Module 2: Program Security (Attacks)
memory errors

Meng Xu (University of Waterloo)
Spring 2023
# Outline

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Why studying memory errors?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Background: how a C program executes on a machine?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Textbook exploitation of a stack overflow vulnerability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A relatively formal definition to memory error</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Case study: Heartbleed vulnerability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Concluding remarks</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory errors are prevalent

Source: BlackHat IL 2019 talk by Matt Miller from Microsoft

Around 70% of all the vulnerabilities in Microsoft products addressed through a security update each year (2006 - 2018) are memory safety issues
Memory errors are prevalent

Source: Chromium Memory Safety Report from Google.
Analysis based on 912 high or critical severity security bugs in Chromium reported in 2015 - 2020
Memory errors are prevalent

Memory Safety Vulnerabilities are Disproportionately Severe

Source: Blog post Memory Safe Languages in Android 13 from Google.

Memory safety vulnerabilities disproportionately represent Android’s most severe vulnerabilities
Memory errors can lead to severe consequences

Heartbleed Vulnerability (CVE-2014-0610)

- A security bug in version 1.0.1 of OpenSSL, which is a widely used implementation of the Transport Layer Security (TLS) protocol
- It was introduced into OpenSSL in 2012 and publicly disclosed in April 2014
- At the time of disclosure, some 17% (around half a million) of the Internet’s secure web servers certified by trusted authorities were believed to be vulnerable to the attack
Memory errors can lead to severe consequences

- The Canada Revenue Agency (CRA) reported a theft of social insurance numbers belonging to 900 taxpayers, and said that they were accessed through an exploit of the bug during a 6-hour period on 8 April 2014.

- After the discovery of the attack, the agency shut down its website and extended the taxpayer filing deadline from 30 April to 5 May.

- On 16 April, the RCMP announced they had charged a computer science student in relation to the theft with unauthorized use of a computer and mischief in relation to data.
Heartbleed explanation

Source: https://imgs.xkcd.com/comics/heartbleed_explanation.png
Heartbleed explanation

Source: https://imgs.xkcd.com/comics/heartbleed_explanation.png
Introduction

Heartbleed explanation

SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server’s master key to "14835038534". Isabel wants pages about snakes but not too long. User Karen wants to change account password to "CofHePaSt". User

HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server’s master key to "14835038534". Isabel wants pages about snakes but not too long. User Karen wants to change account password to "CofHePaSt". User

Source: https://imgs.xkcd.com/comics/heartbleed_explanation.png
Outline

1. Why studying memory errors?
2. Background: how a C program executes on a machine?
3. Textbook exploitation of a stack overflow vulnerability
4. A relatively formal definition to memory error
5. Case study: Heartbleed vulnerability
6. Concluding remarks
A simple C program

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

int main(void) {
    char buff[8];
    int pass = 0;

    printf("Enter the password: ");
    gets(buff);

    if(strcmp(buff, "warriors")) {
        printf("Wrong password\n");
    } else {
        printf("Correct password\n");
        pass = 1;
    }

    if(pass) {
        printf("Root privileges granted\n");
    }

    return 0;
}
```

Try with

gcc -m64 -fno-stack-protector

And password “golden-hawks”
Von Neumann architecture

Computer

Central Processing Unit

Control Unit

Arithmetic / Logic Unit

Registers

Memory

Input

Output

PC  CIR  AC  MAR  MDR
Implications of the Von Neumann architecture

- Code and data reside in the same memory space and can be addressed in a unified way
  - If you manage to get the PC register to point to a memory address contains your logic, you have effectively hijacked the control flow.

- There is only one unified memory, it is the job of the compiler / programming language / runtime to find a way to utilize the memory efficiently.
  - Variables declared in a program (e.g., `int i = 0;`) needs to be mapped to an address in the memory, and the mapping logic needs to be (ideally) consistent on the same architecture.
Q: What is a conventional way of dividing up the “memory”?

A: Four types of memory on a conceptual level:

- Text (where program code is initially loaded to)
- Stack
- Heap
- Global (a.k.a., static)
Memory layout (Linux x86-64 convention)

- **High address**
  - Environment
    - Stack
  - Heap
    - BSS
    - Data
    - Text

- **Low address**

  - *Initialized to zero*
  - *Read from program binary*
Example

```c
#include <stdlib.c>

const char *HELLO = "hello";

long counter;

void main() {
    int val;

    char *msg = malloc(120);

    free(msg);
}

```
# Example (and answers)

```c
#include <stdlib.c>

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

// this is in the BSS section
c
long counter;

void main() {
    // 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
```
Outline

1. Why studying memory errors?
2. Background: how a C program executes on a machine?
3. Textbook exploitation of a stack overflow vulnerability
4. A relatively formal definition to memory error
5. Case study: Heartbleed vulnerability
6. Concluding remarks
Demo
Stack layout (Linux x86-64 convention)

