Please check your attendance using Blackboard!

Lecture 3 – Memory Safety

[COSE451] Software Security

Instructor: Seunghoon Woo

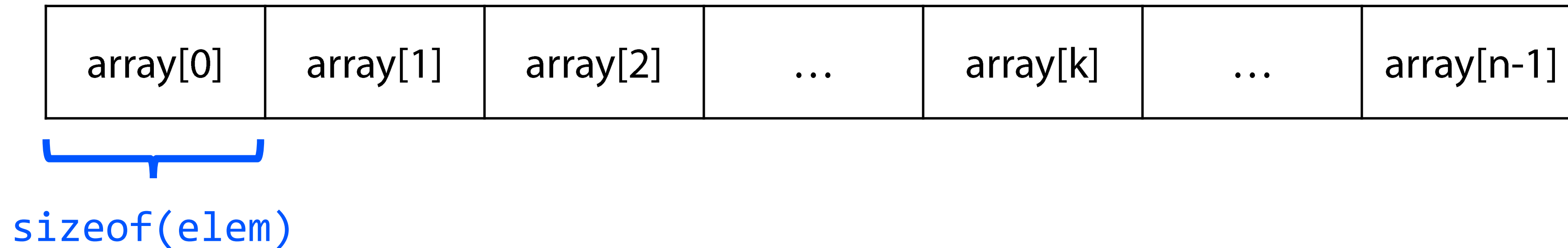
Spring 2024

Overview

- **Out-of-bounds (OOB) vulnerabilities**
- **Defense mechanism against buffer overflow / OOB**

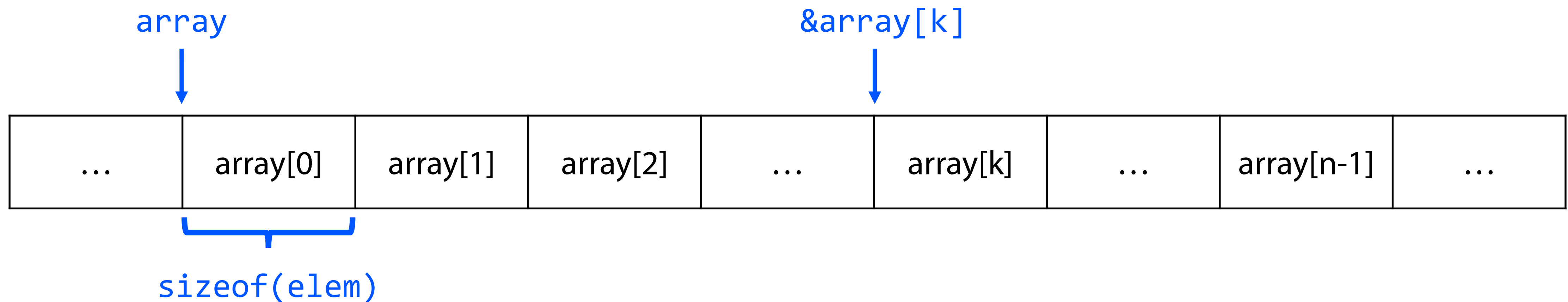
Out-of-bounds (OOB) vulnerabilities

- **Consider an array**
  - Length of array: n
  - Size of Array: `sizeof(elem) * n`



