The Heap

Offensive Security - Summer 2020

Martin Clauß, Ruben Gonzalez (original slides: Konstantin Wurster)

July 2, 2020

Bonn-Rhein-Sieg University o.a.S.

The Heap

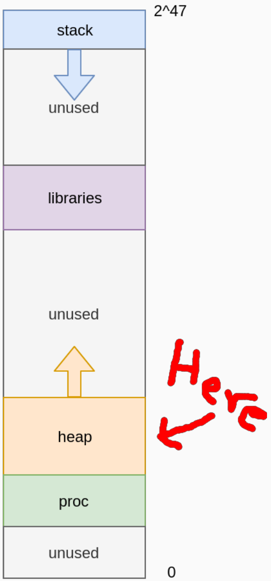
Disclaimer

- Heap management is an advanced and complex topic
- We can only scratch the surface in this lecture
- We will talk about certain aspects of the heap (based on glibc's ptmalloc implementation)
- This lecture will introduce the tcache (thread cache) and two related vulnerability types: Use-after-free (UAF) and Double-free
- One didactic caveat: We will explain the Double-free vulnerability with non-tcache heap chunks (chunks are blocks of heap memory), BUT the knowledge can be transferred to tcache heap chunks

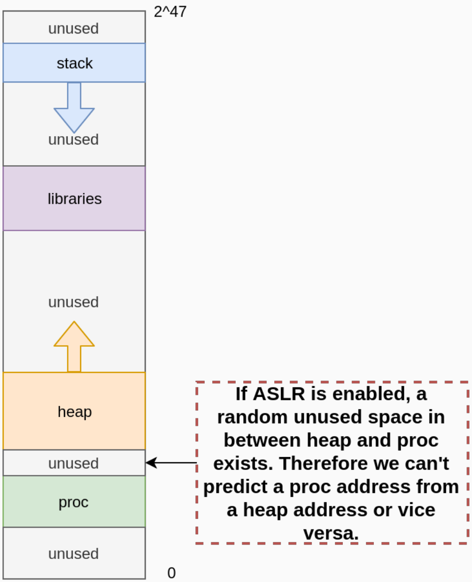
Just another memory region...

- Read and writable memory region
- Located above our proc
- The heap is managed by libc!
- As a programmer, interaction through the *malloc*/*calloc*/*realloc*/*free* functions
- Exploiting the heap needs a good understanding of how those functions work
- There are different implementations available, but we will only take a look at the glibc one

Where is the Heap located?



Usually there is ASLR



2^47
unused
stack
unused
libraries
unused
heap
unused
proc
unused
0
If ASLR is enabled, a random unused space in between heap and proc exists. Therefore we can't predict a proc address from a heap address or vice versa.

Malloc / Free

*The **malloc**() function allocates size bytes and returns a pointer to the allocated memory. **The memory is not initialized**.*
*The **free**() function frees the memory space pointed to by ptr, which must have been returned by a previous call to **malloc**()*

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
```

Example use:

```
char *buffer;
/* request 1000 bytes, returns a pointer into the heap */
buffer = (char*)malloc(1000);
/* we can now use the memory from address
   buffer till buffer + 999 */
/* free the buffer with the base pointer */
free(buffer);
```

How much actual memory does a ***malloc(1)*** *require? What do you think?*

- 1 Byte
- 4 Bytes
- 8 Bytes
- 16 Bytes
- 24 Bytes
- 32 Bytes
- other

Demo

