CS 858: Software Security

Offensive and Defensive Approaches

Attacks: memory corruption

Meng Xu *(University of Waterloo)*

Fall 2022

Outline

Definition: memory

Q: What is “memory” in memory corruption?

Definition: memory

Q: What is “memory” in memory corruption?

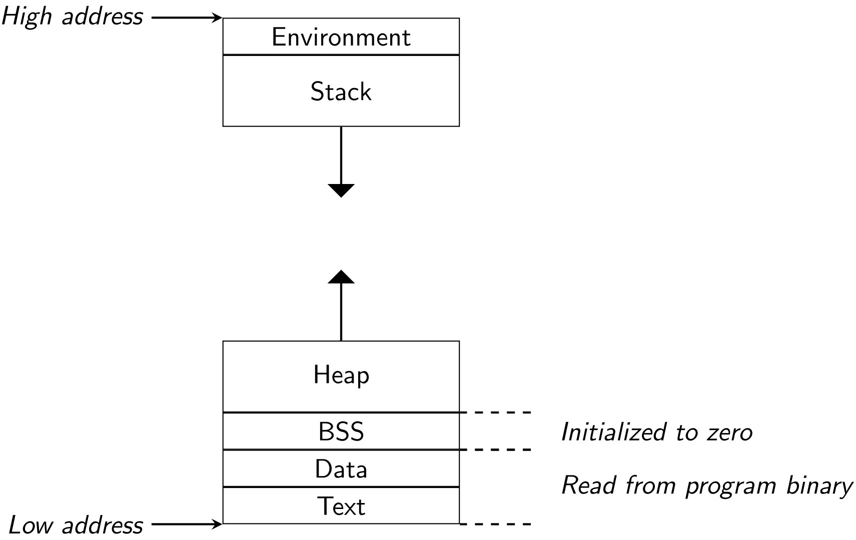
A: Three types of memory in system level:

- Stack
- Heap
- Global (a.k.a., static)

Example

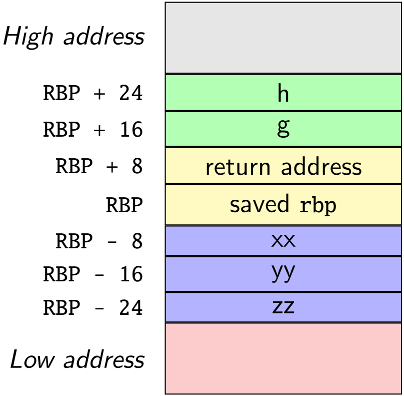
```c
#include <stdlib.c>

// this is in the data section
const char *HELLO = "hello";

// this is in the BSS section
long counter;

void foo() {
    // this is in the stack memory
    int val;

    // the msg pointer is in the stack memory
    // the msg content is in the heap memory
    char *msg = malloc(120);

    // msg content is explicitly freed here
    free(msg);

    // the val and msg pointer is implicitly freed here
}

// the global memory is only destroyed on program exit
```

Memory layout (Linux x86-64 convention)



Stack layout (Linux x86-64 convention)

```c
long foo(
    long a, long b, long c,
    long d, long e, long f,
    long g, long h)
{
    long xx = a * b * c;
    long yy = d + e + f;
    long zz = bar(xx, yy, g + h);
    return zz + 20;
}
```



High address	
RBP + 24	h
RBP + 16	g
RBP + 8	return address
RBP	saved rbp
RBP - 8	xx
RBP - 16	yy
RBP - 24	zz
Low address	

Argument a to `f` passed by registeres.

Heap layout (GNU C library implementation)

Refer to the article from Azeria Labs.

Memory layout

Q: What about stacks and heap in multi-threaded programs?

Memory layout

Q: What about stacks and heap in multi-threaded programs?

- Each thread has its own stack
- All threads in the same process share the heap and global data

For exploitation of memory errors

Smashing The Stack For Fun And Profit

How2Heap — Educational Heap Exploitation

Outline

A quick recap

This presentation is about memory corruption, a.k.a.,

- memory errors, or
- violations of memory safety properties, or
- unsafe programs

Definition: safety

Q: What is “safety” in memory safety?

Definition: safety

Q: What is “safety” in memory safety?

Observation 1: At runtime, memory is a pool of objects

Definition: safety

Q: What is “safety” in memory safety?

Observation 1: At runtime, memory is a pool of objects

Observation 2: Each object has known and limited size and lifetime

Definition: safety

Q: What is “safety” in memory safety?

Observation 1: At runtime, memory is a pool of objects

Observation 2: Each object has known and limited size and lifetime

Observation 3: Once allocated, the size of an object never changes

Definition: safety

Q: What is "safety" in memory safety?

Observation 1: At runtime, memory is a pool of objects

Observation 2: Each object has known and limited size and lifetime

Observation 3: Once allocated, the size of an object never changes

Observation 4: A memory access is always object-oriented, i.e.

