Blog

Analysis of Malloc Protections on Singly Linked Lists

6/5/2020

Introduction

On May 21st 2020, Checkpoint Research released a post that detailed the results of a security patch to multiple variations of Malloc, including the Malloc used in GLibC. This patch attempts to fix the inherit insecurities within *singly linked lists* being used as a bin data structure. In this article, I will be diving into how the security patches work and what this actually means for binary exploitation. For this specific article, we will dive into the patches put into GLibC's Malloc. However, the concepts of the mitigations will directly apply to all other versions of Malloc (just not the specific

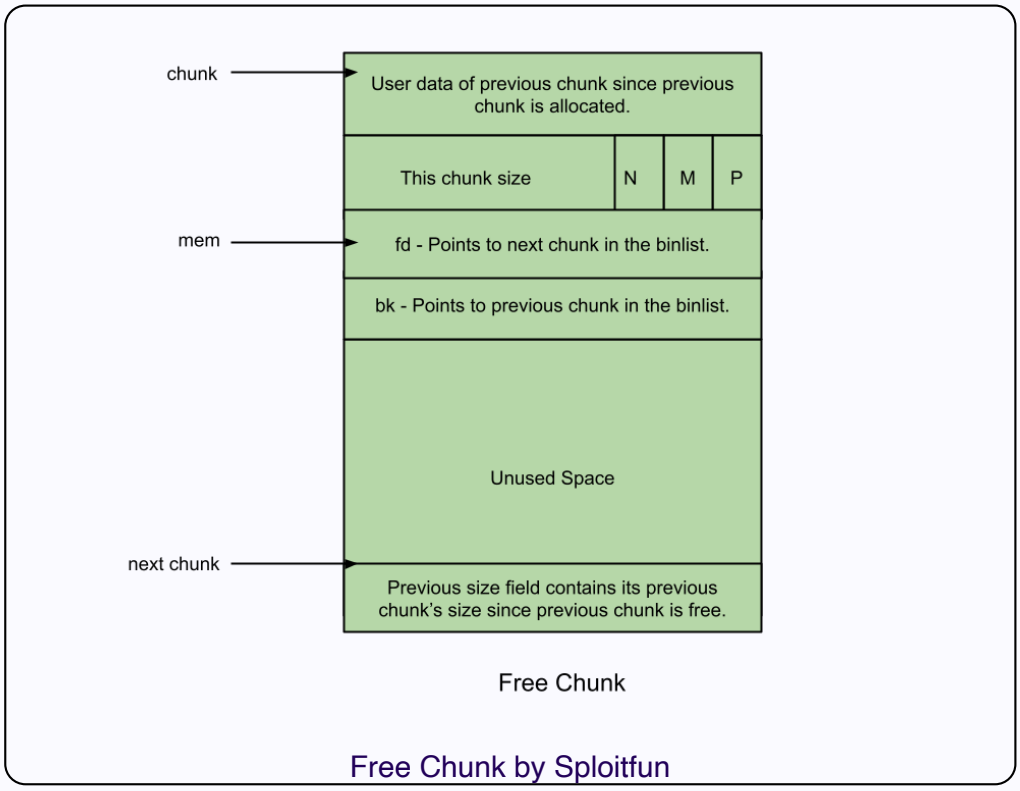
implementation details).

GLibC Malloc

In order to understand the rest of the article, a primer on the data structures used within GLibC's Malloc is required. For the purpose of this article there are two main data structures: *chunks* and *bins*.

Chunks

Chunks are the main object that users interact with. There are two main states to a chunk: *allocated* and *free*. The following picture demonstrates a *free* chunk:



Free Chunk by Sploitfun

The first field is the size of the previous chunk (prev_size). This is only used if the previous chunk is free. The second field is the **chunk size**. The chunk size represents the size of the data of the chunk and the metadata about the chunk (the first 3 bits of the size).

On a free chunk, the third field is used in order to store a pointer to other freed chunks. This field is known as the

forward pointer or the *Fd* field. The fourth field is exactly the same, except that it stores a backwards pointer or *bk*. The fd and bk pointers are unused when the pointer is allocated; they are used as part of the data section for the chunk. For the purposes of this article, that is all that needs to be known. For further information, please to refer to Sploitfun's amazing article about GLibC.

