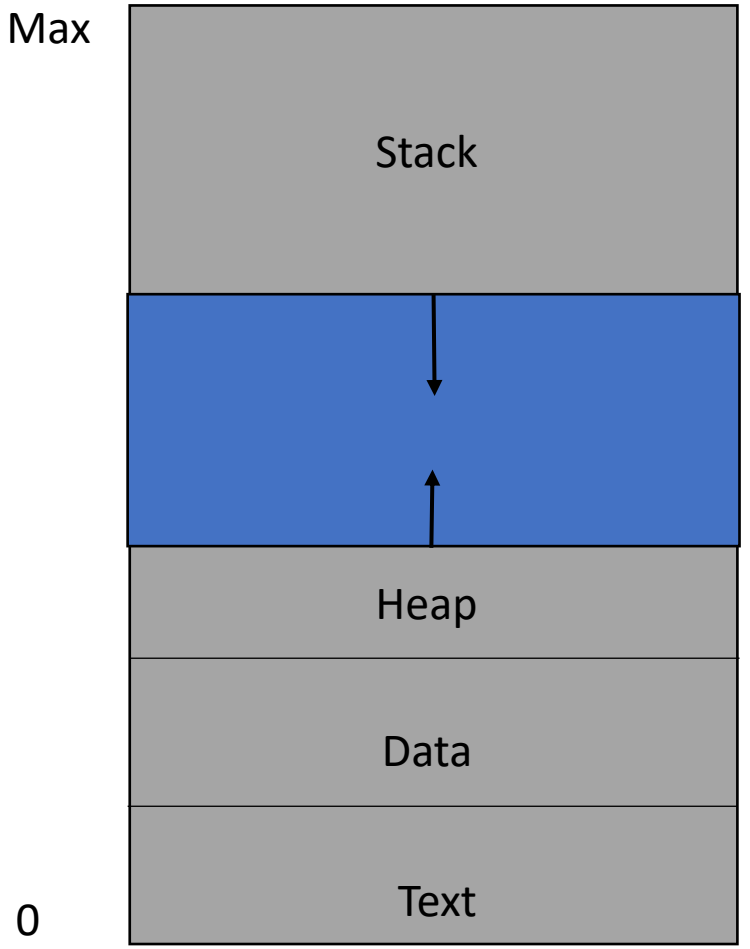


Basics:
The x86 Machine Model



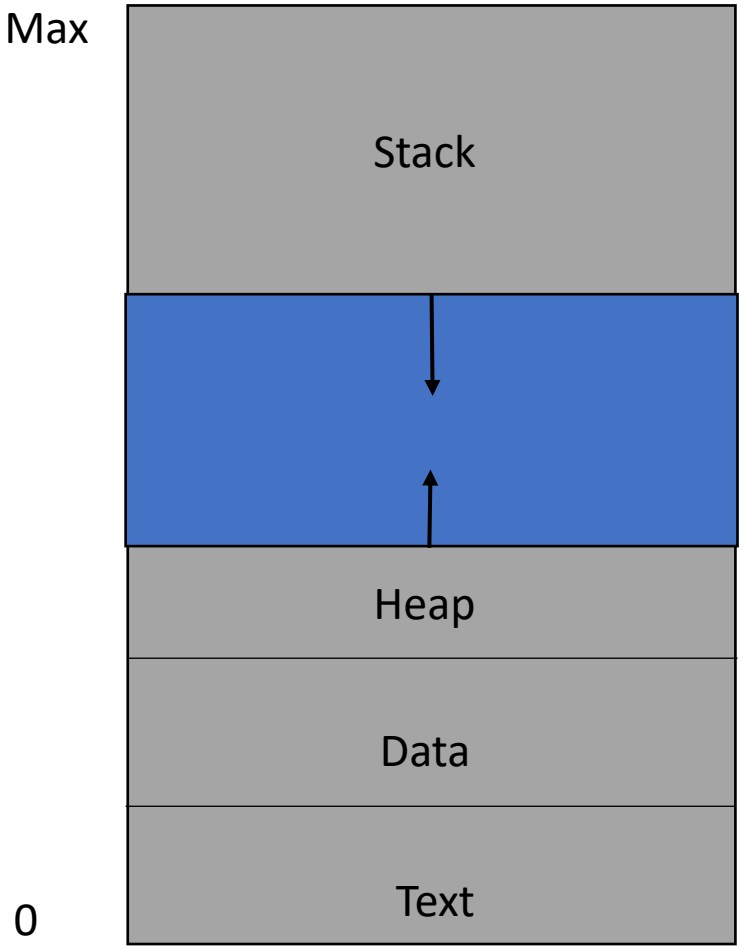
Basics:
The x86 Machine Model

- Registers, Instructions, Stack, EIP
- Addressing modes, offset addresses
  - `mov 0x12[ebp], ecx`
- Stack grows down, other memory accesses move up.

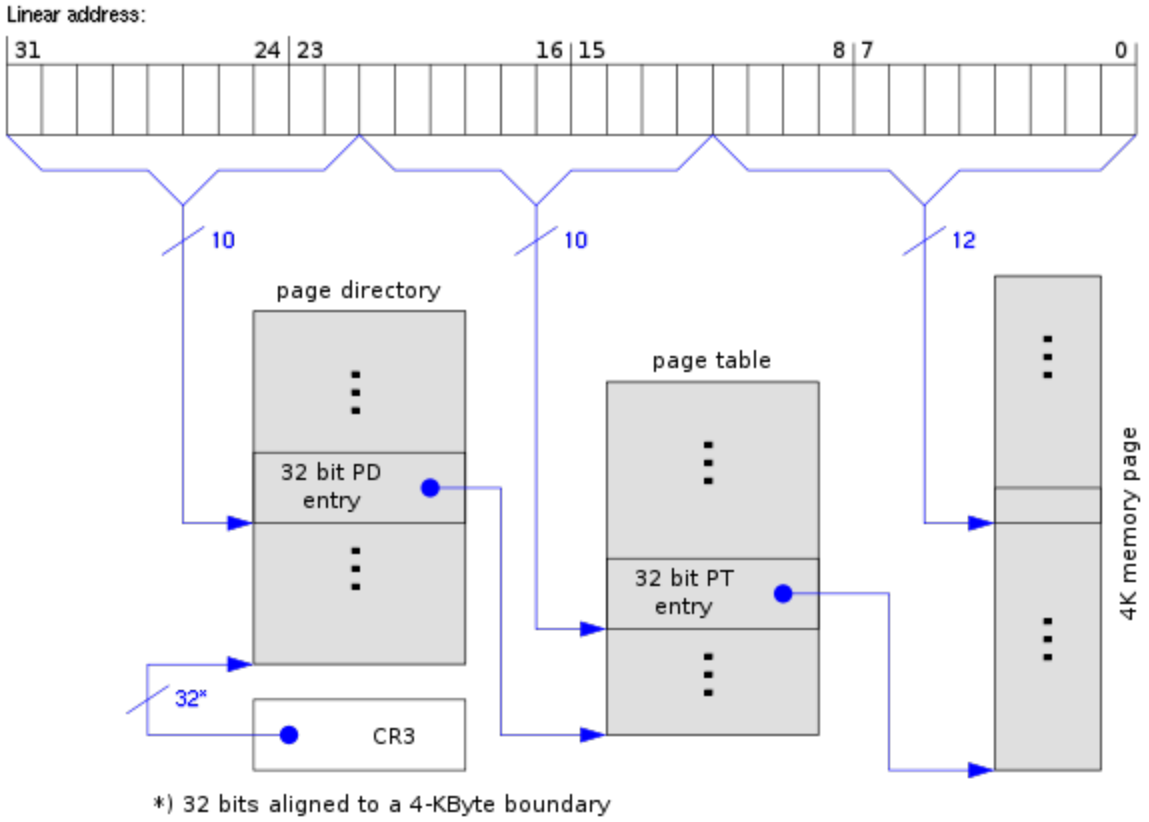


Basics:
The x86 Machine Model

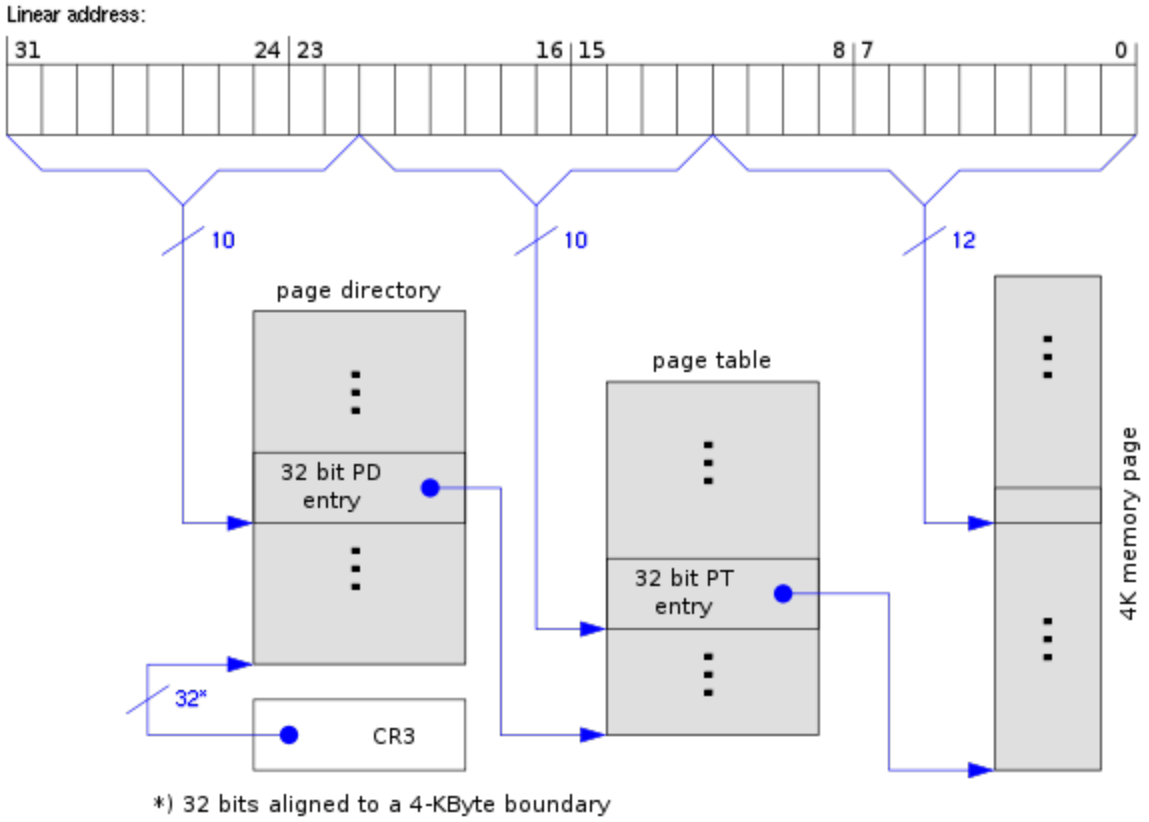
- Registers, Instructions, Stack, EIP
- Addressing modes, offset addresses
  - `mov 0x12[ebp], ecx`
- Stack grows down, other memory accesses move up.



Basics:
The x86 Machine Model



Basics:
The x86 Machine Model



- Page permissions, CPU checks and raises faults
- What is the identity of a process?
  - Processes identified by CR3
- Ring 3 vs. Ring 0

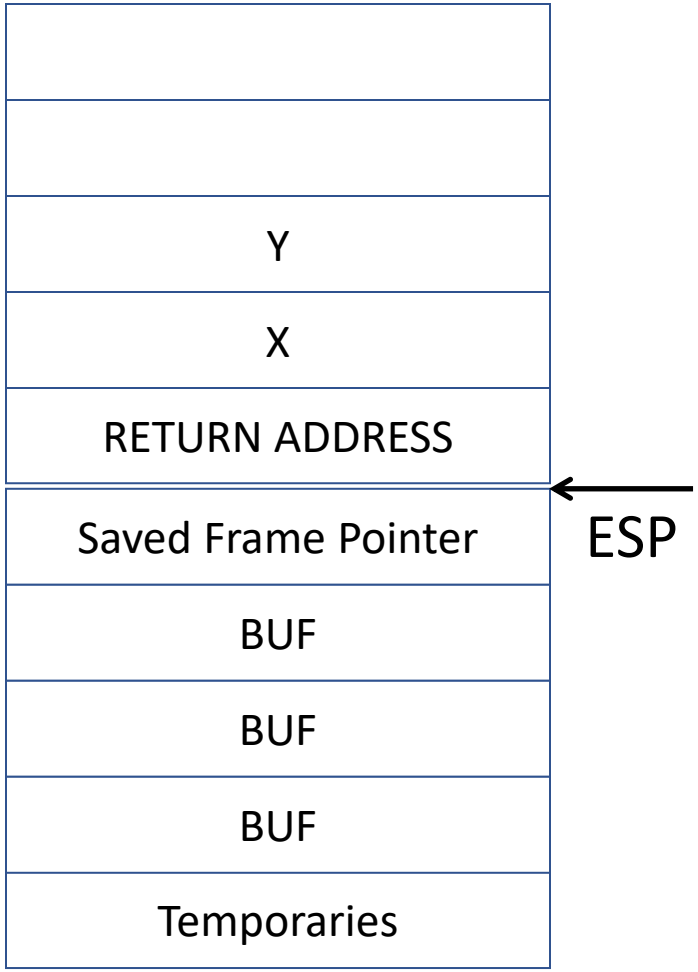
Basics:
The OS Model

Basics:
The OS Model

- Ring-0, runs on behalf of every process
  - E.g. 0x30000000 on 32-bit Linux is kernel's VA space
- Context Switches
  - On an interrupt, CPU switches control to ring-0
  - Ring-0 (OS) sets CR3 to another process, then **`iret`**
- The OS:
  - Identifies processes by CR3, maps to PTs.
  - On page faults, check if faulting VA is allocated
    - If not, raises a **segmentation fault**
    - If mapped, but not in RAM, OS swaps in from disk
      - Demand paging
    - If within the kernel's address range, then **kernel panic**!

