CS 489 / 698: Software and Systems Security

Module 6: Common Bugs and Vulnerabilities

memory errors

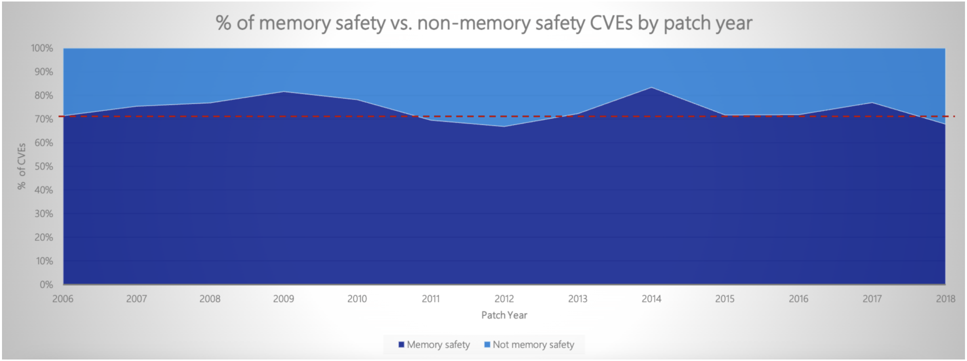
Meng Xu *(University of Waterloo)*

Winter 2024

Outline

1. Why study 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 of memory error
5. Case study: Heartbleed vulnerability
6. Concluding remarks

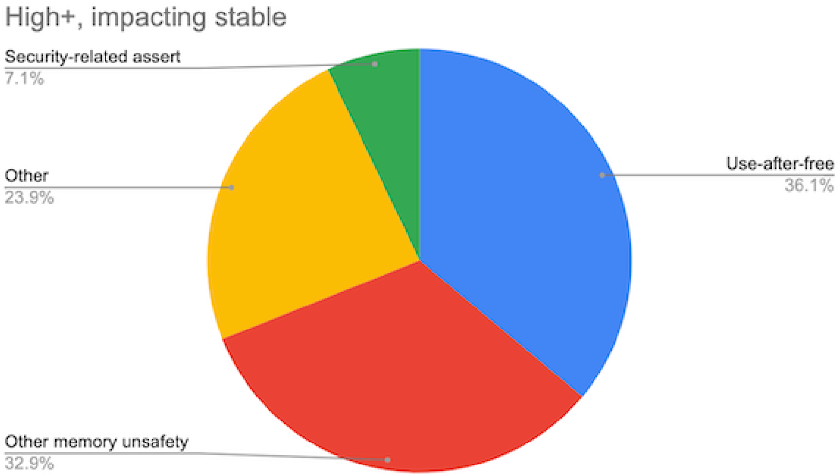
Memory errors are prevalent



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

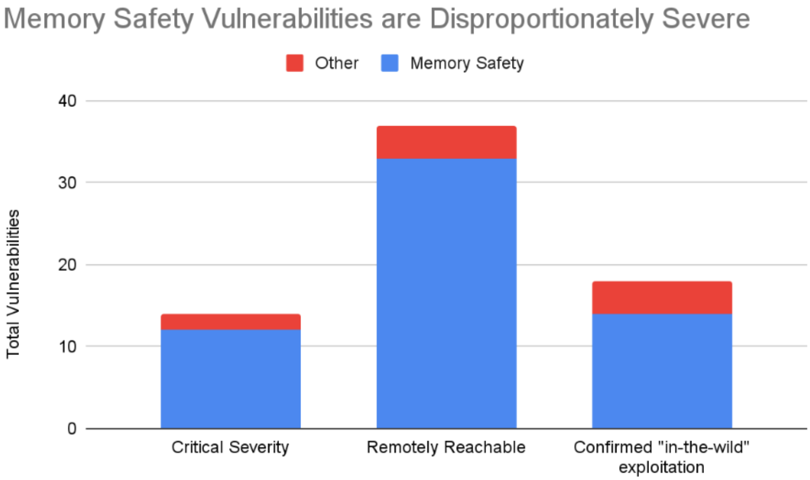
Memory errors are prevalent



Source: Chromium Memory Safety Report from Google.