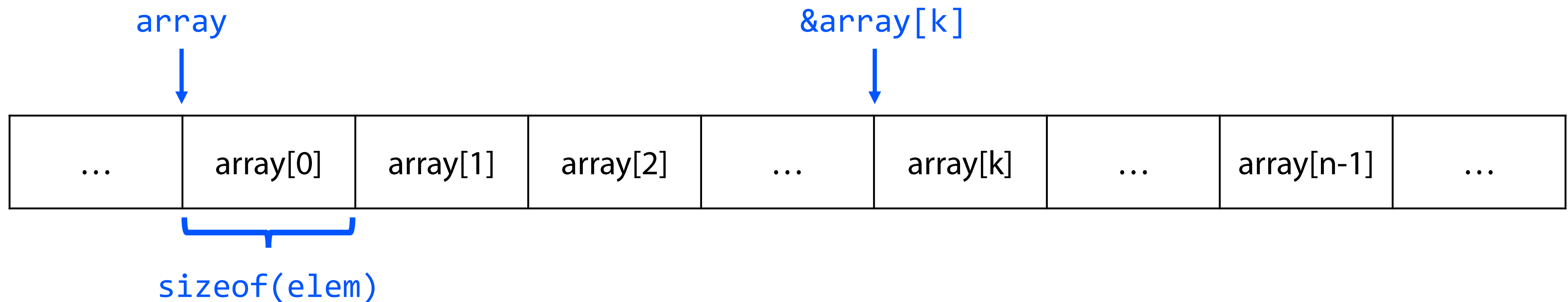
Out-of-bounds (OOB) vulnerabilities

- **Address of each element of array**
  - Calculate using array address, element index, and element data type size



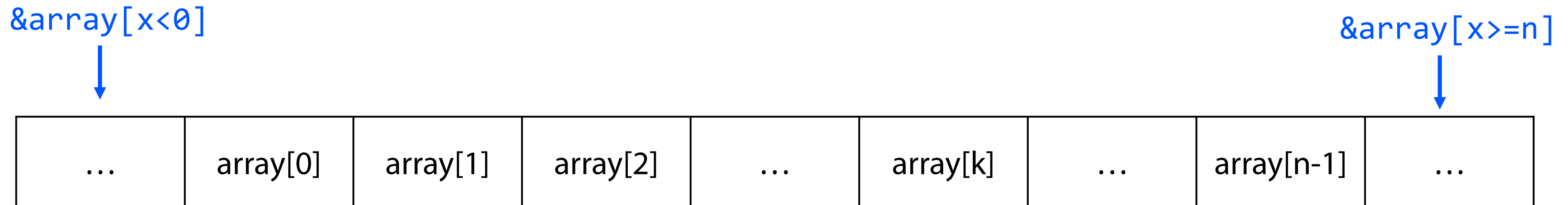
Out-of-bounds (OOB) vulnerabilities

- **Address of each element of array**
  - Calculate using array address, element index, and element data type size
    - `&array[k] = array + sizeof(elem) * k`



Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds**
  - OOB occurs when the index value is negative or outside the length of the array when referring to an element
    - `&array[x] = array + sizeof(elem) * x`



Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds**
  - Example

```c
#include <stdio.h>

int main() {
  int arr[10];

  printf("In Bound: \n");
  printf("arr: %p\n", arr);
  printf("arr[0]: %p\n\n", &arr[0]);

  printf("Out of Bounds: \n");
  printf("arr[-1]: %p\n", &arr[-1]);
  printf("arr[100]: %p\n", &arr[100]);

  return 0;
}
```

https://dreamhack.io/

Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds**
  - Example

```
In Bound:
arr: 0x7ffc8ea71040
arr[0]: 0x7ffc8ea71040

Out of Bounds:
arr[-1]: 0x7ffc8ea7103c
arr[100]: 0x7ffc8ea711d0
```

```c
1  #include <stdio.h>
2  
3  int main() {
4    int arr[10];
5  
6    printf("In Bound: \n");
7    printf("arr: %p\n", arr);
8    printf("arr[0]: %p\n\n", &arr[0]);
9  
10   printf("Out of Bounds: \n");
11   printf("arr[-1]: %p\n", &arr[-1]);
12   printf("arr[100]: %p\n", &arr[100]);
13 
14   return 0;
15 }
```

Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds**
  - Example

```
In Bound:
arr: 0x7ffc8ea71040
arr[0]: 0x7ffc8ea71040

Out of Bounds:
arr[-1]: 0x7ffc8ea7103c
arr[100]: 0x7ffc8ea711d0
```

- The compiler does not issue any warning even though -1 and 100 are used as indices!

```
1  #include <stdio.h>
2
3  int main() {
4    int arr[10];
5
6    printf("In Bound: \n");
7    printf("arr: %p\n", arr);
8    printf("arr[0]: %p\n\n", &arr[0]);
9
10   printf("Out of Bounds: \n");
11   printf("arr[-1]: %p\n", &arr[-1]);
12   printf("arr[100]: %p\n", &arr[100]);
13
14   return 0;
15 }
```

