

# CSE467: Computer Security

## 12. Use After Free & Secure Coding

Seongil Wi

# Notice



- There will be Q&A session for HW2
  - Oct. 24
    - 30 minutes lecture (It is okay to leave the room after the lecture is end)
    - 45 minutes Q&A session

|            |                                  |                                                 |
|------------|----------------------------------|-------------------------------------------------|
| 10/17/2020 | Midterm week (No exam, no class) |                                                 |
| 10/19/2020 | Midterm week (No exam, no class) |                                                 |
| 10/24/2023 | Web Security #1                  | HW2 Q&A session<br><del>HW2 due (11:59PM)</del> |
| 10/26/2023 | Web Security #2                  | HW2 due (11:59PM)                               |

# Recap: Type Confusion



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

Normal

Dog class



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

Abnormal

Person class



# Recap: Type Confusion

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

Normal

Dog class

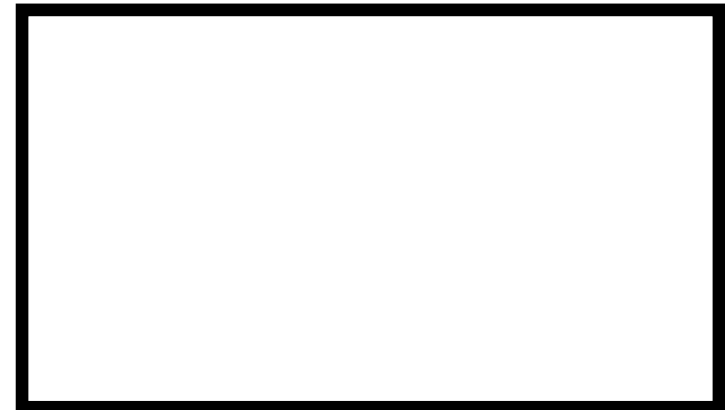


Type Confusion

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

Abnormal

Person class



# 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();
```

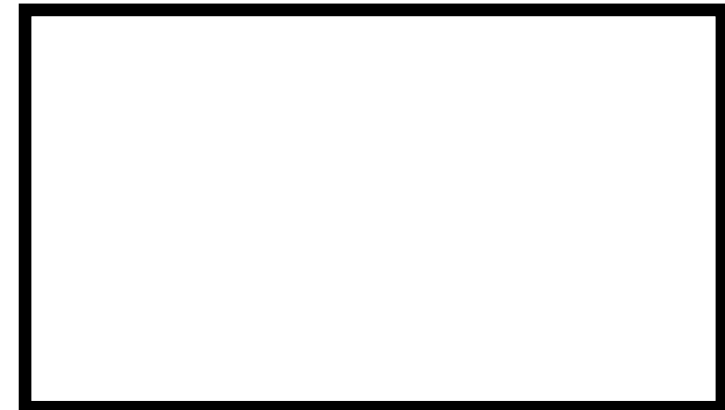
*Control flow*

Dog class



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

Person class



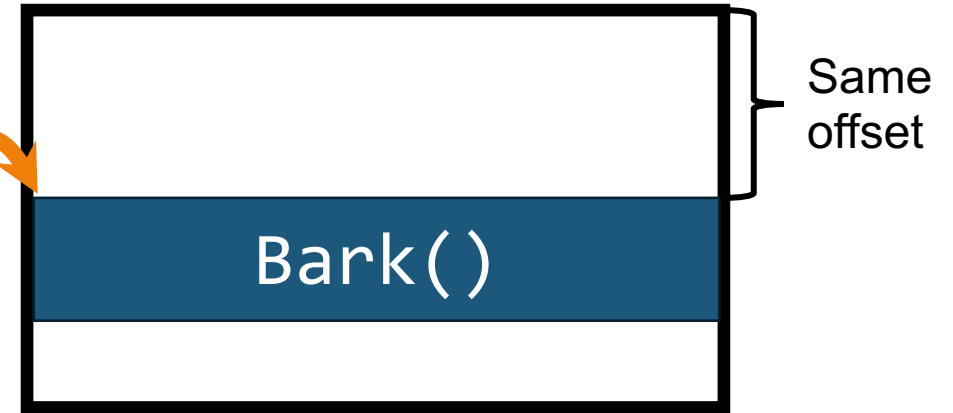
# Recap: Type Confusion Attack

7

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

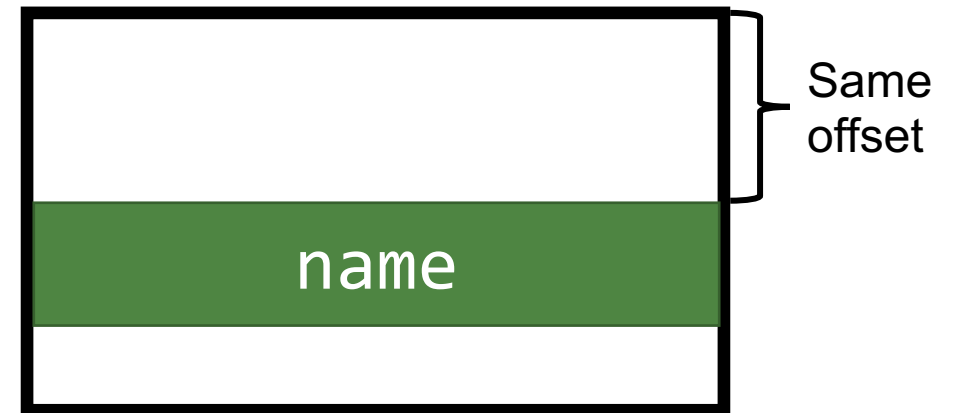
*Control flow*

Dog class



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

Person class



# Recap: Type Confusion Attack

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

*Control flow*

Dog class

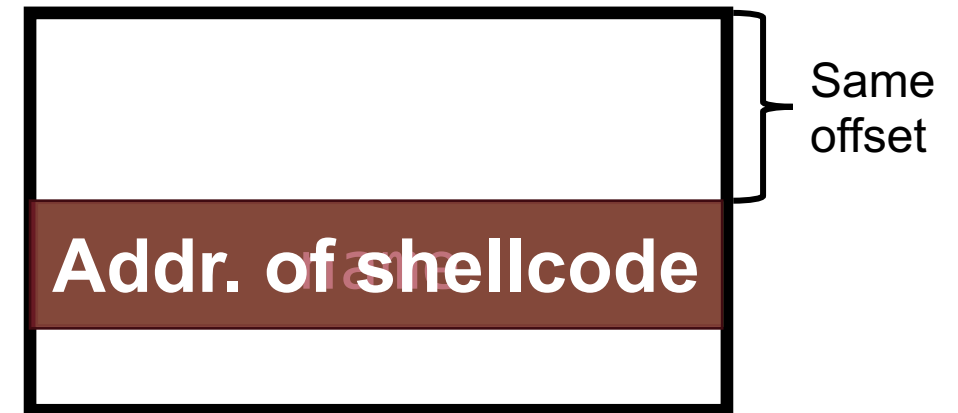


```
some_ptr->name="[shellcode]"
```

...

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

Person class



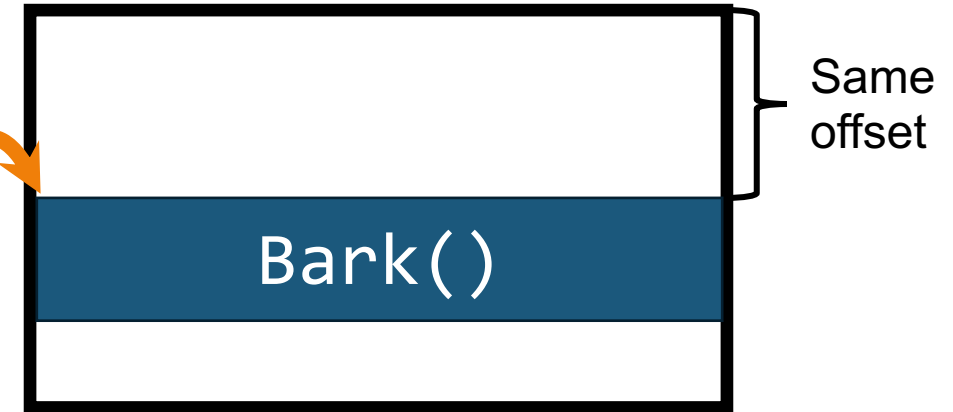


# Recap: Type Confusion Attack

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

Control flow

Dog class