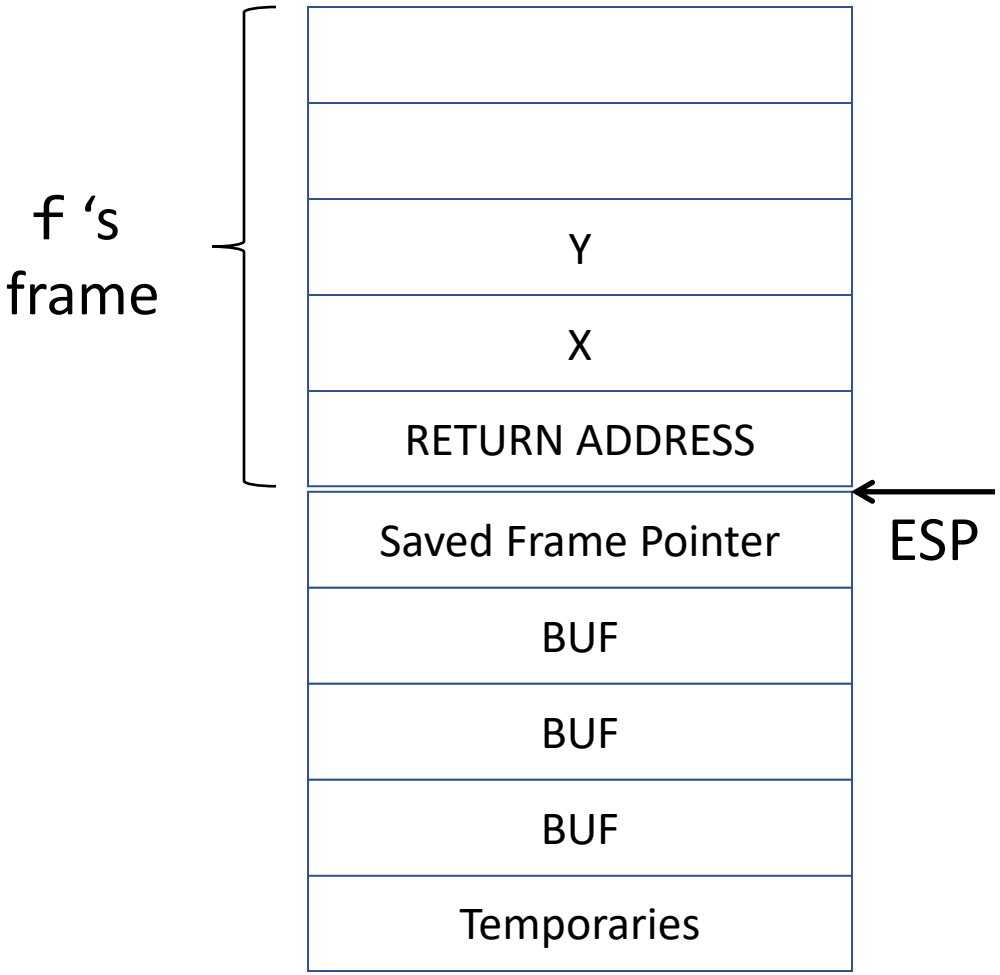
Stack Frames

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



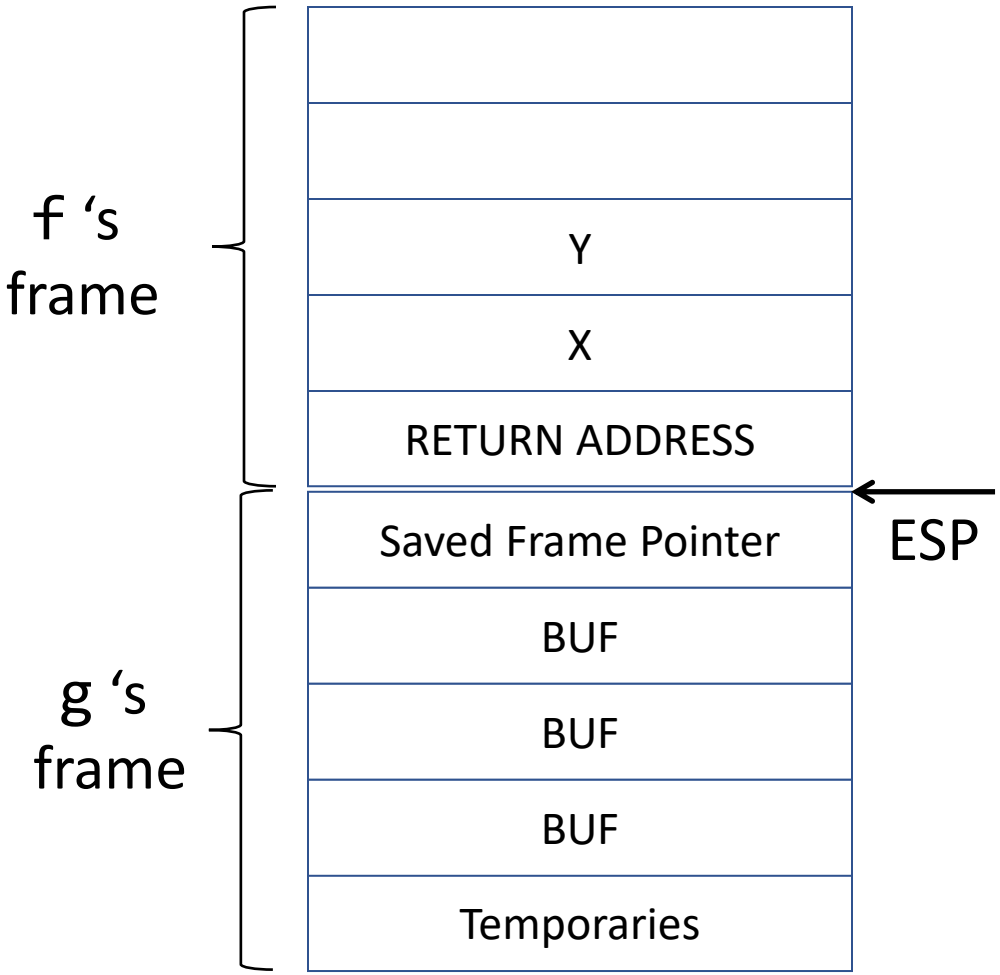
Stack Frames

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



Stack Frames

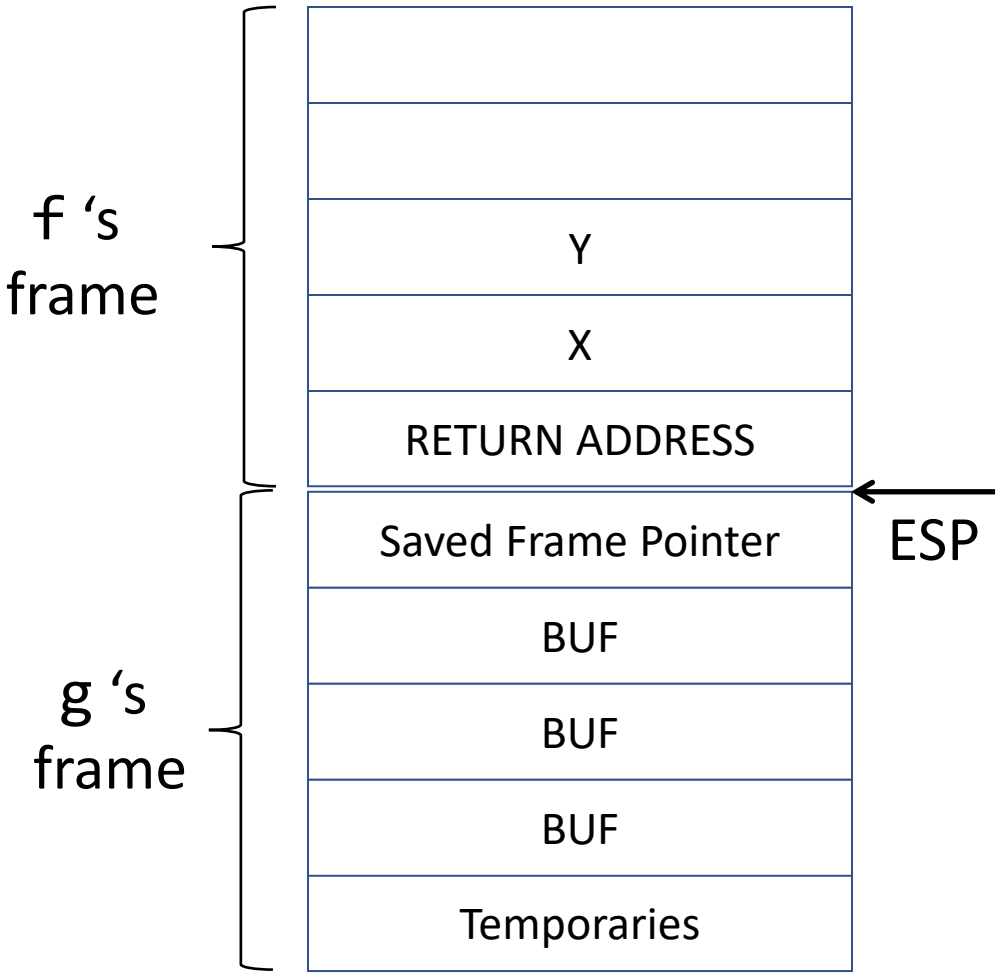
```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



Stack Frames

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```

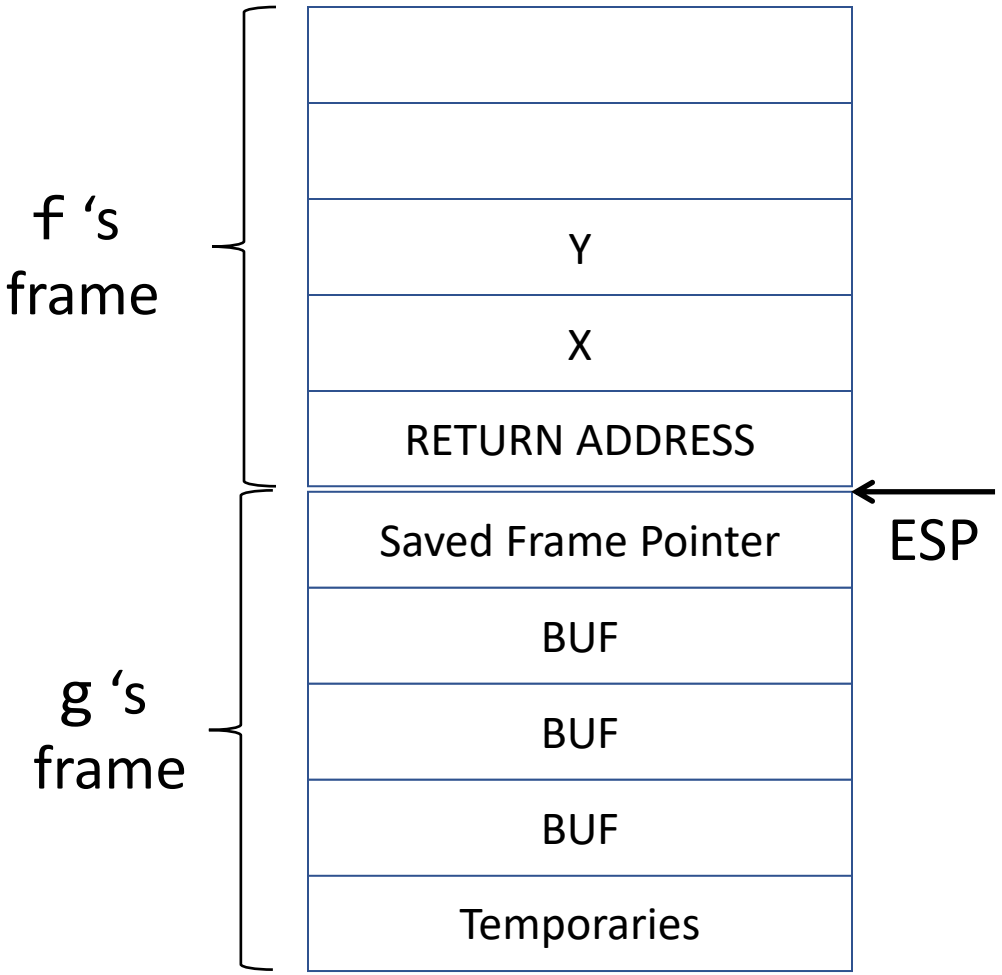
```
.g

push ebp

…
call scanf
…
pop ebp
ret
```



Stack Frames

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```

```
.g

push ebp

…
call scanf
…
pop ebp
ret
```

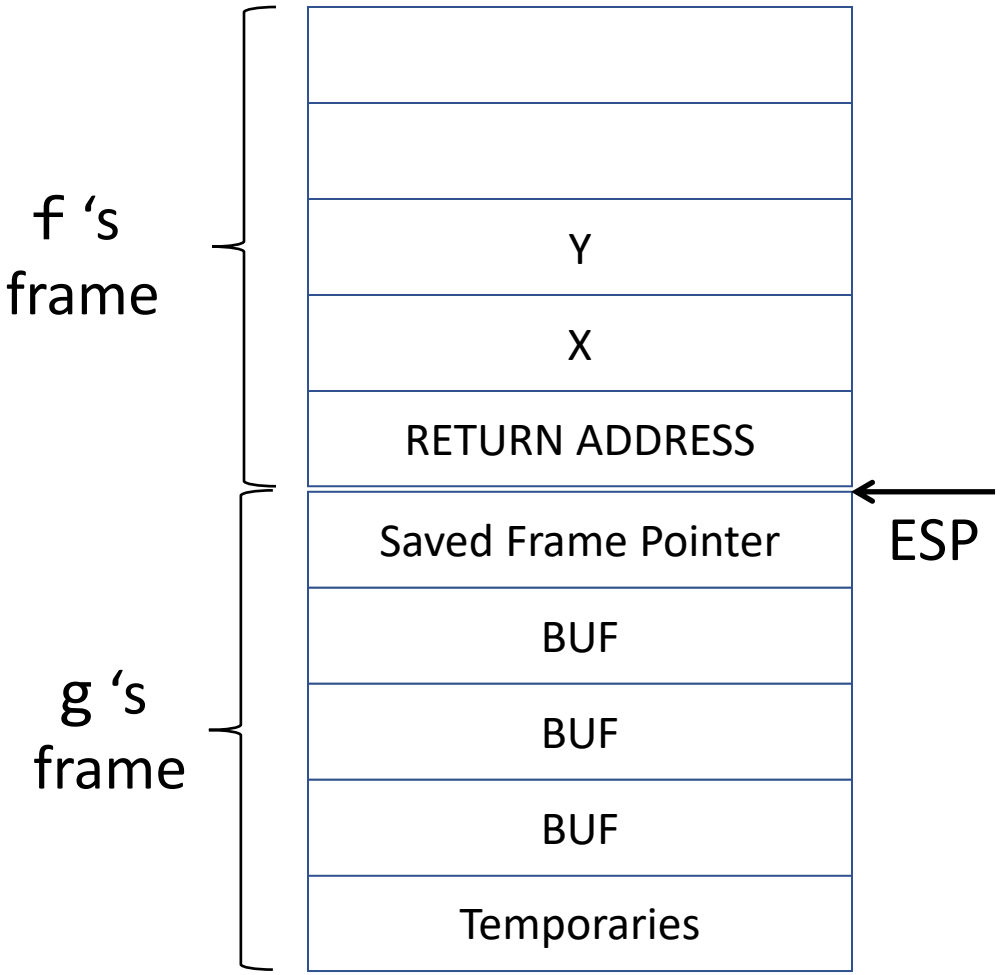


Stack Frames

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```