Out-of-bounds (OOB) vulnerabilities

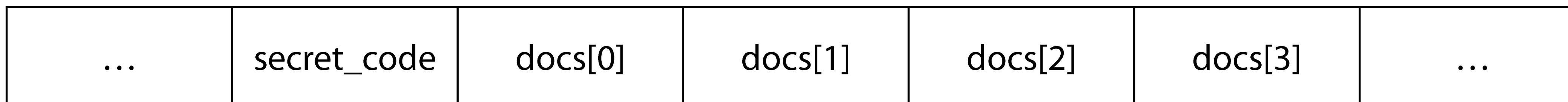
- **Out-of-bounds READ**
  - Example

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

int main() {
  char *docs[] = {"DATA1", "DATA2", "DATA3", "DATA4"};
  char *secret_code = "SECRET DATA";
  int idx;

  // Exploit OOB to print the secret
  puts("What do you want to read?");
  for (int i = 0; i < 4; i++) {
    printf("%d. %s\n", i + 1, docs[i]);
  }
  printf("> ");
  scanf("%d", &idx);

  if (idx > 4) {
    printf("Detect out-of-bounds");
    exit(-1);
  }

  puts(docs[idx - 1]);
  return 0;
}
```

Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds READ**
  - Example

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main() {
6    char *docs[] = {"DATA1", "DATA2", "DATA3", "DATA4"};
7    char *secret_code = "SECRET DATA";
8    int idx;
9
10   // Exploit OOB to print the secret
11   puts("What do you want to read?");
12   for (int i = 0; i < 4; i++) {
13     printf("%d. %s\n", i + 1, docs[i]);
14   }
15   printf("> ");
16   scanf("%d", &idx);
17
18   if (idx > 4) {
19     printf("Detect out-of-bounds");
20     exit(-1);
21   }
22
23   puts(docs[idx - 1]);
24   return 0;
25 }
```

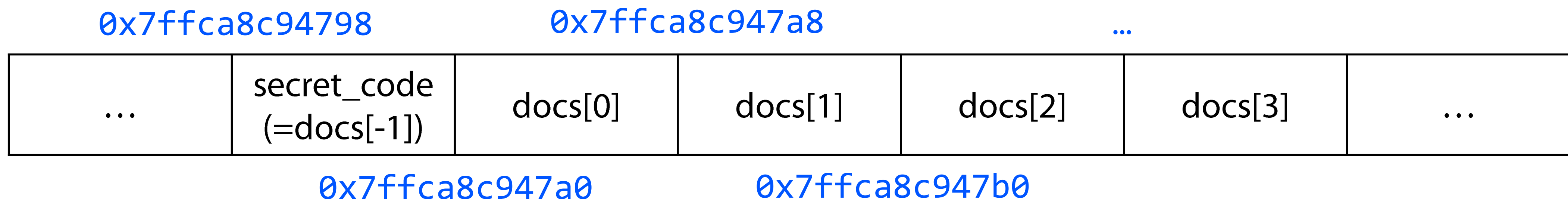


Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds READ**
  - Example

```
What do you want to read?
1. DATA1
2. DATA2
3. DATA3
4. DATA4
> 1
DATA1
Address of docs: 0x7ffca8c947a0
Address of docs[1]: 0x7ffca8c947a8
Address of docs[2]: 0x7ffca8c947b0
Address of docs[-1]: 0x7ffca8c94798
Address of docs[-2]: 0x7ffca8c94790
Address of docs[-3]: 0x7ffca8c94788
Address of secret_code pointer: 0x7ffca8c94798
```

