# CSE467: Computer Security

## 12. Use After Free & Secure Coding

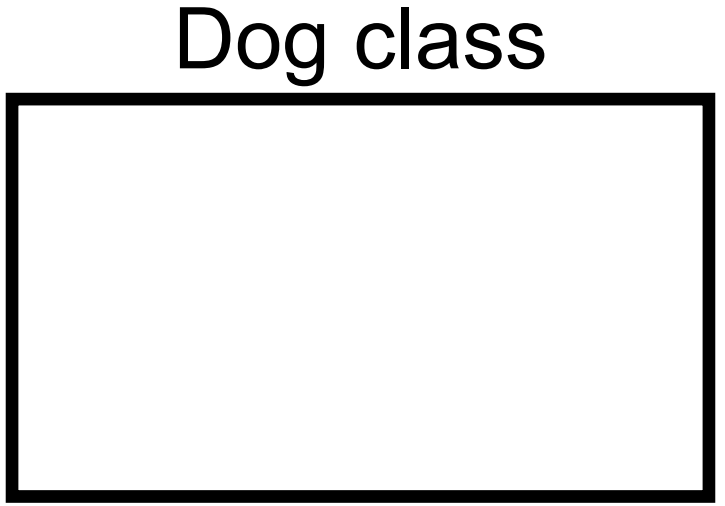
Seongil Wi

Department of Computer Science and Engineering

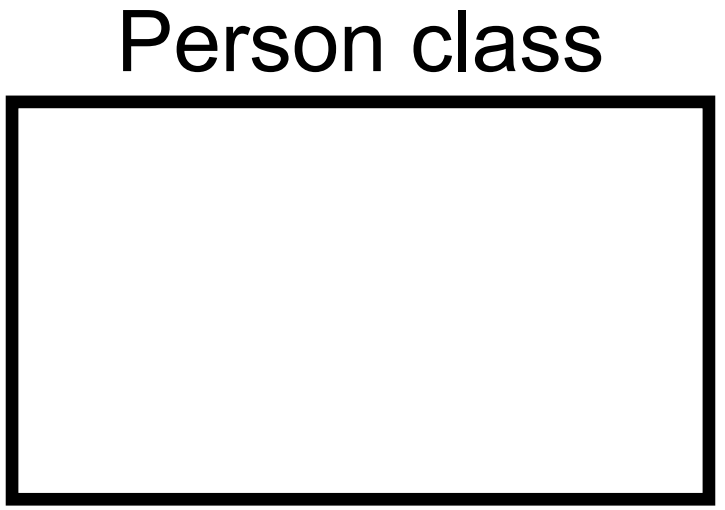
*The slide is based on Prof. Sang Kil Cha's lecture slide*

# Recap: Type Confusion

```
Dog *d = (Dog*) some_ptr;
d->bark();
```



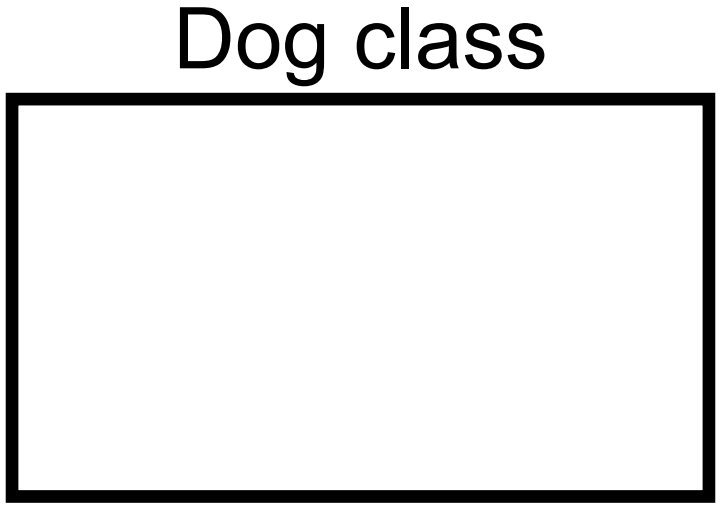

```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```



# Recap: Type Confusion

```
Dog *d = (Dog*) some_ptr;
d->bark();
```

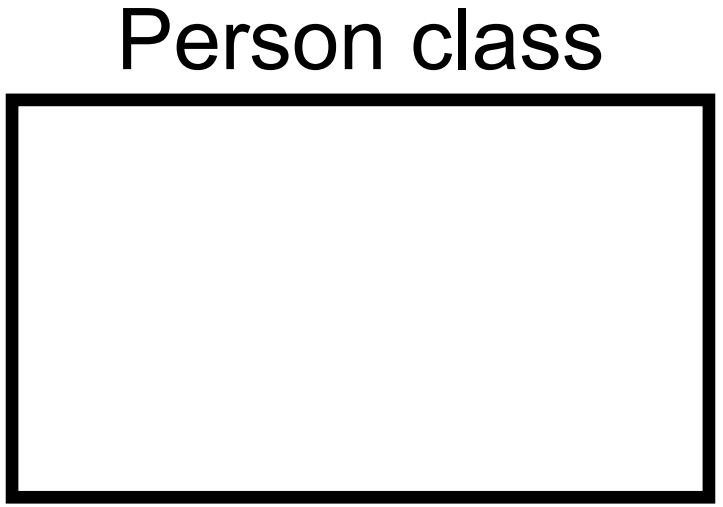
Normal




Type Confusion