Bins

Within GLibC, bins are a list of *currently free chunks*. All bins exist as some sort of linked list data structure. There are multiple types of bins, which vary in chunk size and speed. For the focus of this blog post, we only need to know about two types of bins: *fastbins* and *tcache*.

Both the tcache and fastbins are singly linked lists of free chunks. This means that the Fd ptr (third field of a chunk) points to the *next* chunk that is free (in the bin). Both of these bins do not use the bk pointer. Fastbin and Tcache bins have many differences, such as speed, security checks and more. But, for the purposes of this article, this is all that is needed to be known.

Singly Linked List Security Issues

Since the early days of the original unlink exploit, many security and sanity checks have been put into GLibC to check for corrupted chunks. Other bins (unsorted, small and large) use a doubly linked list, which can be used for many checks to ensure that the chunk or bin has not been corrupted. However, in the case of singly linked lists, these chunk checks are not possible to perform. This is bad. But why?

Given the ability to overwrite a singly linked list Fd pointer, it is possible to set this pointer to anywhere in memory. Then, when a call to Malloc is made, this *fake chunk* (the location of the Fd ptr that we wrote) will be returned to the user. At this point, with the memory pointing to an arbitrary location,

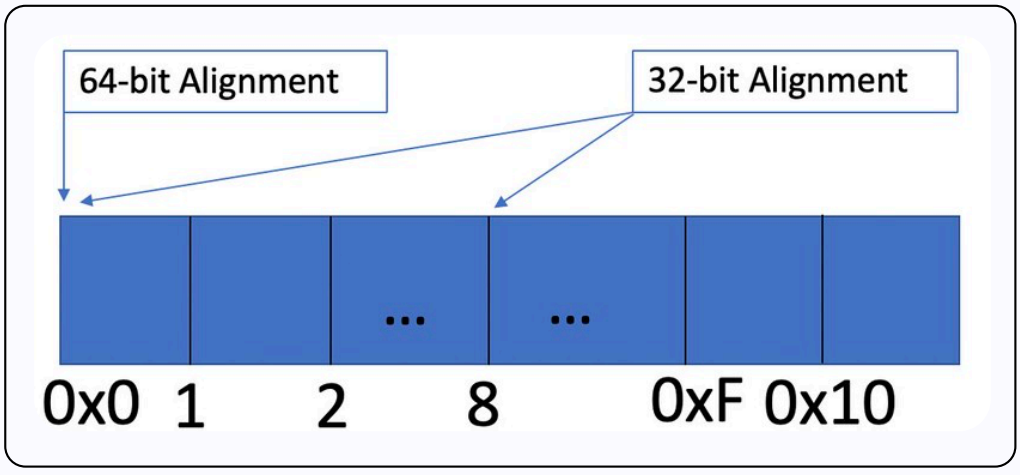
this spot can be written to. From this primitive, it is possible to **gain code execution** in multiple ways by overwriting function pointers or by creating other primitives off of this bug.

Security Changes

As mentioned above, singly linked lists Fd pointers in Malloc are quite exploitable and pose a major threat to security. Because of this, Checkpoint Research decided to add new security safeguards onto singly-linked lists. These boil down to two main upgrades: *chunk alignment enforcement* and *pointer mangling*. These will both be discussed below.

Chunk Alignment Enforcement

Malloc only gives out chunks in 8 byte (32-bit) or 16 byte (64-bit) groupings. Hence, chunks should only end with 0x8 (32-bit only) and 0x0. The added securiy check validates that the alignment for a given chunk is only one of the expected locations. The image below demonstrates the locations where a valid chunk can point to for a set of 16 bytes.



This simple sanity check on the alignment of the chunk restricts the amount of places in which a fake chunk (a chunk address that an attacker invents) can be created at by 7 (out of 8) spots in 32-bit and 15 (out of 16) spots on 64-bit. The implications of this security patch will be discussed further down in the article.

Pointer Mangling

An overview of this feature is that the fastbin and tcache Fd pointers are now obfuscated in order to prevent leakless techniques and relative overwrites from working.