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main() {
6    char *docs[] = {"DATA1", "DATA2", "DATA3", "DATA4"};
7    char *secret_code = "SECRET DATA";
8    int idx;
9
10   // Exploit OOB to print the secret
11   puts("What do you want to read?");
12   for (int i = 0; i < 4; i++) {
13     printf("%d. %s\n", i + 1, docs[i]);
14   }
15   printf("> ");
16   scanf("%d", &idx);
17
18   if (idx > 4) {
19     printf("Detect out-of-bounds");
20     exit(-1);
21   }
22
23   puts(docs[idx - 1]);
24   return 0;
25 }
```

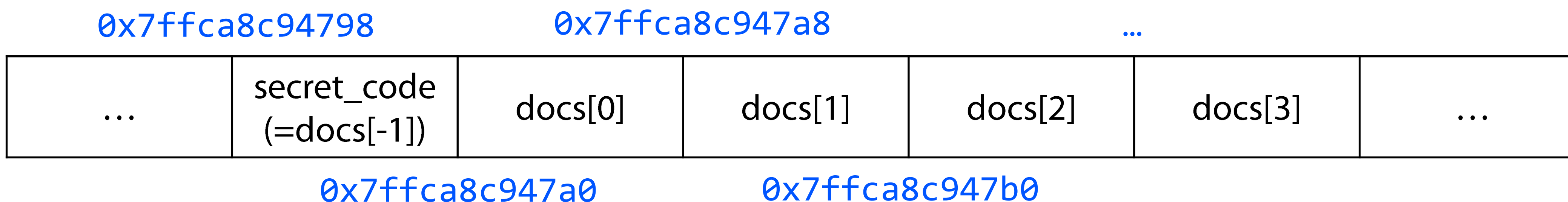


Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds READ**
  - Example

```
What do you want to read?
1. DATA1
2. DATA2
3. DATA3
4. DATA4
> 0
SECRET DATA
```

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

int main() {
  char *docs[] = {"DATA1", "DATA2", "DATA3", "DATA4"};
  char *secret_code = "SECRET DATA";
  int idx;

  // Exploit OOB to print the secret
  puts("What do you want to read?");
  for (int i = 0; i < 4; i++) {
    printf("%d. %s\n", i + 1, docs[i]);
  }
  printf("> ");
  scanf("%d", &idx);

  if (idx > 4) {
    printf("Detect out-of-bounds");
    exit(-1);
  }

  puts(docs[idx - 1]);
  return 0;
}
```



Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds WRITE**
  - Example

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

struct Student {
  long attending;
  char *name;
  long age;
};

struct Student stu[10];
int isAdmin;

int main() {
  unsigned int idx;

  // Exploit OOB to read the secret
  puts("Who is present?");
  printf("(1-10)> ");
  scanf("%u", &idx);

  stu[idx - 1].attending = 1;

  if (isAdmin) printf("Access granted.\n");
  return 0;
}
```

Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds WRITE**
  - Example
    - `Student` : 24bytes (in 64bit)
    - `Stu` : 10 x 24bytes (in 64bit)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Student {
5    long attending;
6    char *name;
7    long age;
8  };
9
10 struct Student stu[10];
11 int isAdmin;
12
13 int main() {
14   unsigned int idx;
15
16   // Exploit OOB to read the secret
17   puts("Who is present?");
18   printf("(1-10)> ");
19   scanf("%u", &idx);
20
21   stu[idx - 1].attending = 1;
22
23   if (isAdmin) printf("Access granted.\n");
24   return 0;
25 }
```

Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds WRITE**
  - Example
    - `Student` : 24bytes (in 64bit)
    - `Stu` : 10 x 24bytes (in 64bit)
    - Address of `isAdmin` = Address of `stu` + 240 bytes

```
pwndbg> i var isAdmin
All variables matching regular expression "isAdmin":

Non-debugging symbols:
0x0000000000004130  isAdmin
pwndbg> i var stu
All variables matching regular expression "stu":

Non-debugging symbols:
0x0000000000004040  stu
pwndbg> print 0x4130-0x4040
$1 = 240
```