```
1 long foo(
2   long a, long b, long c,
3   long d, long e, long f,
4   long g, long h)
5 {
6   long xx = a * b * c;
7   long yy = d + e + f;
8   long zz = bar(xx, yy, g + h);
9   return zz + 20;
10 }
```

High address

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBP + 24</td>
<td>h</td>
</tr>
<tr>
<td>RBP + 16</td>
<td>g</td>
</tr>
<tr>
<td>RBP + 8</td>
<td>return address</td>
</tr>
<tr>
<td>RBP - 8</td>
<td>xx</td>
</tr>
<tr>
<td>RBP - 16</td>
<td>yy</td>
</tr>
<tr>
<td>RBP - 24</td>
<td>zz</td>
</tr>
</tbody>
</table>

Low address

Argument a to f passed by registers.
What is heap and why do we need it?

In C/C++, the heap is used to manually allocate (and free) new regions of process memory during program execution.
Heap vs stack

A stack-based implementation of (roughly) the same functionality
Heap: what happens after malloc()? 

High address →  
(top of heap) 

- `p1 = malloc(50)` 
- `p2 = malloc(35)` 
- `p3 = malloc(64)` 
- `p4 = malloc(27)` 

Low address ←  

- `- - - - Heap base pointer`
Heap: what happens after `free()`?

High address

(top of heap)

Low address

chunk size | used

user data

chunk size | used

user data

chunk size | used

user data

chunk size | used free

user data

chunk size | used

--- Heap base pointer

$p1 = malloc(50)$

$p2 = malloc(35); free(p2)$

$p3 = malloc(64)$

$p4 = malloc(27)$
Real-world heap manager

For implementation details of the glibc\(^1\) memory allocator, refer to the article from Azeria Labs.

\(^{1}\)GNU C library
For exploitation of memory errors

Smashing The Stack For Fun And Profit

How2Heap — Educational Heap Exploitation
Outline

1. Why studying memory errors?
2. Background: how a C program executes on a machine?
3. Textbook exploitation of a stack overflow vulnerability
4. A relatively formal definition to memory error
5. Case study: Heartbleed vulnerability
6. Concluding remarks
A quick recap

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

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

A program is memory safe if it is free of memory errors.
Definition: safety

Q: What is “safe” 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:

```c
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], _ )
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

```
1 int foo(int x) {
2     int arr[16] = {0};
3     return arr[x];
4 }
```

```
1 long foo() {
2     int a = 0;
3     return *(long *)(&a);
4 }
```
Definition: NULL-pointer dereference

```c
1 int foo(int *p) {
2     // it is possible that p == NULL
3     return *p + 42;
4 }
```

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.
At any point of time during the program execution, for any object in memory, we know its \((\text{object\_id} \neq 0, \text{size \[int\]}, \text{alive \[bool\]})\)

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

- Memory read: \((\text{object\_id}, \text{offset \[int\]}, \text{length \[int\]})\)
- Memory write: \((\text{object\_id}, \text{offset \[int\]}, \text{length \[int\]}, _)\)

It is a NULL-pointer dereference if

- \(\text{object\_id} == 0\)
Definition: temporal safety

At any point of time during the program execution, for any object in memory, we know its
\((\text{object} \_\text{id}, \text{size} [\text{int}], \text{alive} [\text{bool}])\)

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

- Memory read: \((\text{object} \_\text{id}, \text{offset} [\text{int}], \text{length} [\text{int}])\)
- Memory write: \((\text{object} \_\text{id}, \text{offset} [\text{int}], \text{length} [\text{int}], _)\)
- Memory free: \((\text{object} \_\text{id})\)