Analysis based on 912 high or critical severity security bugs in Chromium reported in 2015 - 2020

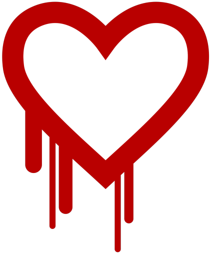
Memory errors are prevalent



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

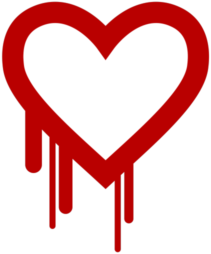
Memory safety vulnerabilities disproportionately represent Android's most severe vulnerabilities

Memory errors can lead to severe consequences



Heartbleed Vulnerability
(CVE-2014-0610)

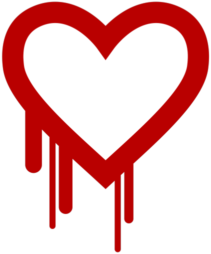
Memory errors can lead to severe consequences



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

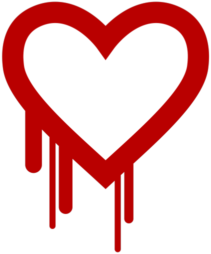
Memory errors can lead to severe consequences



Heartbleed Vulnerability (CVE-2014-0610)

- 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.

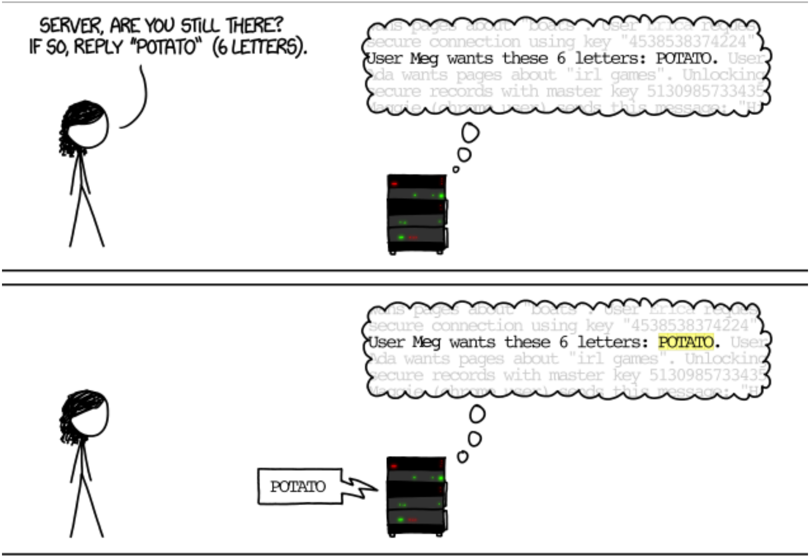
Memory errors can lead to severe consequences



Heartbleed Vulnerability
(CVE-2014-0610)

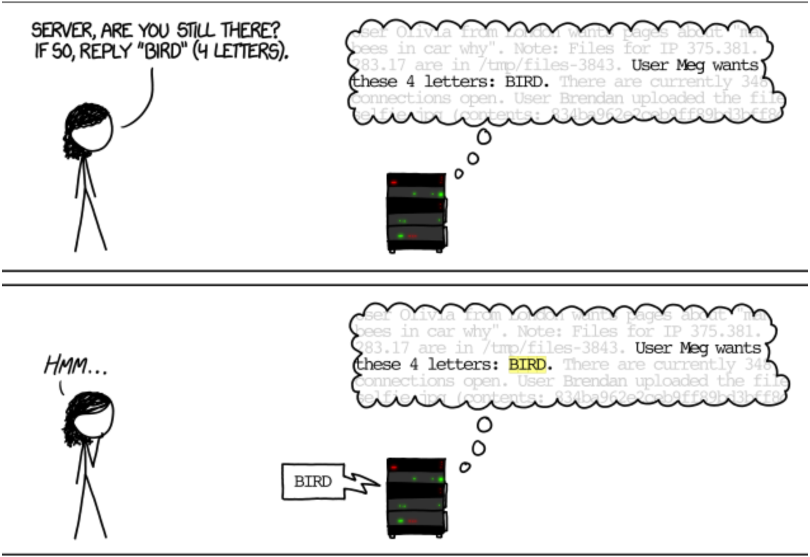
- The Canada Revenue Agency (CRA) reported a theft of social insurance numbers belonging to 900 taxpayers, and said that they were accessed through an exploit of the bug during a 6-hour period on 8 April 2014.
- After the discovery of the attack, the agency shut down its website and extended the taxpayer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