```
malloc(0), size=32, address=0x43a2a0
malloc(8), size=32, address=0x43a6d0
malloc(16), size=32, address=0x43a6f0
malloc(24), size=32, address=0x43a710
malloc(32), size=48, address=0x43a730
malloc(40), size=48, address=0x43a760
malloc(48), size=64, address=0x43a790
malloc(56), size=64, address=0x43a7d0
malloc(64), size=80, address=0x43a810
malloc(72), size=80, address=0x43a860
malloc(80), size=96, address=0x43a8b0
malloc(88), size=96, address=0x43a910
malloc(96), size=112, address=0x43a970
malloc(104), size=112, address=0x43a9e0
...
```

Sizes are always a multiple of 16 (`2 * sizeof(void *)`) and at least 32 bytes (`4 * sizeof(void *)`). This is different on 32 bit!

Malloc internals

- Actually, it is quite complex, to manage memory efficient
- We will only take a look at small memory requests $\leq$ 1032 Bytes
- And, to simplify this even more, **we only look at the recently introduced tcache behaviour and ignore the rest as far as we can**
- Lets start

Malloc Internals

Tcache

- Glibc malloc is implemented thread-safe (ptmalloc: pthread malloc)
- Completely locks, if heap stuff is done
- Bad, as there is a trend to more parallelization and heap functions are heavily used
- We have a bottleneck. Therefore a per-thread cache was introduced. They called it tcache and optimized it for performance.
- Default tcache caches memory requests of $\leq$ 1032 Bytes
- Works as a simple LIFO queue
- If malloc is called, the last freed memory region of the appropriate size is returned
- Default tcache caches up to 7 entries of the same chunksize .

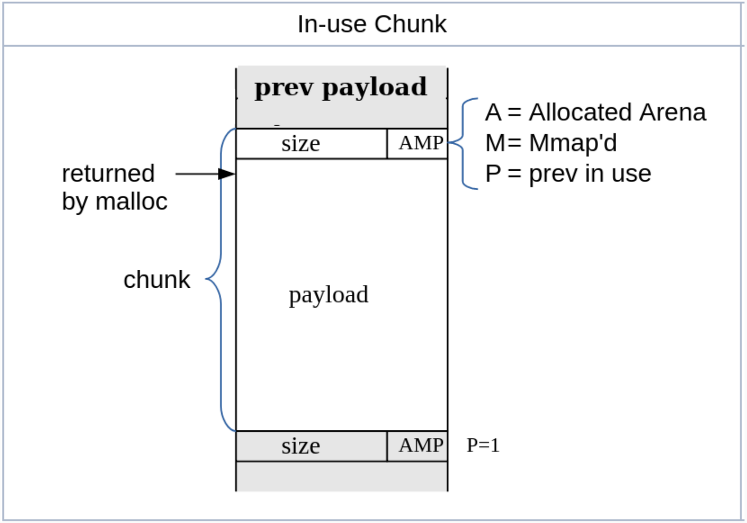
Chunks

- Chunks organize a larger memory region into smaller pieces
- A Chunk consists out of a size header (8 Byte) and a payload section
- The chunk size header is calculated by the payload size + 8
- A malloc request will return a pointer into the chunks payload
- Chunks exist in different chunk sizes, from 32 to 1040 Bytes in steps of 16 Bytes
- For this reason, a 1 Byte malloc will always return 24 Bytes of *usable space*.
- A Chunk has status information as well, like if the chunk located previous in memory is free or not
- The status information is saved in the last bits of the size header

Don't be confused by the different sizes! The chunks size is the size including the header! This means that you can only use

`chunk_size - 8` bytes as a programmer that called malloc.

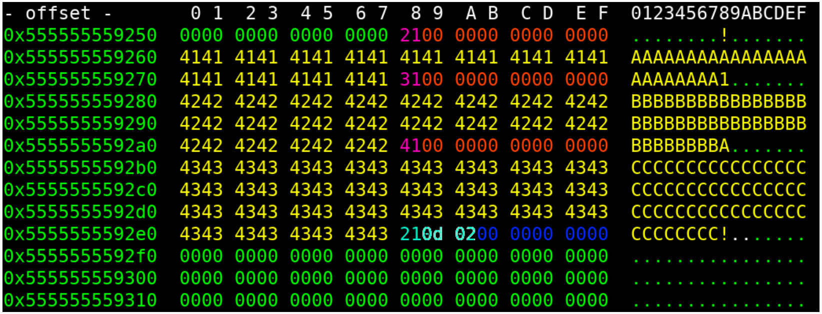
Chunks


In-use Chunk
prev payload
size
AMP
A = Allocated Arena
M= Mmap'd
P = prev in use
returned
by malloc
chunk
payload
size
AMP
P=1

Chunks

```
memset(malloc(24), 'A', 24); // 0x18 bytes => 0x20 chunksize
memset(malloc(40), 'B', 40); // 0x28 bytes
memset(malloc(56), 'C', 56); // 0x38 bytes
```

Our three memory chunks and a fourth mysterious chunk. Notice how the last bit of chunk size is set (prev in use bit).



