CMPSC 447

Heap Attacks

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Heap Memory

- What is heap memory?
Heap Memory

- Another region of memory that may be vulnerable to attacks is heap memory
  - Attacks similar to those on stack memory, such as buffer overflows, are possible
    - Although the attack techniques differ somewhat
      - Target metadata – kinds of similar, but different effect
      - Target data – we didn’t do that on the stack yet
Another region of memory that may be vulnerable to attacks is heap memory.

- However, the complexity of managing heap memory brings other attacks into consideration.
  - While these attacks are also possible on stack memory in theory, exploitable flaws are much less likely on the stack.

Today, we will look at the new attack types and attack techniques for the heap.
Heap Memory

• What is heap memory?

  ‣ The heap memory region is where *dynamic memory allocations* take place

  ‣ It is a contiguous region of virtual memory (can expand)
Heap Memory

- What is heap memory?
  - The heap memory region is where dynamic memory allocations take place
  - An allocation is assigned a contiguous range of virtual memory within the heap (e.g., on malloc)
• What is heap memory?

- The heap memory region is where dynamic memory allocations take place
- An allocation is assigned a contiguous range of virtual memory within the heap (e.g., on malloc)

![Heap Memory Diagram]

- Heap Low
- Heap High
What is *heap memory*?

- The heap memory region is where dynamic memory allocations take place.
- Memory from a specific allocation may be *reclaimed* when no longer needed (e.g., on “free”).

![Heap Memory Diagram](image)

<table>
<thead>
<tr>
<th>Obj A</th>
<th>Obj C</th>
</tr>
</thead>
</table>

Heap Low

Heap High
Heap Memory

- What is heap memory?
  - The heap memory region is where dynamic memory allocations take place.
  - Memory from a specific allocation may be reclaimed when no longer needed (e.g., on “free”) and reused.
Heap Memory

- What is heap memory?
  - The heap memory region is where dynamic memory allocations take place
  - If you forget to reclaim memory no longer in use, that memory region is lost (i.e., memory leak)
Suppose that `PacketRead` causes an overflow on the memory region of the variable “`packet`” below.

- What is the potential impact?

```c
int authenticated = 0;
char packet[1000];

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}
if (authenticated)
    ProcessPacket(packet);
```
Stack Buffer Overflow

- Suppose that **PacketRead** causes an overflow on the memory region of the variable “*packet*” below

  - What is the potential impact? ”**authenticated**” may be set

```c
int authenticated = 0;
char packet[1000];

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
if (authenticated)
  ProcessPacket(packet);
```
What happens if we allocate “packet” on the heap?

A buffer overflow of a buffer allocated on the heap is called a heap overflow – Impact?

```c
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}
if (authenticated)
    ProcessPacket(packet);
```
Heap Buffer Overflow

• While a heap overflow may impact heap memory regions, it won’t impact stack memory (directly)

• “authenticated” is unaffected, but something else may be affected

```c
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessPacket(packet);
```
The Heap Memory Layout below is \textit{idealized}:

- Depends on the \textit{heap allocator}
- Many heap allocators store metadata with objects on the heap to manage the heap region

![Heap Memory Layout Diagram]
The Heap Memory Layout often includes metadata:

- Depends on the heap allocator
- Often placed between objects to store information needed to manage allocation state – e.g., sizes and status.
Heap Memory Layout

• The Heap Memory Layout often includes metadata
  ‣ Depends on the heap allocator
  ‣ Often placed between objects to store information like the “size of chunk,” “size of allocation,” “in use bit,” and reference to the previous or next chunk

char *packet = malloc(1000)
ptr[1000] = 'M';
Heap Memory Layout

- The Heap Memory Layout often includes metadata
  - Depends on the heap allocator
  - So, what are the potential impacts of a heap overflow?