The formula used for pointer mangling looks like the following:

`| New_Ptr = (L >> 12) XOR P |`, where L is the *Storage Location* and the P is the fastbin/tcache *Fd Pointer*.

Because the Fd pointer and storage location are already randomized by *Address Space Layout Randomization* (ASLR), New_ptr benefits from the randomness at no cost.

In words, this simply XOR's the storage location of the pointer and the Fd pointer itself. Prior to this XOR, the storage location is shifted 12 bits to the right. This is done because the least significant 12 bits of the storage location and the Fd pointer are deterministic. So, in order to have randomness on the first 12 bits of the Fd pointer, the shift is needed. Otherwise, a relative overwrite could still easily be performed because the final 12 bits would always be the same.

Demangling the pointer is the exact same formula as above. The mangled pointer is now *P* and the storage location *L* stays the same as before. Once this value is XOR'ed, the original Fd pointer appears again.

This is a good overview of the pointer mangling. However, if you would like to see more, please read the write up from the creators of the GLibC patch here.

Security Implications

Both of the changes to Malloc above have mild to significant effects on the art of binary exploitation. These implications will be discussed below in detail.

Implications of Chunk Alignment Enforcement

Although this seems small, this is a huge change for Malloc. On a 64-bit binary, this restricts the location of a fake chunk to only 1 out of every 16 addresses! Practically, this makes exploitation more difficult, especially in binaries where little control is given to the user on the values in the program. Exploitation is no where near impossible, but each constraint that is added it makes exploitation slightly more complex and difficult to pull off. In the future, all fake fastbin and tcache chunks will need to be aligned in order for the chunk to be usable.

Fastbin Attack on __malloc_hook

Another scenario to consider is the classic attack used to overwrite a function pointer (__malloc_hook) to eventually gain code execution. When allocating a chunk from a fastbin, the chunk size is validated to be the same as the fastbin size itself. If this fails, then Malloc aborts. In order to bypass this security check an address close to the __malloc_hook (__memalign_hook) is used in a *misaligned* way in order to get the valid fastbin chunk size of 0x7F. An example of this can be found here. By adding an alignment constraint, this attack is no longer viable for fastbins.

Implications of Pointer Mangling

In the Checkpoint Research article, the main goal of pointer mangling is outlined: remove partial overwrites and full overwrites (without a memory leak). The implications of this will be discussed below.

Bye Byte Relative Overwrites

Prior to this patch, a partial overwrite on a heap chunk

pointer was quite common. A partial overwrite is exactly what it sounds like: overwrite part of a pointer.

For example, we could point a heap chunk to an entirely different location by changing the pointer from `0x80000080` to `0x80000040`. This one simple trick would give us the ability to point chunks to different locations without knowing the actual addresses on the heap! An additional example of this being used can be seen in the leakless House of Roman, which uses relative overwrites on 3 separate occasions to eventually get remote code execution.

Now, without a heap leak relative overwrites will require brute forcing in order to pull off. In a situation where a particular byte is needed, the exploit would only work *1/256* times, or a full byte of brute forcing. This makes exploitation much more difficult than previous versions of Malloc, as an added level of non-deterministic computation has been added.

Tcache Bin/Fastbin Attacks Become More Difficult on Non-Heap Locations

A common attack is to set a Fd pointer to the location of a function pointer. This works well because when the chunk is allocated over the function pointer, the pointer can be overwritten with something else, such as system or a one_gadget. As mentioned previously the __malloc_hook is a good target. This is an outline of an example attack to overwrite a function pointer:

1. Leak LibC address to determine location of __malloc_hook
2. Free a chunk into the Tcache Bin
3. Overwrite the Fd pointer of the tcache chunk with an address to __malloc_hook
4. Allocate the chunk that points to __malloc_hook
5. Set __malloc_hook value to system or a one_gadget to get code execution

The attack listed above is very popular and common. The attack requires only a LibC address leak. With the addition of pointer mangling, this attack no longer works. Faking a

heap pointer into GLibC would require the pointer to be mangled before being inserted (in step 3). This additional step requires a heap address leak in order to mangle the pointer in step 3, significantly raising the bar for the exploitation process.