```
- offset -        0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x555555559250  0000 0000 0000 0000 2100 0000 0000 0000  ........!.......
0x555555559260  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0x555555559270  4141 4141 4141 4141 3100 0000 0000 0000  AAAAAAAA1.......
0x555555559280  4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
0x555555559290  4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
0x5555555592a0  4242 4242 4242 4242 4100 0000 0000 0000  BBBBBBBBA.......
0x5555555592b0  4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0x5555555592c0  4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0x5555555592d0  4343 4343 4343 4343 4343 4343 4343 4343  CCCCCCCCCCCCCCCC
0x5555555592e0  4343 4343 4343 4343 210d 0200 0000 0000  CCCCCCCC!.......
0x5555555592f0  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x555555559300  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x555555559310  0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Bits and Bit Fiddling

- multiples of 16 look like this in binary:
  - `bit nr:   9876543210`
  - `0:  0000000000000000`
  - `16: 0000000000010000`
  - `32: 0000000000100000`
  - `48: 0000000000110000`
- note that the last 4 bits are always `0`
- those bits can be used, for example, to store additional information
- set bit: `value |= 2**0` to set the least significant bit, `value |= 2**1` to set the second to last bit
- test bit: `if value & 2**1: print("bit 1 is set")`, `if value & 2**4: print("bit 4 is set")`
- get value without the additional information: `without_add_infos = value & ~0b1111` (`17 & ~0b1111 == 16`: clears the last $2^0 = 1$ bit)

The Wilderness

- Notice the last chunk with the huge size, it is called 'Wilderness'
- Every time no existing freed or cached chunk can be used, a new chunk is taken from the space of the wilderness
- If the wilderness grows too little, malloc will request an additional adjacent memory page from the os (brk/sbrk system calls)

Tcache LIFO Behaviour

- Tcache holds a single linked list for every chunksize
- There are 64 of those linked lists (one for chunksize 0x20/32, one for 0x30/48, one for 0x40/64, ...)
- LIFO behavior for each of those lists

```c
void *a, *b, *c;
a = malloc(24); // chunk a size = ?
b = malloc(8);  // chunk b size = ?
c = malloc(40); // chunk c size = ?
free(a); free(b); free(c);
malloc(1);
malloc(25);
malloc(40);
malloc(24);
malloc(8);
```

What chunksizes do we get?

Tcache LIFO Behaviour

```c
void *a, *b, *c;
a = malloc(24); // chunk a size = 0x20
b = malloc(8);  // chunk b size = 0x20
c = malloc(40); // chunk c size = 0x30
free(a); free(b); free(c); // free all in the order a, b, c.
malloc(1); // request size = ?
malloc(25);
malloc(40);
malloc(24);
malloc(8);
```

What chunk size is requested by malloc(1)?

Tcache LIFO Behaviour

```
void *a, *b, *c;
a = malloc(24); // chunk a size = 0x20
b = malloc(8);  // chunk b size = 0x20
c = malloc(40); // chunk c size = 0x30
free(a); free(b); free(c); // free all in the order a, b, c.
malloc(1); // request size = 0x20, got chunk ? reassigned
malloc(25);
malloc(40);
malloc(24);
malloc(8);
```

Which of the previously freed chunks do we get assigned?

Tcache LIFO Behaviour

```c
void *a, *b, *c;
a = malloc(24); // chunk a size = 0x20
b = malloc(8);  // chunk b size = 0x20
c = malloc(40); // chunk c size = 0x30
free(a); free(b); free(c); // free all in the order a, b, c.
malloc(1); // request size = 0x20, got chunk b reassigned
malloc(25); // request size = ?, got chunk ? reassigned
malloc(40);
malloc(24);
malloc(8);
```

Which size do we request? Which chunk do we get?

Tcache LIFO Behaviour

```
void *a, *b, *c;
a = malloc(24); // chunk a size = 0x20
b = malloc(8);  // chunk b size = 0x20
c = malloc(40); // chunk c size = 0x30
free(a); free(b); free(c); // free all in the order a, b, c.
malloc(1); // request size = 0x20, got chunk b reassigned
malloc(25); // request size = 0x30, got chunk c reassigned
malloc(40); // request size = ?, got chunk ? reassigned
malloc(24);
malloc(8);
```

Which size do we request? Which chunk do we get?

Tcache LIFO Behaviour

```
void *a, *b, *c;
a = malloc(24); // chunk a size = 0x20
b = malloc(8);  // chunk b size = 0x20
c = malloc(40); // chunk c size = 0x30
free(a); free(b); free(c); // free all in the order a, b, c.
malloc(1); // request size = 0x20, got chunk b reassigned
malloc(25); // request size = 0x30, got chunk c reassigned
malloc(40); // request size = 0x30, got new chunk d
malloc(24); // request size = ?, got chunk ? reassigned
malloc(8);
```

Which size do we request? Which chunk do we get?

Tcache LIFO Behaviour

```c
void *a, *b, *c;
a = malloc(24); // chunk a size = 0x20
b = malloc(8);  // chunk b size = 0x20
c = malloc(40); // chunk c size = 0x30
free(a); free(b); free(c); // free all in the order a, b, c.
malloc(1); // request size = 0x20, got chunk b reassigned
malloc(25); // request size = 0x30, got chunk c reassigned
malloc(40); // request size = 0x30, got new chunk d
malloc(24); // request size = 0x20, got chunk a reassigned
malloc(8); // request size = ?, got chunk ? reassigned
```

Which size do we request? Which chunk do we get?

