CS 576 – Systems Security
Introduction to C/C++ Vulnerabilities

Georgios (George) Portokalidis

Memory Corruption

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

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

--wikipedia

Common Vulnerabilities

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

Common Weakness Enumeration

A Community-Developed List of Software Weakness Types

CWE and SANS Institute TOP 25 MOST DANGEROUS SOFTWARE ERRORS

ID Lookup: Go

Home | About | CWE List | Scoring | Community | News | Search

CWE™ is a community-developed list of common software security weaknesses. It serves as a common language, a measuring stick for software security tools, and as a baseline for weakness identification, mitigation, and prevention efforts.

View the CWE List

View by Research Concepts | View by Development Concepts | View by Architectural Concepts

Search CWE

Easily find a specific software weakness by performing a search of the CWE List by keywords(s) or by CWE-ID Number. To search by multiple keywords, separate each by a space.

Google Custom Search Search

See the full CWE List page for enhanced information, downloads, and more.

Total Software Weaknesses: 714

Buffer Overflows

- Writing outside the boundaries of a buffer
  - Spatial violation
- Common programmer errors that lead to it …
  - Insufficient input checks/wrong assumptions about input
  - Unchecked buffer size
  - Integer overflows

Example

```
char buf[16];

strcpy(buf, str);

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

return strlen(buf);
```

BO Variations

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

        strcpy(buf, str);

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

        return 0;
}
```

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

        strcpy(buf, str);

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

        return 0;
}
```

```
char buf[16];

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

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

        return 0;
}
```

BO Variations

Stack Buffer Overflow

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

        strcpy(buf, str);

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

        return 0;
}
```

Heap Buffer Overflow

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

        strcpy(buf, str);

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

        return 0;
}
```

Global Buffer Overflow

```
char buf[16];

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

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

        return 0;
}
```

