No Headache - OpenECSC 2024

by Marco Pellero

Overview

> All these complicated memory allocators only give me headaches, so I decided to go with a much simpler implementation. Simpler code = less bugs = no more getting pwned!

The challenge gives us a binary which when run prompts the user with a menu:

```
Available commands:
        n) New object
        s) Set object properties
        p) Print object
        z) Get object size
        d) Delete object
        h) Help
        e) Exit
```

We can create a new object, which adds it at the start of a linked list of objects, at index 0, we can set the properties of the first object, which are just key-value pairs of strings, we can print any object's properties or size, or delete it, which will unlink it from the list.

```
> n
> s
Properties (format: foo=bar;baz=123;...):
foo=bar
> p
Index: 0
{
        "foo": "bar",
}
> z
Index: 0
8
```

```
> d
Index: 0
> p
Index: 0
Invalid index
>
```

Reversing

When decompiling the program, we'll notice that it doesn't use `malloc()` or even `mmap()` to allocate our objects, instead, it uses a baked in allocator which provides `malloc()`, `free()`, `calloc()`, and `realloc()`. When first reversing the program I thought this was a custom allocator, but when I then looked at the `realloc()` function I noticed a condition that looked like this:

```
if (a != b)
        __assert_fail("ptr == alloc_last_block", "src/dl-minimal-malloc.c",
99, "__minimal_realloc");
```

Here's the signature of `__assert_fail()`:

```
void __assert_fail(const char *assertion, const char *file, unsigned int
line, const char *function);
```

So, we can see that this check is at line 99 of a file called dl-minimal-malloc.c, and that this function is called `__minimal_realloc()`. If we lookup this file name online, we find out that this isn't actually a custom allocator, it's a simple allocator used by the linker (dl, dynamic linker) during the initial linking phase. I found the source here, though the line numbers don't match.

I still reversed the allocator, because even though I didn't think this challenge would be about a 0day in the linker's allocator (lol), I thought that it might have been modified to introduce a vulnerability. And besides, understanding how it worked would definitely be essential for the exploitation process, even if the vuln were somewhere else.

The allocator

In the end, the only difference to the source that I found is a missing piece of code in `malloc()` which makes the allocator use the leftover space in the linker's data segment to serve chunks before starting to `mmap()` more memory.

The only references to the heap that the allocator keeps are 3 pointers: `alloc_end` points to the end of the mmap that chunks are being served out of, `alloc_last_block` points to the last allocated chunk, and `alloc_ptr` points to right after it, to the 'top chunk' (different from the standard malloc's top chunk because it doesn't store any metadata in it, nor in any other chunk).

Malloc

If there's not enough space in the mmap chunks are currently being served from to fulfill this request, or if the heap hasn't been initialized yet, a new one is created with enough space for the new chunk plus 2 pages. This moves `alloc_end`, as well as `alloc_ptr` (the top chunk), unless the new mmap is immediately after the current one, meaning it basically just expanded the top chunk.

Now, `alloc_last_block` is updated to the current top chunk, which is then shrunk by the desired size (by *adding* to its pointer); `alloc_last_block` is returned.

Free

If the target chunk is the last given chunk, the one adjacent to the top chunk (`alloc_last_block`), it's consolidated and memset to zero. Otherwise, the memory is lost forever. We can't consolidate more than once in a row though, so even if we free our chunks in a LIFO order, their memory is still leaked.

Calloc

Since freed chunks aren't recycled, except for consolidated chunks which are memset to zero, the chunks given out by `malloc()` are guaranteed to be initialized to zero. Because of this, `calloc()` just calls `malloc()` with `nitems*size`.

Realloc

If the given pointer is null, a new chunk is allocated with `malloc()`.

If the pointer isn't the last allocated chunk (`alloc_last_block`), the function panics; this is the `__assert_fail()` we saw earlier. Only letting the last chunk be reallocated means we can just 'expand' it as long as there's enough space in the top chunk.

The chunk is consolidated just like `free()` would do, and `malloc()` is then called. If there's enough space in the top chunk, the same address will be returned; if there isn't, `malloc()` will `mmap()` more memory and return a different address; the contents of the old chunk will be copied into the new one before returning it.

The program

Objects are stored in a linked list and represented by a struct:

```
typedef struct object_t {
        size_t    size;
        object_t* next;
        char      props[];
} object_t;
```

This is a variable-size struct, the props array's size depends on that size field. A reference to the list containing all objects is stored in a global `object_t* root` variable.