```
.g

push ebp

…
call scanf
…
pop ebp
ret
```

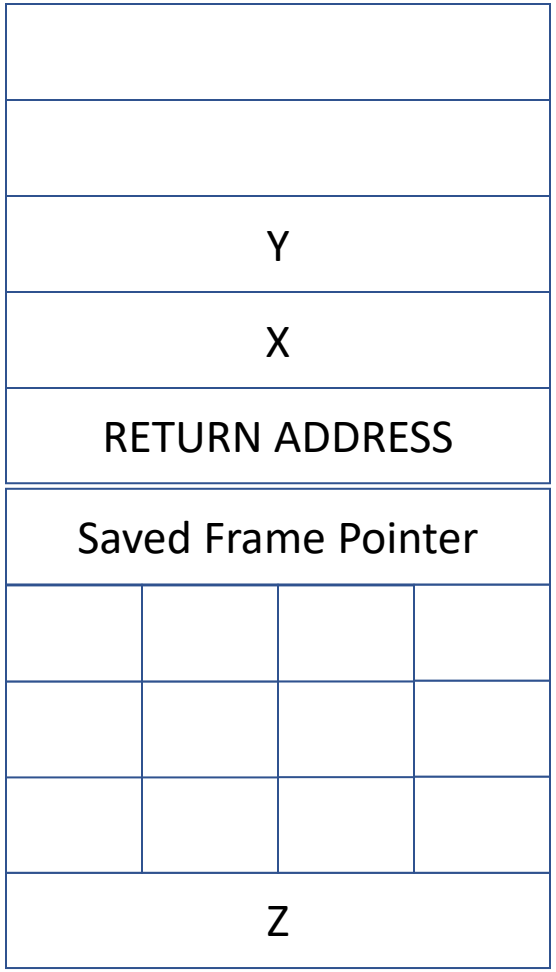
What address will it return to?



Spatial Memory Errors: Buffer Overflows

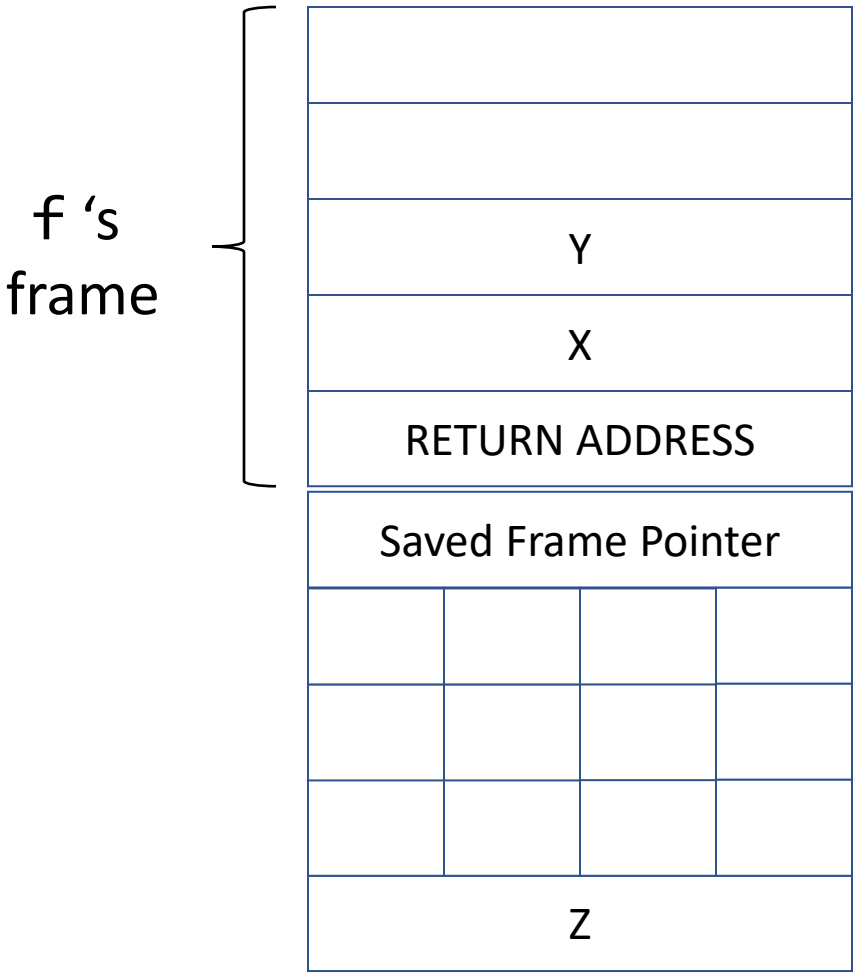
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



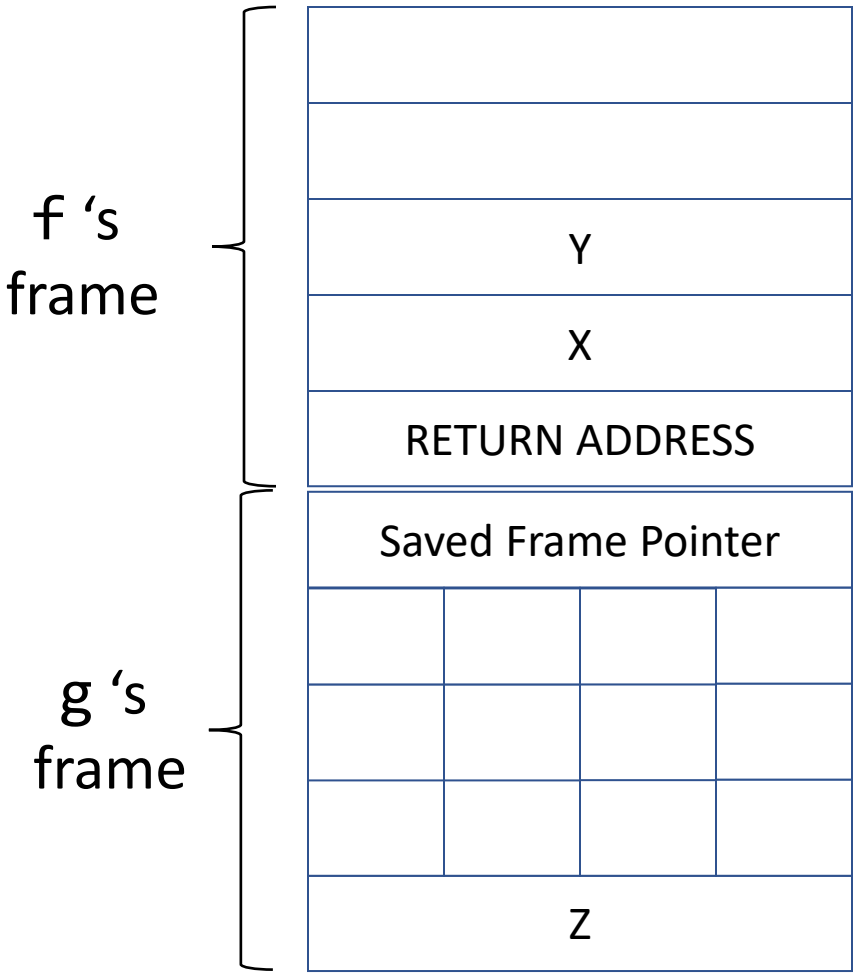
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



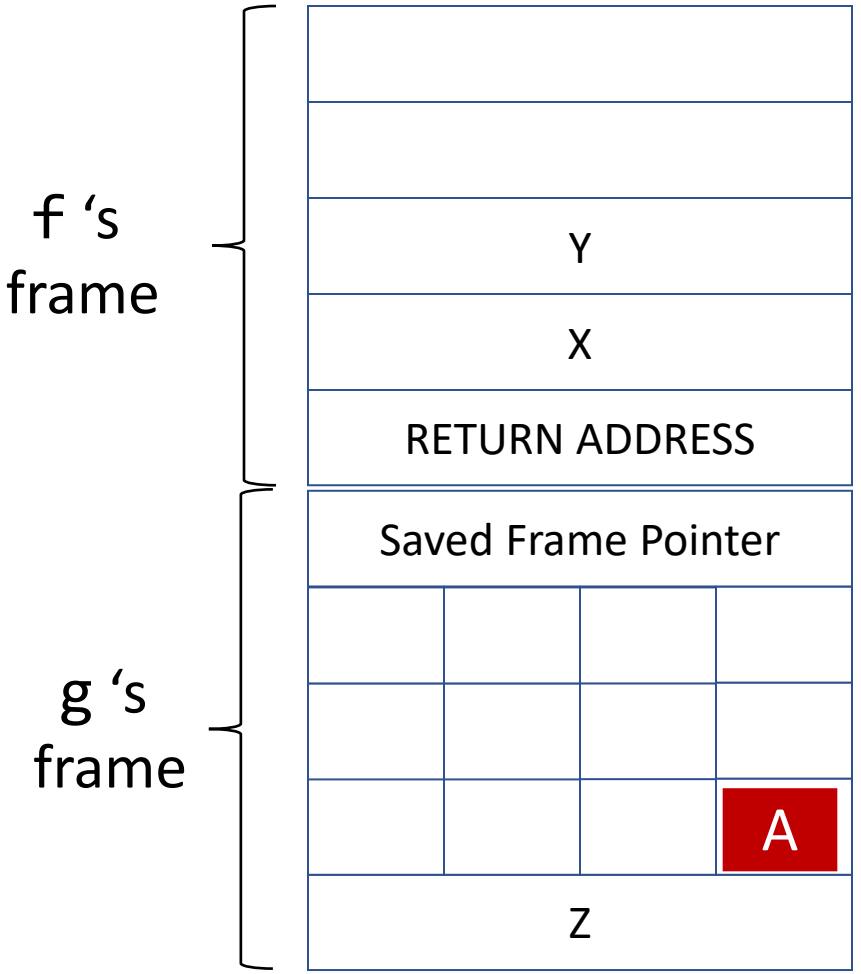
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



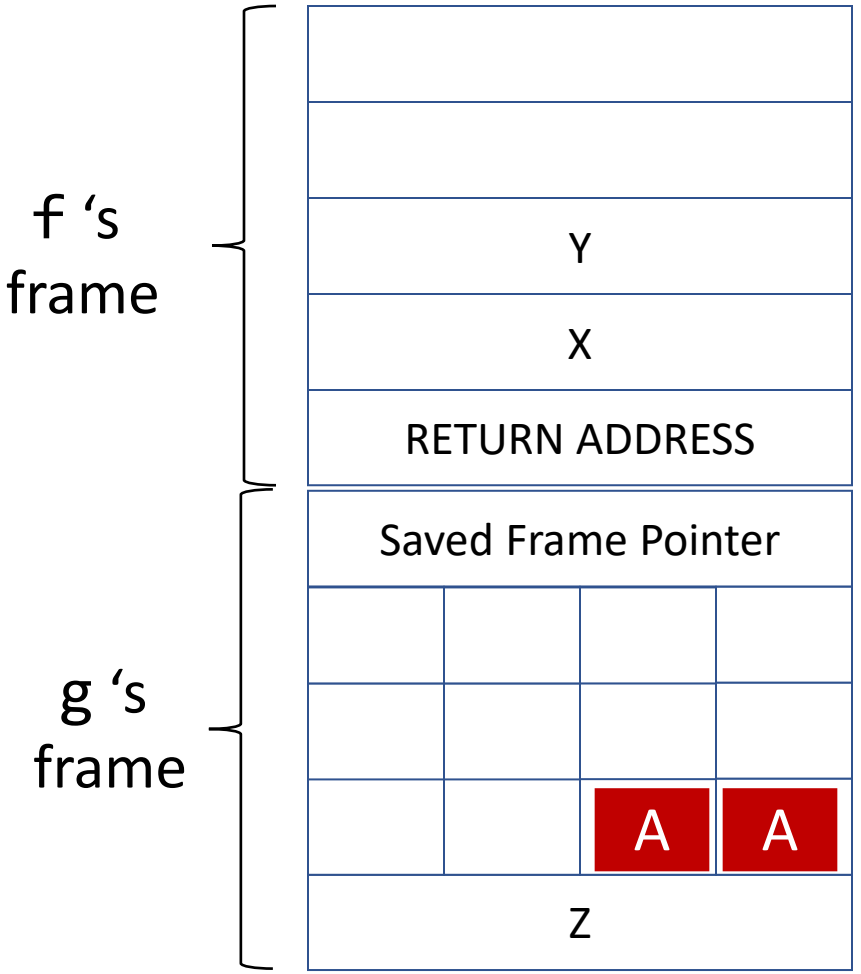
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



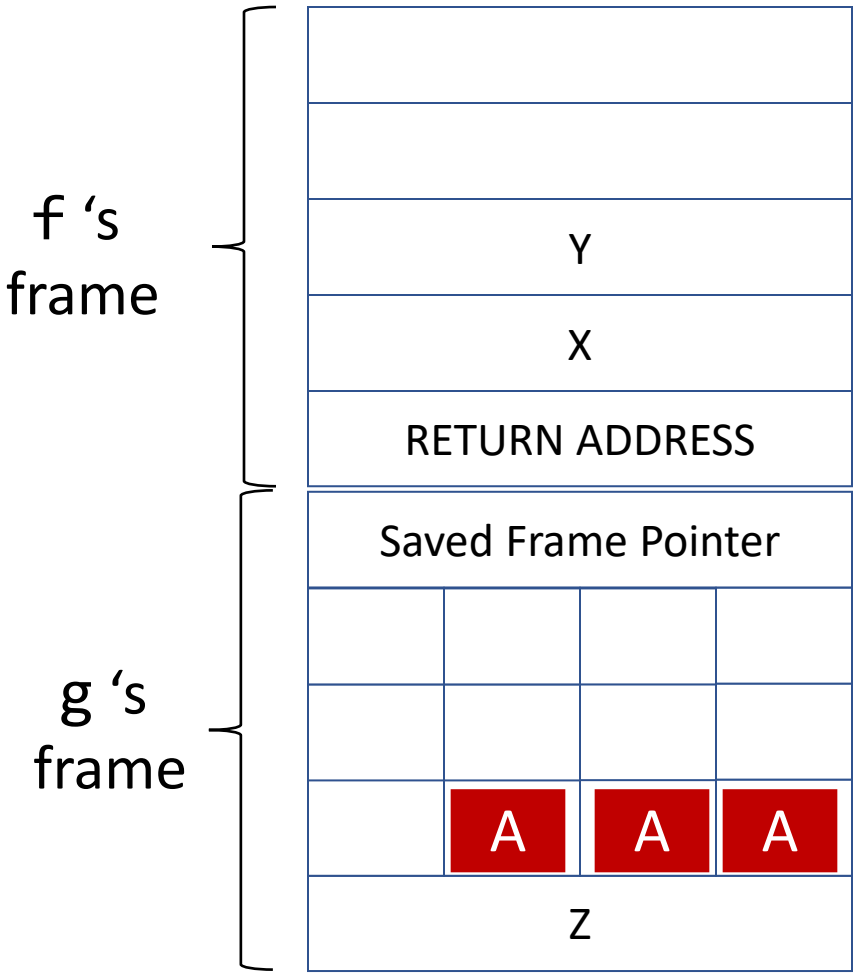
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



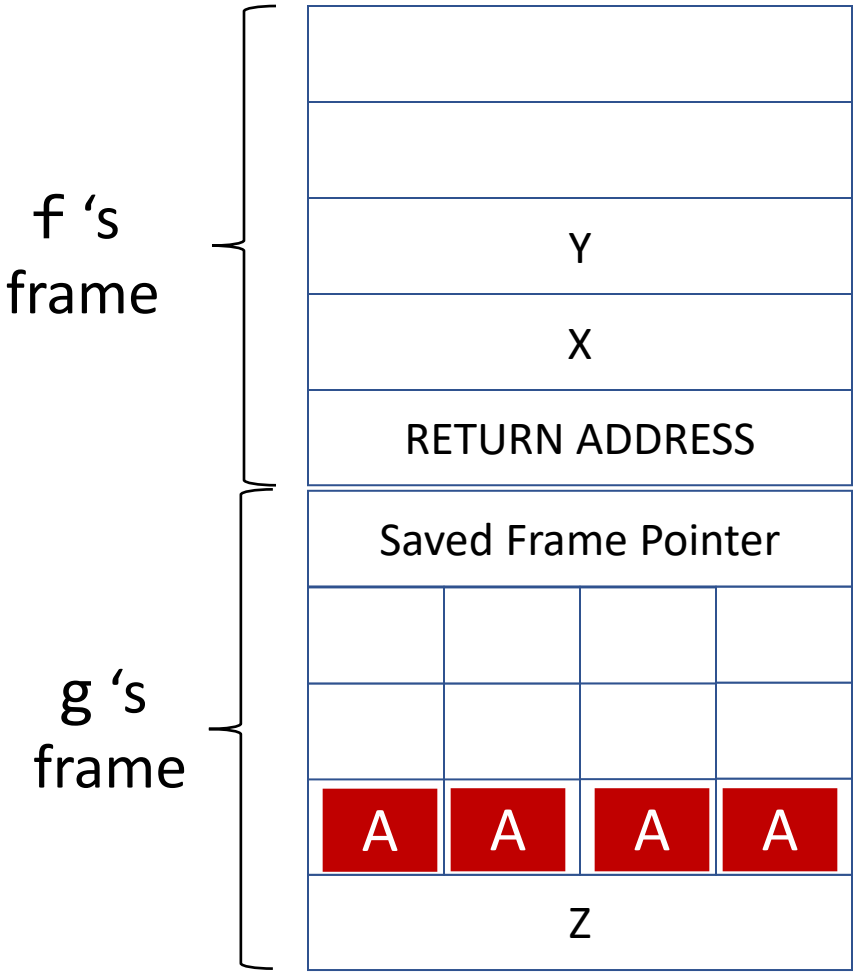
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



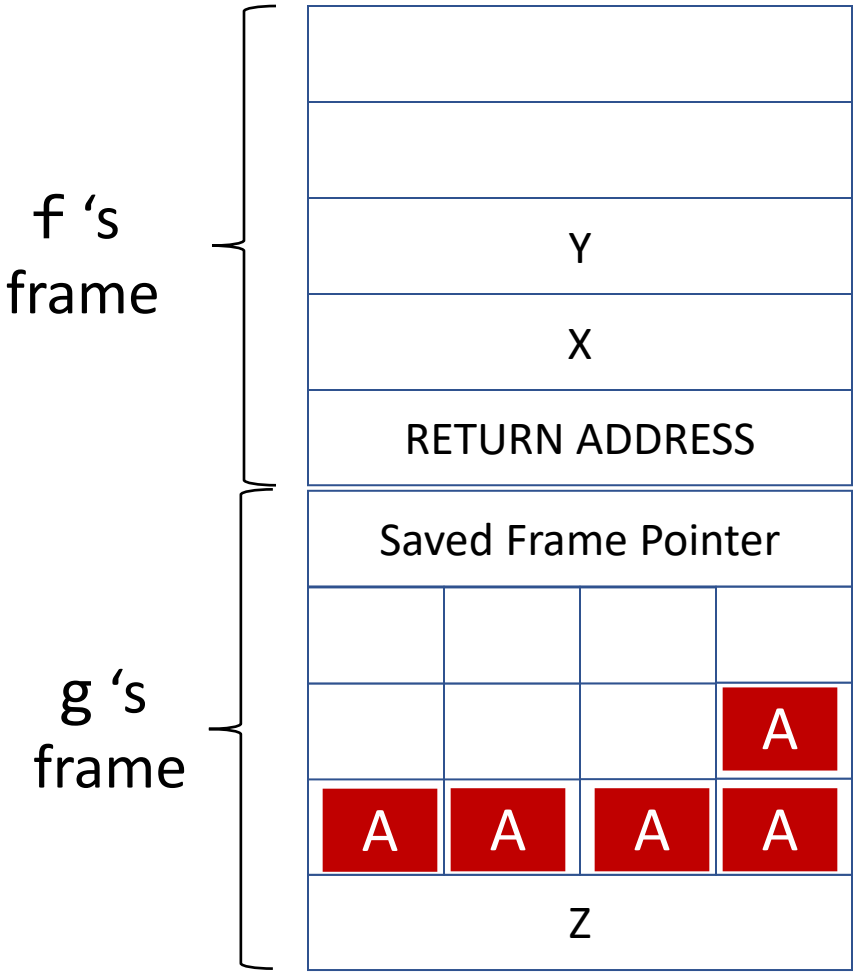
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



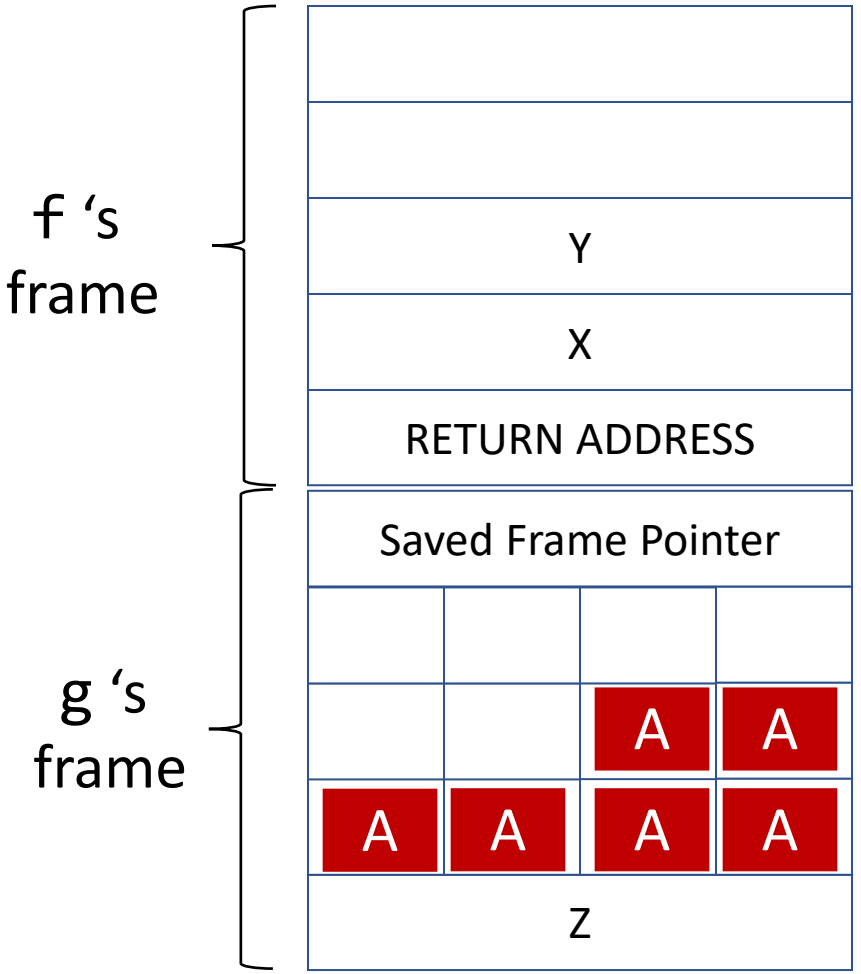
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



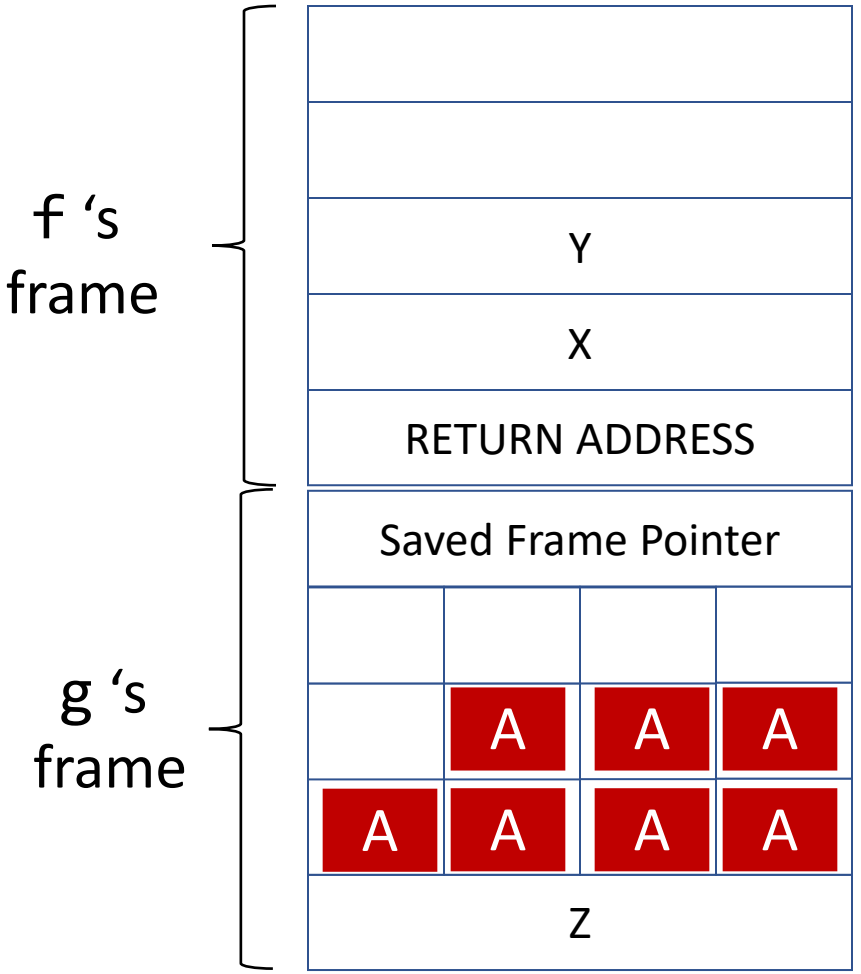
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



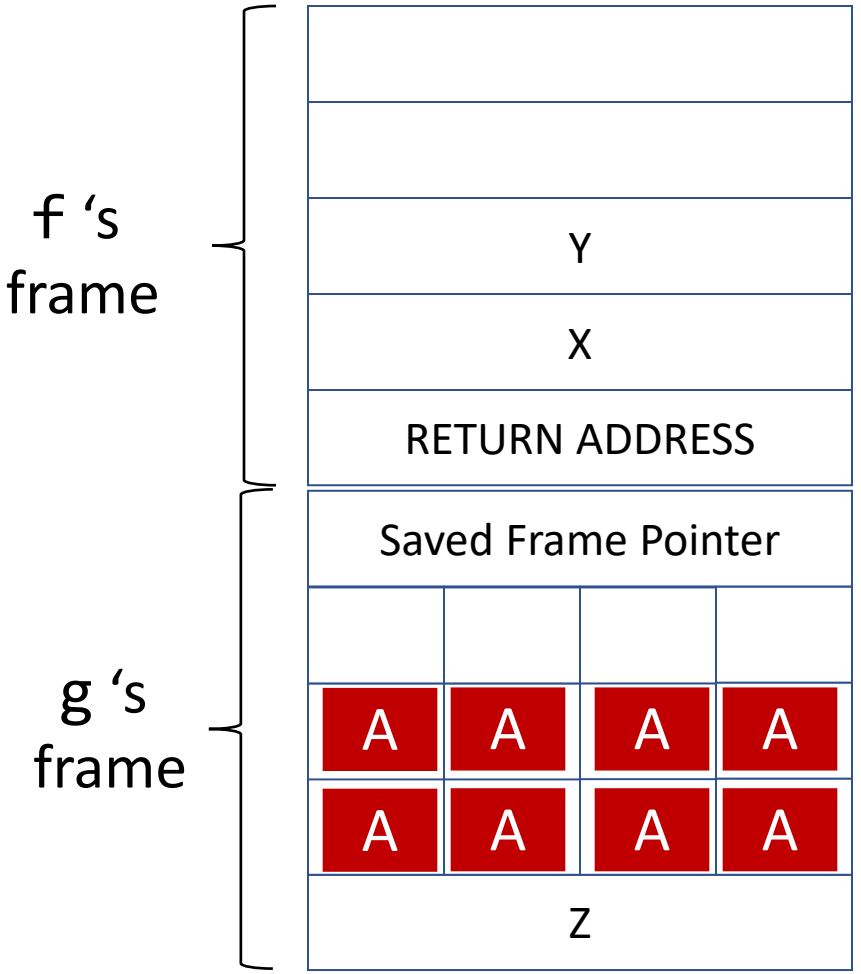
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



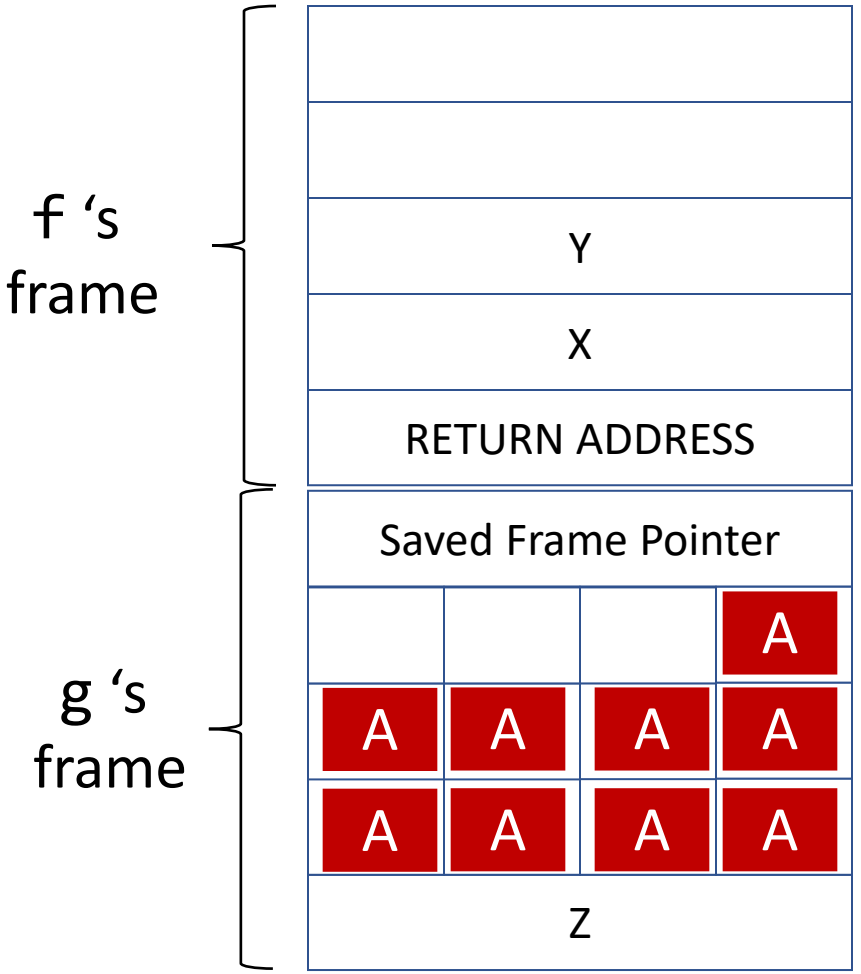
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



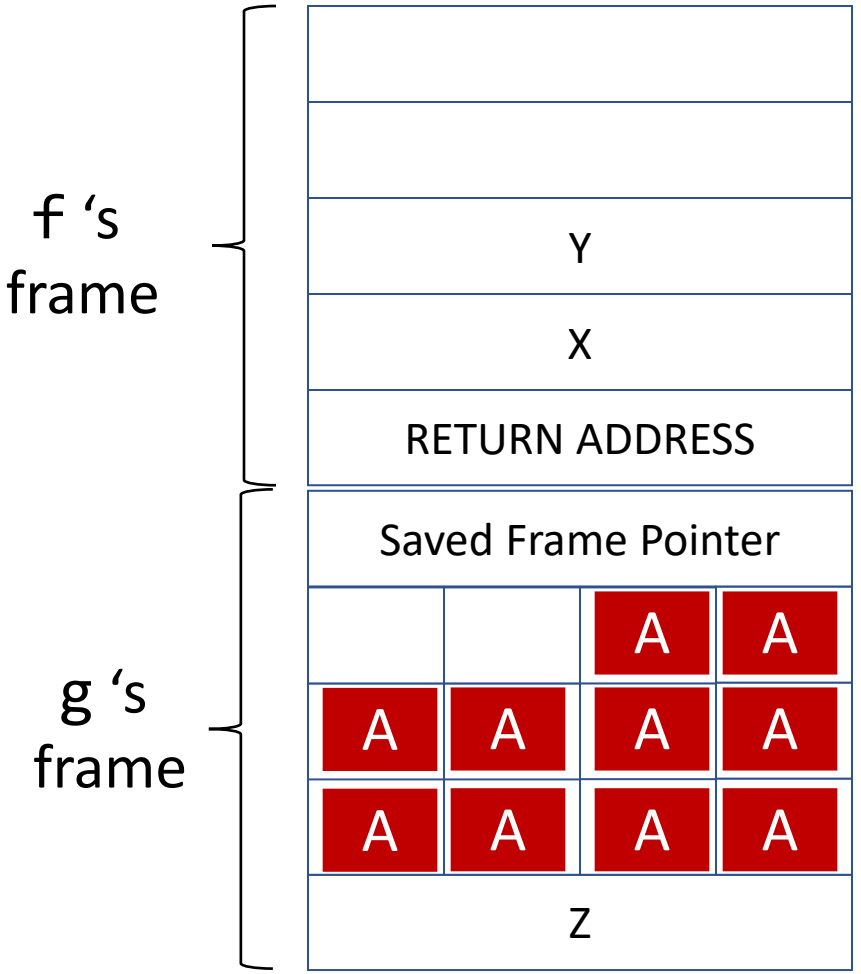
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



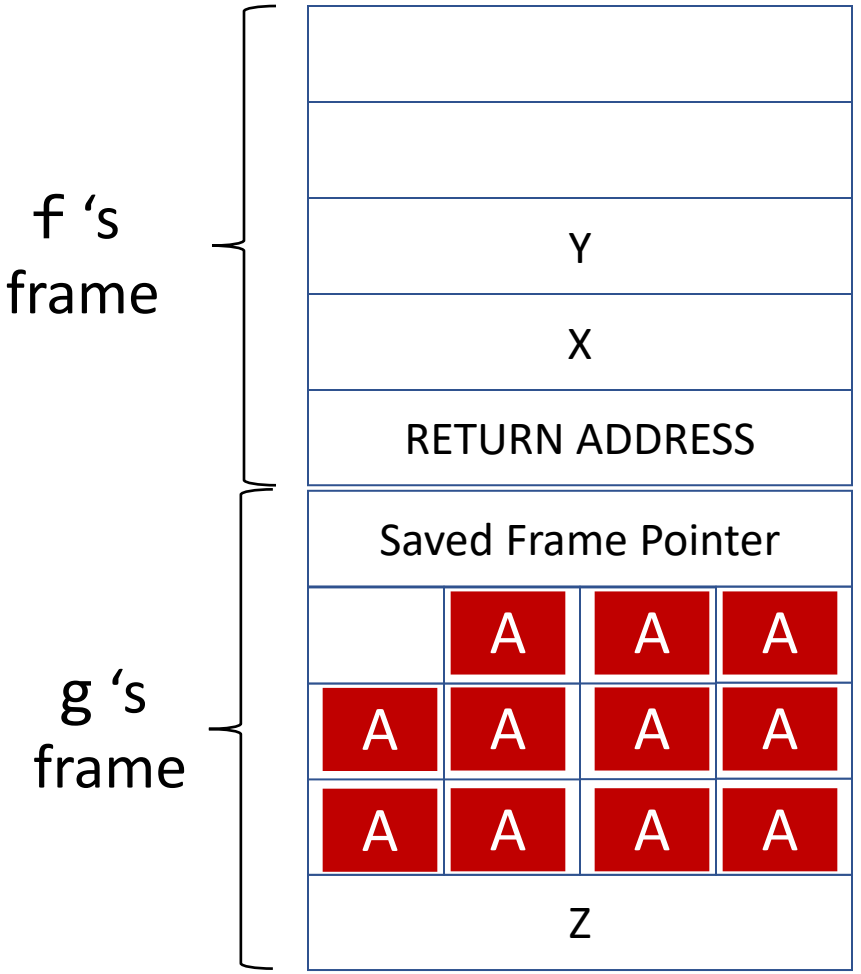
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



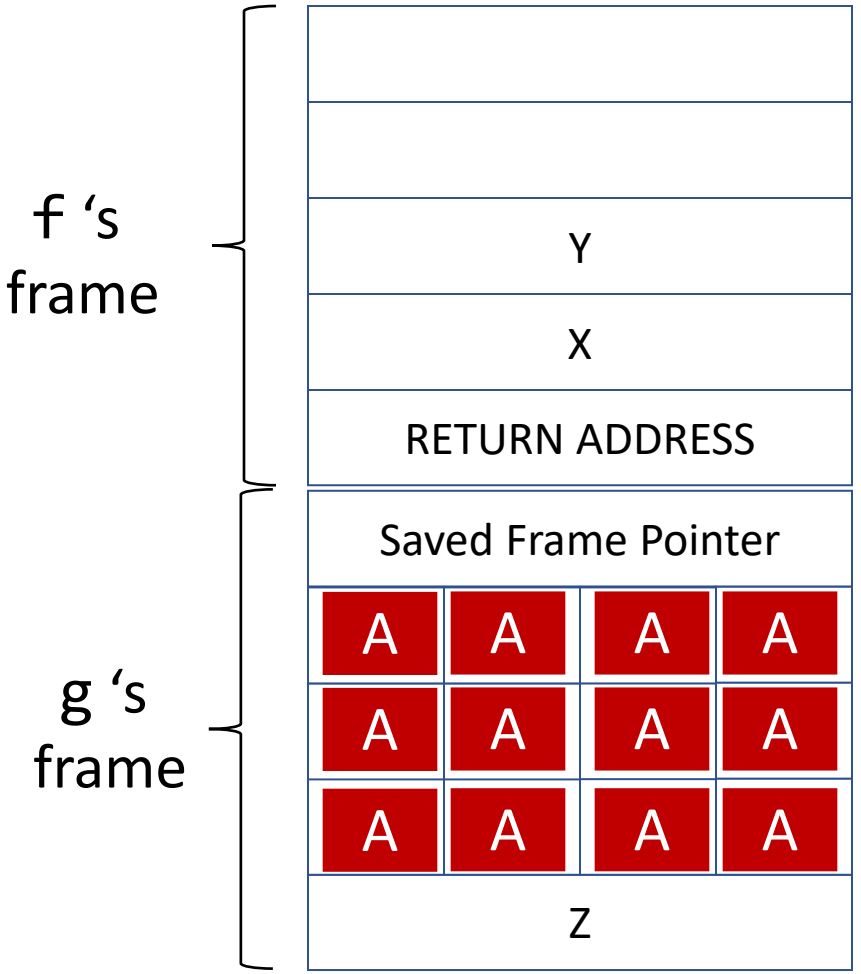
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



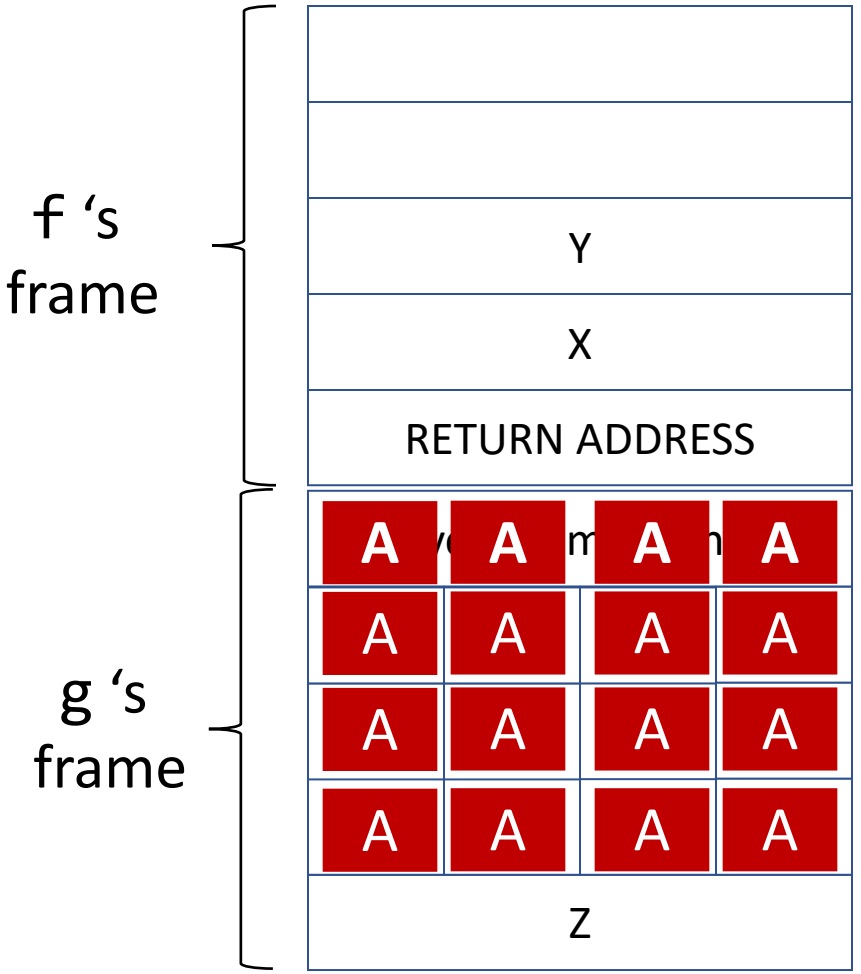
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



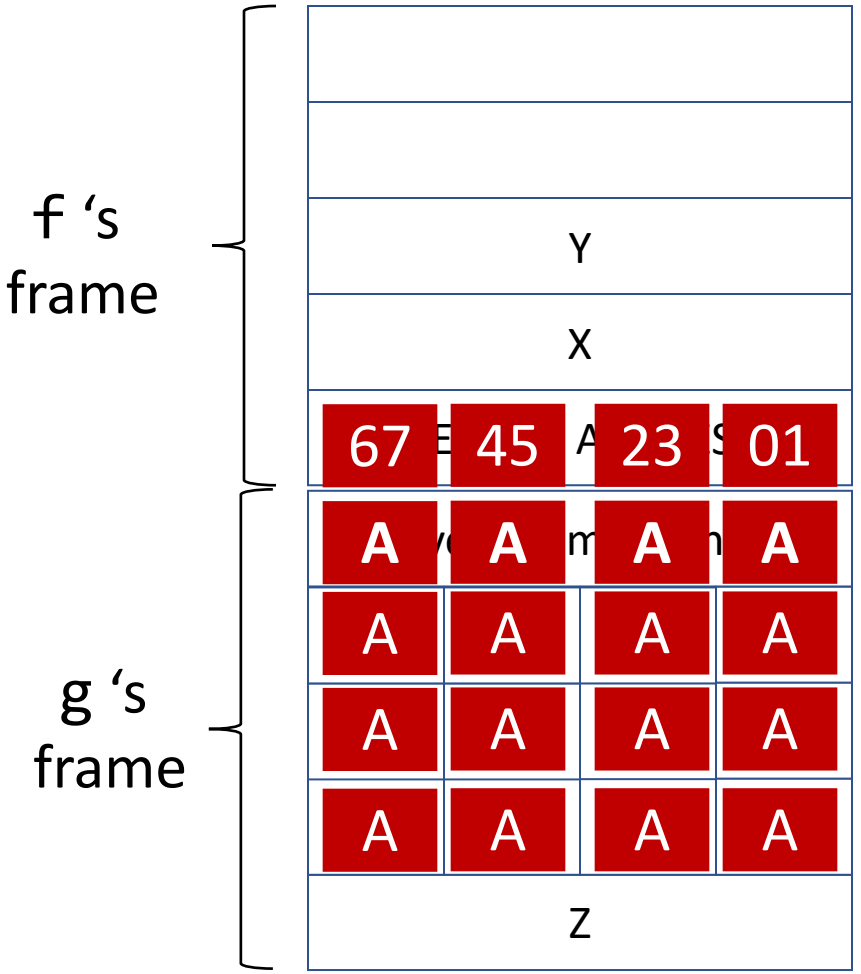
Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



Buffer Overflows

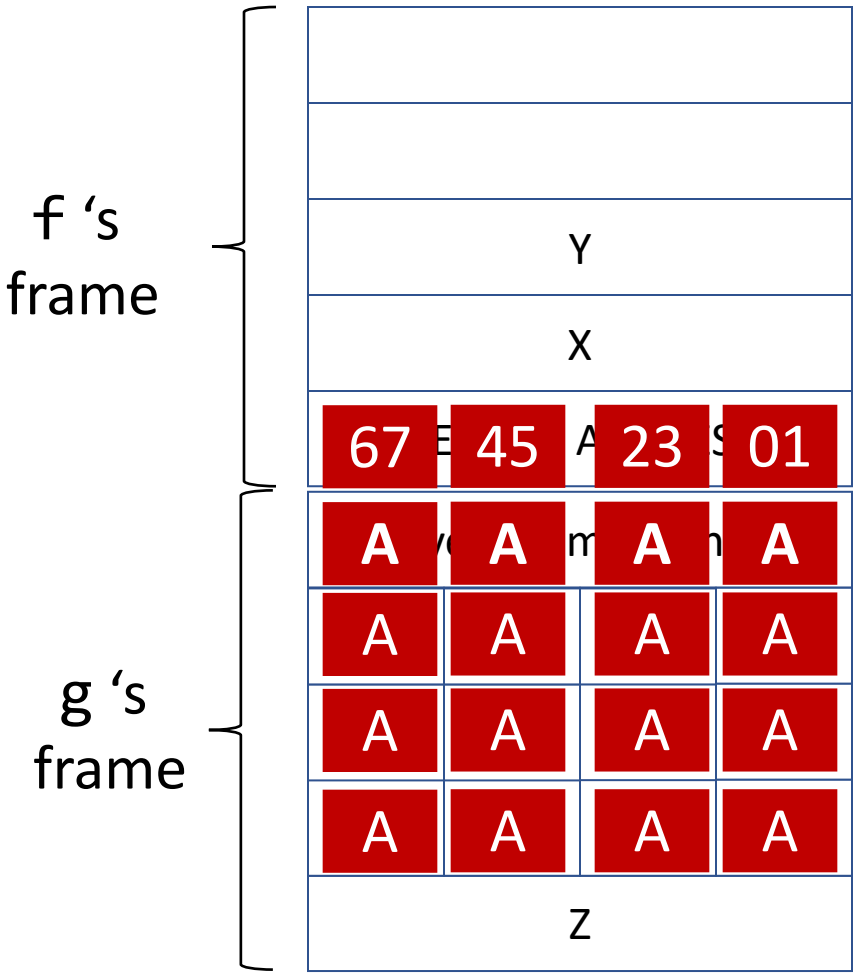
```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```



Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```

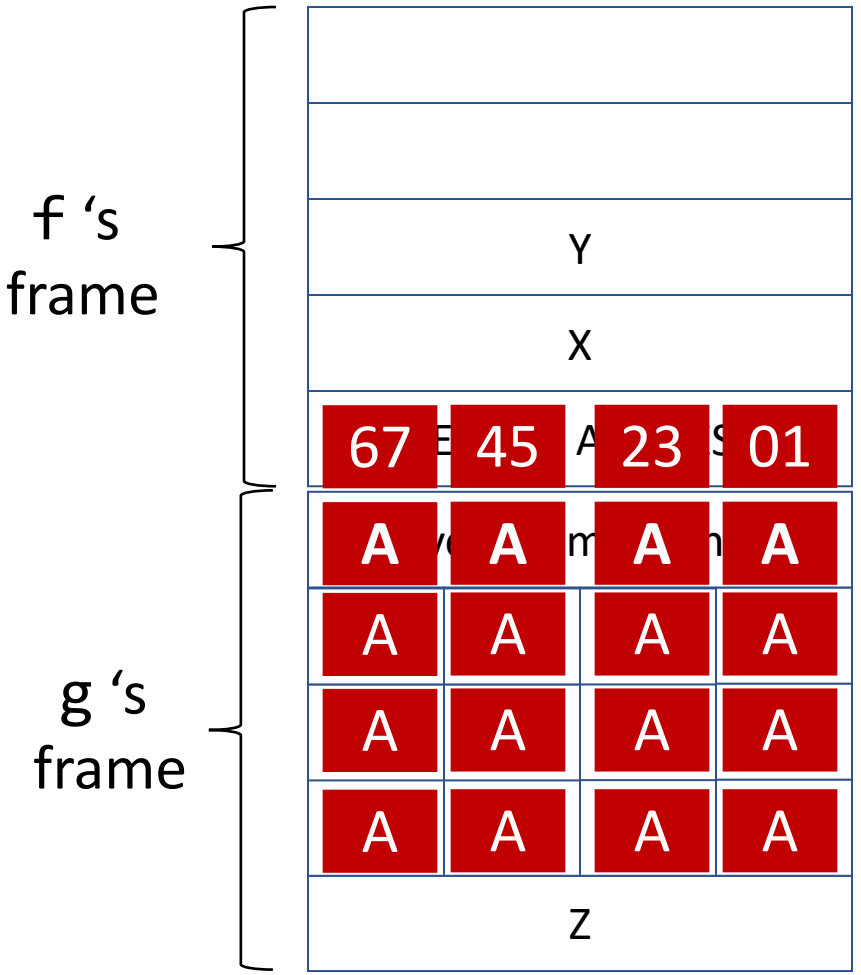
```
.g

push ebp

…
call scanf
…
pop ebp
ret
```



Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```

```
.g

push ebp

…
call scanf
…
pop ebp
ret
```



➡ What address will it return to?