```
some_ptr->name="[shellcode]"
```

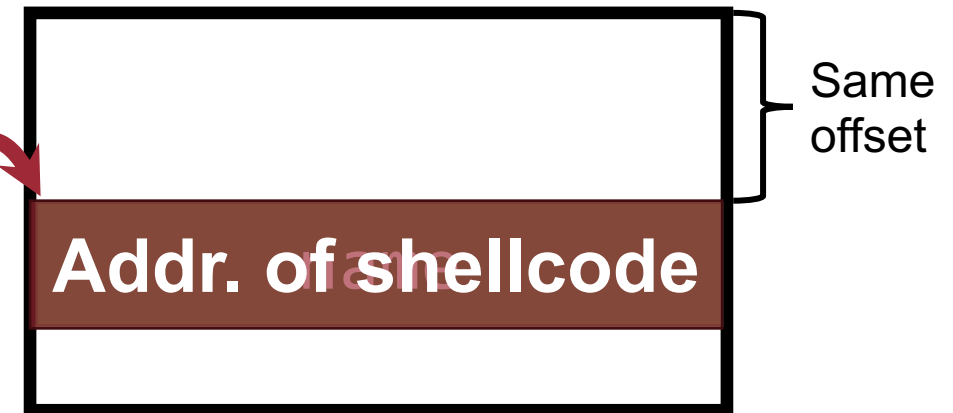
...

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

// ???

Control flow

Person class



# **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

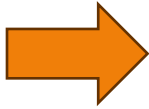


## Class information

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

```
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;
```

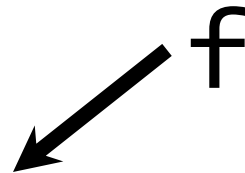
# Use After Free Example



```
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 Foo



## Class information

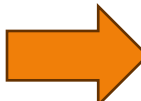
```
class Foo {  
    public:  
        int x;  
};  
class Bar {  
    public:  
        const char* y;  
};
```

# Use After Free Example



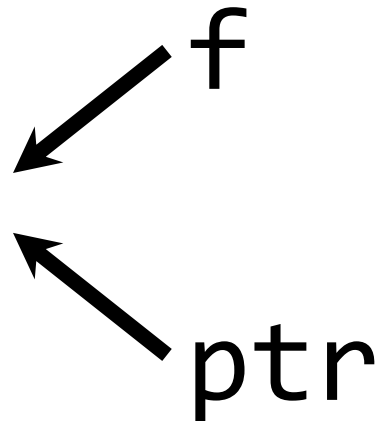
## Class information

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



```
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**



# Use After Free Example



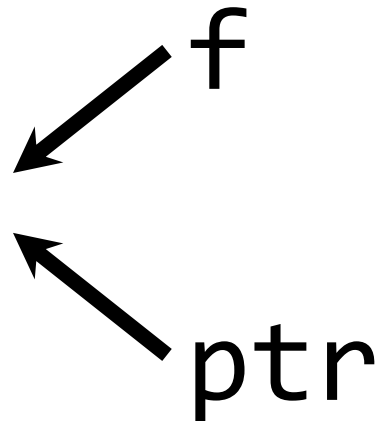
## Class information

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

```
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



# 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

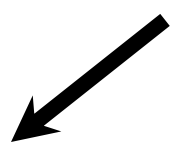
## Class information

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

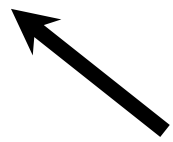
Class Foo

Foo.x = 42

f



ptr





# 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;  
};
```

**Class Foo**

Foo.x = 42

ptr

Often called  
***"Dangling Pointer"***

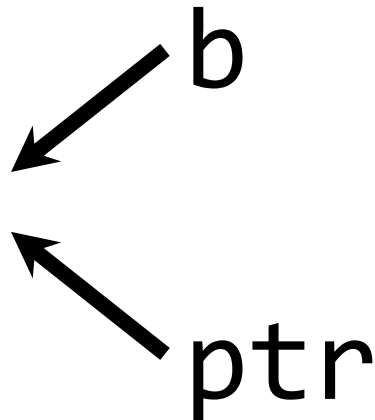
# Use After Free Example