Buffer Overflows

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

Buffer Overflows

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

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

                *dp = *sp;
        }
        *dp = '\0’;
}
```

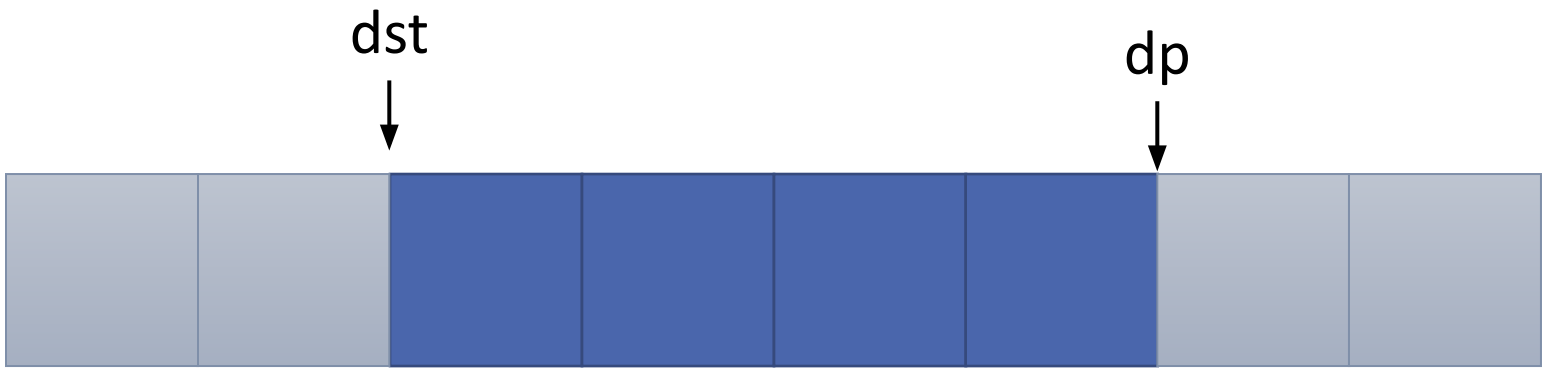
Buffer Underflows

- Writing outside the boundaries of a buffer
  - Spatial violation
- Common programmer errors that lead to it …
  - Insufficient input checks/wrong assumptions about input
  - Unchecked buffer size
  - Integer overflows
- Opposite direction than overflows
  - Otherwise, the same
- Less common

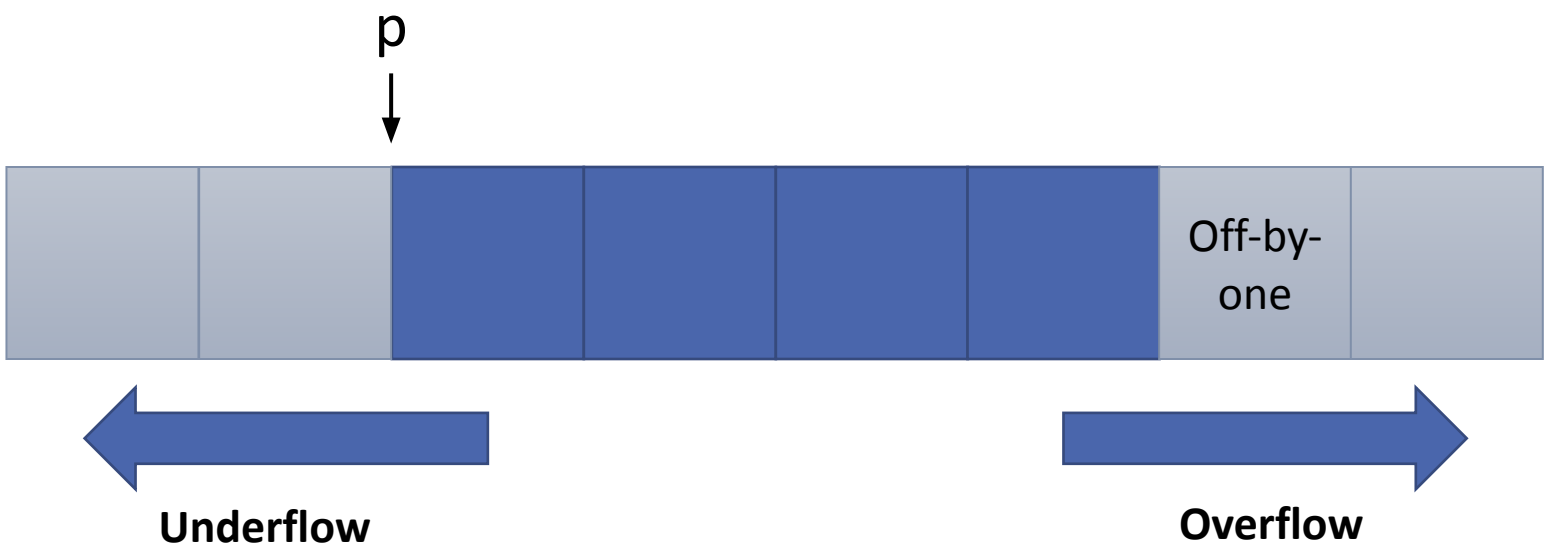
```
static void my_copy(char *dst, const char *src)
{
        char *dp, *sp;
        for (sp = src, dp = dst + strlen(dst) - 1;
             *sp != '\0'; sp++, dp--) {
                *dp = *sp;
        }
}
```