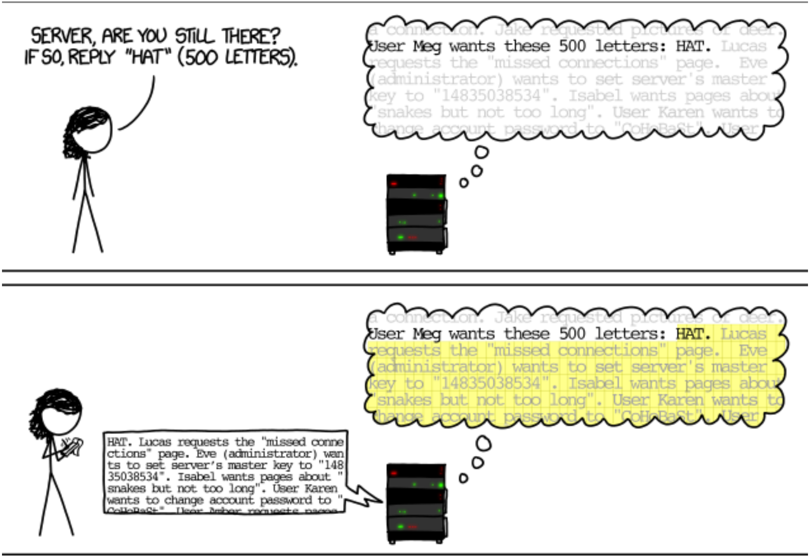
Source: https://imgs.xkcd.com/comics/heartbleed_explanation.png

Heartbleed explanation



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

Heartbleed explanation



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

Outline

1. Why study 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 of 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;
}
```

A simple C program

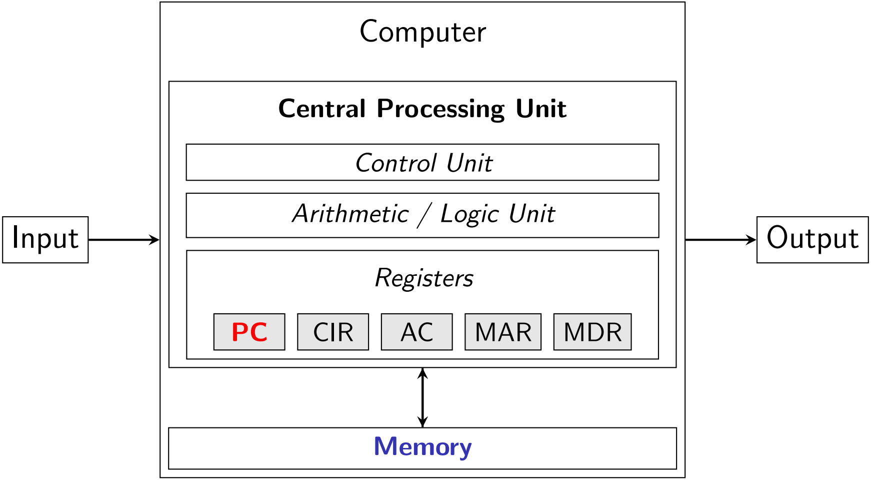
```c
#include <stdio.h>
#include <string.h>

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

PC CIR AC MAR MDR

Memory

Input

Output

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.

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.

Definition: memory

Q: What is a conventional way of dividing up the “memory”?