```
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 Bar**



## Class information

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

# Use After Free Example



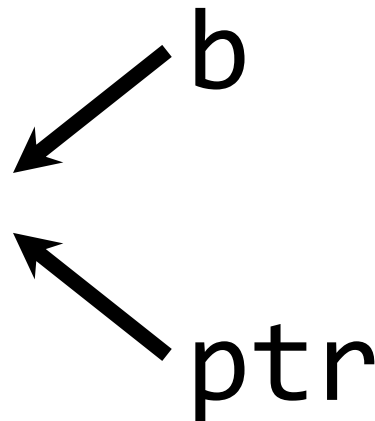
```
Foo * f = new Foo();  
Foo * ptr = f;  
ptr->x = 42;  
delete f;  
f = NULL;  
Bar * b = new Bar();  
b->y = "hello world";  
cout << ptr->x << endl;
```

## Class information

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

**Class Bar**

Bar.y="hello world"



# 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;  
};
```

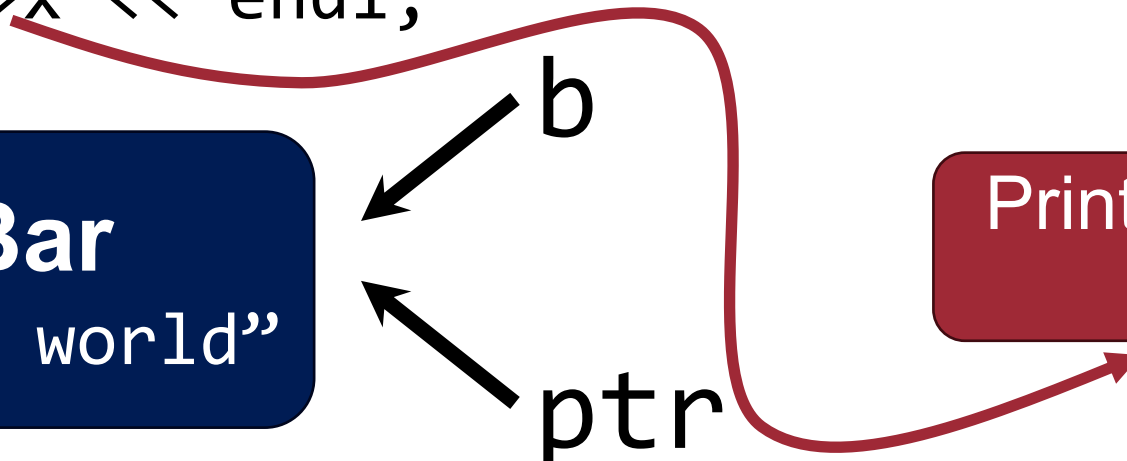
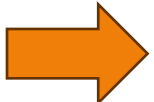
### Class Bar

Bar.y="hello world"

b

ptr

Print the address of  
the Bar.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:  
        Bar* y;  
};
```

We *used* this  
pointer *after free*

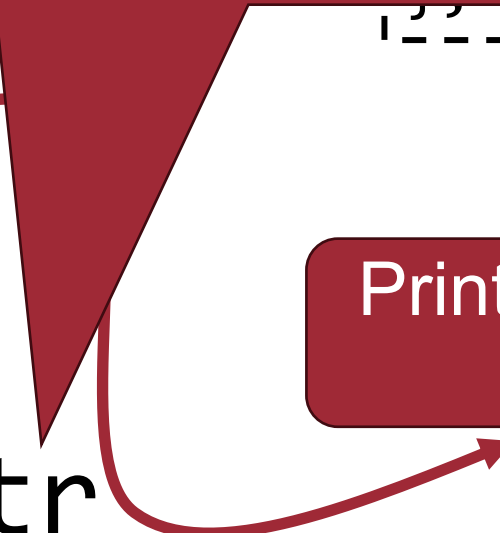
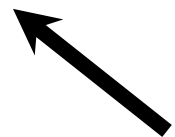
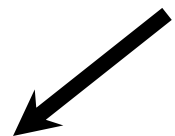
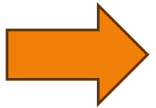
Print the address of  
the Bar.y