Is this code free from buffer overflow?

```
void bad_function(char *input)
{
      char dest_buffer[32];
      char input_len = strlen(input);

      if (input_len < 32)
      {
            strcpy(dest_buffer, input);
            printf("The first command line argument is %s.\n", dest_buffer);
      }
      else
      {
            printf("Error - input is too long for buffer.\n");
      }
```

Type Errors:
Integer Overflow

Integer Overflow

```
void bad_function(char *input)
{
      char dest_buffer[32];
      char input_len = strlen(input);

      if (input_len < 32)
      {
            strcpy(dest_buffer, input);
            printf("The first command line argument is %s.\n", dest_buffer);
      }
      else
      {
            printf("Error - input is too long for buffer.\n");
     }
```

Integer Overflow

```
void bad_function(char *input)
{
      char dest_buffer[32];
      char input_len = strlen(input);

      if (input_len < 32)
      {
            strcpy(dest_buffer, input);
            printf("The first command line argument is %s.\n", dest_buffer);
      }
      else
      {
            printf("Error - input is too long for buffer.\n");
      }
```

Integer Overflow

```
void bad_function(char *input)
{
      char dest_buffer[32];
      char input_len = strlen(input);   // Range? [0, 255]
                                                Or [-128, 127]
      if (input_len < 32)
      {
            strcpy(dest_buffer, input);
            printf("The first command line argument is %s.\n", dest_buffer);
      }
      else
      {
            printf("Error - input is too long for buffer.\n");
      }
```

Integer Overflow

```
void bad_function(char *input)
{
      char dest_buffer[32];
      char input_len = strlen(input);   // Range? [0, 255]
                                              Or [-128, 127]
      if (input_len < 32)
      {
            strcpy(dest_buffer, input);
            printf("The first command line argument is %s.\n", dest_buffer);
      }
      else
      {
            printf("Error - input is too long for buffer.\n");
      }
```

-100

Why Does Integer Overflow Occur?

- Hardware: Arithmetic doesn't distinguish signed and unsigned integers --- works for both

Unsigned $\sum_{i=0}^{n-1} x_i 2^i$

$-x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$ Signed

$i=0$

An advantage of two's complement is that signed and unsigned addition can be performed using the same operation. The same is true for subtraction and multiplication. Historically, this was advantageous because fewer instructions needed to be implemented. Also, unlike the *ones' complement* and *sign-magnitude* representations, two's complement has only one representation for zero. A drawback of two's complement is that its range, $-2^{n-1} \ldots 2^{n-1} - 1$, is asymmetric. Thus, there is a representable value, -2^{n-1}, that does not have a representable additive inverse—a fact that programmers can and do forget.

When an n-bit addition or subtraction operation on unsigned or two's complement integers overflows, the result "wraps around," effectively subtracting 2^n from, or adding 2^n to, the true mathematical result. Equivalently, the result can be considered to occupy $n+1$ bits; the lower n bits are placed into the result register and the highest-order bit is placed into the processor's carry flag.

Why Does Integer Overflow Occur?

- Type Promotions: add/sub signed/unsigned int?

3.2. The Usual Arithmetic Conversions

Most integer operators in C/C++ require that both operands have the same type and, moreover, that this type is not narrower than an `int`. The collection of rules that accomplishes this is called *the usual arithmetic conversions*. The full set of rules encompasses both floating point and integer values; here we will discuss only the integer rules. First, both operands are *promoted*:

> If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. These are called the integer promotions. All other types are unchanged by the integer promotions.

If the promoted operands have the same type, the usual arithmetic conversions are finished. If the operands have different types, but either both are signed or both are unsigned, the narrower operand is converted to the type of the wider one.

If the operands have different types and one is signed and the other is unsigned, then the situation becomes slightly more involved. If the unsigned operand is narrower than the signed operand, and if the type of the signed operand can represent all values of the type of the unsigned operand, then the unsigned operand is converted to signed. Otherwise, the signed operand is converted to unsigned.

These rules can interact to produce counterintuitive results. Consider this function:

```
int compare (void) {
  long a = -1;
  unsigned b = 1;
  return a > b;
}
```

For a C/C++ implementation that defines `long` to be wider than `unsigned`, such as GCC for x86-64, this function returns zero. However, for an implementation that defines `long` and `unsigned` to have the same width, such as GCC for x86, this function returns one. The issue is that on x86-64, the comparison is between two signed integers, whereas on x86, the comparison is between two unsigned integers, one of which is very large. Some compilers are capable of warning about code like this.

C/C++ don't define exactly for all cases!

Where they do define, the behavior is implementation (size) specific to compilers, and varies by architecture!

Temporal Memory Errors: Use-after-free & Double Free

Heap Overflow

```
int g(int x, int y) {
  char* buf;

  buf = malloc (50);
  scanf(“%s”, buf);

  free (buf);
}
```

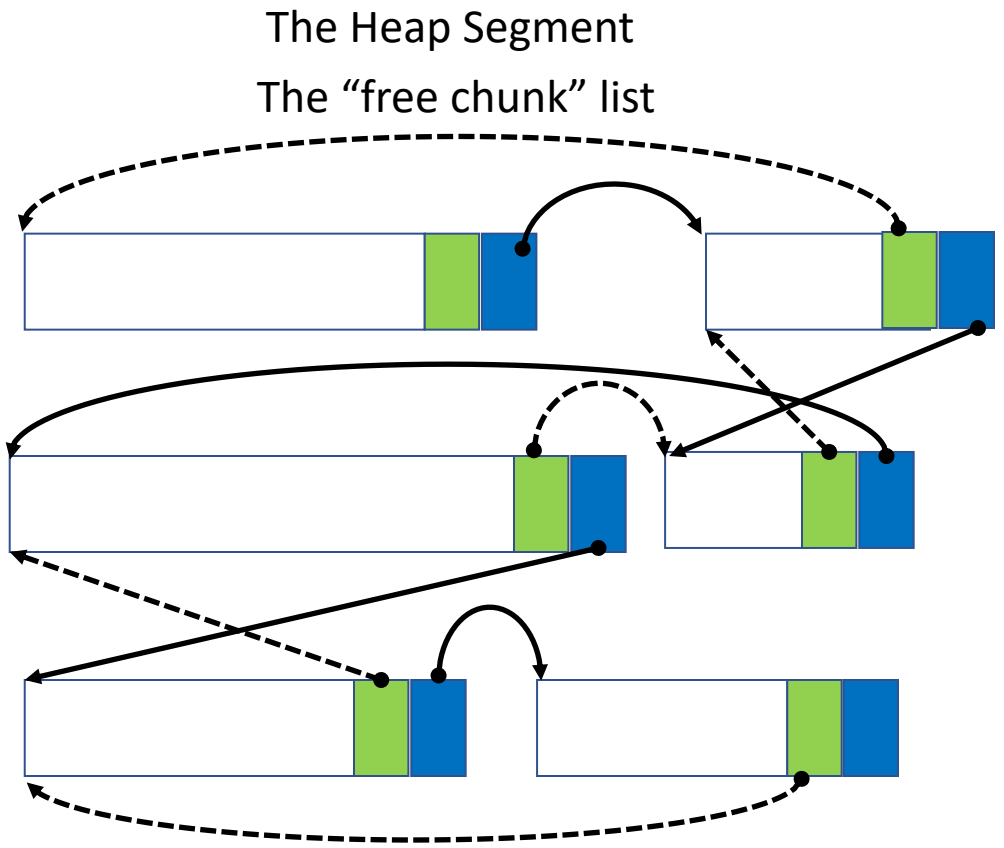
Heap Overflow

```
int g(int x, int y) {
  char* buf;

  buf = malloc (50);
  scanf(“%s”, buf);

  free (buf);
}
```

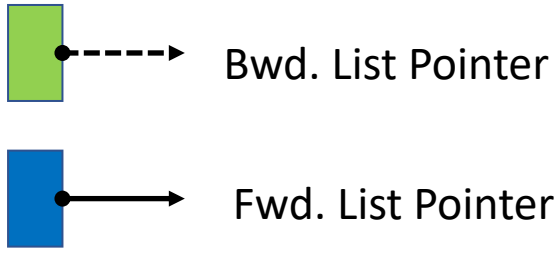
- Usually, the C library manages memory allocations
- Stores each allocated block in a linked list
- Can re-allocate a previously freed blocks
- Requests the OS for pages when given pages are full

Heap Overflow

```
int g(int x, int y) {
  char* buf;

  buf = malloc (50);
  scanf("%s", buf);

  free (buf);
}
```

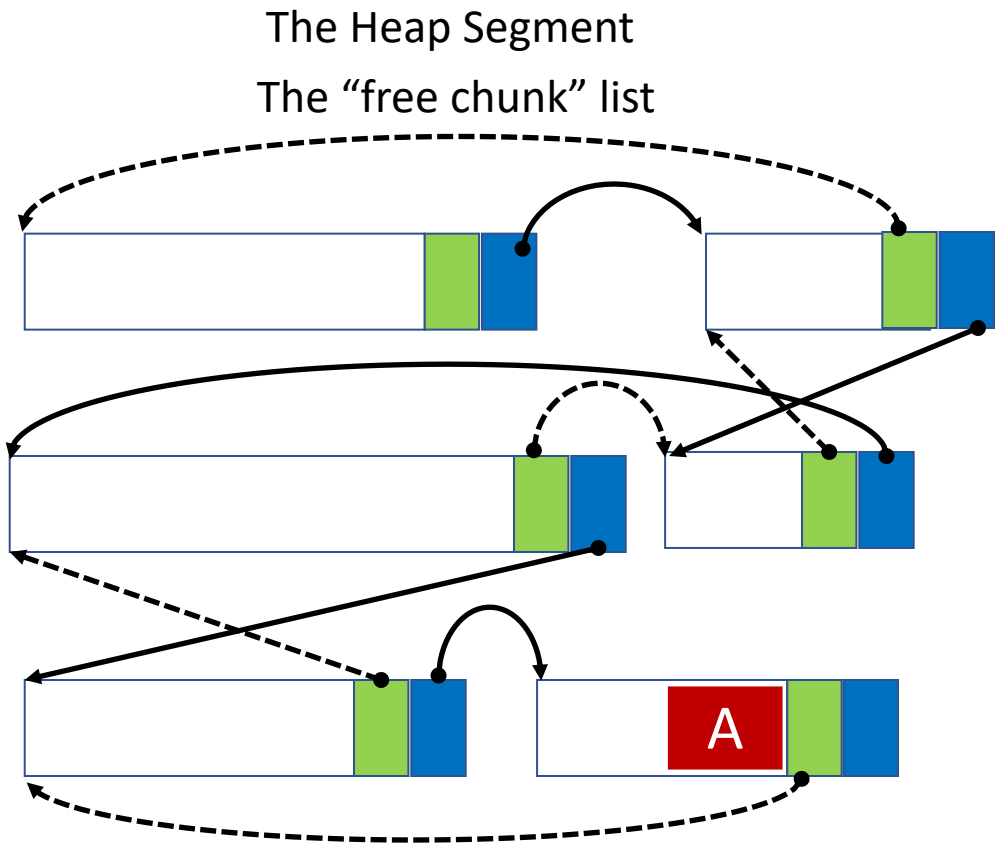


- Usually, the C library manages memory allocations
- Stores each allocated block in a linked list
- Can re-allocate a previously freed blocks
- Requests the OS for pages when given pages are full

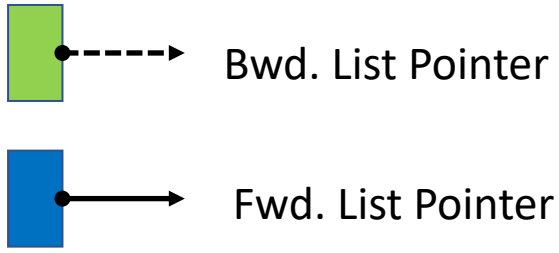


Heap Overflow

```
int g(int x, int y) {
  char* buf;

  buf = malloc (50);
  scanf("%s", buf);

  free (buf);
}
```

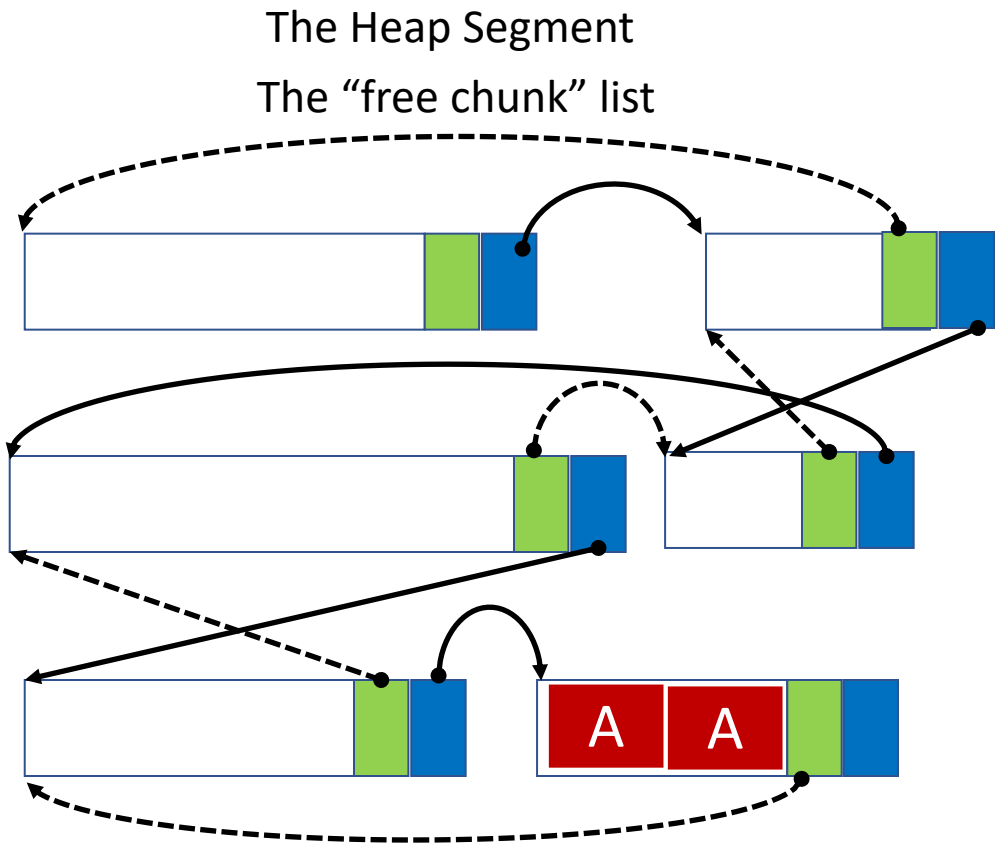


- Usually, the C library manages memory allocations
- Stores each allocated block in a linked list
- Can re-allocate a previously freed blocks
- Requests the OS for pages when given pages are full

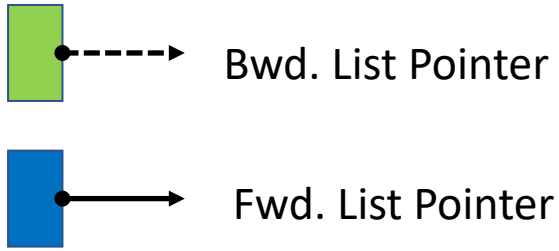


Heap Overflow

```
int g(int x, int y) {
  char* buf;

  buf = malloc (50);
  scanf("%s", buf);

  free (buf);
}
```

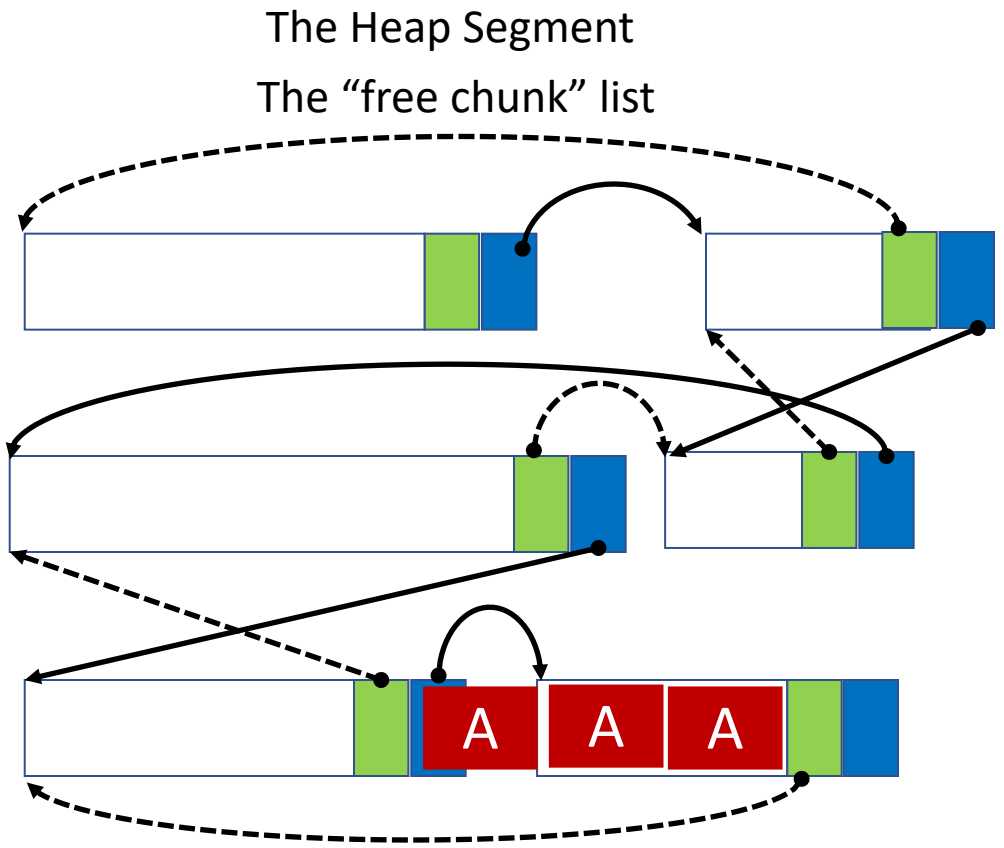


- Usually, the C library manages memory allocations
- Stores each allocated block in a linked list
- Can re-allocate a previously freed blocks
- Requests the OS for pages when given pages are full

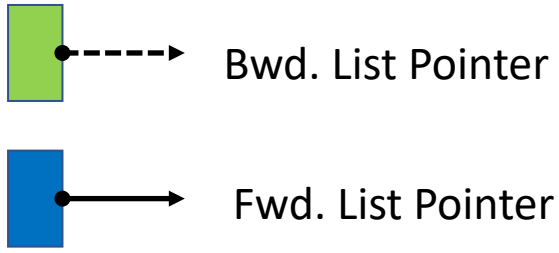


Heap Overflow

```
int g(int x, int y) {
  char* buf;

  buf = malloc (50);
  scanf("%s", buf);

  free (buf);
}
```