Most of the commands are pretty simple: the 'new' command allocates a new object and sets it as the new root, linking it to the rest of the list. The 'delete' command unlinks the target object from the list. The 'get object size' command just prints the target object's size field, and the 'print' command just prints all the keys and values as parsed when reading them. The 'set object properties' command though…

It first reads the raw properties into a global `char props_buf[4096]` buffer, then it checks if the new properties fit in the object, reallocating it if they don't, and finally it parses them into the `<key1 str>00<val1 str>00<key2 str>00<val2 str>00` format, putting the resulting data in the first object's props buffer.

The interesting code here is the parser, since it's the only function that writes into the objects on the heap (except for touching the size & next fields), and it's also easily the most complicated function in the whole program, a perfect place for bugs to crop up.

The parser algorithm

It first copies the raw input into the object's buffer and sets up two iterators for that buffer, one used for reading (`char* it`), and one used for writing (`size_t written`, used as an index); all reads and writes advance these iterators, 'consuming' the input. The reason for having separate read and write iterators is to ignore garbage (like a value with no key `"foo;"`), overwriting it.

It then enters the main parsing loop, where it parses a key-value pair in each iteration; the loop:

1. Finds the size of the key by looking for an '=', exiting the loop if it gets to the end of the input, or jumping to the next cycle if it finds a ';' (a value with no key, considered garbage; the read iterator is advanced, consuming it, but the write iterator isn't, eventually overwriting it).

2. Finds the size of the value by looking for a ‘;’, exiting the loop if it gets to the end of the input.
3. In this step, if this isn’t the first cycle, it outputs a null byte, null-terminating the last value string that was outputted (done in later steps); the parser lazily null-terminates the strings it outputs, either doing it before outputting the next one, or before returning completely.
4. Outputs the key, null terminates it, and outputs the value.
5. Unless this was the last value, it consumes the value by advancing the read iterator. The idea here is that if this is the last value, even if it isn’t consumed, the parser will stop in the next cycle while trying to find the key size, since it’s going to find a null byte instead of an ‘=’.

After leaving the main loop, it outputs a final null byte, null-terminating the last value string.

Here’s a commented pseudo code of the parser, to better understand it:

```
void parse_props(char* dst) {
        // Copy input & setup iterators
        strcpy(dst, props_buf);
        char* it = dst;
        size_t written = 0;

        while (true) {
                // Step 1: find the key size
                char* key = it;
                size_t key_size;
                for (key_size = 0; it[key_size] != '=' && it[key_size] !=
';' && it[key_size] != '\0'; ++key_size);

                // Handle garbage
                if (it[key_size] == '\0')
                        break;
                else if (it[key_size] == ';') {
                        it += key_size + 1;
                        continue;
                }

                // Step 2: find the value size
                it += key_size + 1;
                size_t val_size;
                for (val_size = 0; it[val_size] != ';' && it[val_size] !=
'\0'; ++val_size);

                // Step 3: null-terminate the previous value
                if (written > 0)
```

```
                        dst[written++] = '\0';

                // Step 4: output the key & value
                while (*key != '=')
                        dst[written++] = *key++;
                // Null terminate the key
                dst[written++] = '\0';
                for (size_t i = 0; i < val_size; ++i)
                        dst[written++] = props_buf[it - dst + i]; // looks
'sus' but works

                // Step 5: if this wasn't the last value, consume it
                if (it[val_size] != '\0')
                        it += val_size + 1;
        }

        // Null terminate the last value
        dst[written] = '\0';
}
```

Breaking the parser

Even though the parser algorithm isn't very complicated, I was still reluctant to study it to find the vulnerability, which is why I thought that before doing that, I might as well try to see if I could break it with sheer brute force, after all everyone knows that pwn is really just about sending a lot of AAAA's and making the program explode 😁

In this case, sending a lot of A's didn't do anything, nor did sending a lot of ;'s. However, when I sent a lot of ='s…

```
> n
> s
Properties (format: foo=bar;baz=123;...):
================================================================================
================================================================
Segmentation fault (core dumped)
```

Sweet! But why does this happen? Well, let's try to go through the steps of the main parser loop and think about how it interpreted this input:

1. Finding the key size: it's going to stop at the first character, finding a key of size 0 and advancing the read iterator by 1 byte.
2. Finding the value size: it's going to count the rest of the input as a single value, only stopping at the last null byte.
3. This doesn't do anything since this is the first cycle.
4. Outputting the key and the value: the key is empty, so it's just going to output its null terminator. Then, it'll output our value of N-1 (with an input of size N) ='s.
5. Consuming the value: it considered the whole input as a single value, ending with a null byte, meaning that it won't advance the read pointer, and will instead go to the next cycle, assuming that it's going to notice the input has ended and stop parsing.

