CYBER
CHALLENGE
CyberChallenge.IT

The
Roman
Xpl0it

A gentle introduction to the art of exploitation

Part 2a : Common Vulnerabilities

by anticlockwise & malweisse

What can possibly go wrong?

What can possibly go wrong?

EVERYTHING

What can possibly go wrong?

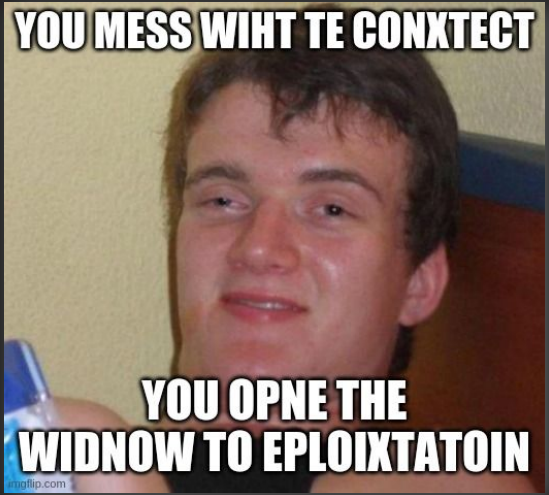
```
0x58 0x33 0x58 0x30
```

Any time you mess with the context you open the window to exploitation

What can possibly go wrong?

```
0x58 0x33 0x58 0x30
```

Any time you mess with the context you open the window to exploitation


YOU MESS WIHT TE CONXTECT
YOU OPNE THE WIDNOW TO EPLOIXTATOIN
imgflip.com

WTF does this mean?
Give me examples bro !!11!1

Lots of examples mate

- Integers Overflow
- Type Confusions
- Writes out of bound
- Reads out of bound
- Uninitialized Memory
- ...Leet Tricks

Integer overflows/underflows - 1

- Numeric values have a fixed size
- Math operations could wrap around the limit values and produce unexpected behavior

```
unsigned short n = width * height;
char *buf = (char *)malloc(n);
for (i=0; i< height; i++)
    memcpy(&buf[i*width], rows[i], width);
```

Integer overflows/underflows - 1

- Numeric values have a fixed size
- Math operations could wrap around the limit values and produce unexpected behavior

```
unsigned short n = width * height;
char *buf = (char *)malloc(1);
for (i=0; i< height; i++)
    memcpy(&buf[i*width], rows[i], width);
```

```
0xffff * 0xffff = 0xfffe0001
```

Integer overflows/underflows - 2

- Numeric values have a fixed size
- Math operations could wrap around the limit values and produce unexpected behavior

```
int length = input();
char buf[1024] = {0};
if(length < 0 || length + 1 >= 1024){
    die("bad length: %d", value);
}
memcpy(buf, input, length);
```

Integer overflows/underflows - 2

- Numeric values have a fixed size
- Math operations could wrap around the limit values and produce unexpected behavior

```
0x7fffffff + 0x1 = 0x80000000 = -1
```

```c
int length = input();
char buf[1024] = {0};
if(length < 0 || length + 1 >= 1024){
    die("bad length: %d", value);
}
memcpy(buf, input, length);
```

Type confusion

- **signed/unsigned** confusion in one of the most common type confusion

```
int length = input();
char buf[1024] = {0};
if(length >= 1024){
    die("bad length: %d", value);
}
memcpy(buf, input, length);
```

Type confusion

- **signed/unsigned** confusion in one of the most common type confusion

```
int length = input();
char buf[1024] = {0};
if(length >= 1024){
    die("bad length: %d", value);
}
memcpy(buf, input, length);
```

-1

-1 => 0xffffffff

Writes out of bound - 1

- **buffer overflows**: data copied in a location exceeds the size of the reserved destination area

```
strcpy(char *dest, const char *src);
strcat(char *dest, const char *src);
sprintf(char* str, "%s", char* message);
gets(char *s);
scanf("%s", buffer);
```

Writes out of bound - 2

- **buffer overflows**: data copied in a location exceeds the size of the reserved destination area
- **off-by-one errors**: length calculation incorrect by one array element

```
char buf[1024];
if(strlen(input) > sizeof(buf))
    die("error: user string too long\n");
strcpy(buf, input);
```

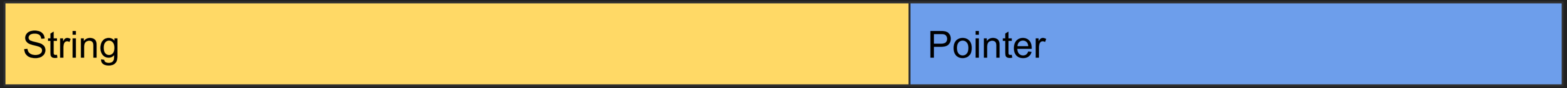
Reads out of bound

- Just like writing out of bound, reading out of bound is a ~~serious~~ useful vulnerability
- Disclose randomized data:
  1. Leak libc addresses
  2. Leak PIE base
  3. Leak heap address
  4. Leak stack canary