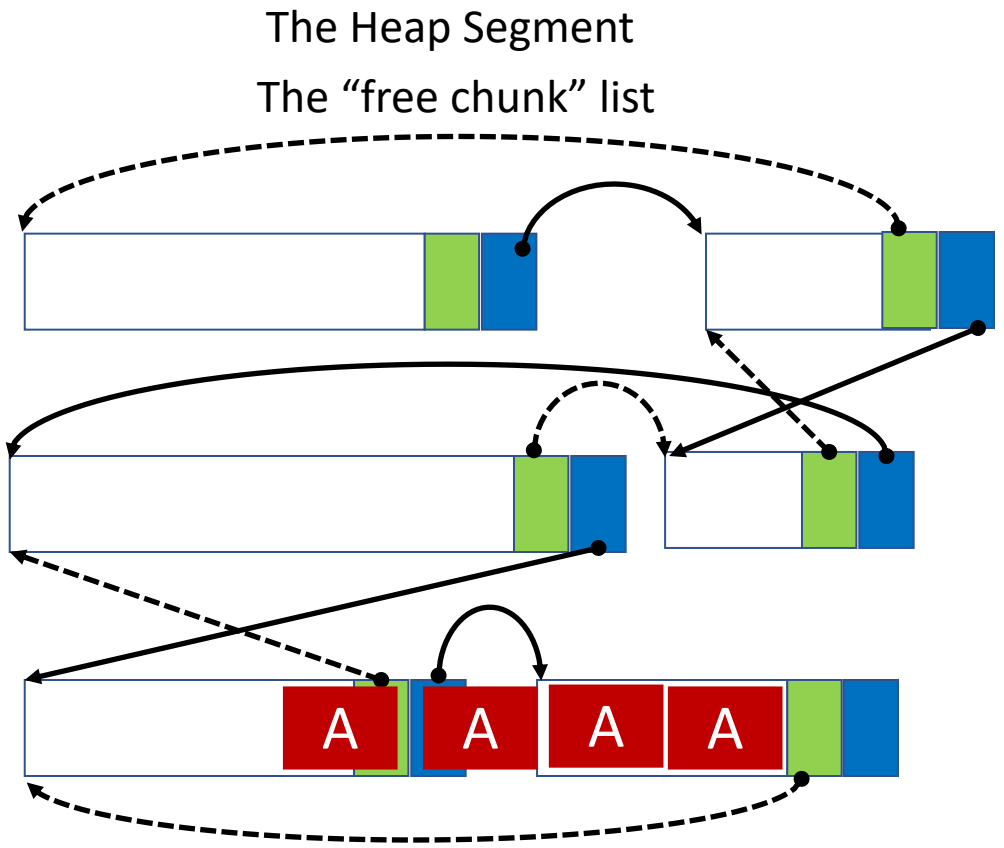


- Usually, the C library manages memory allocations
- Stores each allocated block in a linked list
- Can re-allocate a previously freed blocks
- Requests the OS for pages when given pages are full

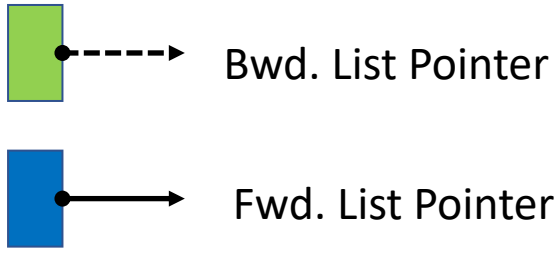


Heap Overflow

```
int g(int x, int y) {
  char* buf;

  buf = malloc (50);
  scanf("%s", buf);

  free (buf);
}
```



- Usually, the C library manages memory allocations
- Stores each allocated block in a linked list
- Can re-allocate a previously freed blocks
- Requests the OS for pages when given pages are full

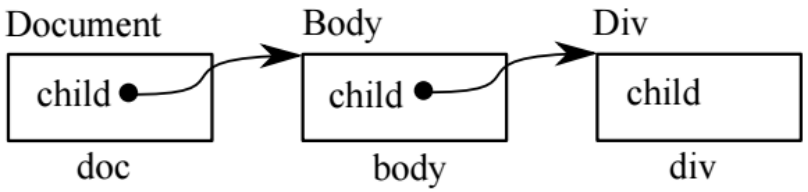


Use-after-free

```
1  class Div: Element;
2  class Body: Element;
3  class Document {
4    Element* child;
5  };
6
7  // (a) memory allocations
8  Document *doc = new Document();
9  Body *body = new Body();
10 Div *div = new Div();
11
12 // (b) using memory: propagating pointers
13 doc->child = body;
14 body->child = div;
15
16 // (c) memory free: doc->child is now dangled
17 delete body;
18
19
20
21
```

Use-after-free

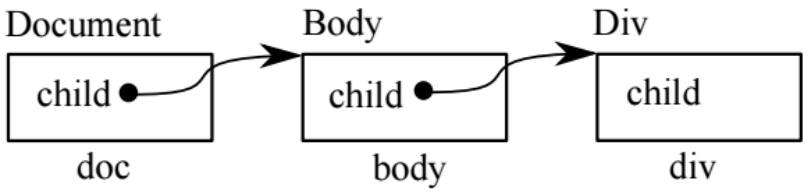
```
1  class Div: Element;
2  class Body: Element;
3  class Document {
4    Element* child;
5  };
6
7  // (a) memory allocations
8  Document *doc = new Document();
9  Body *body = new Body();
10 Div *div = new Div();
11
12 // (b) using memory: propagating pointers
13 doc->child = body;
14 body->child = div;
15
16 // (c) memory free: doc->child is now dangled
17 delete body;
18
19
20
21
```



(a-b) objects are allocated and linked

Use-after-free

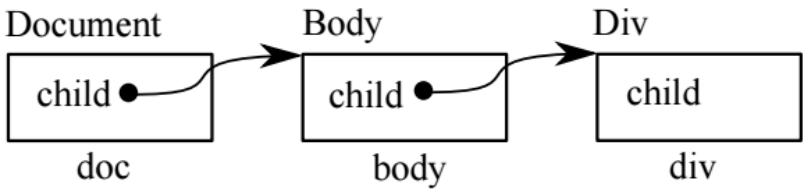
```
1  class Div: Element;
2  class Body: Element;
3  class Document {
4    Element* child;
5  };
6
7  // (a) memory allocations
8  Document *doc = new Document();
9  Body *body = new Body();
10 Div *div = new Div();
11
12 // (b) using memory: propagating pointers
13 doc->child = body;
14 body->child = div;
15
16 // (c) memory free: doc->child is now dangled
17 delete body;
18
19 // (d) use-after-free: dereference the dangled pointer
20 if (doc->child)
21     doc->child->getAlign();
```



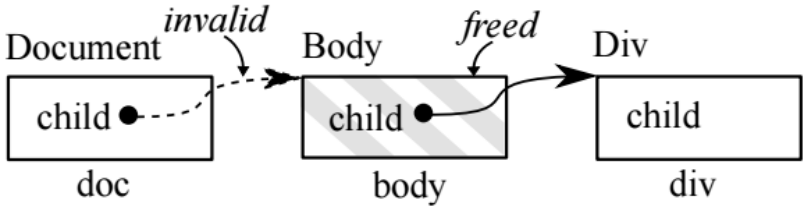
(a-b) objects are allocated and linked

Use-after-free

```
1  class Div: Element;
2  class Body: Element;
3  class Document {
4    Element* child;
5  };
6
7  // (a) memory allocations
8  Document *doc = new Document();
9  Body *body = new Body();
10 Div *div = new Div();
11
12 // (b) using memory: propagating pointers
13 doc->child = body;
14 body->child = div;
15
16 // (c) memory free: doc->child is now dangled
17 delete body;
18
19 // (d) use-after-free: dereference the dangled pointer
20 if (doc->child)
21     doc->child->getAlign();
```



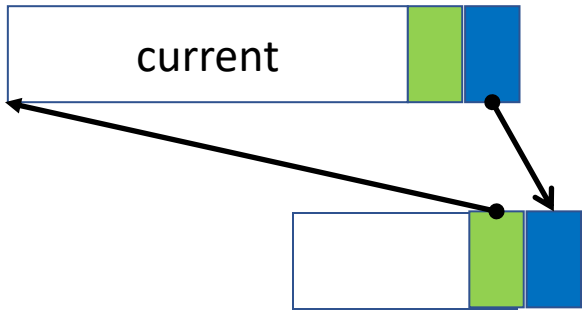
(a-b) objects are allocated and linked



(c-d) body is freed (so dangled), and doc reads the invliad memory

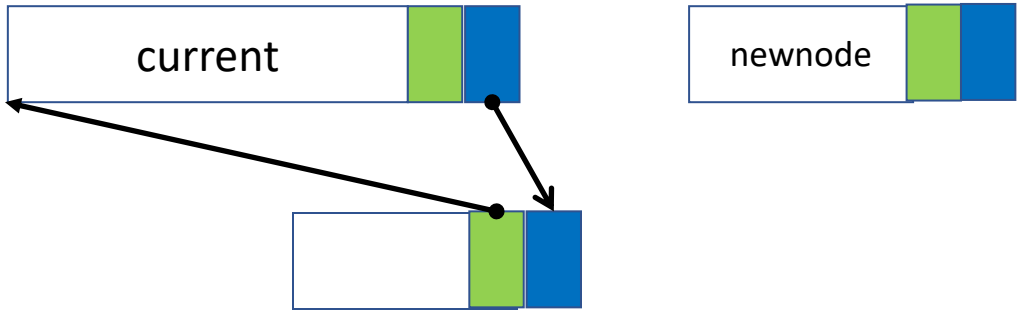
Double Free:
Setup (Benign Working)

```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
    newnode->next = current->next;
    if (current->next) current->next->prev = newnode;
    current->next = newnode;
    newnode->prev = current;
  }
```



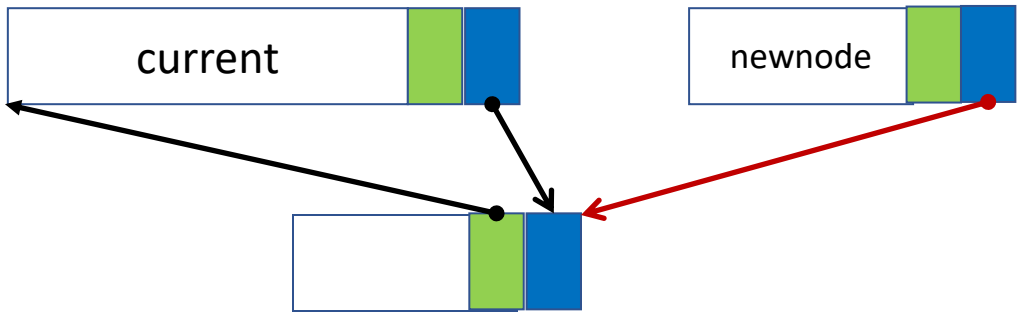
Double Free:
Setup (Benign Working)

```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
    newnode->next = current->next;
    if (current->next) current->next->prev = newnode;
    current->next = newnode;
    newnode->prev = current;
  }
```



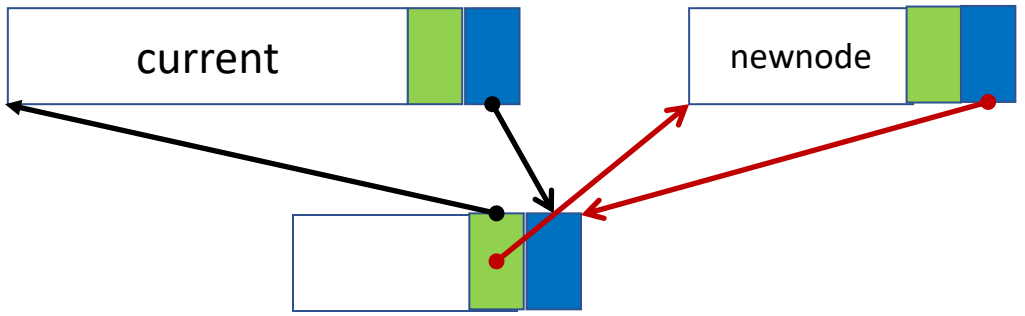
Double Free:
Setup (Benign Working)

```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
    newnode->next = current->next;
    if (current->next) current->next->prev = newnode;
    current->next = newnode;
    newnode->prev = current;
  }
```



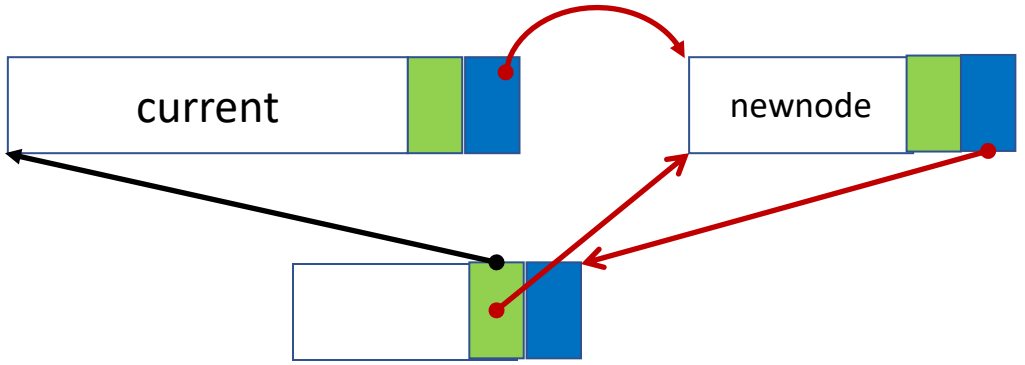
Double Free:
Setup (Benign Working)

```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
    newnode->next = current->next;
    if (current->next) current->next->prev = newnode;
    current->next = newnode;
    newnode->prev = current;
  }
```



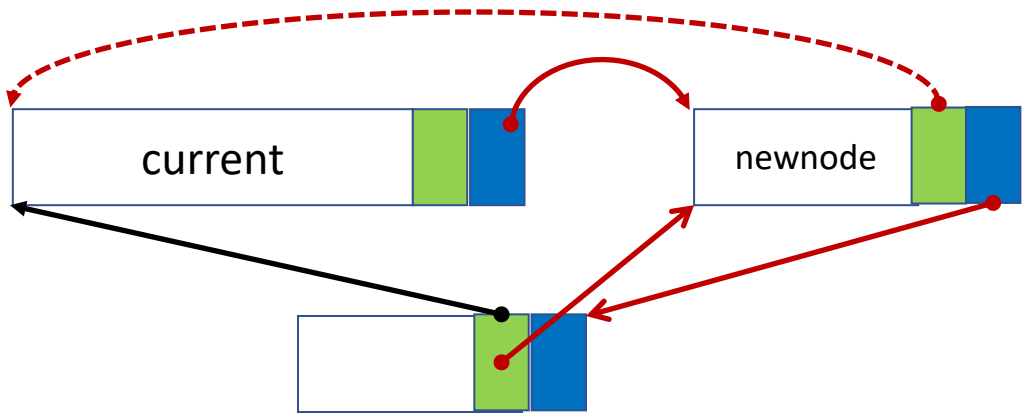
Double Free:
Setup (Benign Working)

```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
    newnode->next = current->next;
    if (current->next) current->next->prev = newnode;
    current->next = newnode;
    newnode->prev = current;
  }
```



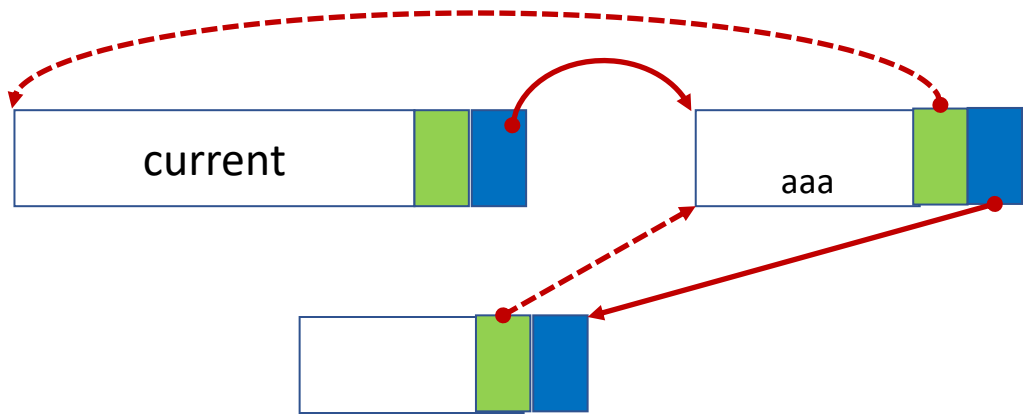
Double Free:
Setup (Benign Working)

```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
    newnode->next = current->next;
    if (current->next) current->next->prev = newnode;
    current->next = newnode;
    newnode->prev = current;
  }
```



Double Free:
The Write-Anywhere Exploit

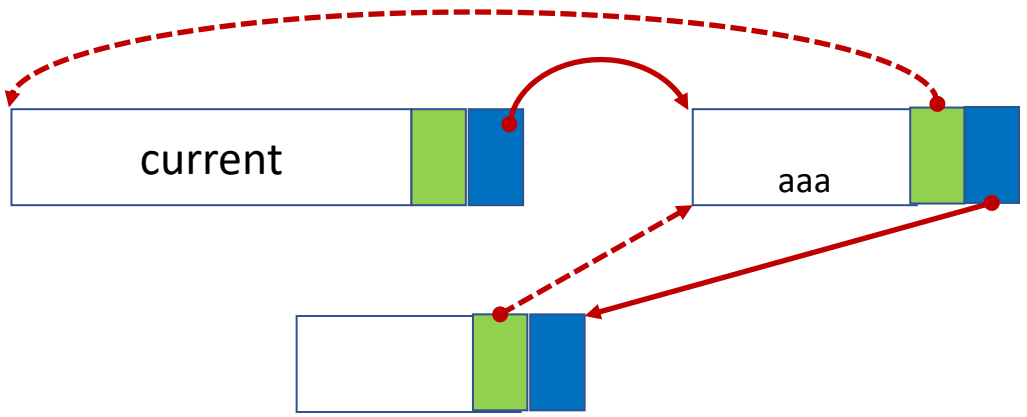
```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
    newnode->next = current->next;
    if (current->next) current->next->prev = newnode;
    current->next = newnode;
    newnode->prev = current;
  }
```