![Heap Memory Layout diagram]

- Metadata
  - Obj A
  - Obj D
  - Obj C
  - Heap Low
  - Heap High
Heap Overflows

- Heap allocators maintain a doubly-linked list of allocated and free chunks
- `malloc()` and `free()` modify this list

```plaintext
Chunks1, 2, and 3 are joined by a doubly-linked list

```

```plaintext
Chunk2 may be unlinked by rewriting 2 pointers

```

```plaintext
Chunk2 is now unlinked

```
Heap Overflows

• free() removes a chunk from allocated list

\[
\text{chunk2->bk->fd} = \text{chunk2->fd}
\]

\[
\text{chunk2->fd->bk} = \text{chunk2->bk}
\]

• By overflowing chunk1, attacker controls bk and fd
  ‣ Controls both *where* and *what* data is written!
  • Arbitrarily change memory

![Diagram of heap chunk relationships]
Heap Overflows

• By overflowing chunk1, attacker controls bk and fd
  ‣ Controls both where and what data is written!
    • Assign chunk2->fd to value to want to write
    • Assign chunk2->bk to address X (where you want to write)
      • Less an offset of the fd field in the structure
  • Free() removes a chunk from allocated list
    \[\text{chunk2->bk->fd} = \text{chunk2->fd}\]
    \[\text{chunk2->fd->bk} = \text{chunk2->bk}\]

• What’s the result?
Heap Overflows

- By overflowing chunk2, attacker controls bk and fd
  - Controls both where and what data is written!
    - Assign chunk2->fd to value to want to write
    - Assign chunk2->bk to address X (where you want to write)
      - Less an offset of the fd field in the structure
- Free() removes a chunk from allocated list

\[
\begin{align*}
\text{chunk2->bk->fd} &= \text{chunk2->fd} \\
\text{addrX->fd} &= \text{value} \\
\text{chunk2->fd->bk} &= \text{chunk2->bk} \\
\text{value->bk} &= \text{addrX}
\end{align*}
\]

- What’s the result?
  - Change a memory address to a new pointer value (in data)
Heap Overflow Defenses

• Separate data and metadata
  ‣ e.g., OpenBSD’s allocator (Variation of PHKmalloc)

• Sanity checks during heap management
  \[\text{free(chunk2)} \rightarrow\]
  \[
  \text{assert(chunk2->fd->bk == chunk2)}
  \]
  \[
  \text{assert(chunk2->bk->fd == chunk2)}
  \]
  ‣ Added to GNU libc 2.3.5
Other Heap Attacks

- Other Types of Attacks
  - Buffer Overread or Disclosure
  - Use-After-Free
  - Type Confusion

- While these are all also possible attacks on stack objects, they are often more significant attacks on heap objects
  - We will take a look
Buffer Overread/Disclosure

- A buffer overread (disclosure) attack enables an adversary to read memory outside of a region.
Buffer Overread/Disclosure

• A **buffer overread** (disclosure) attack enables an adversary to read memory outside of a region
  ‣ Benign task: Copy from “buffer X” to “buffer Y”
  ‣ Read beyond the memory region of “buffer X”
  ‣ To access other objects’ data
  ‣ And copy into “buffer Y”

• Why would that be a problem?
Buffer Overread/Disclosure

- A buffer overread (disclosure) attack enables an adversary to read memory outside of a region
  - Benign task: Copy from “buffer X” to “buffer Y”
  - Read beyond the memory region of “buffer X”
  - To access other objects’ data
  - And copy into “buffer Y”

- While also possible for stack objects, often more sensitive data is stored on the heap
  - Heap data is longer lived (more than a function) and often more diverse and complex (structures)
Heartbleed

• The **Heartbleed vulnerability** was a significant threat to the security of OpenSSL
  ‣ OpenSSL – crypto library for the SSL/TLS protocols
  ‣ Buffer overread vulnerability in the library that allowed an adversary to steal web servers’ private keys
  ‣ About 500,000 secure web servers were at risk
Heartbleed

- The **Heartbleed vulnerability** was a significant threat to the security of OpenSSL
  - OpenSSL – crypto library for the SSL/TLS protocols
- Caused by a **heap overread**
  - Send a message of length K, but say its length is N > K
  - Allocate N-byte buffer, but only copy K bytes into the buffer from the original message
  - Return all the memory in the N-byte buffer
Attacks on Memory Reuse

- Attacks also exploit the inconsistencies caused in the reuse of memory on the heap