```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

Abnormal



# Recap: Type Confusion

```
Dog *d = (Dog*) some_ptr;
d->bark();
```

Normal

Dog class

Type Confusion

```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

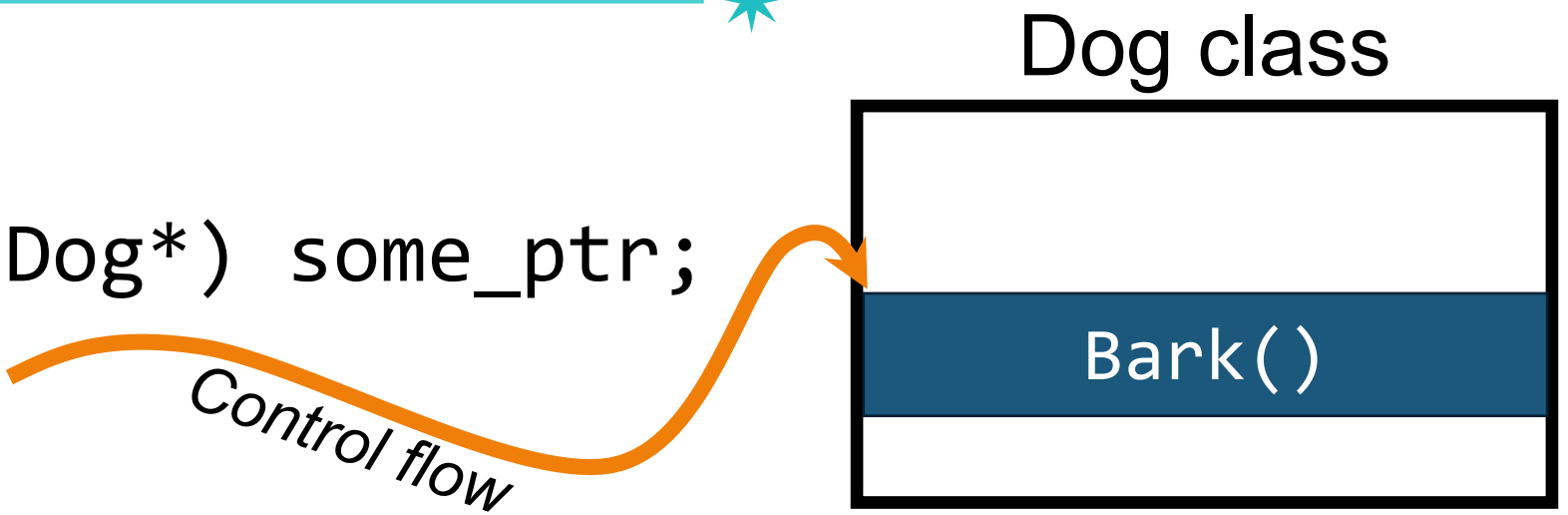
Abnormal

Person class

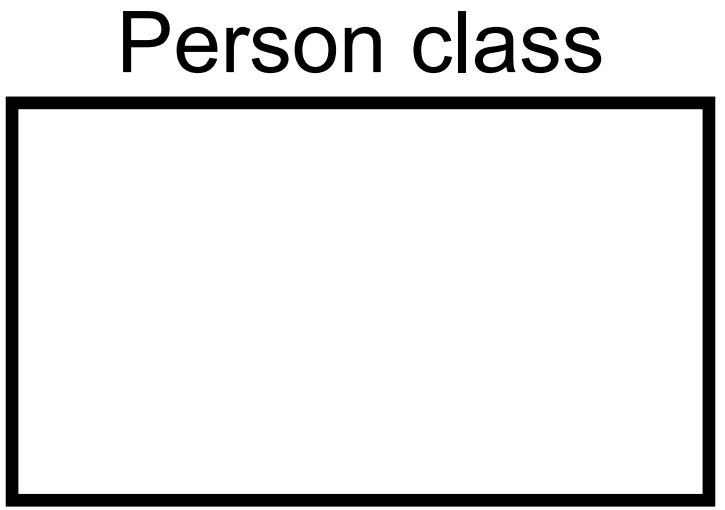
Invoke person's something

# Recap: Type Confusion Attack

```
Dog *d = (Dog*) some_ptr;
d->bark();
```



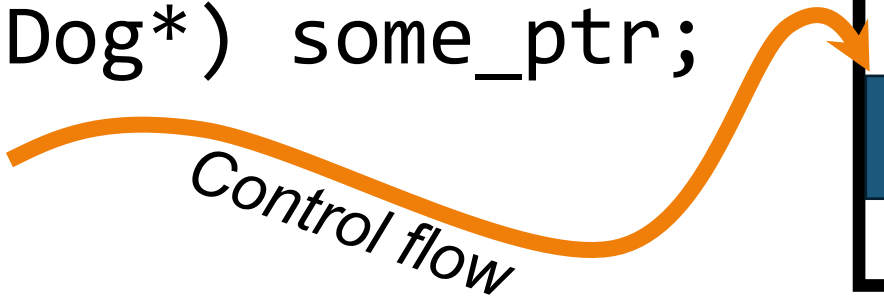