Off-by-One Bariation

- Writing outside the boundaries of a buffer by one byte
  - Spatial violation
- `dp = dst + strlen(dst) -1`

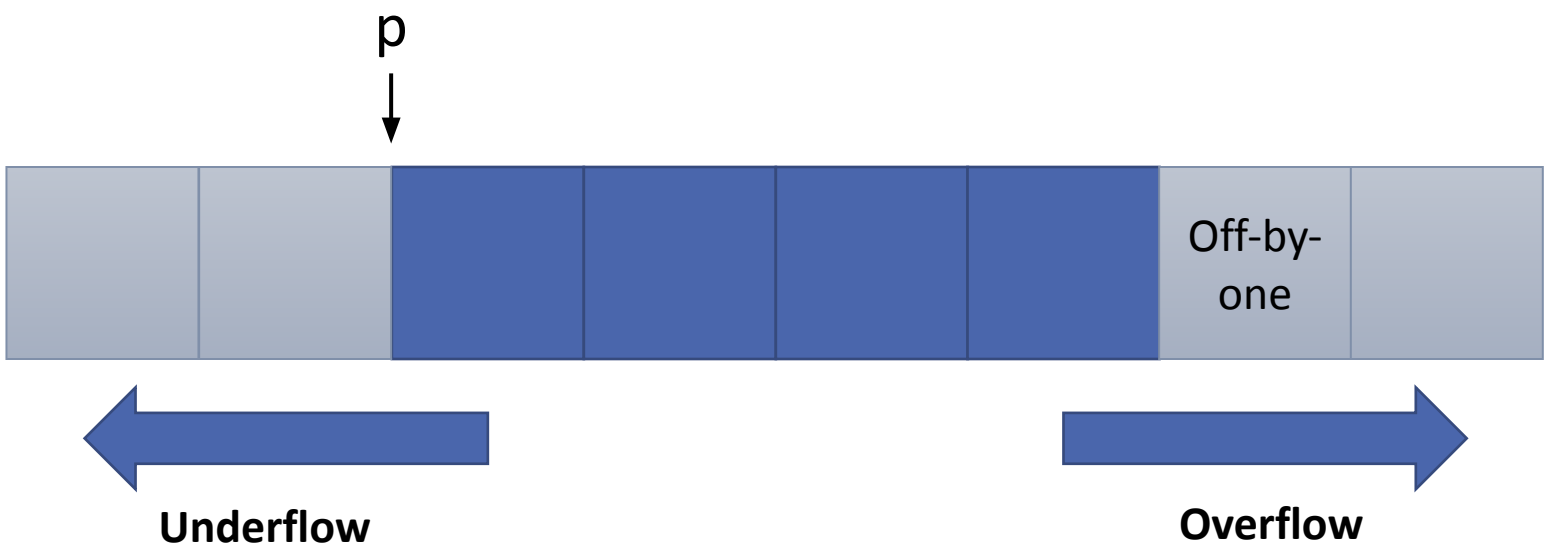


To Summarize



To Summarize

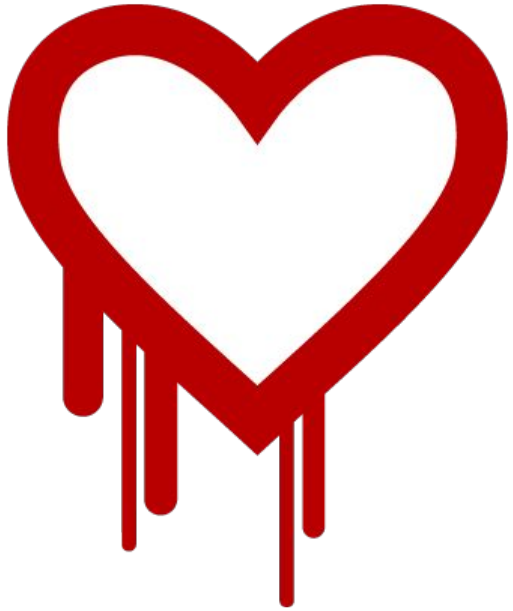
- Write □ over/under-write □ Corrupt neighboring memory areas
- Read □ over/under-read □ Leak data from neighboring memory areas

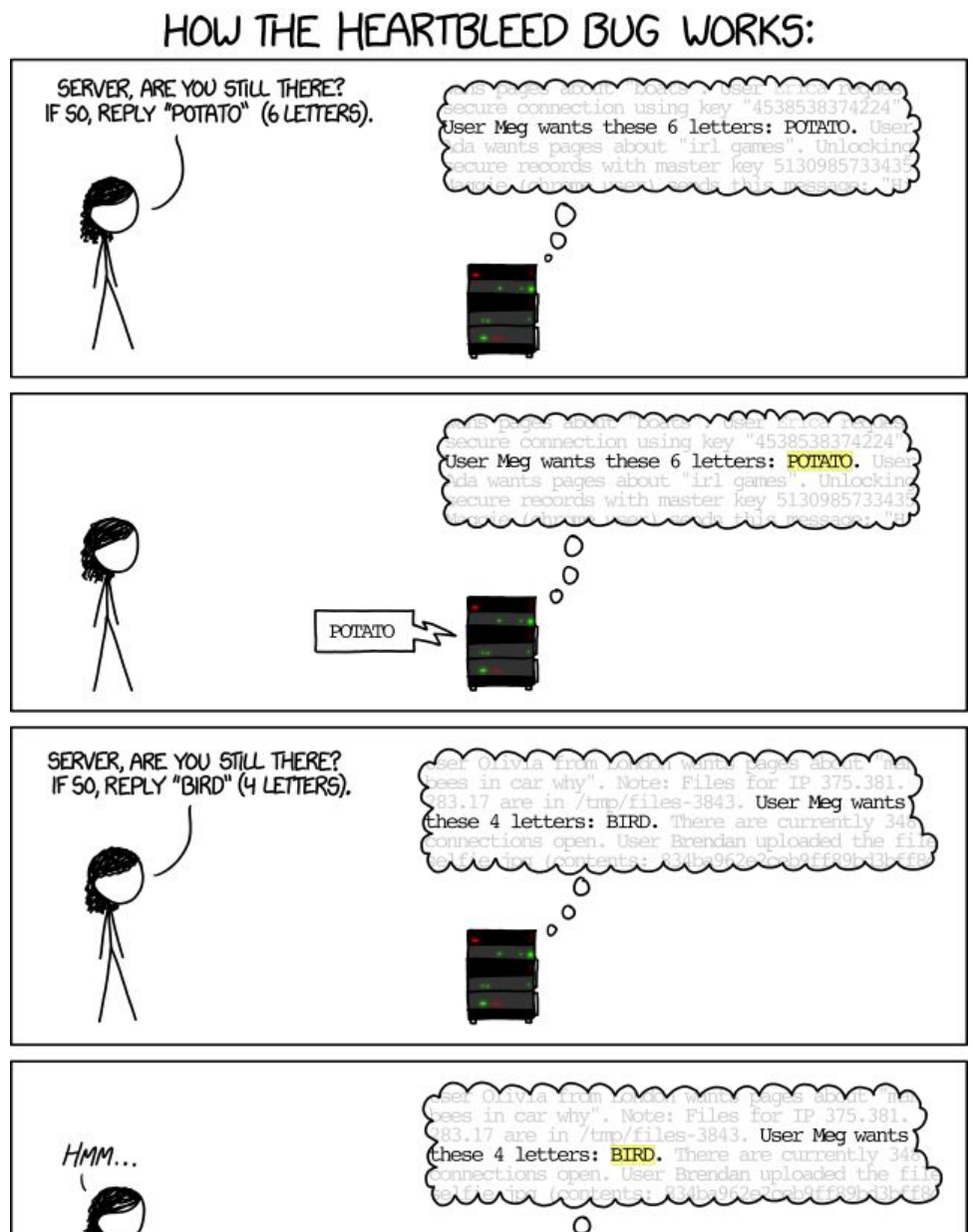


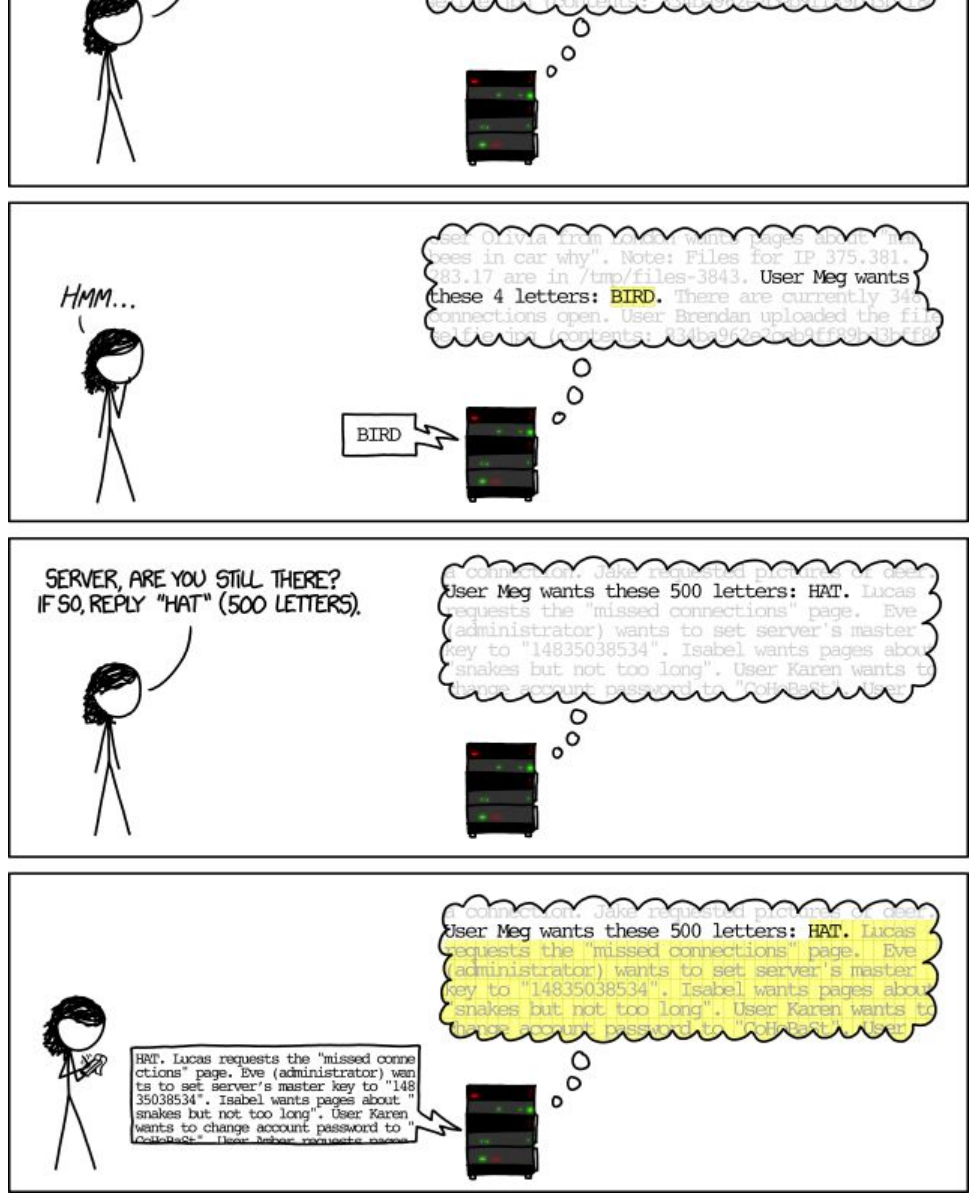
Are These Vulnerabilities

- If user input can trigger them **YES**
- Effects of over/under-writes:
  - Crash the application (DoS)
  - Take over the application
    - If remote Remote code execution
    - Otherwise Arbitrary code execution
  - Corrupt application state
- Effects of over/under-read:
  - Leak sensitive data

Overreads Are Serious Too: Heartbleed




HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
User Meg wants these 6 letters: POTATO.
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
these 4 letters: BIRD.
HMM...


HMM...
these 4 letters: BIRD.
BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT.
User Meg wants these 500 letters: HAT.
HAT. Lucas requests the "missed conne ctions" page. Eve (administrator) wan ts to set server's master key to "148 35038534". Isabel wants pages about " snakes but not too long". User Karen wants to change account password to "

Format String Bugs

- Exploits functions formatted output functions like printf, sprint, snprintf, etc.
- int **printf**(const char * restrict format, ...);
- printf is a variadic function a function which accepts a variable number of arguments
- Problem Evaluating input string as format string
- Other variadic functions may have similar problems

Format String Vulnerabilities: Examples