- Inconsistencies
  - Your program may reclaim memory
    - And reuse that memory region for another object
  - But, the pointers to the original object (i.e., memory location prior to reclamation) may remain
    - And be used after the reuse

- Examples
  - Use-after-free and type confusion
Use After Free

- **Flaw**: Program frees data on the heap, but then references that memory as if it were still valid
  - E.g., pointer to Obj B (say “b”)
- **Accessible**: Adversary can control data written using the freed pointer
  - `memcpy(b, adv-data, size);`
- **Exploit**: Obtain a “write primitive”
Use After Free

- **Flaw**: Program frees data on the heap, but then references that memory as if it were still valid
  - E.g., pointer to Obj B (say “b”)
- **Accessible**: Adversary can control data written using the freed (stale) pointer
  - `memcpy(b, adv-data, size);`
- **Exploit**: Obtain a “write primitive”

![Diagram](image)
Use After Free

- **Flaw**: Program frees data on the heap, but then references that memory as if it were still valid

- **Accessible**: Adversary can control data written using the freed pointer

- **Exploit**: Obtain a “write primitive”

- **Hold on**: just using a reference to freed memory isn’t really a problem, is it?
  
  - What is missing from above?
Use After Free

- **Flaw**: Program frees data on the heap, but then references that memory as if it were still valid
  - E.g., pointer to Obj B (say “b”)

- **Accessible**: Adversary can control data written using the freed pointer
  - `memcpy(b, adv-data, size);`

- **Exploit**: Obtain a “write primitive” to a new object

```
Obj A  Obj D  Obj C
```
Use After Free

• What happens here?

```c
int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf2R2;
    char *buf3R2;

    buf1R1 = (char *) malloc(BUFSIZER1);
    buf2R1 = (char *) malloc(BUFSIZER1);

    free(buf2R1);

    buf2R2 = (char *) malloc(BUFSIZER2);
    buf3R2 = (char *) malloc(BUFSIZER2);

    strncpy(buf2R1, argv[1], BUFSIZER1-1);
    free(buf1R1);
    free(buf2R2);
    free(buf3R2);
}
```
Use After Free

• When the second R1 buffer (buf2R1) is freed that memory is available for reuse right away

```c
buf1R1 = (char *) malloc(BUFSIZER1);
buf2R1 = (char *) malloc(BUFSIZER1);
free(buf2R1);
```

• Then, the R2 buffers could be allocated within that memory region (buf2R1s)

```c
buf2R2 = (char *) malloc(BUFSIZER2);
buf3R2 = (char *) malloc(BUFSIZER2);
```

• Finally, the write using the freed pointer will overwrite the R2 buffers (and metadata between)

```c
strncpy(buf2R1, argv[1], BUFSIZER1-1);
```
Type Confusion Attacks

- A **type confusion** attack exploits when a program uses a pointer one type to reference a memory region of another type
  - A common way of **utilizing a use-after-free vulnerability** to go from a “write primitive” to an “arbitrary write primitive”
  - Let’s see how…
Type Confusion

- Most effective attacks exploit data of another type

```c
struct A {
    struct C *c;
    char buffer[40];
};

struct B {
    int B1;
    int B2;
    char info[32];
};
```
Type Confusion

- Free A and allocate B – assume in A’s location

```c
struct A {
    struct C *c;
    char buffer[40];
};

struct B {
    int B1;
    int B2;
    char info[32];
};

x = (struct A *)malloc(sizeof(struct A));
free(x);
y = (struct B *)malloc(sizeof(struct B));
```
Type Confusion

• How do you think you exploit this?

```c
struct A {
    struct C *c;
    char buffer[40];
};

struct B {
    int B1;
    int B2;
    char info[32];
};

x = (struct A *)malloc(sizeof(struct A));
free(x);
y = (struct B *)malloc(sizeof(struct B));
```
Type Confusion

- Arbitrary write primitive!

```c
struct A {
    struct C *c;
    char buffer[40];
};

struct B {
    int B1;
    int B2;
    char info[32];
};

x = (struct A *)malloc(sizeof(struct A));
free(x);
y = (struct B *)malloc(sizeof(struct B));
y->B1 = address_of_where_to_write;
x->c->field = value_to_write;
```
Use After Free