```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

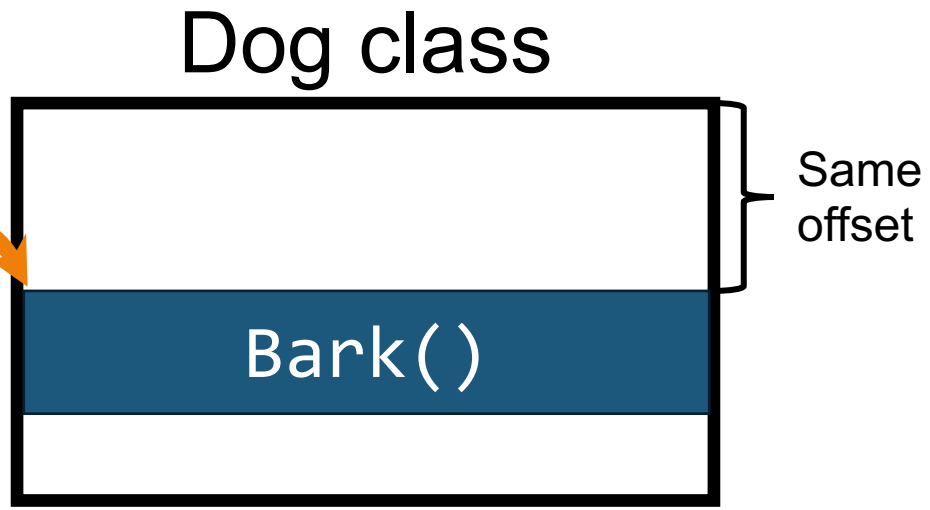


# Recap: Type Confusion Attack

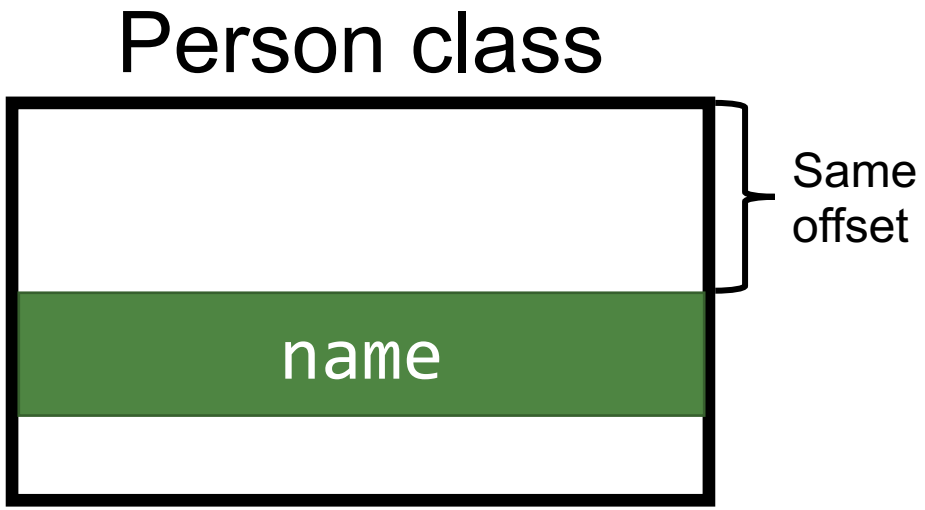


```
Dog *d = (Dog*) some_ptr;
d->bark();
```



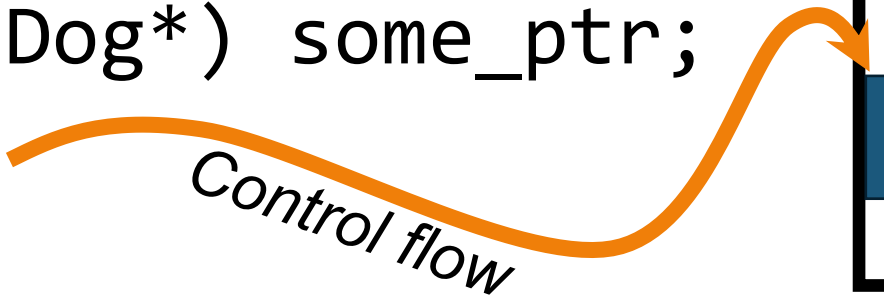



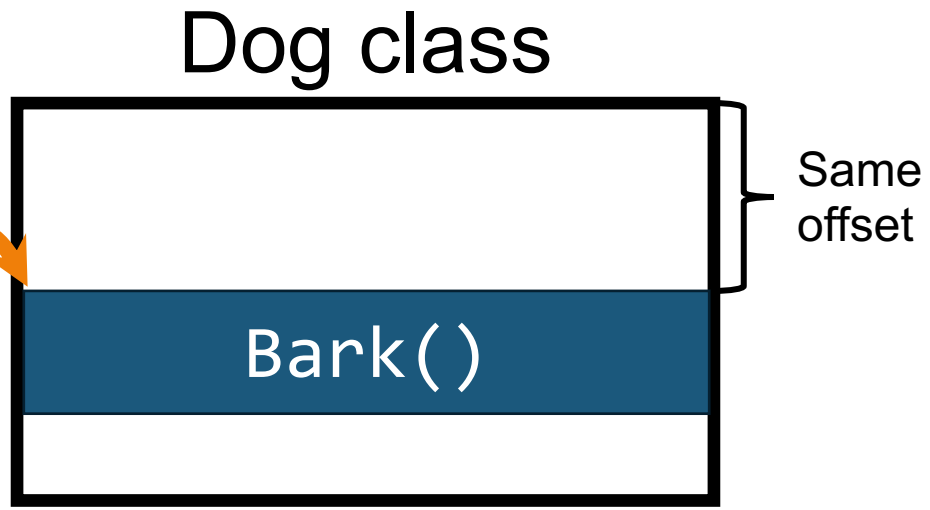

```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```



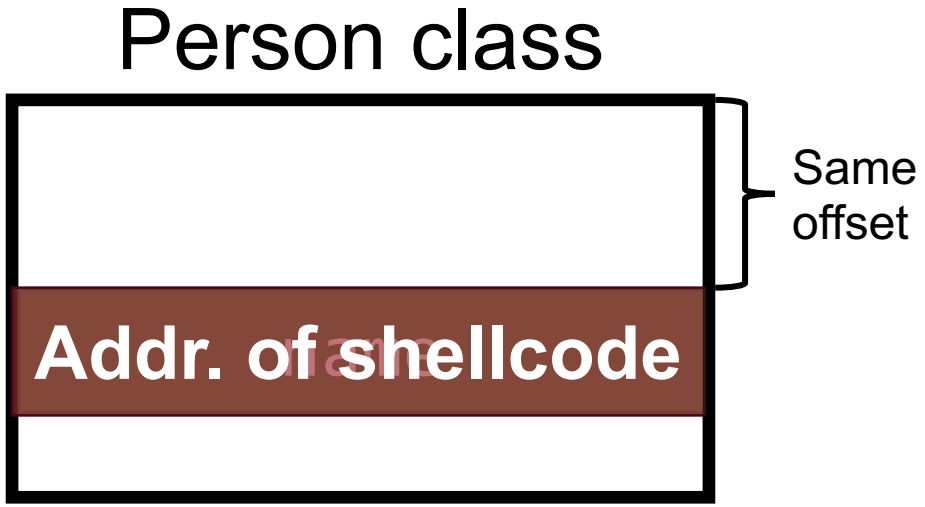
# Recap: Type Confusion Attack