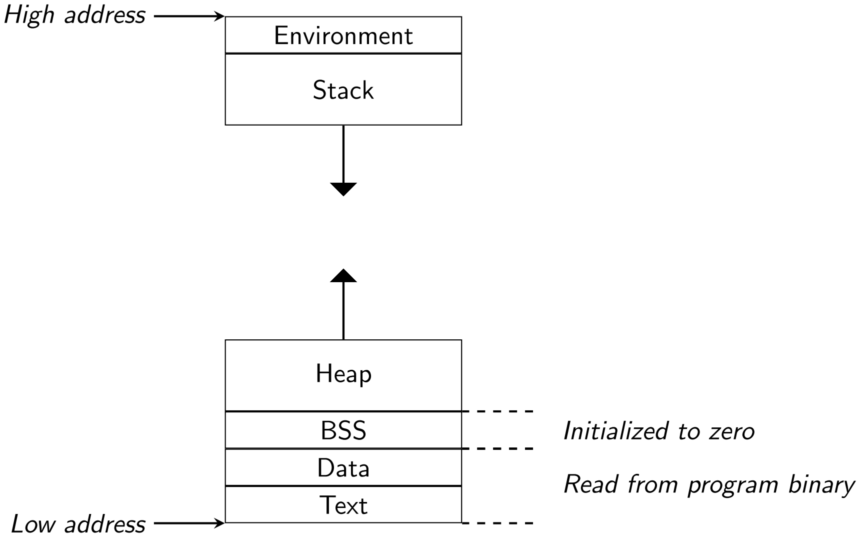
Definition: memory

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>

//! where is this variable hosted?
const char *HELLO = "hello";

//! where is this variable hosted?
long counter;

void main() {
    //! where is this variable hosted?
    int val;

    //! where is this variable hosted?
    //! where is its content allocated?
    char *msg = malloc(120);

    //! what is freed here?
    free(msg);

    //! what is freed here (at end of function)?
}

//! what is freed here (at end of execution)?
```

Example (and answers)

```c
#include <stdlib.c>

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

// this is in the BSS section
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

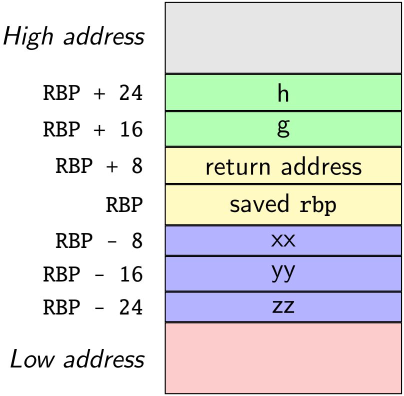
1. Why study memory errors?
2. Background: how a C program executes on a machine?
3. Textbook exploitation of a stack overflow vulnerability
4. A relatively formal definition of memory error
5. Case study: Heartbleed vulnerability
6. Concluding remarks

Demo

Demo

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.

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

```c
typedef struct Response {
  int status;
  char message[40];
} response_t;

response_t *say_hello() {
  response_t* res =
    malloc(sizeof(response_t));
  if (res != NULL) {
    res->status = 200;
    strncpy(res->message, "hello", 6);
  }
  return res;
}
void send_back(response_t *res) {
  // implementation omitted
}
void process() {
  response_t *res = say_hello();
  send_back(res);
  free(res);
}
```

Heap vs stack

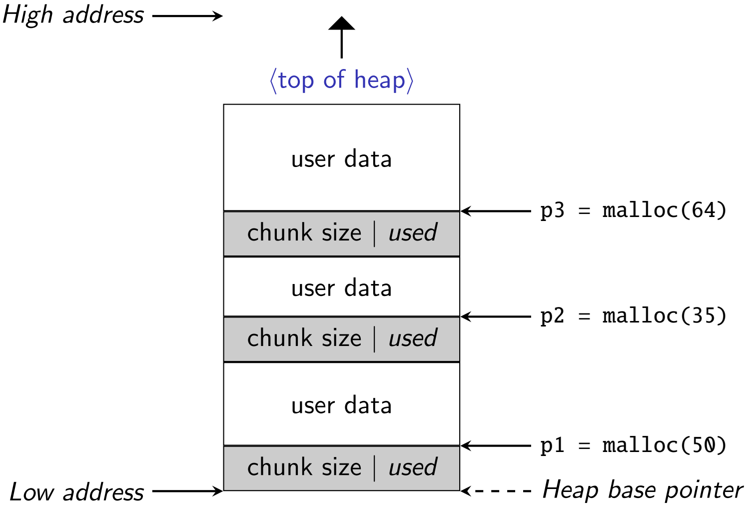
```c
typedef struct Response {
  int status;
  char message[40];
} response_t;

response_t *say_hello() {
  response_t* res =
    malloc(sizeof(response_t));
  if (res != NULL) {
    res->status = 200;
    strncpy(res->message, "hello", 6);
  }
  return res;
}
void send_back(response_t *res) {
  // implementation omitted
}
void process() {
  response_t *res = say_hello();
  send_back(res);
  free(res);
}
```

```c
typedef struct Response {
  int status;
  char message[40];
} response_t;

void say_hello(response_t *res) {
  res->status = 200;
  strncpy(res->message, "hello", 6);
}
void send_back(response_t *res) {
  // implementation omitted
}
void process() {
  struct Response res;
  say_hello(&res);
  send_back(&res);
}
```