The problem is in the fifth step. It should always consume the value, or outright exit the loop if it sees it's just parsed the last value. Since it doesn't do this, the next cycle will start with a read iterator advanced by 1 byte, a write iterator advanced by N bytes (the key's null terminator and the N-1 bytes of the value), and an input of still just ='s, just 1 less. It won't detect that the input has ended because it's going to stop at the '=' rather than at the null byte.

The parser outputted N bytes but only consumed 1, and it's going to do that again in the next cycle, never performing any bounds checks. This gives us a huge buffer overflow! But how huge exactly?

In the first cycle we wrote N bytes, one null byte to null terminate the key and N-1 for the value. All the next cycles will write two null bytes, to null terminate the previous value and the current key, plus N-i for the value, for N+2-i bytes in total.

This can be expressed as N plus the sum of the numbers in the range [2;N], meaning that with N bytes of input we can generate $N + \sum_{i=2}^{n} i$ bytes of output, which can also be written as $N + \frac{N(N+1)}{2} - 1$.

That's a lot! Also, sorry for the math 🤓

We can send stuff other than ='s too, for example, if we want to write "ABCDEF", we can send "======…ABCDEF". The only problem is that we can't easily control where we place it, since adding just one '=' increases the overflow size by a lot. We can add a constant offset to the payload by just adding garbage bytes before the '=' chain, like this: "AAAAA======… ABCDEF". This is enough control for the exploit.

Exploitation

The program has all protections turned on, and it's seccomp'ed to only allow exit, exitgroup, mmap, open, openat, read, and write syscalls.

We have a heap overflow from the last allocated object (the only one whose properties we can set), but since the allocator doesn't recycle freed chunks our chunk will be the one bordering the top chunk. Chunks don't store metadata, so how can we exploit this overflow?

If we allocate enough memory we're going to fill the heap's underlying mmap, and the allocator is going to `mmap()` a new one. If we look at these 2 mmaps' addresses in gdb, we'll see that the new one is right before the old one, acting as a big contiguous piece of memory.

This means that we can overflow from a chunk in this mmap back into a chunk allocated earlier in the first mmap! We can now overwrite a chunk's size and make it bigger allowing us to read out of bounds and get some leaks; these mmaps are right before the TLS, letting us leak pointers to all the major areas of memory: the program, stack, libc and linker.

I actually found that these leaks weren't reliable at all, and I had to use heuristics to filter them and get what I needed; I exclude anything that isn't 6 bytes long, the length of most userspace addresses when ASLR is enabled, I leak the libc via a pointer to `_nl_global_locale`, finding the right address by subtracting this symbol's offset from libc's base and checking if the result is page-aligned, I leak the stack by excluding addresses that don't have `0x7ff` as their highest 12 bits, and by removing possible libc addresses by filtering for addresses after the libc base plus its size as seen from gdb. I ignore linker and program leaks since I don't need them for my exploit, and it also made it not work remotely.

We can now overwrite a next pointer of the linked list to forge our own objects that we can read from and write to, achieving arbitrary rw. I used this to fake an object in main's stack frame to ROP.

The only obstacle in doing this is that we can't send null bytes directly since the parser works on strings, meaning that we can't send the whole chain at once. This is easily fixable though, we just need to send the last, furthest, gadget first, and then use the null terminator put at the end of the properties to fix the previous gadget's high null bytes.

We also have to place our object after a big enough size value to be able to send our chain.

The last problem is the seccomp, but this can be bypassed by using the ROP chain to `mmap()` a rwx page, read it, and return to it, and then sending a simple shellcode that opens the flag file, reads the flag into memory, prints it, and exits.

And boom 💥 we're done!

This challenge gave me 2 headaches while trying to find the vulnerability, before I understood that the allocator wasn't actually a custom implementation. So much for "No Headache".. So

yeah, I enjoyed it quite a bit 😄

I think the challenge could have been more fun if it had been made clearer where the allocator came from though.