```
Dog *d = (Dog*) some_ptr;
d->bark();
```



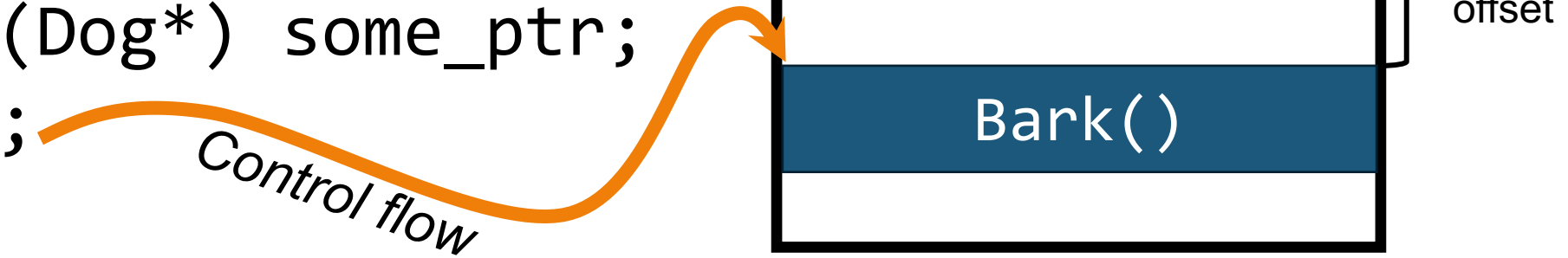




```
some_ptr->name="[shellcode]"
…
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

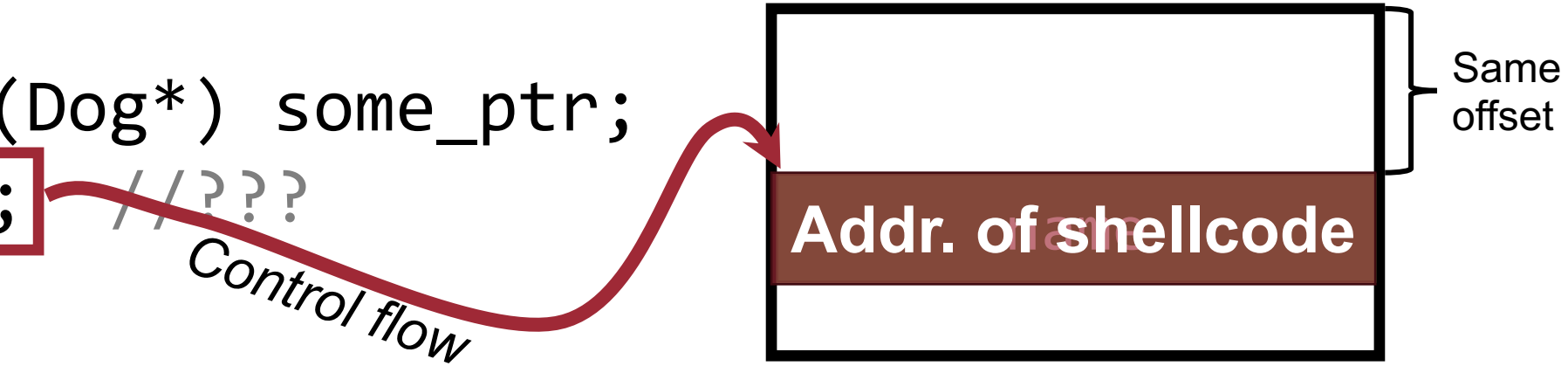


# Recap: Type Confusion Attack

```
Dog *d = (Dog*) some_ptr;
d->bark();
```




```
some_ptr->name="[shellcode]"
…
Dog *d = (Dog*) some_ptr;
d->bark(); //???
```



# Use After Free

## (A popular source of type confusion)

# Use After Free



- If after freeing a memory location, a program does not clear the pointer to that memory, an attacker can use it to hack the program

# Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Class information

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

# Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Allocate a memory block on the heap