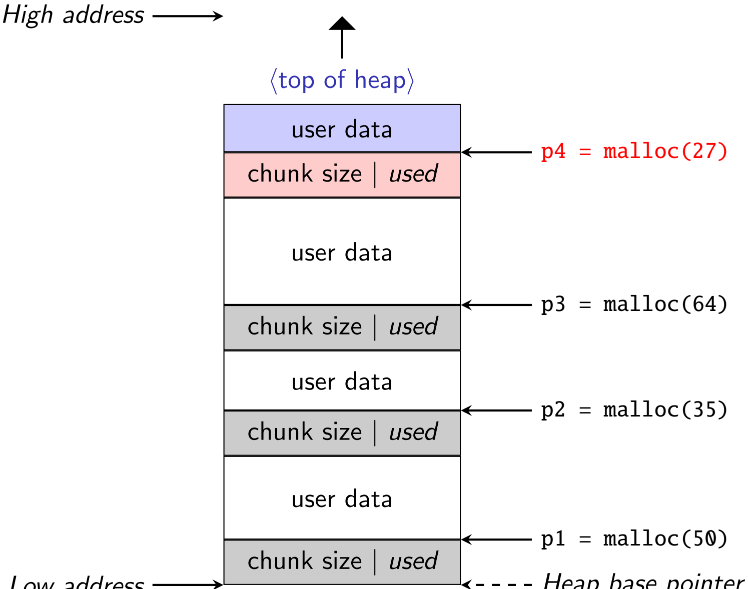
A stack-based implementation of (roughly) the same functionality

Heap: what happens after `malloc()`?

Heap: what happens after `malloc()`?



Heap: what happens after `malloc()`?