```
printf("%d %s\n", i, s);

char *fmt = "%d %s\n";
printf(fmt, i, s);
```

Not possible when format string is a constant

```
static void my_print(char *prefix, char *formatter, long num)
{
        char fmt[1000];
        strcpy(fmt, "%s: received number ");
        strcat(fmt, formatter);
        printf(fmt, prefix, num);
}
```

Format string needs to be controllable by input

What Is Possible?

- Format strings are very expressive!
- Controlling a format string allows to **read or write arbitrary data from memory**
- Same results as overflows
  - Writes
    - Crash the application (DoS)
    - Take over the application
      - If remote Remote code execution
      - Otherwise Arbitrary code execution
    - Corrupt application state
  - Reads
    - Leak sensitive data

Uninitialized Memory

- Simplest of errors
- Using a variable before initializing it with a value
- Modern compilers warn of this
  - Developers may ignore errors

```
void my_fync(void)
{
    int i;
    printf("%d\n", i);
    ...
}
```

What will be the value of *i*?

This is also classified as ”undefined behavior”

Whatever you guessed may be correct

Bugs or Vulnerabilities

- Value uninitialized variable could be stale program value more of a bug but all bugs can cause security issues under certain conditions
- Value could be stale value controlled by attacker definitely a problem
- Much bigger issue if what is controlled is a pointer

```
void my_fync(char *user_data)
{
    char *s;
    sprintf(s, "this is %s\n", user_data);
    ...
}
```

Where will the data be written?

```
void my_fync(char *user_data)
{
    int (*f)(char *);
    ...
    return f(user_data);
}
```

What happens here?

Not all bugs are the same, but worst outcome is similar as before

Null Pointer Dereferences

- Happens when a null (0) pointer is dereferenced
- Could happen when variable is uninitialized

```
void my_fync(char *user_data)
{
    char *s;
    sprintf(s, “this is %s\n”, user_data);
    ...
}
```

- Or when previously cleared variable is used

```
void my_fync(char *user_data)
{
    char *s = malloc(100);
    ...
    free(s); s = NULL;
    ...
    sprintf(s, “this is %s\n”, user_data);
}
```

What Is Possible?

- Depends what is at address 0
- Usually not allocated/mapped application crashes
- Sometimes not memory corruption
  - Similar exploitation scenarios as before become possible

Use-After-Free (UAF) Vulnerabilities

- Writing using a pointer no longer pointing to a valid buffer
  - Temporal violation
  - Such pointers are called dangling