Double Free:
The Write-Anywhere Exploit

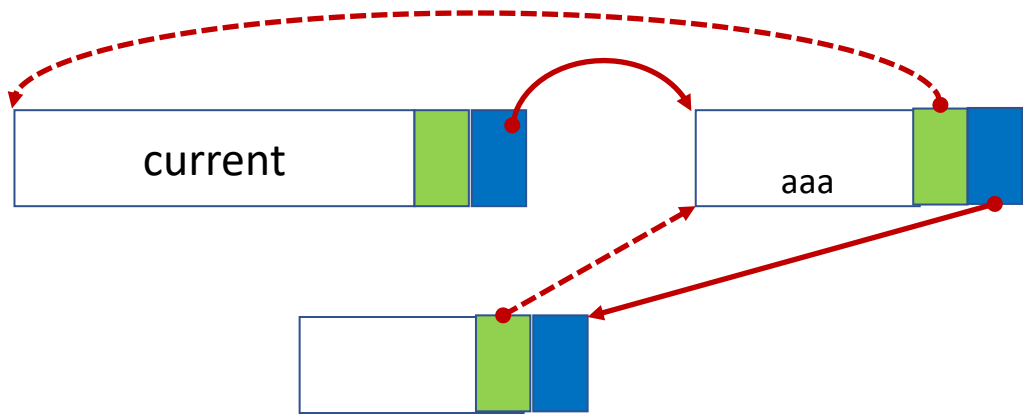
```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
    newnode->next = current->next;
    if (current->next) current->next->prev = newnode;
    current->next = newnode;
    newnode->prev = current;
  }
```

aaa aaa



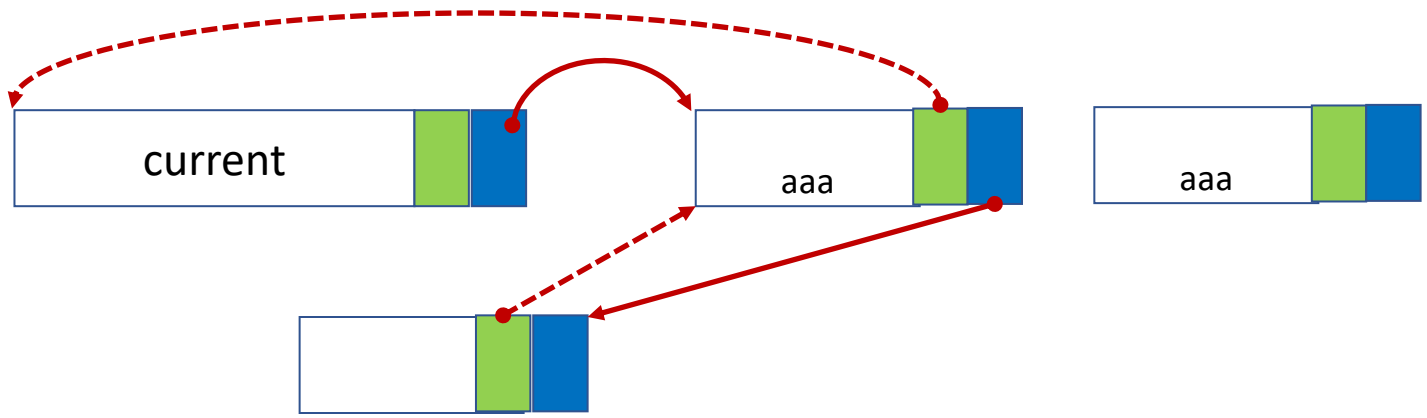
Double Free:
The Write-Anywhere Exploit

```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */                                          aaa                    aaa

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
    aaa ->next = aaa ->next;
    if (current->next) aaa ->next->prev = aaa ;
      aaa ->next = aaa ;
      aaa ->prev = aaa ;
  }
```



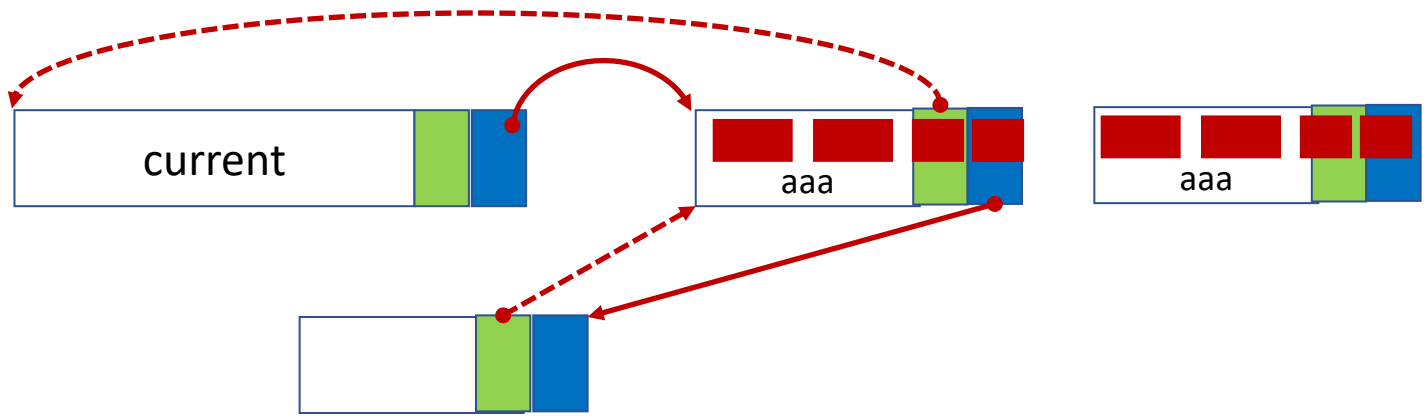
Double Free:
The Write-Anywhere Exploit

```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */                                          aaa                    aaa

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
    aaa ->next = aaa ->next;
    if (current->next) aaa ->next->prev = aaa ;
      aaa ->next = aaa ;
      aaa ->prev = aaa ;
  }
```



Double Free:
The Write-Anywhere Exploit

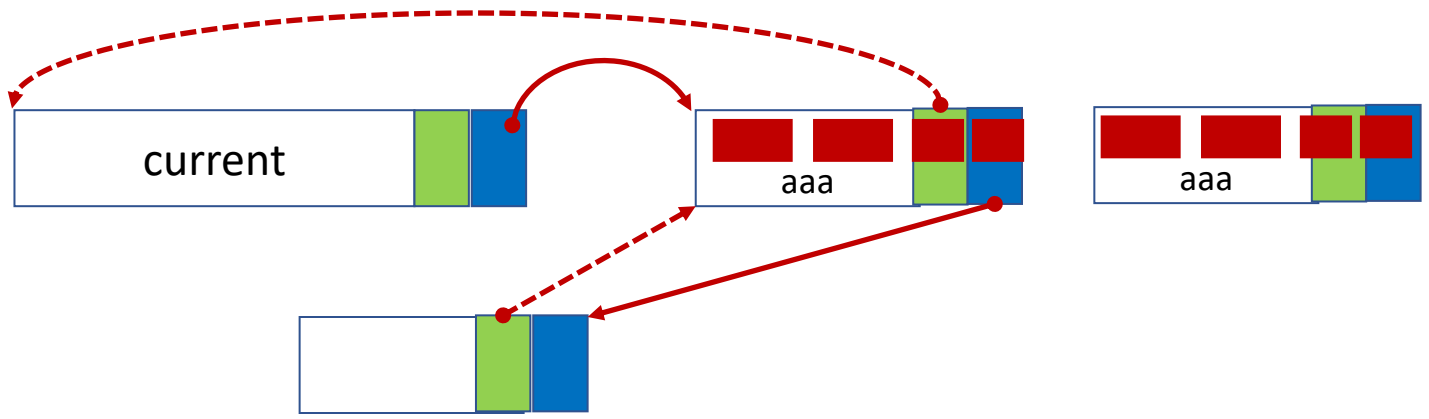
```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */                                          aaa                    aaa

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
    aaa ->next = aaa ->next;
    if (current->next) aaa ->next->prev = aaa ;
      aaa ->next = aaa ;
      aaa ->prev = aaa ;
  }
```



Double Free:
The Write-Anywhere Exploit

```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
     aaa ->next = aaa ->next;
     if (current->next) aaa ->next->prev = aaa ;
     aaa ->next = aaa ;
     aaa ->prev = aaa ;
  }
```

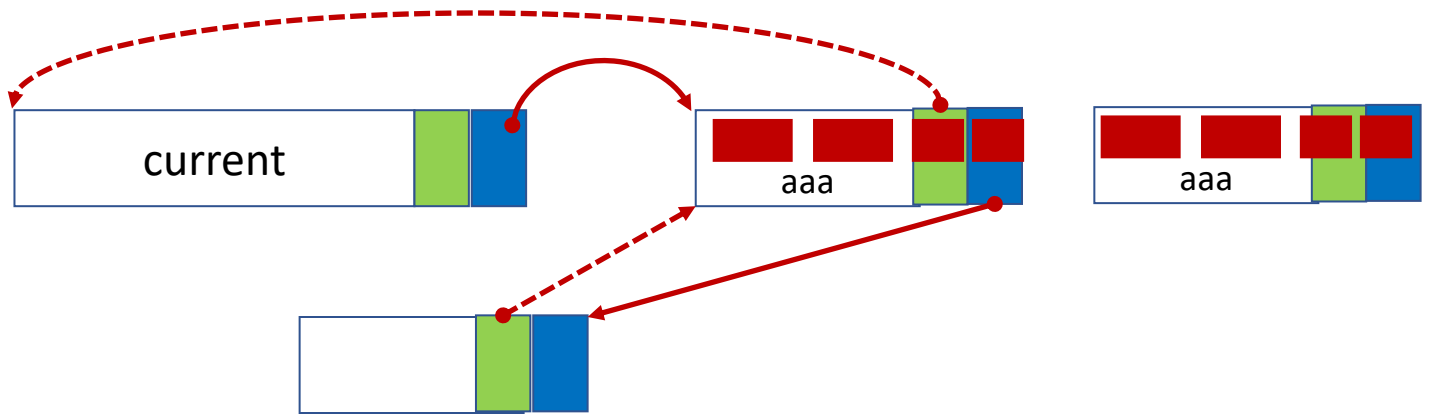
aaa aaa



Double Free:
The Write-Anywhere Exploit

```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
     aaa  ->next =  aaa  ->next;
     if (current->next)  aaa  ->next->prev =  aaa  ;
     aaa  ->next =  aaa  ;
     aaa  ->prev =  aaa  ;
  }
```

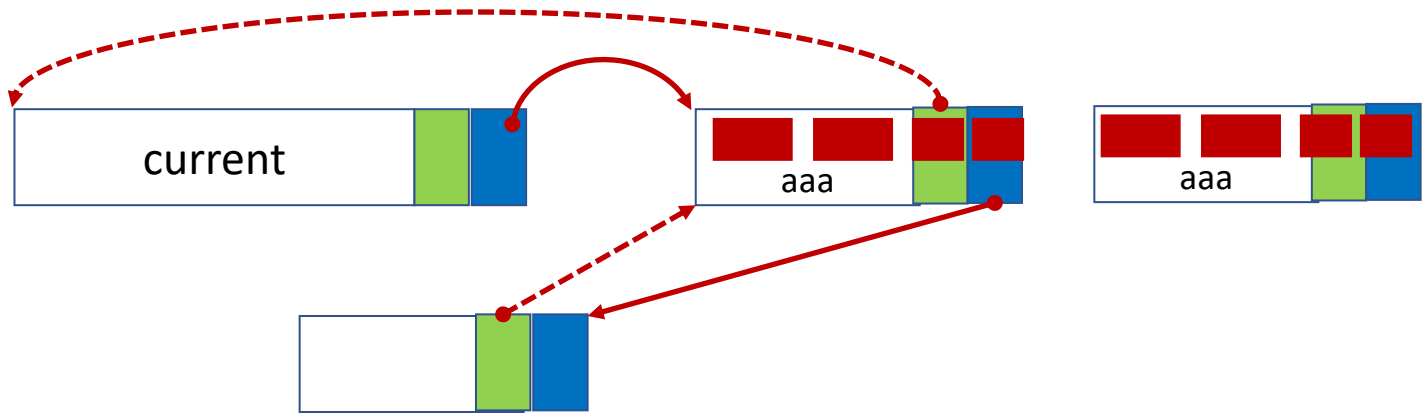
aaa aaa



Double Free:
The Write-Anywhere Exploit

```
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
     aaa  ->next =  aaa  ->next;
     if (current->next)  aaa  ->next->prev =  aaa  ;
     aaa  ->next =  aaa  ;
     aaa  ->prev =  aaa  ;
  }
```

aaa aaa

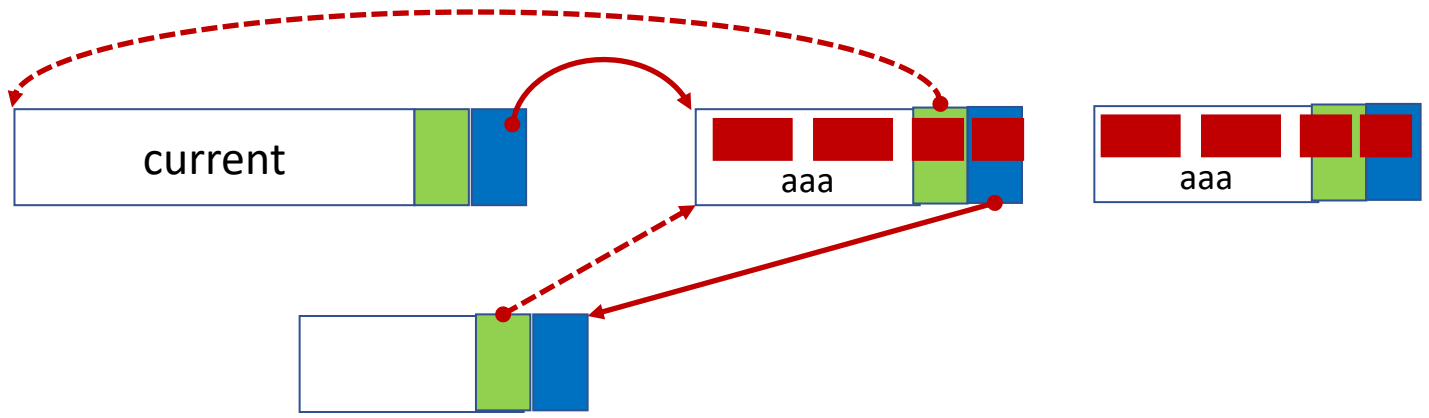


Double Free:
The Write-Anywhere Exploit

```
* @ current : Pointer to the chunk after which the new chunk
*             has to be inserted
* @ newnode : Pointer to the new chunk
*/

void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
  if (current) {
     aaa  ->next =  aaa  ->next;
     if (current->next)  aaa  ->next->prev =  aaa ;
     aaa  ->next =  aaa  ;
     aaa  ->prev =  aaa  ;
  }
```

aaa aaa

Write-Anywhere



Double Free:
The Write-Anything-Anywhere Exploit

```
void delete_from_list (struct chunk * p) {
  if (!p) return NULL;
  if (p->next) p->next->prev = p->prev;
  if (p->prev) p->prev->next = p->next;
  else free_list_head = p->next;
}
```

The attacker has corrupted p's next and previous pointers.

Double Free:
The Write-Anything-Anywhere Exploit

```
void delete_from_list (struct chunk * p) {
  if (!p) return NULL;
  if (p->next) p->next->prev = p->prev;
  if (p->prev) p->prev->next = p->next;
  else free_list_head = p->next;
}
```

The attacker has corrupted p's next and previous pointers.

Double Free:
The Write-Anything-Anywhere Exploit

```
void delete_from_list (struct chunk * p) {
  if (!p) return NULL;
  if (p->next) p->next->prev = p->prev;
  if (p->prev) p->prev->next = p->next;
  else free_list_head = p->next;
}
```

Write-Anywhere

Write Anything

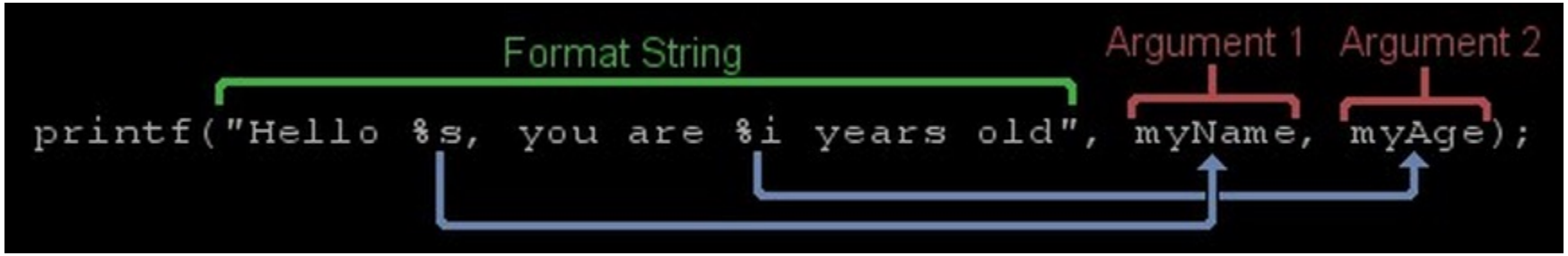
The attacker has corrupted p's next and previous pointers.

Spatial Memory Errors: Format String Bugs

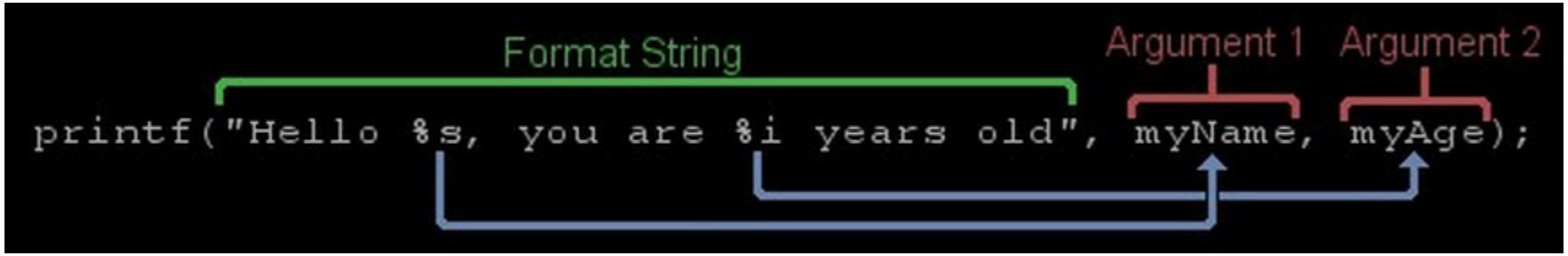
Format String Vulnerabilities

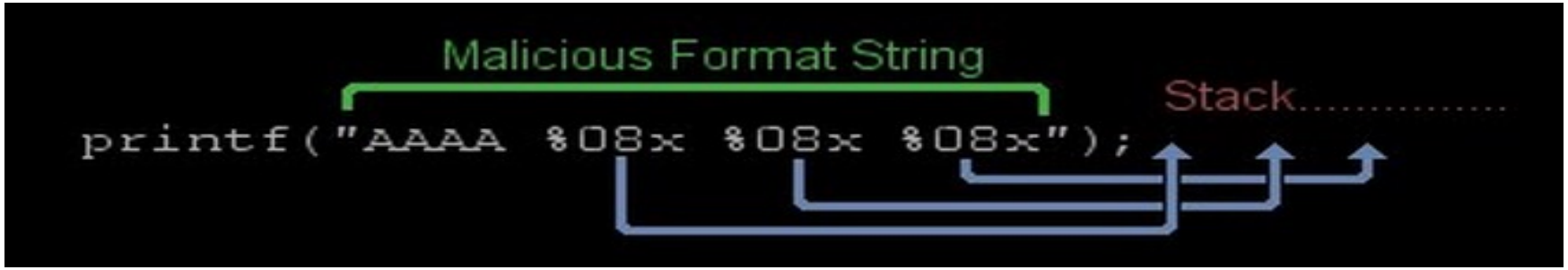
```
#include <stdio.h>

int main()
{
    srand(time(NULL));

    char localStr[100];
    int magicNumber = rand() % 100;
    int userCode = 0xBBBBBBBB;

    printf("Username? ");
    fgets(localStr, sizeof(localStr), stdin);

    printf("Hello ");
    printf(localStr);
    printf("What is the access code? ");

     scanf(“%d”, &userCode);
    if (userCode == magicNumber)
        printf("You win!\n");
…}
```

Format String Vulnerabilities



Format String Vulnerabilities

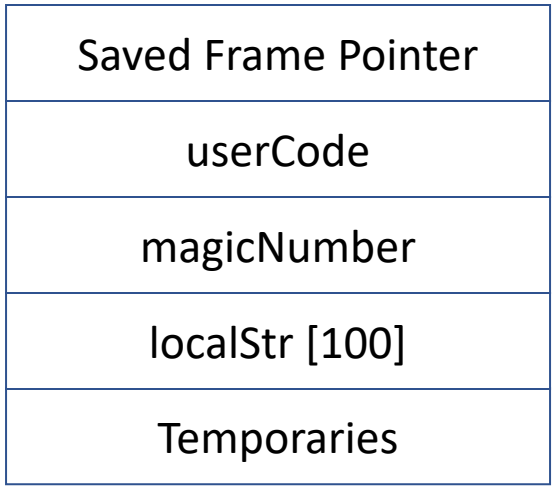




Format String Vulnerabilities

```
#include <stdio.h>

int main()
{
    srand(time(NULL));

    char localStr[100];
    int magicNumber = rand() % 100;
    int userCode = 0xBBBBBBBB;

    printf("Username? ");
    fgets(localStr, sizeof(localStr), stdin);

    printf("Hello ");
    printf(localStr);
    printf("What is the access code? ");

    scanf(“%d”, &userCode);
    if (userCode == magicNumber)
        printf("You win!\n");
… }
```