Reads out of bound - 1

An example how PIE can be leaked.

```
char string[16];
void* pointer_to_data_section;
read(0, string, sizeof(string));
printf(“%s”, string);
```



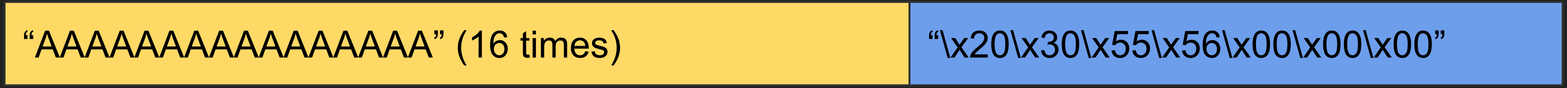
String

Pointer

Reads out of bound - 1

An example how PIE can be leaked.

```
char string[16];
void* pointer_to_data_section;
read(0, string, sizeof(string));
printf(“%s”, string);
```



“AAAAAAAAAAAAAAAA” (16 times)

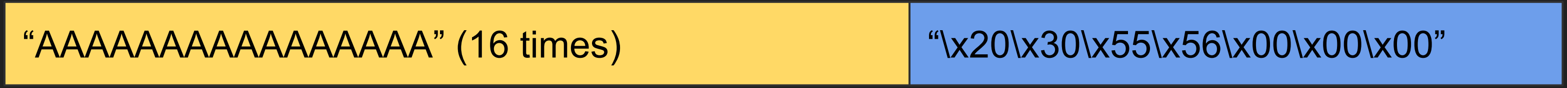
“\x20\x30\x55\x56\x00\x00\x00”

Reads out of bound - 1

An example how PIE can be leaked.

```
char string[16];
void* pointer_to_data_section;
read(0, string, sizeof(string));
printf(“%s”, string);
```

```
=> "AAAAAAAAAAAAAAAA\x20\x30\x55\x56"
```



“AAAAAAAAAAAAAAAA” (16 times)

“\x20\x30\x55\x56\x00\x00\x00”

Reads out of bound - 2

```
void * pointer = get_ptr_to_data();
int array[16];
int index = input();
if (index >= 16) die(“index out of bounds”);
printf(“%d\n”, array[index]);
```

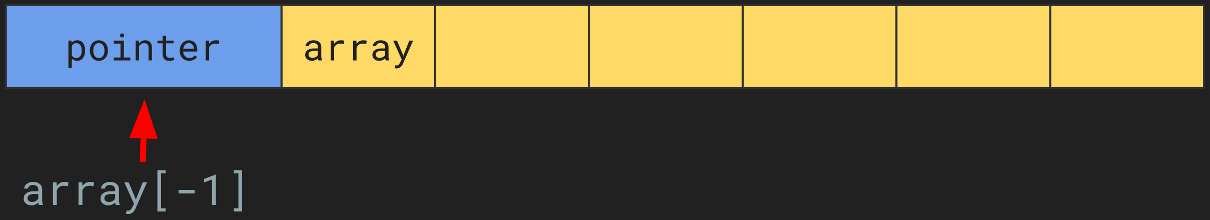
Reads out of bound - 2

```
void * pointer = get_ptr_to_data();
int array[16];
int index = input();
if (index >= 16) die(“index out of bounds”);
printf(“%d\n”, array[index]);
```

What happens if we insert -1?

Reads out of bound - 2

```
void * pointer = get_ptr_to_data();
int array[16];
int index = input();
if (index >= 16) die(“index out of bounds”);
printf(“%d\n”, array[index]);
```



What happens if we insert -1?

Uninitialized Memory

- Using uninitialized variables is BAD:
    1. Unexpected values could lead to memory corruption
    2. Variable usage may disclose randomized data

```
char buffer[1024];
read(0, buffer, 1024);
write(1, buffer, 1024);
```

Uninitialized Memory

- Using uninitialized variables is BAD:
    1. Unexpected values could lead to memory corruption
    2. Variable usage may disclose randomized data

```
char buffer[1024];

read(0, buffer, 1024);

write(1, buffer, 1024);
```

```
\x00\xf0\x38\xb9\xe8\x7f\x00\x00\x00\x00\x00\x00\x00\x00...
```

What if I send just one byte?