## Class Bar

Bar.y="hello world"

b

ptr



# Use After Free can Trigger Type Confusion

22

- 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

---

28

- 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](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] = '\n';  
    // ...  
}
```

# 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.

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

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

# 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/materials/secure\\_coding.pdf](https://websec-lab.github.io/courses/2023f-cse467/materials/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 40



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

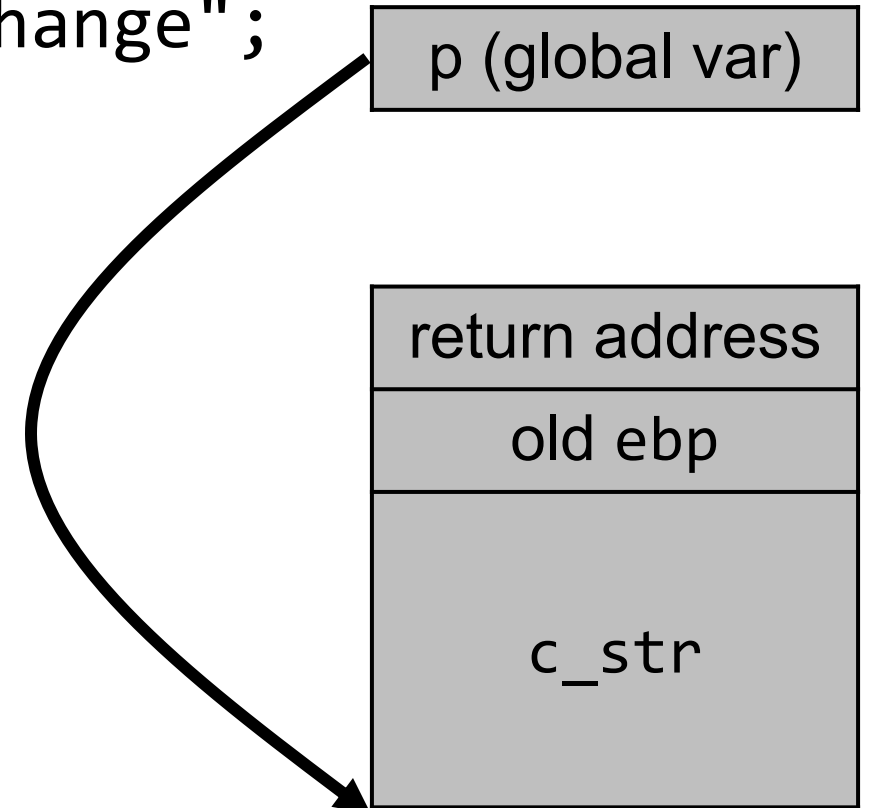
The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. ***If an object is referred to outside of its lifetime, the behavior is undefined.*** 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 41



```
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 42



```
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);  
}
```

p (global var)

return address

old ebp

c\_str

When the function  
dont\_do\_this is  
terminated

# Declare Objects with Appropriate Storage Duration 43



```
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);  
}
```

p (global var)

return address

old ebp

c\_str

Dangling pointer  
(Can be used for Use  
After Free)

## Declare Objects with Appropriate Storage Duration 44

---



```
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;  
}
```



What if number is 0?