**Class information**

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

f

**Class Foo**

# Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

**Class information**

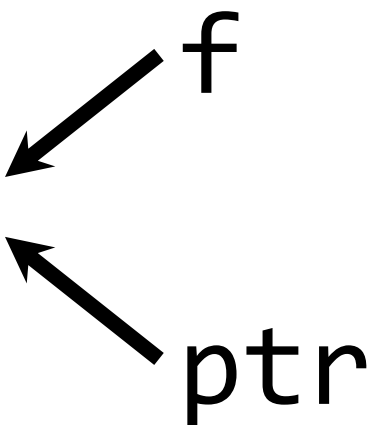
```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

# Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

**Class Foo**

Foo.x = 42




**Class information**

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

# Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Return the block to the free list

**Class information**

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

# Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

**Class information**

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```


Often called
“***Dangling Pointer***”

# Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

Find an appropriate block from the list of free blocks

**Class information**

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

# Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

**Class information**

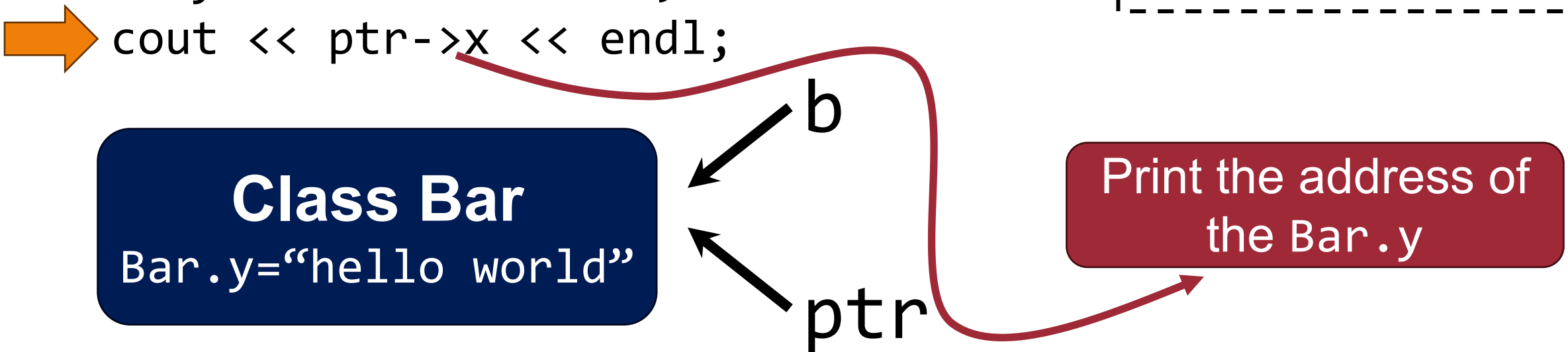
```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```

# Use After Free Example

```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

**Class information**

```
class Foo {
  public:
    int x;
};
class Bar {
  public:
    const char* y;
};
```



# Use After Free Example

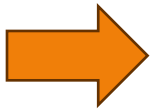
```
Foo * f = new Foo();
Foo * ptr = f;
ptr->x = 42;
delete f;
f = NULL;
Bar * b = new Bar();
b->y = “hello world”;
cout << ptr->x << endl;
```

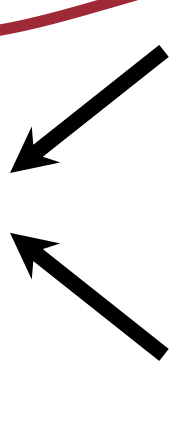
**Class information**

```
class Foo {
  public:
    int x;
};
```

```
ar* y;
```

We ***used*** this pointer ***after free***






**Class Bar**
Bar.y=“hello world”

Print the address of the Bar.y

# Use After Free can Trigger Type Confusion



- A dangling pointer's type and the corresponding reallocated data's type can be different => Trigger type confusion!

# Example: OpenSSL UAF Bug



```
...
dtls1_hm_fragment_free(frag);
pitem_free(item);
if (al==0) {
    *ok = 1;
    return frag->msg_header.frag_len;
}
```

# Example: OpenSSL UAF Bug



```
...
dtls1_hm_fragment_free(frag);
pitem_free(item);
if (al==0) {
    *ok = 1;
    return frag->msg_header.frag_len;
}
```

frag is freed

# Example: OpenSSL UAF Bug



```
...
dtls1_hm_fragment_free(frag);
pitem_free(item);
if (al==0) {
     *ok = 1;
     return frag->msg_header.frag_len;
}
```

frag is freed

Read after the free

# Key Points



- Type confusion bugs happen when a program misuses types
- ***Type confusion allows attackers to trigger memory corruption or disclosure***
- Use After Free is one of the major causes of type confusion

# Secure Coding

# Software Bug = Root of Evil



- Bugs can be exploited by an attacker to compromise the entire system

# Our Goal



- Error-free Software!

# Defensive Programming

Making the software behave in a predictable manner despite unexpected inputs or user actions*

Secure coding is a type of defensive programming that mainly concerns with computer security

* https://en.wikipedia.org/wiki/Defensive_programming

# Insecure vs. Secure

```
int func(char *input) {
    char buf[8];
    strcpy(buf, input);
    // ...
}
```

```
int func(char *input) {
    char buf[8];
    strncpy(buf, input, 8);
    buf[7] = 0;
    // ...
}
```

# Problem of Defensive Programming



```
int x = 1;
int y = 2;
int z = x + y;
if ( z > x && z > y )
    return z;
else
    abort();
```

- Introduce redundancy

# Problem of Defensive Programming



```
int x = 1;
int y = 2;
int z = x + y;
if ( z > x && z > y )
    return z;
else
    abort();
```

- Introduce redundancy
- Introduce hard-to-read code
- Slow down the program

# Offensive Programming



- A category of defensive programming (***not the opposite***)
- Make defensive checks to be unnecessary by failing fast

# Defensive vs. Offensive Programming



```
void addElement(list <int>* lst, int el) {
  if (lst != NULL) {
    lst -> push_back(el);
  }
}
```

VS.

```
void addElement(list <int>* lst, int el) {
  assert(lst);   // fail-fast!
  lst -> push_back(el);
}
```

# Defensive vs. Offensive Programming



```
void addElement(list <int>* lst, int el) {
  if (lst != NULL) {
    lst -> push_back(el);
  }
}
```

VS.