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Student {
5    long attending;
6    char *name;
7    long age;
8  };
9
10 struct Student stu[10];
11 int isAdmin;
12
13 int main() {
14   unsigned int idx;
15
16   // Exploit OOB to read the secret
17   puts("Who is present?");
18   printf("(1-10)> ");
19   scanf("%u", &idx);
20
21   stu[idx - 1].attending = 1;
22
23   if (isAdmin) printf("Access granted.\n");
24   return 0;
25 }
```

Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds WRITE**
  - Example
    - `Student` : 24bytes (in 64bit)
    - `Stu` : 10 x 24bytes (in 64bit)
    - Address of `isAdmin` = Address of `stu` + 240 bytes
      - If we refer to the index 10 of `stu` (11th index), we can manipulate `isAdmin`

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Student {
5    long attending;
6    char *name;
7    long age;
8  };
9
10 struct Student stu[10];
11 int isAdmin;
12
13 int main() {
14   unsigned int idx;
15
16   // Exploit OOB to read the secret
17   puts("Who is present?");
18   printf("(1-10)> ");
19   scanf("%u", &idx);
20
21   stu[idx - 1].attending = 1;
22
23   if (isAdmin) printf("Access granted.\n");
24   return 0;
25 }
```

Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds WRITE**
  - Example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Student {
5    long attending;
6    char *name;
7    long age;
8  };
9
10 struct Student stu[10];
11 int isAdmin;
12
13 int main() {
14   unsigned int idx;
15
16   // Exploit OOB to read the secret
17   puts("Who is present?");
18   printf("(1-10)> ");
19   scanf("%u", &idx);
20
21   stu[idx - 1].attending = 1;
22
23   if (isAdmin) printf("Access granted.\n");
24   return 0;
25 }
```

Out-of-bounds (OOB) vulnerabilities

- **Out-of-bounds WRITE**
  - Example

```
seunghoonwoo@seunghoonwoo-virtual-machine:~$ ./oob_write_ex
Who is present?
(1-10)> 11
Access granted.
```

```
pwndbg> print (int)isAdmin
$1 = 1
```

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Student {
5      long attending;
6      char *name;
7      long age;
8  };
9
10 struct Student stu[10];
11 int isAdmin;
12
13 int main() {
14     unsigned int idx;
15
16     // Exploit OOB to read the secret
17     puts("Who is present?");
18     printf("(1-10)> ");
19     scanf("%u", &idx);
20
21     stu[idx - 1].attending = 1;
22
23     if (isAdmin) printf("Access granted.\n");
24     return 0;
25 }
```

Out-of-bounds (OOB) vulnerabilities

2023 CWE Top 25 Most Dangerous Software Weaknesses

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

1 Out-of-bounds Write
CWE-787 | CVEs in KEV: 70 | Rank Last Year: 1

2 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-79 | CVEs in KEV: 4 | Rank Last Year: 2

3 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-89 | CVEs in KEV: 6 | Rank Last Year: 3