```
void my_fync(char *user_data)
{
    char *s = malloc(100);
    ...
    free(s); s = NULL;
    ...
    sprintf(s, "this is %s\n", user_data);
}
```

Where will the data be written?

Risky Scenario

1. Program allocates objectA
2. Program frees objectA
3. Program allocates objectB reusing the memory previously allocated to objectA
4. User input is written into objectB
5. Program uses pointer to objectA (dangling)

```
int main(int argc, char **argv)
{
        struct objectA *objA;
        struct objectB *objB;


        objA = malloc(sizeof(struct object A));
        funcA(objA);  /* frees objA */
        objB = malloc(sizeof(struct object B));
       funcB(objhB) /* writes on objB */
        …
     funcAA(objA); /*accesses freed objA */
```

Risky Scenario

1. Program allocates objectA
2. Program frees objectA
3. Program allocates objectB reusing the memory previously allocated to objectA
4. User input is written into objectB
5. Program uses pointer to objectA (dangling)

```
struct objectA {
    void (*fptr)();
    char *string;
 }
```

```
struct objectB {
    …  …
    int a;
    long b;
    …
 }
```

```
int main(int argc, char **argv)
{
        struct objectA *objA;
        struct objectB *objB;


        objA = malloc(sizeof(struct object A));
        funcA(objA);  /* frees objA */
        objB = malloc(sizeof(struct object B));
       funcB(objhB) /* writes on objB */
        …
     funcAA(objA); /*accesses freed objA
  */
```

Risky Scenario

1. Program allocates objectA
2. Program frees objectA
3. Program allocates objectB reusing the memory previously allocated to objectA
4. User input is written into objectB
5. Program uses pointer to objectA (dangling)

```
struct objectA {
    void (*fptr)(); □ a
    char *string;
 }
```

```
struct objectB {
    …  …
    int a;
    long b;
    …
 }
```

```
int main(int argc, char **argv)
{
        struct objectA *objA;
        struct objectB *objB;


        objA = malloc(sizeof(struct object A));
        funcA(objA);  /* frees objA */
        objB = malloc(sizeof(struct object B));
       funcB(objhB) /* writes on objB */
        …
     funcAA(objA); /*accesses freed objA
  */
```

UAF Vulnerabilities

- Serious and hard to discover/defend against
- Become more complex due to threading/concurrency
- Severity depends on the type of the object the dangling pointer points-to
  - As severe as other memory corruption errors

Type Confusion

- An object is accessed using a pointer of the wrong type

```
class ClassA {
…
void (*fptr)();
void func();
};
```

```
class ClassB {
…
int user_data;
void func();
}
```

```
int main(int argc, char **argv)
{
        ClassA *a  = new ClassA()
        ClassB *b = new ClassB();

        func('A', a);
        func('A', b);
```

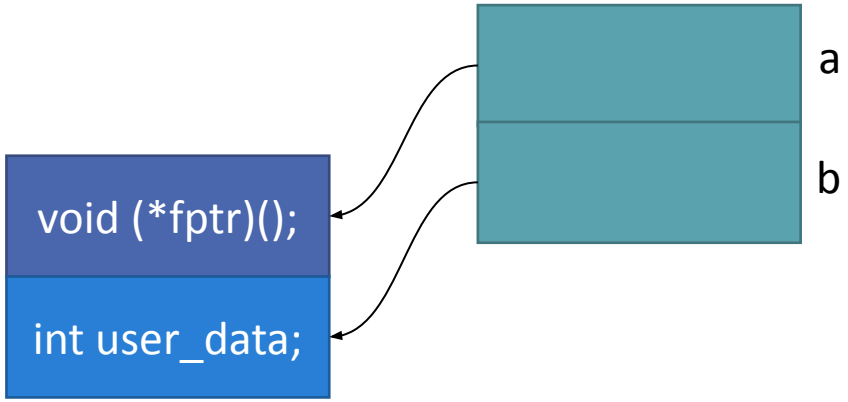
```
void func(char type, void *cptr)
{
    if (type == 'A')
       ((ClassA *)cptr))->func();
    else (if type == 'B')
        ((ClassB *)cptr))->func();
```

Type Confusion

- An object is accessed using a pointer of the wrong type