It is a violation of temporal safety if:
- \(!\text{alive}\)
Example: temporal safety violations

```c
1 int foo() {
2     int *p = malloc(sizeof(int));
3     *p = 42;
4     free(p);
5     return *p;
6 }
```

```c
1 int *ptr;
2
3 void foo() {
4     int p = 100;
5     ptr = &p;
6 }
7 int bar() {
8     return *ptr;
9 }
```

```c
1 int foo() {
2     int *p = malloc(sizeof(int));
3     *p = 42;
4     free(p);
5     free(p);
6     return *p;
7 }
```
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

```c
1 int foo() {
2    int p;
3    return p;
4    // what is the value returned?
5 }
```

```c
1 int foo() {
2    int *p = malloc(sizeof(int));
3    return *p;
4    // what is the value returned?
5 }
```
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. Why studying memory errors?
2. Background: how a C program executes on a machine?
3. Textbook exploitation of a stack overflow vulnerability
4. A relatively formal definition to memory error
5. **Case study: Heartbleed vulnerability**
6. Concluding remarks
Heartbleed vulnerability I

```c
int dtls1_process_heartbeat(SSL *s) {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;

    /* ... redacted ... */

    if (hbtype == TLS1_HB_REQUEST) {
        unsigned char *buffer, *bp;

        /* Allocate memory for the response */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
    }
}
```
Heartbleed vulnerability II

```c
23    s2n(payload, bp);
24    memcpy(bp, pl, payload);
25
26    /* Random padding */
27    RAND_pseudo_bytes(bp, padding);
28
29    /* Send out the response */
30    r = dtls1_write_bytes(
31        s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding
32    );
33
34    /* ... redacted ... */
35
36    /* Clean-up used resources */
37    OPENSSL_free(buffer);
38    return r;
39 }
40
41    else { /* ... redacted ... */ }
42 }
```
Patch for the Heartbleed vulnerability I

diff --git a/ssl/d1_both.c b/ssl/d1_both.c
index 7a5596a6b3..2e8cf681ed 100644
@@ -1459,26 +1459,36 @@ dtls1_process_heartbeat(SSL *s)
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */

- /* Read type and payload length first */
- hbtype = *p++;
- n2s(p, payload);
- pl = p;
- if (s->msg_callback)
-     s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
-                   &s->s3->rrec.data[0], s->s3->rrec.length,
-                   s, s->msg_callback_arg);
+ /* Read type and payload length first */
+ if (1 + 2 + 16 > s->s3->rrec.length)
+     return 0; /* silently discard */
+     hbtype = *p++;
+     n2s(p, payload);
Patch for the Heartbleed vulnerability II

```c
23  +  if (1 + 2 + payload + 16 > s->s3->rrec.length)
24  +     return 0; /* silently discard per RFC 6520 sec. 4 */
25  +  pl = p;
26  +
27  if (hbtype == TLS1_HB_REQUEST)
28  {
29     unsigned char *buffer, *bp;
30     unsigned int write_length = 1 /* heartbeat type */ +
31     2 /* heartbeat length */ + payload + padding;
32     int r;
33
34     if (write_length > SSL3_RT_MAX_PLAIN_LENGTH)
35     return 0;
36 +
37     /* Allocate memory for the response, size is 1 byte
38      * message type, plus 2 bytes payload length, plus
39      * payload, plus padding
40      */
41     - buffer = OPENSSL_malloc(1 + 2 + payload + padding);
42     + buffer = OPENSSL_malloc(write_length);
43     bp = buffer;
```
Outline

1. Why studying memory errors?
2. Background: how a C program executes on a machine?
3. Textbook exploitation of a stack overflow vulnerability
4. A relatively formal definition to memory error
5. Case study: Heartbleed vulnerability
6. Concluding remarks
Memory errors are prevalent

Source: Chromium Memory Safety Report from Google.
Analysis based on 912 high or critical severity security bugs in Chromium reported in 2015 - 2020
Statistics can be misleading...

This is a personal note: one explanation why we have a disproportionately high number of memory errors reported amongst all security vulnerabilities is that — we know memory errors too well.

- Memory errors have universally accepted definitions (e.g., why the website is named Stack Overflow?)
  - Once you find a memory error, you do not need to diligently argue that this is a bug and not a feature
- Memory errors often lead to a set of known consequences that are generally considered severe (e.g., data leak or denial-of-service)
  - Once you find a memory error, you do not need to construct a working exploit to justify it
- Finding memory errors typically do not require program-specific domain knowledge (the bug is rooted in C/C++ language semantics instead of program logic)
  - If you have a technique that can find memory errors in one codebase, you can scale it up to millions of codebases developed in C/C++.

In fact, very few types of vulnerabilities meet these requirements.
Gradual adoption of memory-safe languages

Source: Blog post Memory Safe Languages in Android 13 from Google.

Number of memory safety vulnerabilities starts to decrease with the adoption of memory-safe languages.
Gradual adoption of memory-safe languages

Memory unsafe code and Memory safety vulnerabilities

- New memory unsafe code
- Memory safety vulns

<table>
<thead>
<tr>
<th>Year (Android release)</th>
<th>New memory unsafe code</th>
<th>Memory safety vulns</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019 (10)</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>2020 (11)</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>2021 (12)</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>2022 (13)</td>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>

Source: Blog post Memory Safe Languages in Android 13 from Google.
Number of memory safety vulnerabilities correlates to the portion of unsafe code
Gradual adoption of memory-safe languages

New Native Code

Source: Blog post Memory Safe Languages in Android 13 from Google.

Rust on the rise in Android native implementations