4 Use After Free
CWE-416 | CVEs in KEV: 44 | Rank Last Year: 7 (up 3) ▲

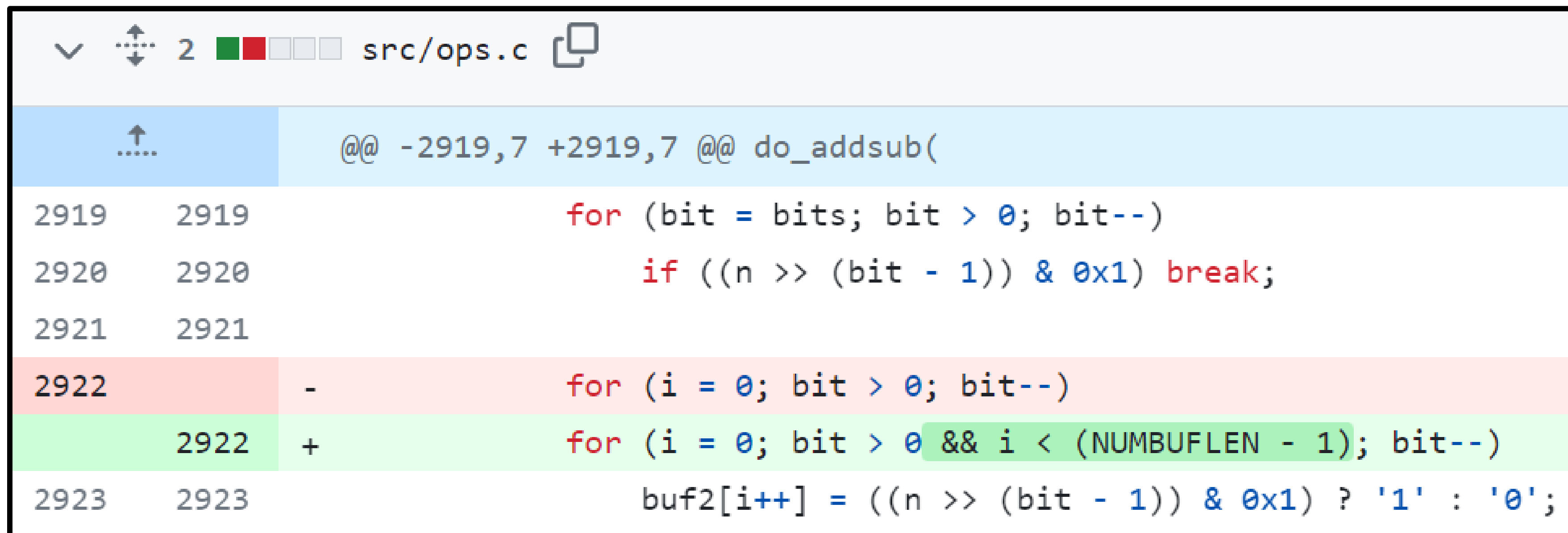
5 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-78 | CVEs in KEV: 23 | Rank Last Year: 6 (up 1) ▲

6 Improper Input Validation
CWE-20 | CVEs in KEV: 35 | Rank Last Year: 4 (down 2) ▼

7 Out-of-bounds Read
CWE-125 | CVEs in KEV: 2 | Rank Last Year: 5 (down 2) ▼

Out-of-bounds (OOB) vulnerabilities

- **Real-world out-of-bounds examples**
  - CVE-2023-4735 (discovered in VIM)



Memory safety

- **How to defense overflow attacks / out-of-bounds vulnerabilities?**
  - One of the most effective way is to check the inputs / the size condition of buffers

```
1  #include <stdio.h>
2  int main() {
3    int buf[0x10];
4    unsigned int index;
5
6    scanf("%d", &index);
7
8    [A]
9
10   printf("%d\n", buf[index]);
11   return 0;
12 }
```

Memory safety

- **How to defense overflow attacks?**
  - Choosing a programming language that is relatively safe in memory management
    - C/C++: Dangerous
    - Java/C#/Python: Safe
    - However, C/C++ have significant benefits in memory optimization and performance
      - It is important to choose the language based on the intended use case

Memory safety

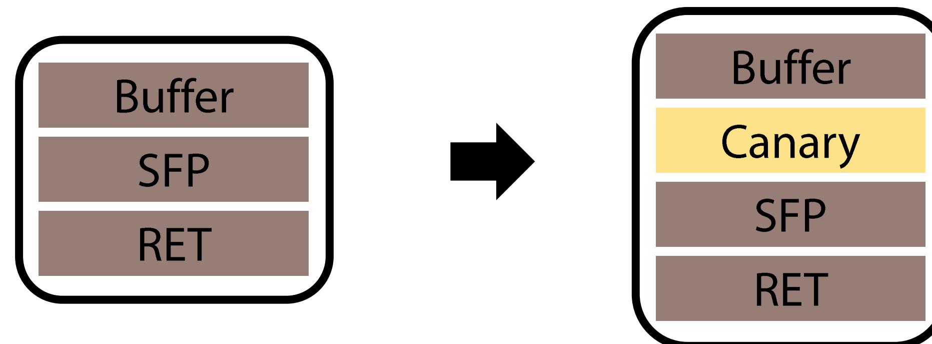
- **How to defense overflow attacks?**
  - Secure-coding / vulnerability detection
    - Avoiding the use of risky functions (e.g., `strcpy`)
    - Proactively utilizing exception handling statements
    - Using static/dynamic analysis tools