```
class ClassA {
…
void (*fptr)();
void func();
};
```

```
class ClassB {
…
int user_data;
void func();
}
```




```
int main(int argc, char **argv)
{
        ClassA *a  = new ClassA()
        ClassB *b = new ClassB();

        func('A', a);
        func('A', b);
```

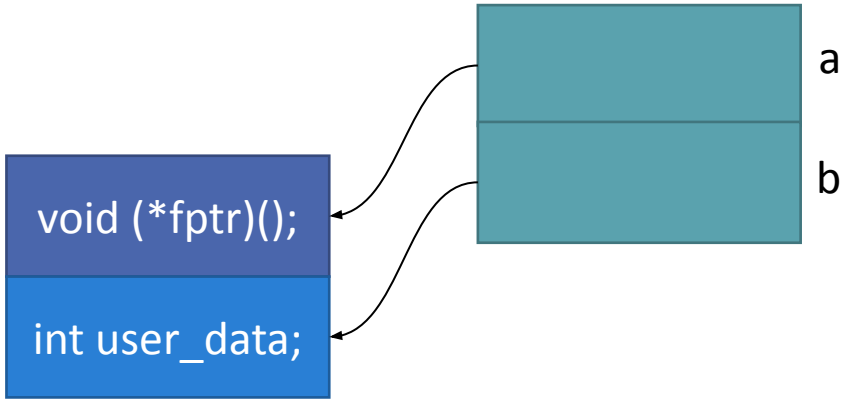
```
void func(char type, void *cptr)
{
    if (type == 'A')
       ((ClassA *)cptr))->func();
    else (if type == 'B')
        ((ClassB *)cptr))->func();
```

Type Confusion

- An object is accessed using a pointer of the wrong type

```
class ClassA {
…
void (*fptr)();
void func();
};
```

```
class ClassB {
…
int user_data;
void func();
}
```




```
int main(int argc, char **argv)
{
        ClassA *a  = new ClassA()
        ClassB *b = new ClassB();

        func('A', a);
        func('A', b);
```

```
void func(char type, void *cptr)
{
    if (type == 'A')
        ((ClassA *)cptr))->func();
    else (if type == 'B')
        ((ClassB *)cptr))->func();
```

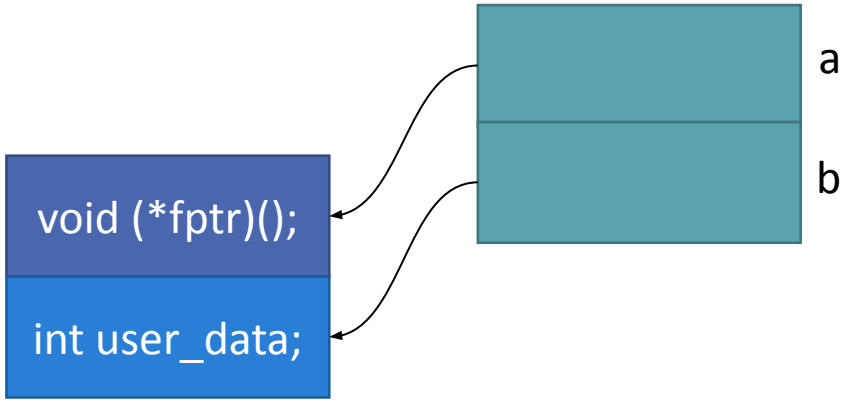
ClassA::func(this)

Type Confusion

- An object is accessed using a pointer of the wrong type

```
class ClassA {
…
void (*fptr)();
void func();
};
```

```
class ClassB {
…
int user_data;
void func();
}
```




```
int main(int argc, char **argv)
{
        ClassA *a  = new ClassA()
        ClassB *b = new ClassB();

        func('A', a);
        func('A', b);
```

```
void func(char type, void *cptr)
{
    if (type == 'A')
        ((ClassA *)cptr))->func();
    else (if type == 'B')
        ((ClassB *)cptr))->func();
```

this = cptr

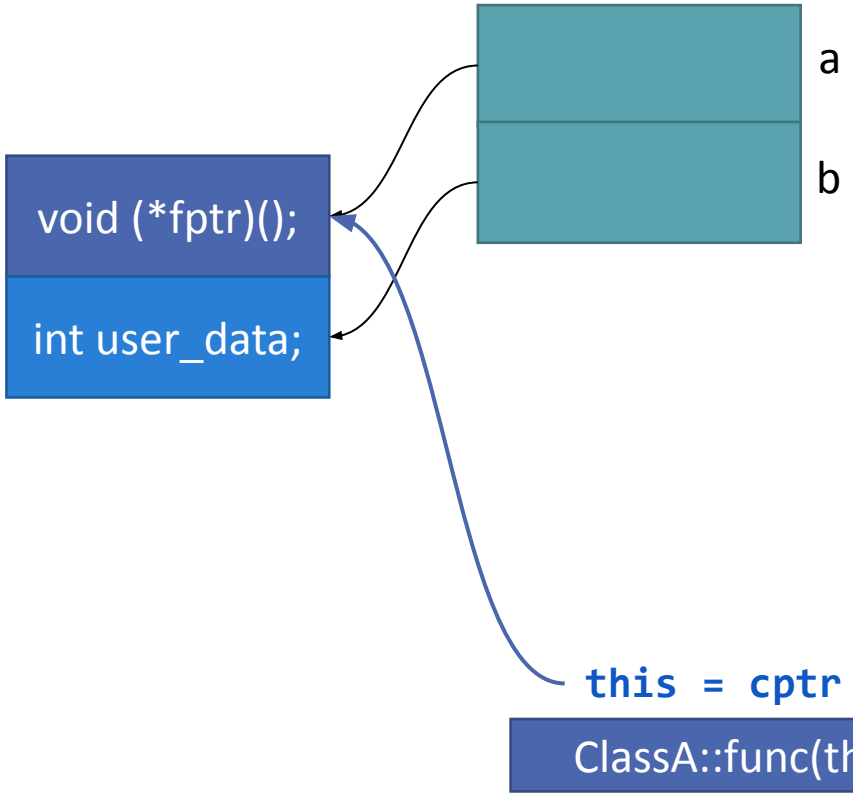
ClassA::func(this)

Type Confusion

- An object is accessed using a pointer of the wrong type