- Memory read: `(object_id, offset, length)`
- Memory write: `(object_id, offset, length, value)`

Definition: safety

Q: What is “safety” in memory safety?

Observation 1: At runtime, memory is a pool of objects

Observation 2: Each object has known and limited size and lifetime

Observation 3: Once allocated, the size of an object never changes

Observation 4: A memory access is always object-oriented, i.e.

- Memory read: `(object_id, offset, length)`
- Memory write: `(object_id, offset, length, value)`

Wait..., in C/C++, pointers are just 32/64-bit integers. I can do:
`int *p = 0xdeadbeef; int v = *p;` Which object I refer to here?

Definition: safety

Q: What is “safety” in memory safety?

At any point of time during the program execution,
for any object in memory, we know its
`(object_id, size [int], alive [bool])`

At the same time, for each memory access, we know:

- Memory read: `(object_id, offset [int], length [int])`
- Memory write: `(object_id, offset [int], length [int], _)`

Outline

Definition: spatial safety

At any point of time during the program execution,
for any object in memory, we know its
(**`object_id`**, `size [int]`, `alive [bool]`)

At the same time, for each memory access, we know:

- Memory read: (**`object_id`**, `offset [int]`, `length [int]`)
- Memory write: (**`object_id`**, `offset [int]`, `length [int]`, `_`)

Definition: spatial safety

At any point of time during the program execution, for any object in memory, we know its `(`**`object_id`**`, size [int], alive [bool])`

At the same time, for each memory access, we know:

- Memory read: `(`**`object_id`**`, offset [int], length [int])`
- Memory write: `(`**`object_id`**`, offset [int], length [int], _)`

It is a violation of spatial safety if:

- `offset + length >= size` or
- `offset < 0`

Example: spatial safety violations

```
int foo(int x) {
    int arr[16] = {0};
    return arr[x];
}
```

Example: spatial safety violations

```c
int foo(int x) {
    int arr[16] = {0};
    return arr[x];
}
```

```c
long foo() {
    int a = 0;
    return *(long *)(&a);
}
```

Definition: NULL-pointer dereference

```c
int foo(int *p) {
    // it is possible that p == NULL
    return *p + 42;
}
```

Definition: NULL-pointer dereference

```c
int foo(int *p) {
    // it is possible that p == NULL
    return *p + 42;
}
```

NULL-pointer dereference is sometimes considered as undefined behavior — meaning, its behavior is not given in the C language specification, although most operating systems chooses to panic the program on such behavior.

Definition: NULL-pointer dereference

At any point of time during the program execution,
for any object in memory, we know its
(**object_id** $\neq$ `0`, `size [int], alive [bool])`

At the same time, for each memory access, we know:

- Memory read: `(`**`object_id`**`, offset [int], length [int])`
- Memory write: `(`**`object_id`**`, offset [int], length [int], _)`

Definition: NULL-pointer dereference

At any point of time during the program execution,
for any object in memory, we know its
`(`**`object_id`** $\neq$ `0, size [int], alive [bool])`

At the same time, for each memory access, we know:

- Memory read: `(`**`object_id`**`, offset [int], length [int])`
- Memory write: `(`**`object_id`**`, offset [int], length [int], _)`

It is a NULL-pointer dereference if

- `object_id == 0`

Outline

1. Introduction
2. Intuition
3. Spatial safety
4. Temporal safety
5. Countermeasures

Definition: temporal safety

At any point of time during the program execution,
for any object in memory, we know its
`(`**`object_id`**`, size [int], alive [bool])`

At the same time, for each memory access, we know:

- Memory read: `(`**`object_id`**`, offset [int], length [int])`
- Memory write: `(`**`object_id`**`, offset [int], length [int], _)`
- Memory free: `(`**`object_id`**`)`

Definition: temporal safety

At any point of time during the program execution, for any object in memory, we know its `(`**`object_id`**`, size [int], alive [bool])`

At the same time, for each memory access, we know:

- Memory read: `(`**`object_id`**`, offset [int], length [int])`
- Memory write: `(`**`object_id`**`, offset [int], length [int], _)`
- Memory free: `(`**`object_id`**`)`

It is a violation of temporal safety if:

- `!alive`

Example: temporal safety violations

```
int foo() {
    int *p = malloc(sizeof(int));
    *p = 42;
    free(p);
    return *p;
}
```

Example: temporal safety violations

```c
int foo() {
    int *p = malloc(sizeof(int));
    *p = 42;
    free(p);
    return *p;
}
```

```c
int *ptr;

void foo() {
    int p = 100;
    ptr = &p;
}
int bar() {
    return *ptr;
}
```

Example: temporal safety violations

```c
int foo() {
    int *p = malloc(sizeof(int));
    *p = 42;
    free(p);
    return *p;
}
```

```c
int *ptr;

void foo() {
    int p = 100;
    ptr = &p;
}
int bar() {
    return *ptr;
}
```

```c
int foo() {
    int *p = malloc(sizeof(int));
    *p = 42;
    free(p);
    free(p);
    return *p;
}
```