# Ensure that signed/Unsigned Integer Operations Do Not Wrap

48



```
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 49



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

Integer overflow can  
be occurred

# Ensure that signed/Unsigned Integer Operations Do Not Wrap

50



## 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

53

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

55

```
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

56

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

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

Memory disclosure

# And Many More ...

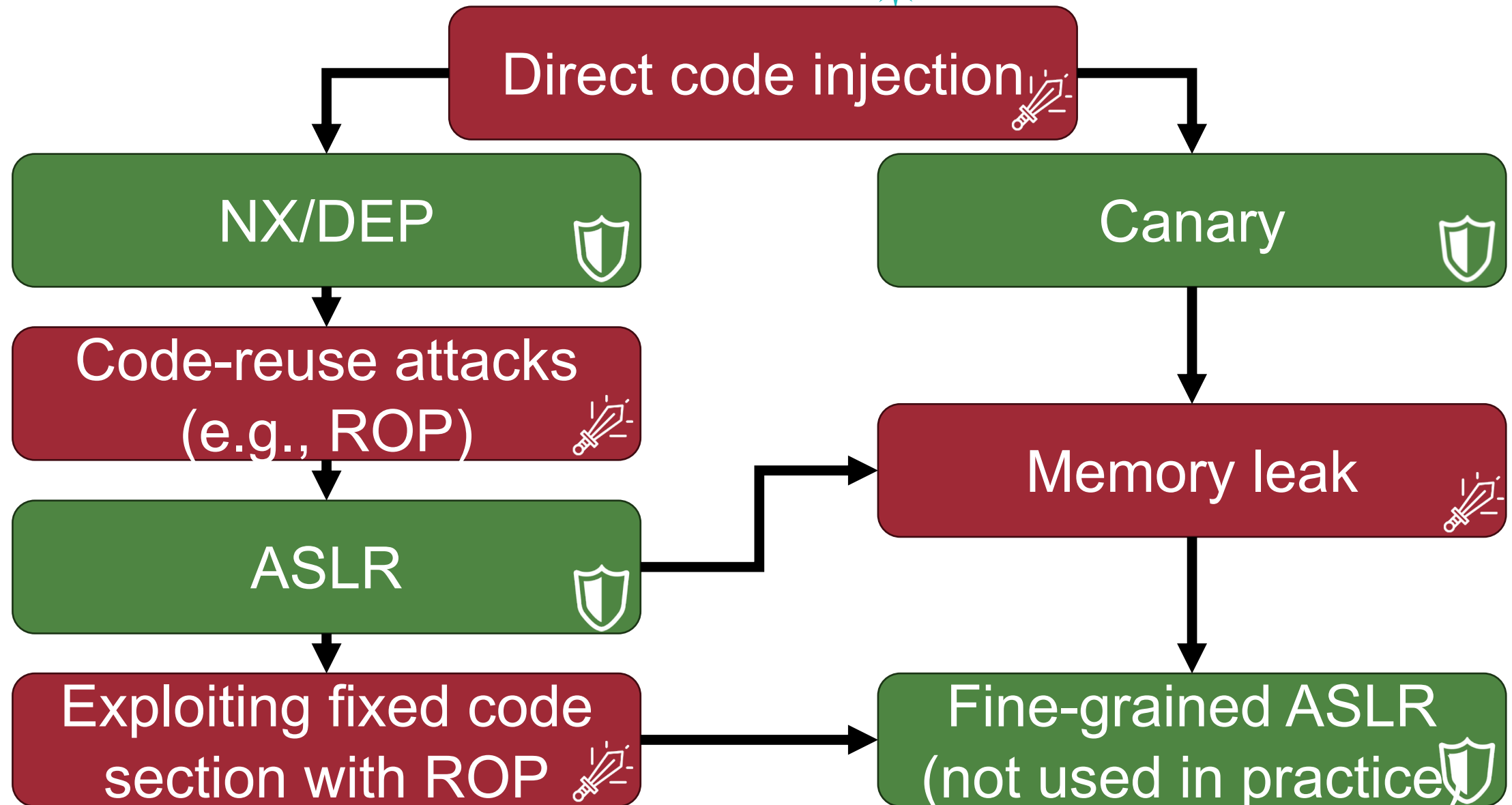
---



Check out the pdf: [https://websec-lab.github.io/courses/2023f-cse467/materials/secure\\_coding.pdf](https://websec-lab.github.io/courses/2023f-cse467/materials/secure_coding.pdf)

# Summary of Software Security

58





# 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?**