High address

⟨top of heap⟩

user data

chunk size | *used*

user data

chunk size | *used*

user data

chunk size | *used*

user data

chunk size | *used*

Low address

`p4 = malloc(27)`

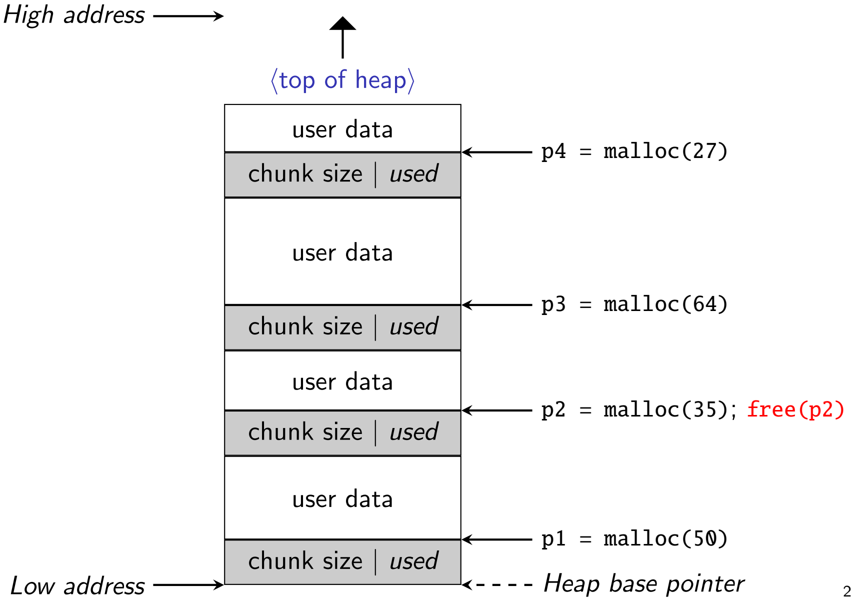
`p3 = malloc(64)`

`p2 = malloc(35)`

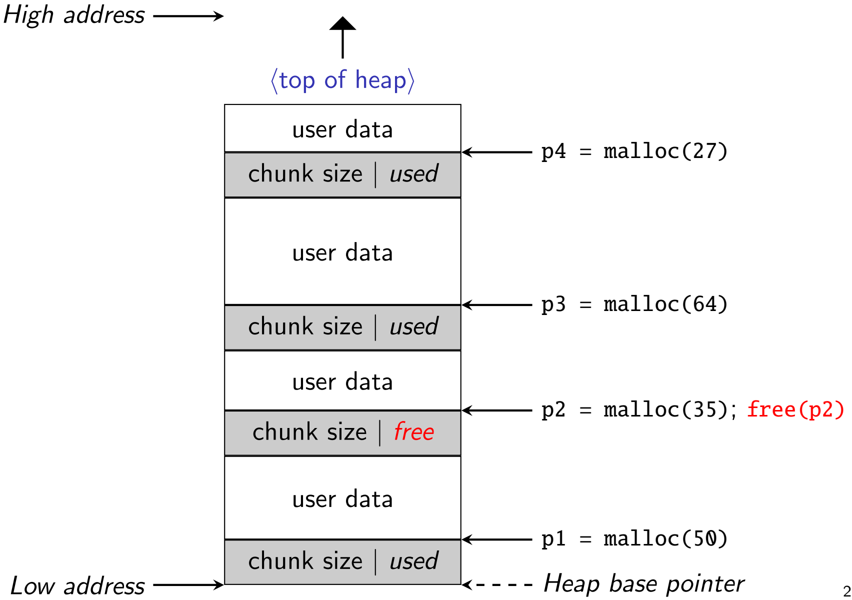
`p1 = malloc(50)`

Heap base pointer

Heap: what happens after `free()`?



Heap: what happens after `free()`?



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 study 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 of 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 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?

Definition: safety

Q: What is “safe” in memory safety?

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

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

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

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)`

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: `int *p = 0xdeadbeef; int v = *p;` Which object do 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], _)`

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

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

```
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

```
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`

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

```c
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

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

```
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. Why study 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 of 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

```
    s2n(payload, bp);
    memcpy(bp, pl, payload);

    /* Random padding */
    RAND_pseudo_bytes(bp, padding);

    /* Send out the response */
    r = dtls1_write_bytes(
        s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding
    );

    /* ... redacted ... */

    /* Clean-up used resources */
    OPENSSL_free(buffer);
    return r;
  }

  else { /* ... redacted ... */ }
}
```

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);
+
```

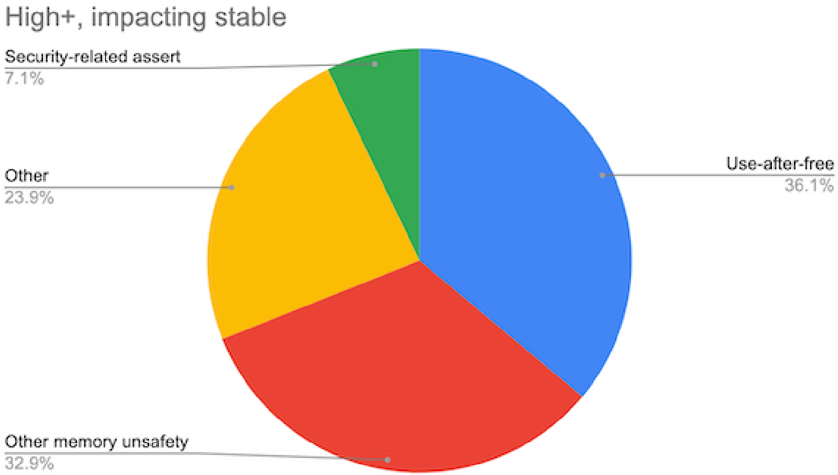
Patch for the Heartbleed vulnerability II

```
+   if (1 + 2 + payload + 16 > s->s3->rrec.length)
+       return 0; /* silently discard per RFC 6520 sec. 4 */
+   pl = p;
+
    if (hbtype == TLS1_HB_REQUEST)
        {
        unsigned char *buffer, *bp;
+       unsigned int write_length = 1 /* heartbeat type */ +
+                           2 /* heartbeat length */ + payload + padding;
        int r;