Uninitialized Memory

- Using uninitialized variables is BAD:
  1. Unexpected values could lead to memory corruption
  2. Variable usage may disclose randomized data

```
char buffer[1024];
read(0, buffer, 1024);
write(1, buffer, 1024);
```

```
\xaa\xf0\x38\xb9\xe8\x7f\x00\x00\x00\x00\x00\x00\x00\x00...
```

Uninitialized Memory

- Using uninitialized variables is BAD:
  1. Unexpected values could lead to memory corruption
  2. Variable usage may disclose randomized data

```c
char buffer[1024];
read(0, buffer, 1024);
write(1, buffer, 1024);
```

```
=> "\xaa\xf0\x83\xb9\xe8\x7f\x00\x00.."
```

```
\xaa\xf0\x38\xb9\xe8\x7f\x00\x00\x00\x00\x00\x00\x00\x00...
```

Leet tricks - 1

```
char buf[100];
int i = input();



use(buf[i % 100]);
```

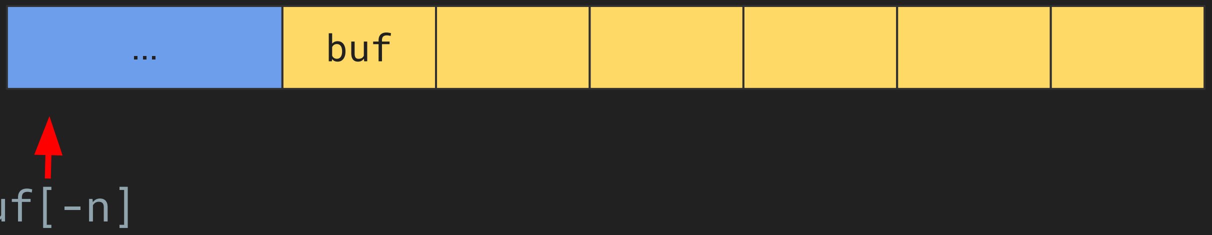
Is it safe?

Leet tricks - 1

```
char buf[100];
int i = input();


use(buf[i % 100]);
```

```
i < 0 => (i % 100) < 0
```

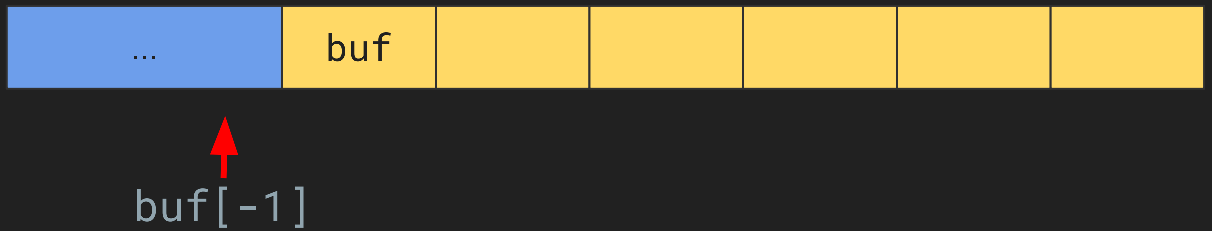


Is it safe?

Leet tricks - 1

```
char buf[100];
int i = input();    -1


use(buf[i % 100]);    -1
```



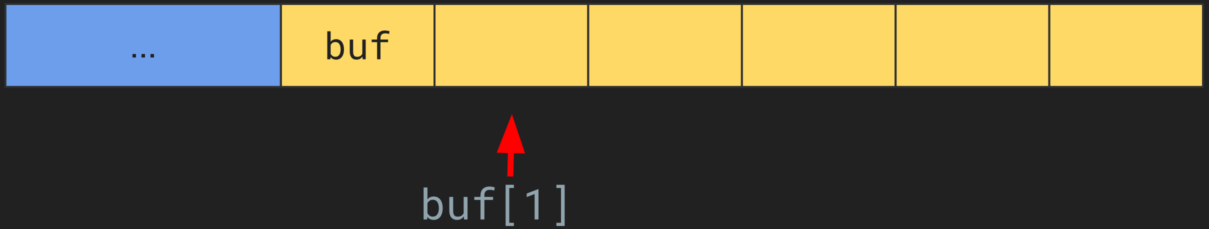
Is it safe?