- **Flaw**: program frees data on the heap, but then references that memory as if it were still valid
- **Accessible**: Adversary can control data written using the freed pointer
- **Exploit**: Obtain an “arbitrary write primitive”
- Become a popular vulnerability to exploit – over 60% of CVEs
Type Confusion

• How do you think you exploit this?

```c
struct A {
    void (*fnptr)(char *arg);
    char buffer[40];
};

struct B {
    int B1;
    int B2;
    char info[32];
};

x = (struct A *)malloc(sizeof(struct A));
free(x);
y = (struct B *)malloc(sizeof(struct B));
```
Type Confusion

- Arbitrary code reuse!

```c
struct A {
    void (*fnptr)(char *arg);
    char buffer[40];
};

struct B {
    int B1;
    int B2;
    char info[32];
};

x = (struct A *)malloc(sizeof(struct A));
free(x);
y = (struct B *)malloc(sizeof(struct B));
y->B1 = execve@PLT;
x->fnptr("/bin/sh");
```
Type Confusion

- Adversary chooses function pointer value (set as int)
- Adversary may also be able to choose value for “arg”
- To implement arbitrary code reuse

```c
struct A {
    void (*fnptr)(char *arg);
    char buffer[40];
};

struct B {
    int B1;
    int B2;
    char info[32];
};

x = (struct A *)malloc(sizeof(struct A));
free(x);
y = (struct B *)malloc(sizeof(struct B));
y->B1 = execve@PLT;
x->fnptr("/bin/sh");
```
Heap Spraying

• How do adversaries use such flaws?
  ‣ May be hard to get an object of ”struct Y” in the location of the freed “struct X” object

• Use heap spraying to fill the heap with lots of “struct Y” objects
  ‣ Eventually, one will be placed in the location of the freed “struct X” object, so we can use the pointer to access to target memory or code
Type Confusion

- Does type confusion require a use-after-free?
  - What other C operation enables a programmer to reference data one location via multiple type signatures?
Type Casts

- Does type confusion require a use-after-free?
  - What other C operation enables a programmer to reference data one location via multiple type signatures?

- Type Cast
  - A type cast enables you to create a pointer of a different type to the same memory region
    - Also, reasoning about multiple types is common in object-oriented languages (C++)
Type Confusion Via Casts

• Cause the program to process data of one type when it expects data of another type
  ‣ Provides same affect as we did with use-after-free
  ‣ But, without the “free” – just need an ambiguous “use”

• Where’s the error below?

```c++
class Ancestor { int x; }
class Descendent : Ancestor { int y; }
Ancestor *A = new A;
Descendant *D = static cast <Ancestor *> A;
D->y = 7;
```

HexType – Jeon et al. ACM CCS 2017
Type Confusion

• Cause the program to process data of one type when it expects data of another type
  ‣ Provides same affect as we did with use-after-free
  ‣ But, without the “free” — just need an ambiguous “use”
  • Where’s the error below?

```cpp
class Ancestor { int x; }
class Descendent : Ancestor { int y; }

Ancestor *A = new A;
Descendant *D = static cast <Ancestor *> A;
D->y = 7; // not within memory region allocated to A
```

HexType – Jeon et al. ACM CCS 2017
Type Hierarchies

- C++ allows you to construct type hierarchies

HexType – Jeon et al. ACM CCS 2017
Type Hierarchies

- C++ allows you to construct type hierarchies
  - Which type of cast is safe and why?

HexType – Jeon et al. ACM CCS 2017
Un/Safe Type Casts

- **Upcasts** are always safe because they only reduce the type structure
  - That is, only subtypes extend the structure definitions
- Thus, **downcasts** (as in the example) and arbitrary casts (that do not follow the hierarchy) are unsafe
  - However, programming environments trust programmers to do the right thing
Take Away

- Heaps provide a wide variety of options for adversaries, depending on the software flaw
- Can attack either heap metadata or other heap data, including pointers to access arbitrary memory
- Heaps are susceptible to more types of powerful attacks than stacks
  - Disclosure attacks, use-after-free, and type confusion
  - These attacks are all somewhat related
- We will explore defenses for all of these