```
void addElement(list <int>* lst, int el) {
  assert(lst);   // fail-fast!
  lst -> push_back(el);
}
```

Better design! having a NULL pointer should be impossible (Fail fast)

# Where to Put Guards?



Defensive/offensive programming is a good start, but you should really know ***where*** to put your guards

# Secure Coding Guideline

# SEI CERT C Coding Standard



- https://websec-lab.github.io/courses/2023f-cse467/metarials/secure_coding.pdf

- Similar guidelines available for other languages, too
- Quite a lot of rules
- We will discuss only a few of them in this lecture

# Declare Objects with Appropriate Storage Duration



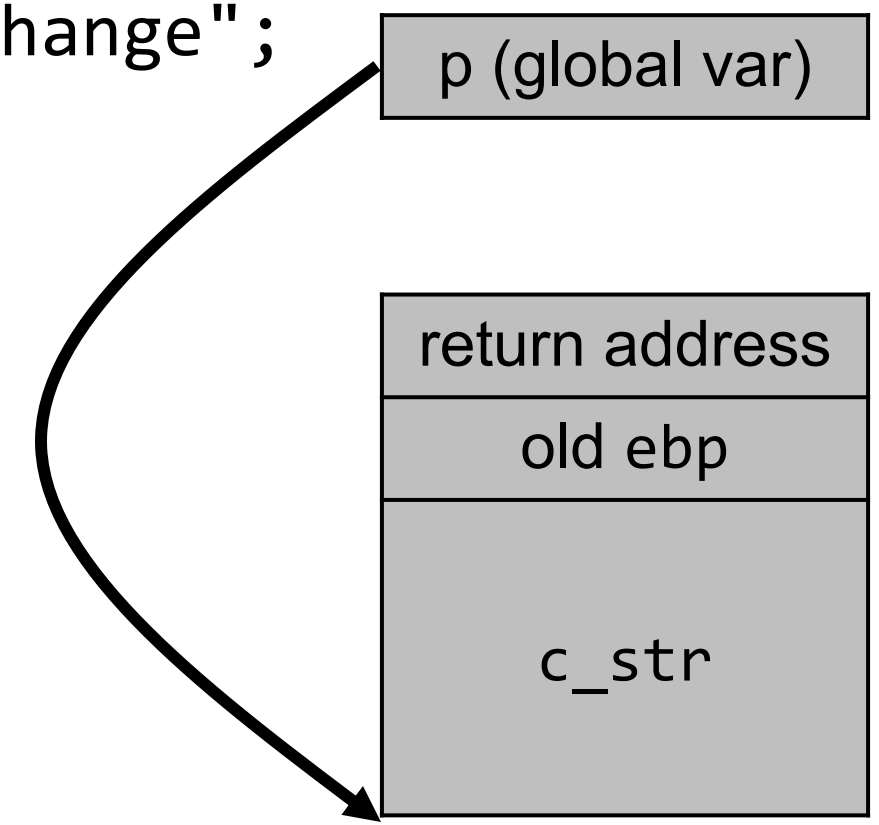
In the C Standard, 6.2.4, paragraph 2 [ISO/IEC 9899:2011]

The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. ***If an object is referred to outside of its lifetime, the behavior is undefined***. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

# Declare Objects with Appropriate Storage Duration



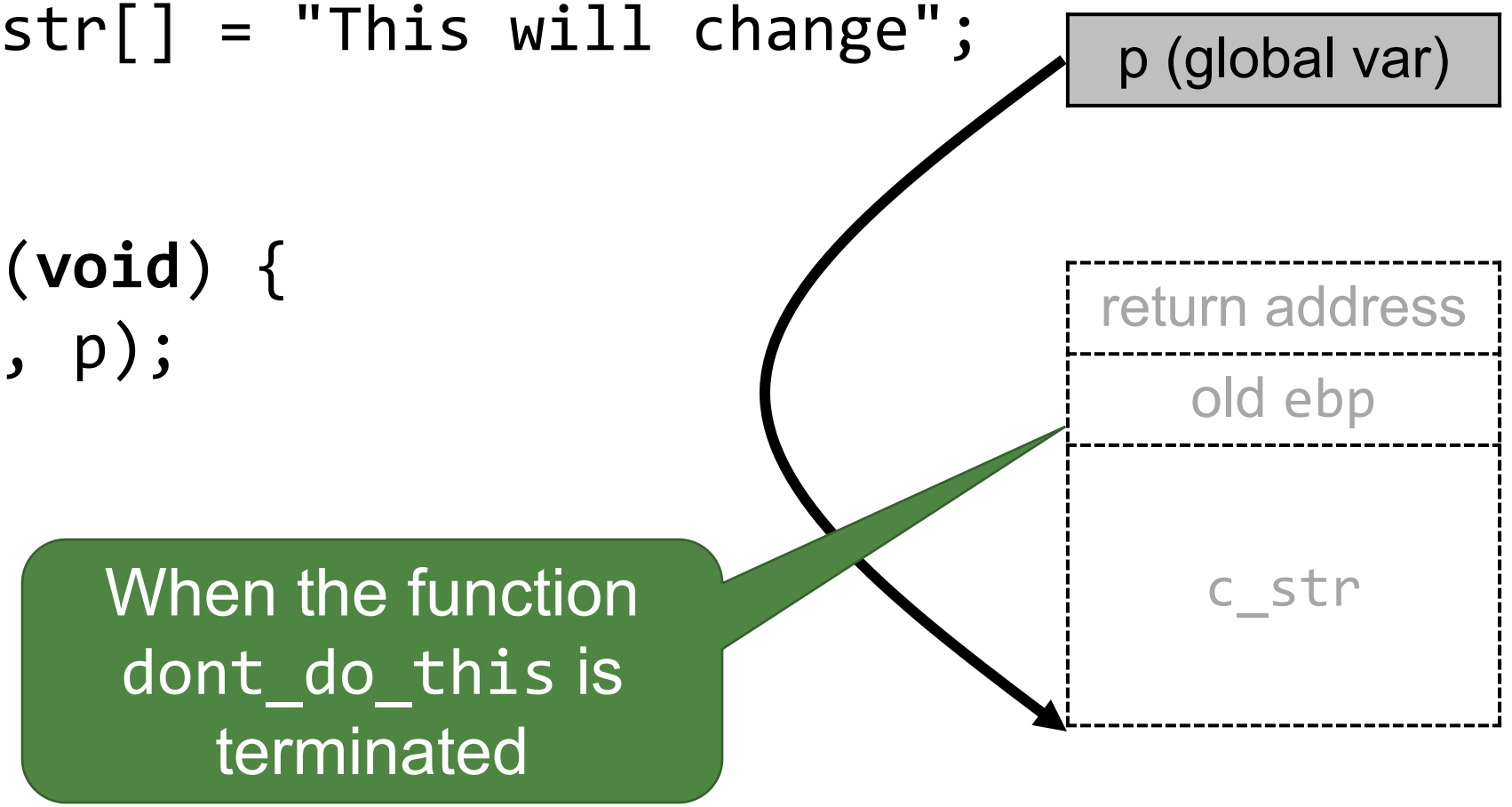
```
const char * p;
void dont_do_this (void) {
  const char c_str[] = "This will change";
  p = c_str;
}
void innocuous (void) {
  printf("%s\n", p);
}
```



# Declare Objects with Appropriate Storage Duration



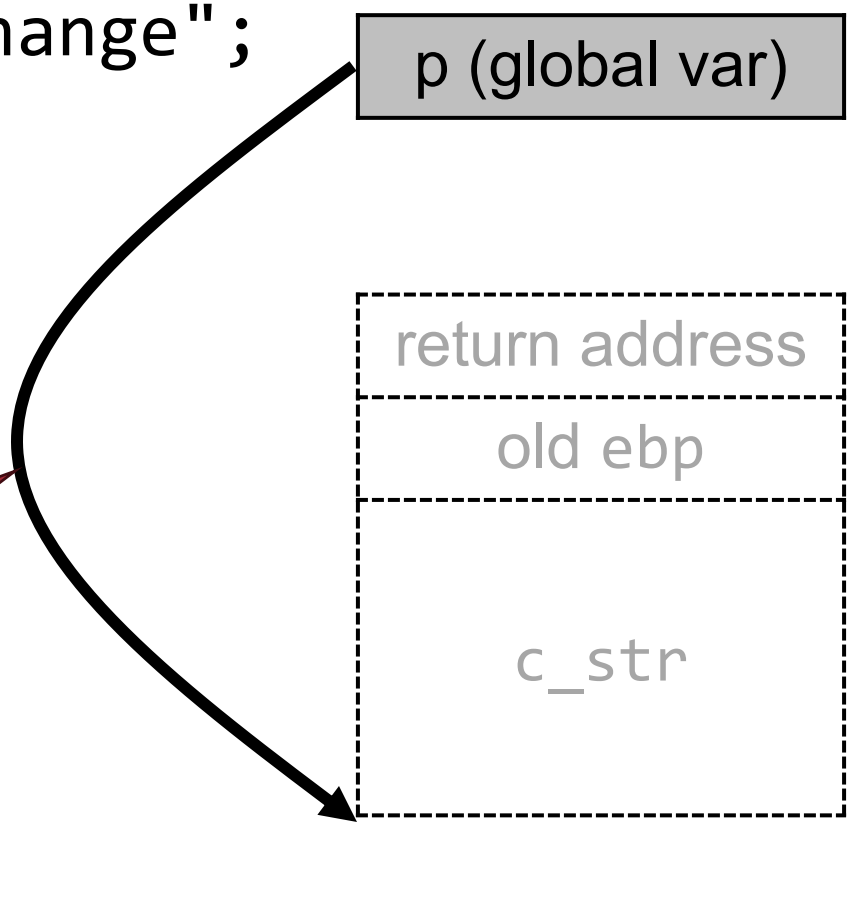
```
const char * p;
void dont_do_this (void) {
  const char c_str[] = "This will change";
  p = c_str;
}
void innocuous (void) {
  printf("%s\n", p);
}
```



# Declare Objects with Appropriate Storage Duration