Leet tricks - 1

```
char buf[100];
int i = input();
if (i < 0)
    i = -i;
use(buf[i % 100]);
```

-1

1



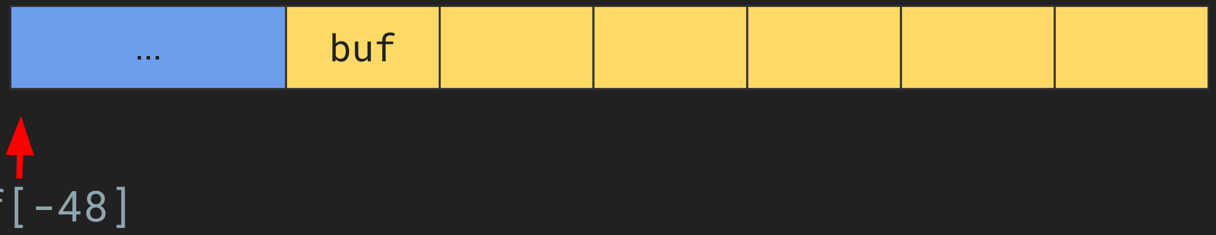
Leet tricks - 1

```
char buf[100];
int i = input();
if (i < 0)
    i = -i;
use(buf[i % 100]);
```

```
INT_MIN == 0x80000000 == -2147483648
```

```
-0x80000000 == 0x80000000 == -2147483648
```

```
(-2147483648 % 100) == -48
```



Leet tricks - 2

```
int snprintf(char *dest, size_t size, "%s", char* src);
```

What does `snprintf` returns?

Leet tricks - 2

```
int snprintf(char *dest, size_t size, "%s", char* src);
```

What does `snprintf` returns?

- **Common belief**: the number of characters written

Leet tricks - 2

```
int snprintf(char *dest, size_t size, "%s", char* src);
```

What does `snprintf` returns?

- ~~**Common belief**: the number of characters written~~ NO
- **Reality**: the number of characters (excluding the terminating null byte) which would have been written to the final string if enough space had been available

Leet tricks - 2

```
int snprintf(char *dest, size_t size, "%s", char* src);
```

What does `snprintf` returns?

- ~~**Common belief**: the number of characters written~~ NO
- **Reality**: the number of characters (excluding the terminating null byte) which would have been written to the final string if enough space had been available

so that:

```
int total_len = snprintf(NULL, 0, "%s", char* src);
char* dest = malloc(total_len + 1);
snprintf(dest, total_len + 1, "%s", char* src);
```

Leet tricks - 2

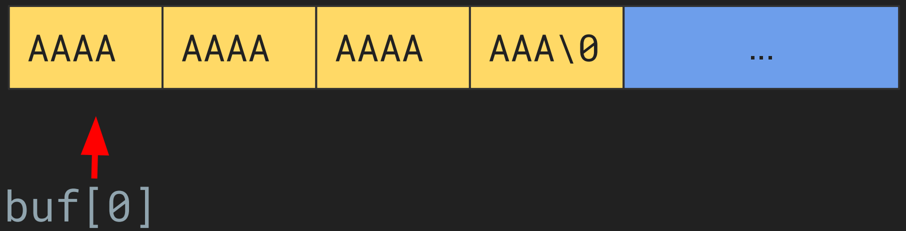
```c
char buf[16];
char* str1 = input();
int n = snprintf(buf, 16, "%s", str1);
char* str2 = input();
n += snprintf(&buf[n], 16 - n, "%s", str2);
```

Leet tricks - 2

```
char buf[16];
char* str1 = input();    <= "A"*17
int n = snprintf(buf, 16, "%s", str1);
char* str2 = input();
n += snprintf(&buf[n], 16 - n, "%s", str2);
```

Leet tricks - 2

```
char buf[16];
char* str1 = input();    <= "A"*17
int n = snprintf(buf, 16, "%s", str1);
char* str2 = input();
n += snprintf(&buf[n], 16 - n, "%s", str2);
```

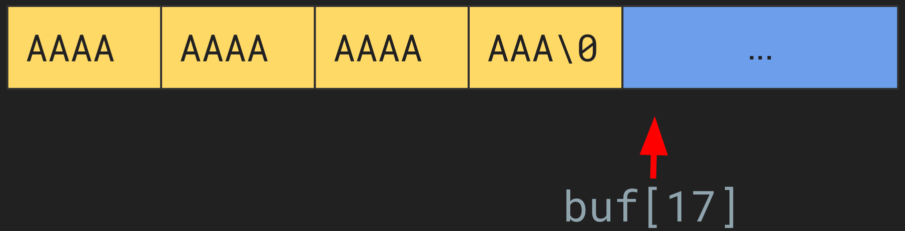


AAAA	AAAA	AAAA	AAA\0	...

buf[0]

Leet tricks - 2

```
char buf[16];
char* str1 = input();       <= "A"*17
int n = snprintf(buf, 16, "%s", str1);
char* str2 = input();
n += snprintf(&buf[17], 16 - n, "%s", str2);
                        0xffffffff
```



AAAA | AAAA | AAAA | AAA\0 | ...

buf[17]

Read more on...

The Art of Software Security Assessment - Identifying and Preventing Software Vulnerabilities