Tcache LIFO Behaviour

```
void *a, *b, *c;
a = malloc(24); // chunk a size = 0x20
b = malloc(8);  // chunk b size = 0x20
c = malloc(40); // chunk c size = 0x30
free(a); free(b); free(c); // free all in the order a, b, c.
malloc(1); // request size = 0x20, got chunk b reassigned
malloc(25); // request size = 0x30, got chunk c reassigned
malloc(40); // request size = 0x30, got new chunk d
malloc(24); // request size = 0x20, got chunk a reassigned
malloc(8); // request size = 0x20, got new chunk e
```

Demo

see `example_1.c` and its comments

Note

- There are more bin types: fast bins, small bins, large bins, unsorted bins
- Heap behavior is very complex: `https://raw.githubusercontent.com/cloudburst/libheap/master/heap.png`
- This lecture only scratches the surface! `malloc.c` (`https://code.woboq.org/userspace/glibc/malloc/malloc.c.html`) is a very detailed reference but also: `https://sourceware.org/glibc/wiki/MallocInternals`

Use After Free

What is Use After Free?

- We still use memory after we freed it
- We have a so called ’Dangling pointer’ (non-NULL pointer that points into `free()`ed memory)
- A call to malloc with the same chunk size will happily return the erroneously freed memory

How To Exploit UAF?

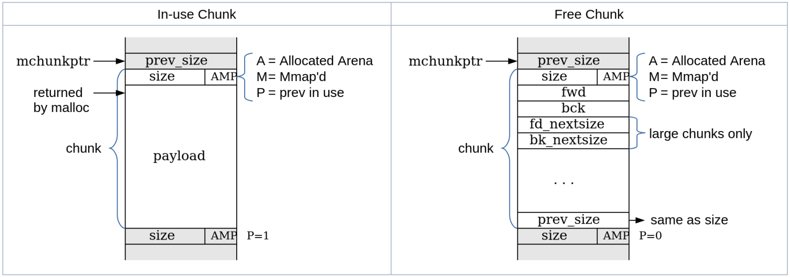
1. The chunk affected by the UAF bug is allocated
2. Trigger the bug, the chunk will be freed, but is still in use
3. Allocate user controlled data with the same chunk size, due to LIFO we will get the just freed chunk of memory
4. We can overwrite the still ’in use’ memory as it belongs to us now
5. Profit

UAF Example (stripped)

```
importantstruct *important = malloc(sizeof(*important));
important->function = importantfunction;
important->number = 42;
char *userdata = NULL;
for(;;) {
  int choice;
  if(scanf("%d", &choice) != 1 || !choice)
    break;
  switch (choice) {
    case 1:
      important->function(important->number); break;
    case 2:
      free(important); break;
    case 3:
      if (!userdata) {userdata = malloc(24);}
      read(STDIN_FILENO, userdata, 24); break;
    case 4:
      if (userdata) {puts(userdata);} break;
  }
}
```

Double Free

In Use vs. Free Chunks



What is Double Free?

The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc(), or realloc(). Otherwise, or if free(ptr) has already been called before, undefined behavior occurs.

What is Double Free?

see `https://en.cppreference.com/w/cpp/language/ub`

What is Double Free?

Have a look at the handwritten notes now!

What is Double Free?

- We free the same memory chunk twice
- The LIFO queue [1] now contains two entries belonging to the same address
- Those entries are returned by subsequent calls of malloc and a matching chunksize

[1] This might be different for different chunks types: tcache, fastbin, small, large and unsorted)

How To Exploit Double Free?

1. Free some memory twice
2. Allocate, so you get one of the double freed chunks for user-controlled data
3. Some other piece of code allocates the other chunk for some functionality we want to abuse
4. As we both got the same chunk, we now have the same condition as in use after free
5. Therefore profit

Double Free Example (stripped)

```c
importantstruct *important = NULL;
char *userdata = NULL;
for(;;) {
  switch (choice) {
    case 1:
      if (!important) {
        important = malloc(sizeof(*important));
        important->function = importantfunction;
        important->number = 42;
      }
      important->function(important->number); break;
    case 2:
      free(userdata); break;
    case 3:
      userdata = malloc(24);
      read(STDIN_FILENO, userdata, 24); break;
    case 4:
      if (userdata) {puts(userdata);} break;
  }
}
```

Demo

Have a look at the CTF example!