Memory safety

- **How to defense overflow attacks?**
  - The root cause of the problem
    1. The return address (RET) could be covered with a random address
    2. The address of the buffer where the user could input data was known
    3. The buffer was executable

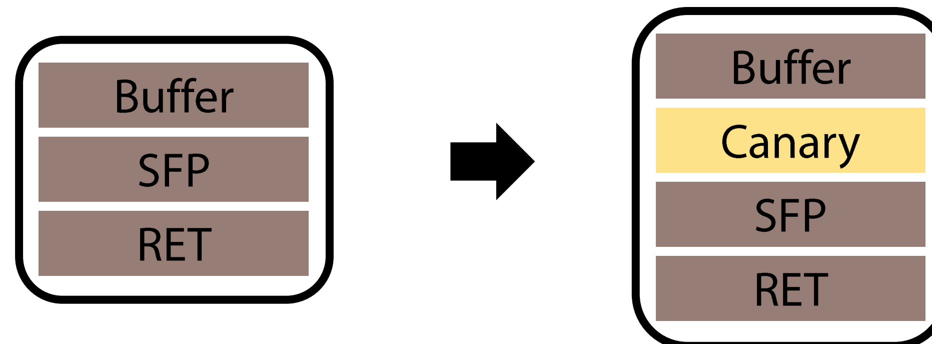
Memory safety

- **How to defense overflow attacks?**
  - Using stack protection mechanisms
    - **CANARY**
      - Insert random data between the buffer and SFP to detect buffer overflow



Memory safety

- **How to defense overflow attacks?**
  - Using stack protection mechanisms
    - **CANARY**
      - When a buffer overflow occurs (e.g., modifying RET), the canary value is also changed
      - This allows the identification of buffer overflow attacks



Memory safety

- **How to defense overflow attacks?**
  - Using stack protection mechanisms
    - **CANARY**
      - Because the attacker should not predict the canary, random values are used
      - There are also disadvantages in that the system to be protected must be recompiled and stack frame analysis becomes more complicated

Memory safety

- **How to defense overflow attacks?**
  - Using Address Space Layout Randomize (ASLR)
    - Randomly changes memory addresses when running a program
    - Attacker needs to decide where to place executable code
      - Prevents buffer overflows by making it difficult for attackers to identify memory addresses

```
1  $ gcc addr.c -o addr -ldl -no-pie -fno-PIE
2
3  $ ./addr
4  buf_stack addr: 0x7ffcd3fcffc0
5  buf_heap addr: 0xb97260
6  libc_base addr: 0x7fd7504cd000
7  printf addr: 0x7fd750531f00
8  main addr: 0x400667
9  $ ./addr
10 buf_stack addr: 0x7ffe4c661f90
11 buf_heap addr: 0x176d260
12 libc_base addr: 0x7ffad9e1b000
13 printf addr: 0x7ffad9e7ff00
14 main addr: 0x400667
15 $ ./addr
16 buf_stack addr: 0x7ffcf2386d80
17 buf_heap addr: 0x840260
18 libc_base addr: 0x7fed2664b000
19 printf addr: 0x7fed266aff00
20 main addr: 0x400667
```

Memory safety

- **How to defense overflow attacks?**
  - Using NX-Bit (Never eXecute Bit)
    - Also known as XD (eXecute disable) or DEP (Data Execution Prevention)
    - A hardware-based security feature implemented in modern computer processors
    - The Processor distinguishes between executable and non-executable areas of memory
      - All memory regions designated with the NX-Bit are used for storage and cannot be executed
    - Even if attackers inject malicious code into the memory, the processor prevents it from being executed as instructions

Memory safety

- **How to defense overflow attacks?**
  - Such protection mechanisms are applied in recent versions of Ubuntu

```
seunghoonwoo@seunghoonwoo-virtual-machine:~$ ./overflow_guarded
BADINPUTBADINPUT
Buffer1: str1(START), str2(BADINPUTBADINPUT), valid(0)
*** stack smashing detected ***: terminated
```

Next Lecture

- **Memory safety: memory leak, use after free, double free**
- **Access controls**