```
class ClassA {
…
void (*fptr)();
void func();
};
```

```
class ClassB {
…
int user_data;
void func();
}
```




```
int main(int argc, char **argv)
{
        ClassA *a  = new ClassA()
        ClassB *b = new ClassB();

        func('A', a);
        func('A', b);
```

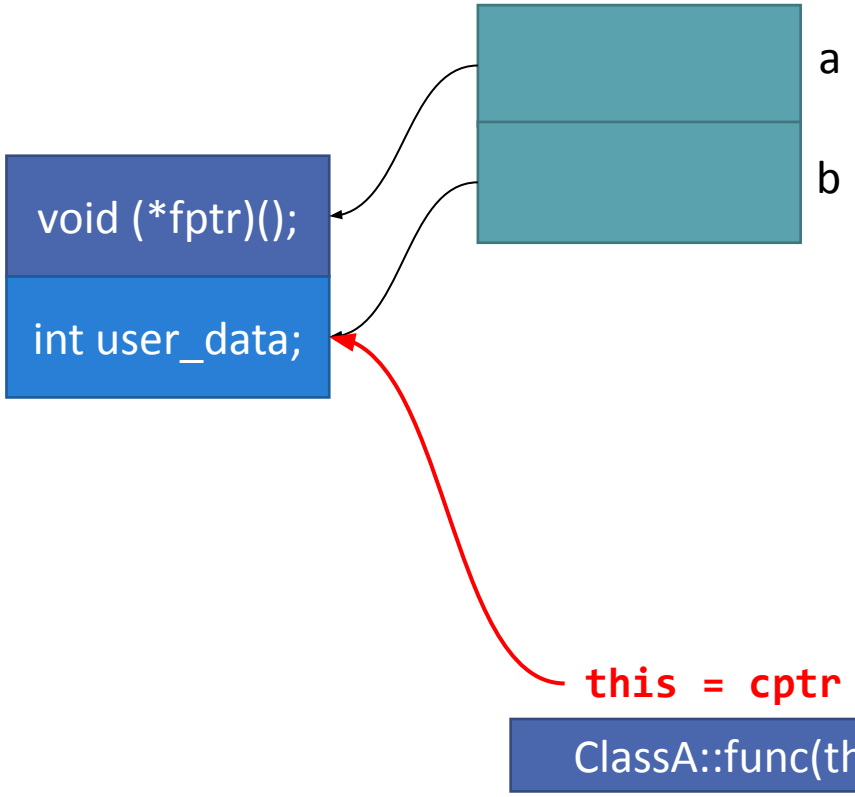
```
void func(char type, void *cptr)
{
    if (type == 'A')
        ((ClassA *)cptr))->func();
    else (if type == 'B')
        ((ClassB *)cptr))->func();
```

Type Confusion

- An object is accessed using a pointer of the wrong type

```
class ClassA {
…
void (*fptr)();
void func();
};
```

```
class ClassB {
…
int user_data;
void func();
}
```




```
int main(int argc, char **argv)
{
        ClassA *a  = new ClassA()
        ClassB *b = new ClassB();

        func('A', a);
        func('A', b);
```

```
void func(char type, void *cptr)
{
    if (type == 'A')
        ((ClassA *)cptr))->func();
    else (if type == 'B')
        ((ClassB *)cptr))->func();
```