+       if (write_length > SSL3_RT_MAX_PLAIN_LENGTH)
+           return 0;
+
        /* Allocate memory for the response, size is 1 byte
         * message type, plus 2 bytes payload length, plus
         * payload, plus padding
         */
-       buffer = OPENSSL_malloc(1 + 2 + payload + padding);
+       buffer = OPENSSL_malloc(write_length);
        bp = buffer;
```

Outline

1. Why study 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 of 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.

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

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

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++.

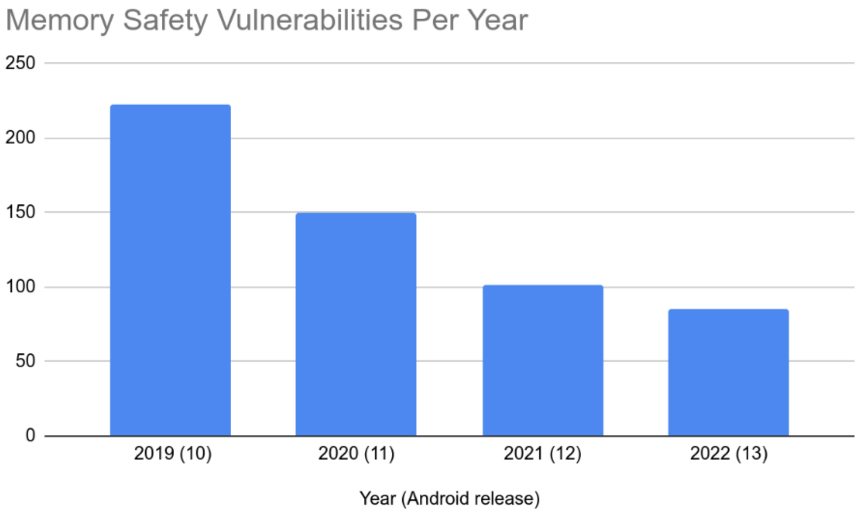
Statistics can be misleading...

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.

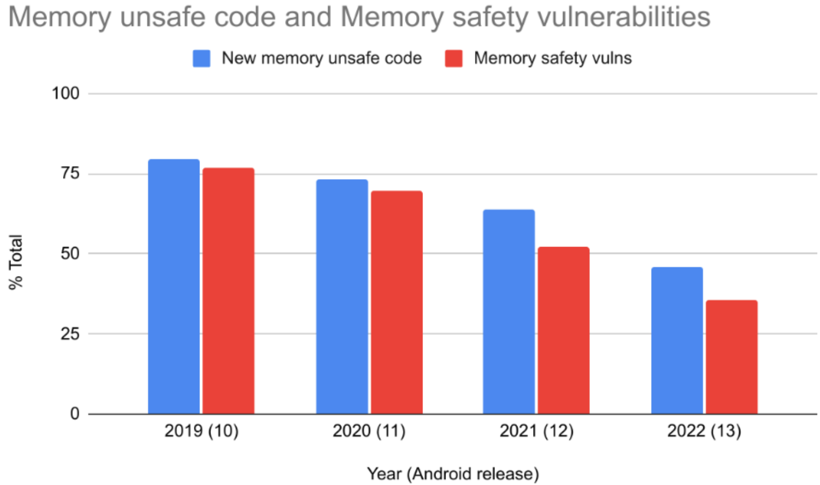
Gradual adoption of memory-safe languages



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

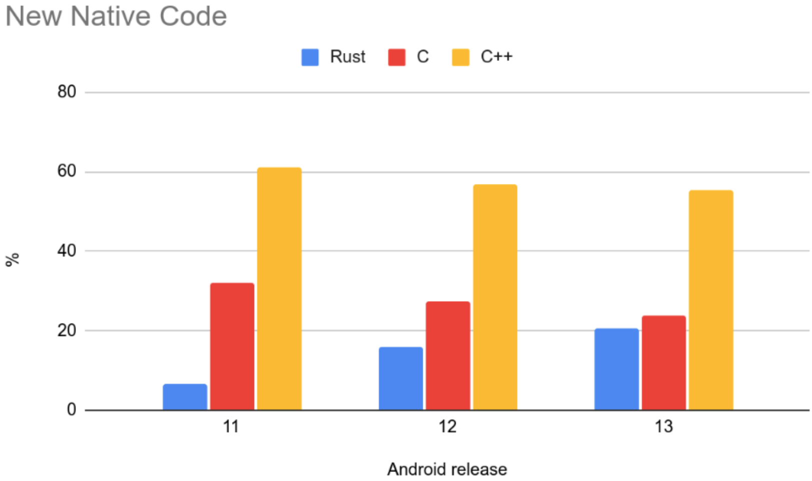
Gradual adoption of memory-safe languages



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



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

Rust on the rise in Android native implementations

Looking into the future

White House Press Release: Future Software Should Be Memory Safe on February 26, 2024.

ONCD Technical Report: Back to the Building Blocks: A Path Toward Secure and Measurable Software published in February 2024.

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