```
const char * p;
void dont_do_this (void) {
  const char c_str[] = "This will change";
  p = c_str;
}
void innocuous (void) {
  printf("%s\n", p);
}
```



# Declare Objects with Appropriate Storage Duration



```
char* init_array(void) {
  char array[10]; /* Initialize array */
  return array;
}
```

# Do Not Depend on the Order of Evaluation for Side Effects



```
i = i + 1; // okay
i = ++i + 1; // not okay

a[i] = i; // okay
a[i++] = i; // not okay
```

# Do Not Read Uninitialized Memory



```
void set_flag(int number, int *sign_flag) {
  if (NULL == sign_flag) return;
  if (number > 0) {
    *sign_flag = 1;
  } else if (number < 0) {
    *sign_flag = -1;
  }
}
int is_negative(int number) {
  int sign;
  set_flag(number, &sign);
  return sign < 0;
}
```

# Do Not Read Uninitialized Memory



```
void set_flag(int number, int *sign_flag) {
  if (NULL == sign_flag) return;
  if (number > 0) {
    *sign_flag = 1;
  } else if (number < 0) {
    *sign_flag = -1;
  }
}
int is_negative(int number) {
  int sign;
  set_flag(number, &sign);
  return sign < 0;
}
```

# Ensure that signed/Unsigned Integer Operations Do Not Wrap



```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  /* ... */
}
```

# Ensure that signed/Unsigned Integer Operations Do Not Wrap



```
void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum = ui_a + ui_b;
  /* ... */
}
```

Integer overflow can be occured

# Ensure that signed/Unsigned Integer Operations Do Not Wrap



Fixed version