The “main” stack frame



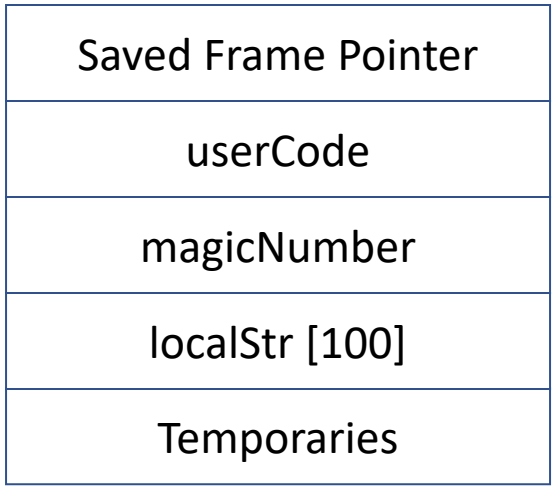
Format String Vulnerabilities

```
#include <stdio.h>

int main()
{
    srand(time(NULL));

    char localStr[100];
    int magicNumber = rand() % 100;
    int userCode = 0xBBBBBBBB;

    printf("Username? ");
    fgets(localStr, sizeof(localStr), stdin);

    printf("Hello ");
    printf(localStr);
```

AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x

```
    scanf(“%d”, &userCode);
    if (userCode == magicNumber)
        printf("You win!\n");
… }
```

The “main” stack frame



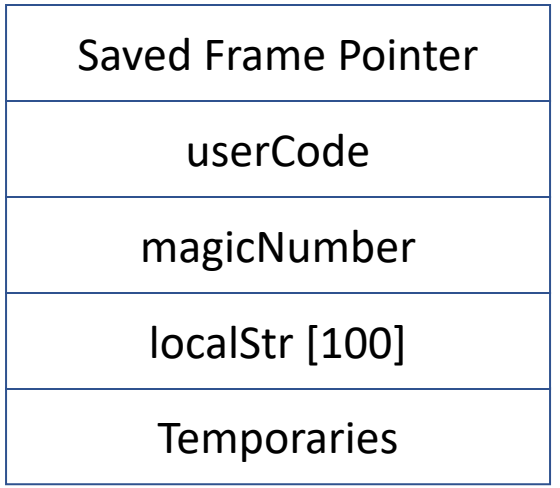
Format String Vulnerabilities

```
#include <stdio.h>

int main()
{
    srand(time(NULL));

    char localStr[100];
    int magicNumber = rand() % 100;
    int userCode = 0xBBBBBBBB;

    printf("Username? ");
    fgets(localStr, sizeof(localStr), stdin);

    printf("Hello ");
    printf(localStr);
```

AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x

```
    scanf("%d", &userCode);
    if (userCode == magicNumber)
        printf("You win!\n");
… }
```

The "main" stack frame



Format String Vulnerabilities

```
#include <stdio.h>

int main()
{
    srand(time(NULL));

    char localStr[100];
    int magicNumber = rand() % 100;
    int userCode = 0xBBBBBBBB;

    printf("Username? ");
    fgets(localStr, sizeof(localStr), stdin);

    printf("Hello ");
    printf(localStr);
```

The "main" stack frame

Saved Frame Pointer
userCode
magicNumber
localStr [100]
Temporaries

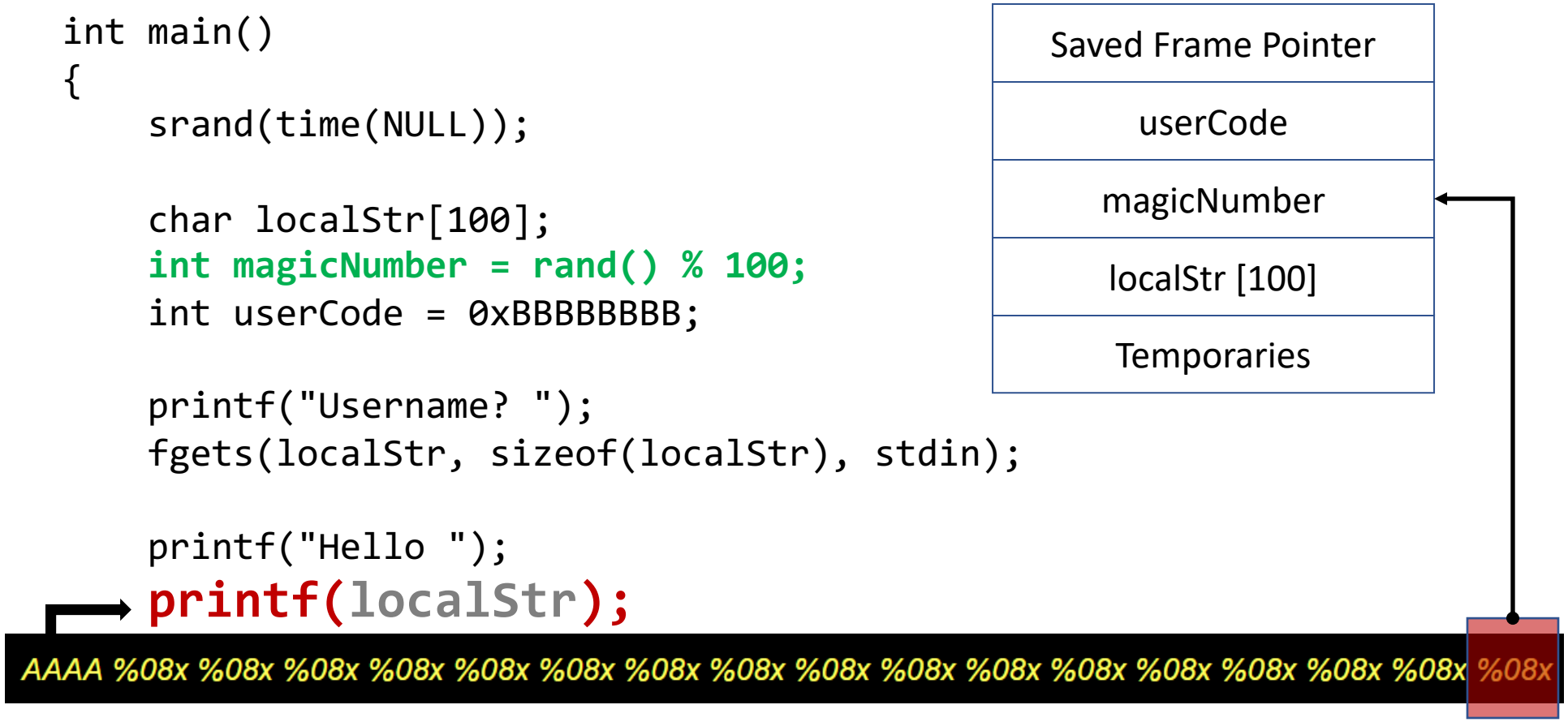
AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x

```
    scanf("%d", &userCode);
    if (userCode == magicNumber)
        printf("You win!\n");
… }
```

Format String Vulnerabilities

```
#include <stdio.h>

int main()
{
    srand(time(NULL));

    char localStr[100];
    int magicNumber = rand() % 100;
    int userCode = 0xBBBBBBBB;

    printf("Username? ");
    fgets(localStr, sizeof(localStr), stdin);

    printf("Hello ");
    printf(localStr);

    scanf("%d", &userCode);
    if (userCode == magicNumber)
        printf("You win!\n");
… }
```

The "main" stack frame

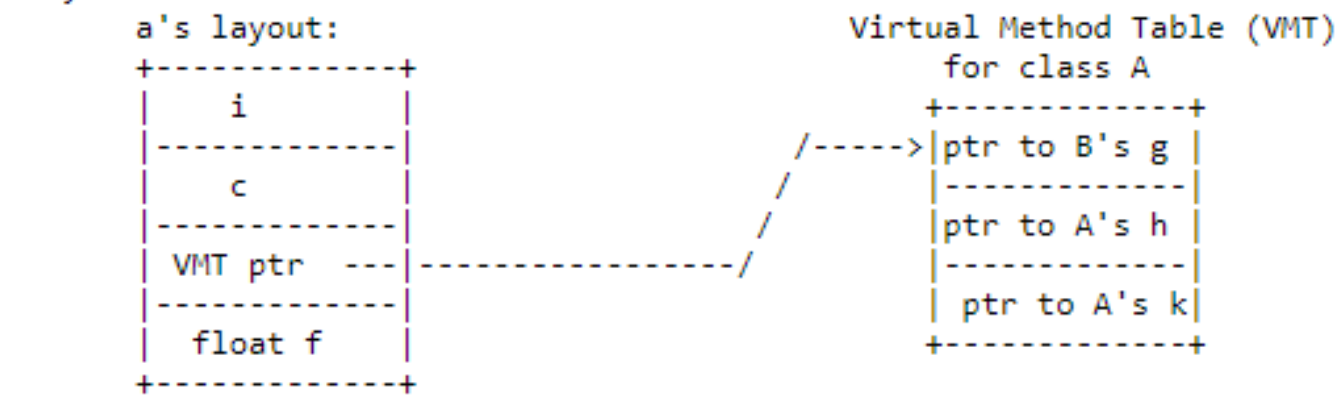


Advanced Type Errors: Bad Casting

Bad Casting

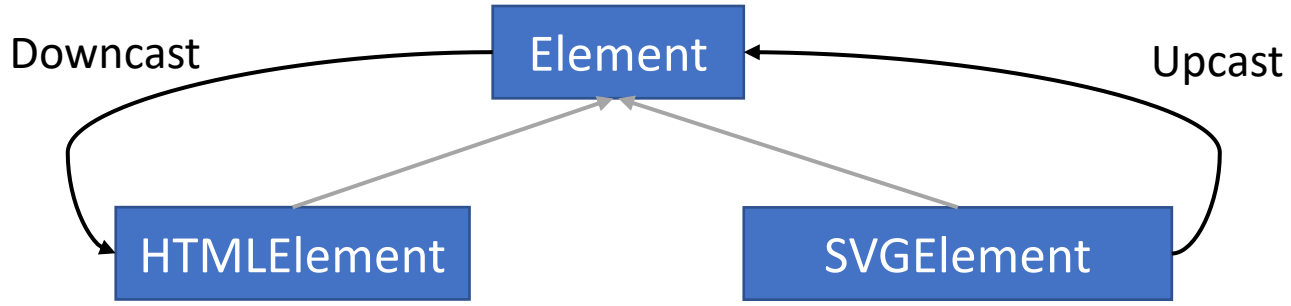
- Background:
  - C++ class subtyping & binding
  - A pointer to class T’ can safely point to a superclass T
    - All functions in class T are implemented in subclass T’
    - Upcasting is safe, down casting is not

```
class A: public B {
      float f;
      void h(); // Redefines h, but reuses implementation of G from B;
      virtual void k();
}

A a;
```



Upcasting and Downcasting

- Upcasting – From a derived class to its parent class
- Downcasting – From a parent class to one of its derived classes

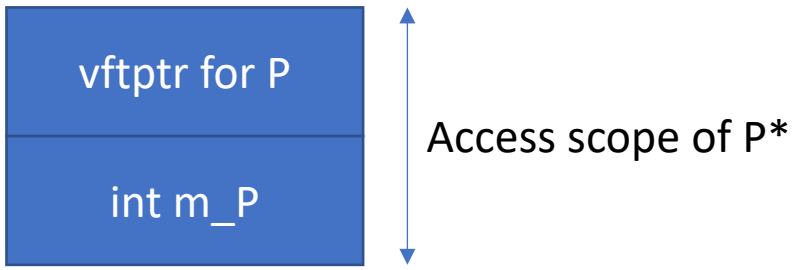


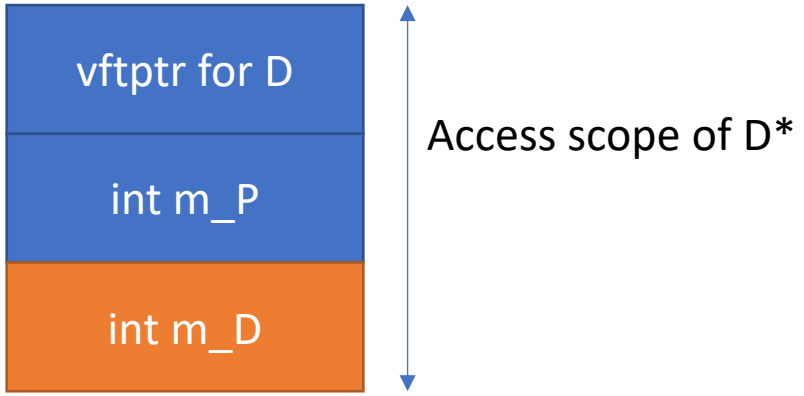
- Upcasting is always safe, but downcasting is not

Downcasting is not always safe

```
class P {
   virtual ~P() {}
   int m_P;
};
```

```
class D: public P {
   virtual ~D() {}
   int m_D;
};
```



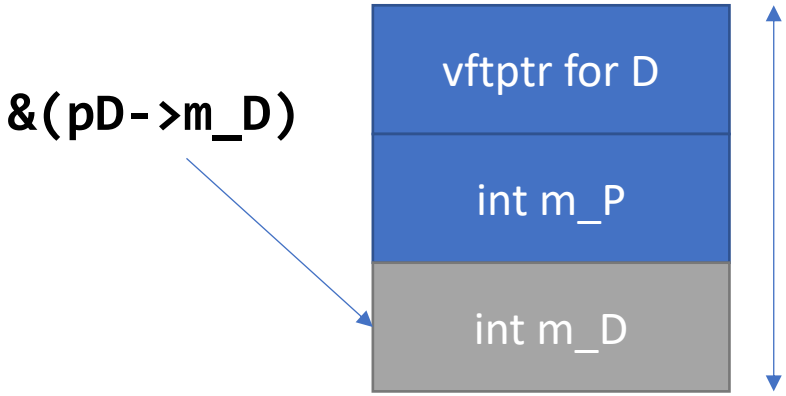


Downcasting can be Bad-casting

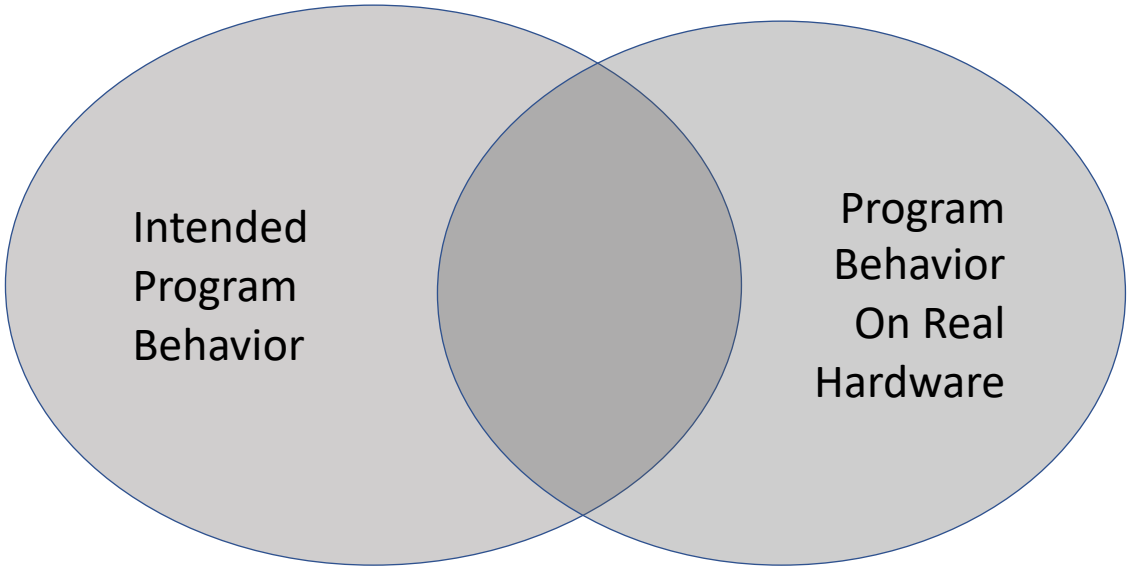
Bad-casting occurs: D is not a sub-object of P → Undefined behaviour

```
P *pS = new P();
D *pD = static_cast(pS);
pD->m_D;
```

Memory corruptions

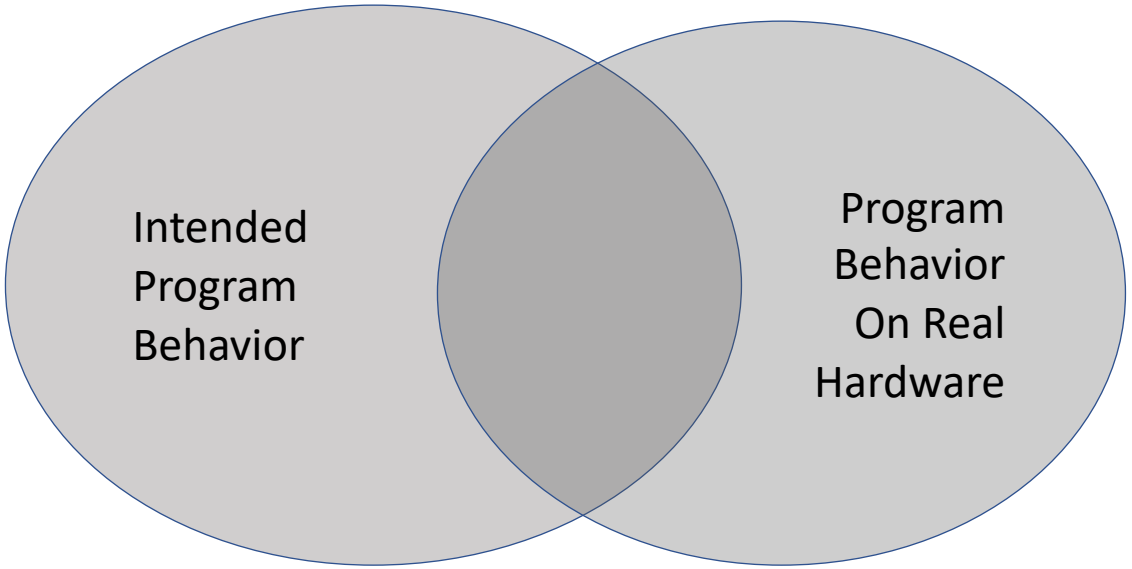


Summary


Intended
Program
Behavior
Program
Behavior
On Real
Hardware

Summary

- Memory Vulnerabilities
  - Spatial Memory Errors
  - Temporal Memory Errors
  - Type Errors
- Hardware does not give memory & type safety



Exploiting Vulnerabilities

Classes of Exploits

Classes of Exploits

- **Control-oriented**
  - Goal: Divert or Hijack Control Flow
  - Main Tricks:
    - Corrupt Code Pointers
    - Corrupt Non-code pointers
  - Outcome:
    - Code Injection
    - Code-reuse
      - Return to libc, ROP, Control-flow Bending
- **Data-oriented**
  - Goal: Hijack Data-flow
  - Outcomes:
    - Privilege Escalation, Data Leakage, …