Definition: temporal safety (revisited)

At any point of time during the program execution,
for any object in memory, we know its
`(`**`object_id`**`, size [int], status [alloc|init|dead])`

At the same time, for each memory access, we know:

- Memory read: `(`**`object_id`**`, offset [int], length [int])`
- Memory write: `(`**`object_id`**`, offset [int], length [int], _)`
- Memory free: `(`**`object_id`**`)`

Definition: temporal safety (revisited)

At any point of time during the program execution, for any object in memory, we know its `(`**`object_id`**`, size [int], status [alloc|init|dead])`

At the same time, for each memory access, we know:

- Memory read: `(`**`object_id`**`, offset [int], length [int])`
- Memory write: `(`**`object_id`**`, offset [int], length [int], _)`
- Memory free: `(`**`object_id`**`)`

It is a violation of temporal safety if:

- Read: `status != init`
- Write: `status == dead`
- Free: `status == dead`

Example: temporal safety violations

```
int foo() {
    int p;
    return p;
    // what is the value returned?
}
```

Example: temporal safety violations

```c
int foo() {
    int p;
    return p;
    // what is the value returned?
}
```

```c
int foo() {
    int *p = malloc(sizeof(int));
    return *p;
    // what is the value returned?
}
```

Definition: memory leak

At any point of time during the program execution,
for any object in memory, we know its
`(`**`object_id`**`, size [int], status [alloc|init|dead])`

At the same time, for each memory access, we know:

- Memory read: `(`**`object_id`**`, offset [int], length [int])`
- Memory write: `(`**`object_id`**`, offset [int], length [int], _)`
- Memory free: `(`**`object_id`**`)`

Definition: memory leak

At any point of time during the program execution,
for any object in memory, we know its
`(`**`object_id`**`, size [int], status [alloc|init|dead])`

At the same time, for each memory access, we know:

- Memory read: `(`**`object_id`**`, offset [int], length [int])`
- Memory write: `(`**`object_id`**`, offset [int], length [int], _)`
- Memory free: `(`**`object_id`**`)`

It is a memory leak if exists one **`object_id`** whose:

- `status != dead`

Example: memory leak

```c
int foo() {
    int *p = malloc(sizeof(int));
    int *q = malloc(sizeof(int));
    *p = 42;
    free(q);
    return *p;
}
```

Outline

1. Introduction
2. Intuition
3. Spatial safety
4. Temporal safety
5. Countermeasures

Detecting memory errors

- Static analysis
- Dynamic analysis

What is so hard about static analysis?

What is so hard about static analysis?

```
void foo(bool cond) {
    char *p = malloc(sizeof(char) * 16);
    char[4] q;

    char *r;
    if (cond) {
        r = p;
    } else {
        r = q;
    }
    memcpy(r, "HELLO", 6);

    free(p);
}
```

It is possible that one pointer may points to multiple locations.

What is so hard about static analysis?

```
struct S {
    char *field;
}

void foo() {
    char *p = malloc(sizeof(char) * 16);
    struct *s = malloc(sizeof(struct S));
    s->field = p;
    free(s);
    free(p);
}

void bar(struct S* s) {
    free(s->field);
}
```

It is possible that one location may have two aliased pointers.

Preventing memory errors

- (Selective) hardening
- Use a safer language (Java / Rust / Modern C++)

Why not harden everything?

There is actually a very simple way of preventing memory errors completely!

At any point of time during the program execution,
for any object in memory, we track its
`(`**`object_id`**`, size [int], status [alloc|init|dead])`

At the same time, for each memory access, we check:

- Memory read: `(`**`object_id`**`, offset [int], length [int])`
- Memory write: `(`**`object_id`**`, offset [int], length [int], _)`

Why not harden everything?

There is actually a very simple way of preventing memory errors completely!

At any point of time during the program execution,
for any object in memory, we track its
`(`**`object_id`**`, size [int], status [alloc|init|dead])`

At the same time, for each memory access, we check:

- Memory read: `(`**`object_id`**`, offset [int], length [int])`
- Memory write: `(`**`object_id`**`, offset [int], length [int], _)`

This is essentially what is implemented in AddressSanitizer and MemorySanitizer. The result? Over 100% performance overhead...

Zero-cost abstraction

Zero-cost abstraction

```
int foo(int *arr, size_t len) {
    int sum = 0;
    for(size_t i = 0; i < len; i++) {
        // memory access check at each access
        sum += arr[i];
    }
    return sum;
}
```

Zero-cost abstraction

```c
int foo(int *arr, size_t len) {
    int sum = 0;
    for(size_t i = 0; i < len; i++) {
        // memory access check at each access
        sum += arr[i];
    }
    return sum;
}
```

```rust
fn foo(arr: &Vec<i32>) -> i32 {
    let mut sum = 0;
    arr.iter().map(
        // no need to check memory access here
        |e| sum += e
    );
    return sum;
}
```

$\langle$ **End** $\rangle$