```
#include <limits.h>

void func(unsigned int ui_a, unsigned int ui_b) {
  unsigned int usum;
  if (UINT_MAX - ui_a < ui_b) {
    /* Handle error */
  } else {
    usum = ui_a + ui_b;
  }
  /* ... */
}
```

# Do Not Form or Use Out-of-Bounds Pointers or Array Subscripts



```
enum { TABLESIZE = 100 };


static int table[TABLESIZE];

int* f (int index) {
  if (index < TABLESIZE) {
    return table + index;
  }
  return NULL;
}
```

# Guarantee that Storage for Strings has Sufficient Space



Length of the variable src

```
void copy(size_t n, char* src, char* dest) {
  size_t i;
  for (i = 0; src[i] && (i < n); ++i) {
    dest[i] = src[i];
  }
  dest[i] = '\0’;
}
```

# Do Not Pass a Non-Null-Terminated Character Sequence to a String Argument



```
void func() {
  char c_str[3] = "abc";
  printf("%s\n", c_str);
}
```

# Do Not Pass a Non-Null-Terminated Character Sequence to a String Argument



```
void func() {
  char c_str[3] = "abc";
  printf("%s\n", c_str);
}
```

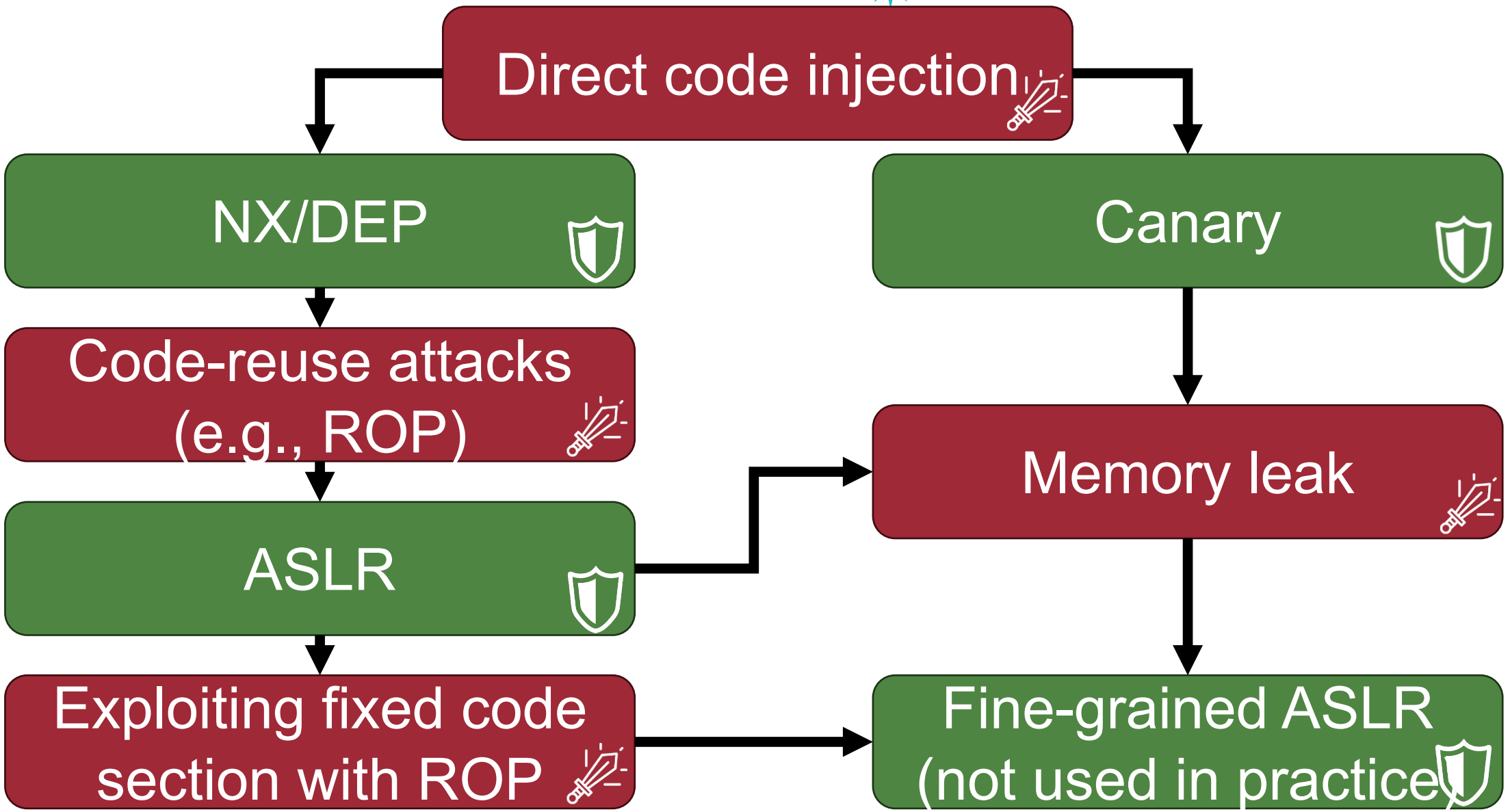
| a | b | c | @#$!#DF!#& |
|---|---|---|---|

Memory disclosure

# And Many More …

Check out the pdf: https://websec-lab.github.io/courses/2023f-cse467/metarials/secure_coding.pdf

# Summary of Software Security



# Summary of Software Security



- Software security can affect physical & data security
  - SW can manipulate machines and read / write data
- SW bugs can lead to security problems
- Growing interest as SW is eating the world!
  - Traditional SW: financial, military, privacy, et.
  - Emerging concerns: security of AI such as fairness or morality

# Question?