This feature also makes a heap leak required to create a fake chunk within the *stack*, *.bss* section or in the *PLT/GOT*. This is the same as the explanation above for LibC, just expanded to all other non-heap locations to create a fake chunk. Overall, this makes heap exploitation significantly more difficult.

Leaking Heap Addresses

Overall, the Pointer mangling will force hackers to go after the pointers in the unsorted, small and large bins in order to get heap addresses. This is because the pointers in the unsorted, small and large bins are not mangled. The Fd pointers in fastbin and tcache bins are no longer viable options for heap leaks. Well, kind of...

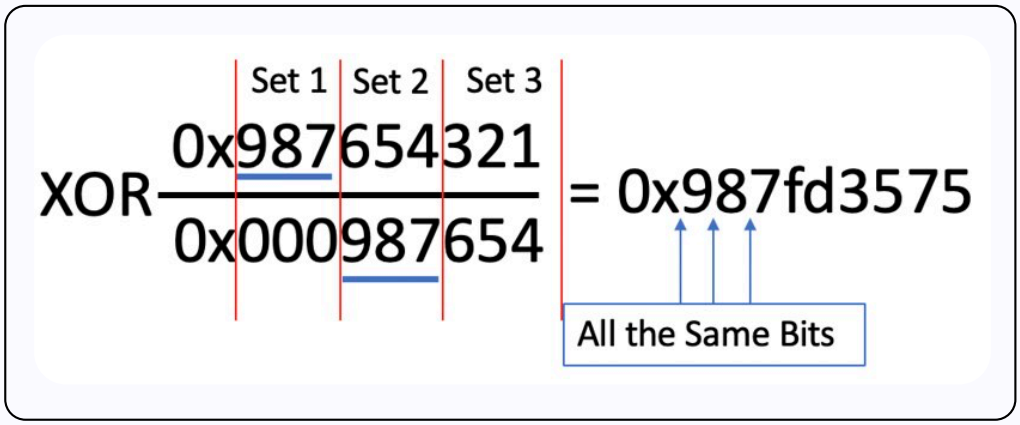
When I originally read about the pointer mangling, I assumed that all leaks from fastbin and tcache pointers would be completely useless. However, after playing around with this for a while I discovered a way to **demangle a mangled pointer** that can be done prior to a heap leak (with some restrictions).

Demangling Pointers Prior to Heap Leak

Recall the algorithm for mangling and demangling pointers: `| New_Ptr = (L >> 12) XOR P |`, where *L* is the storage location and *P* is the Fd pointer.

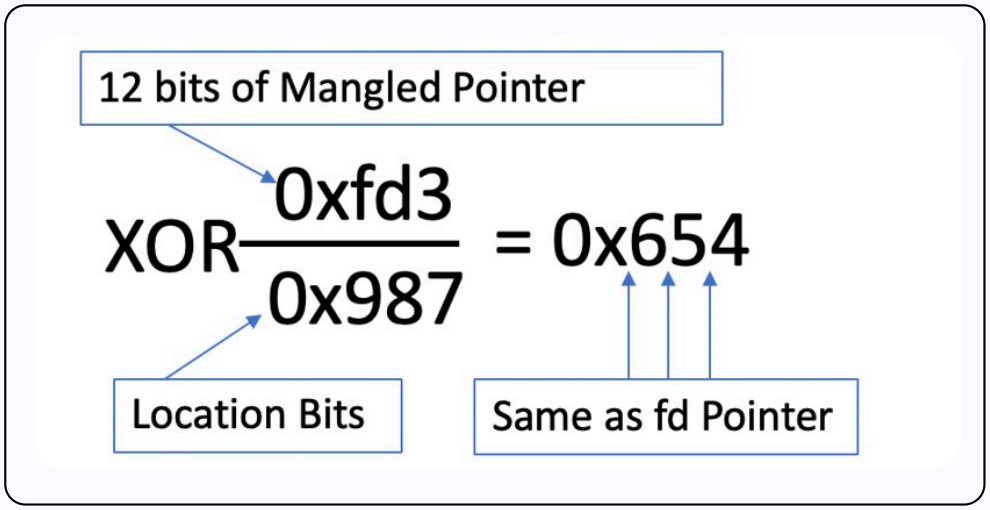
The storage location and the Fd pointer will likely have the same number of bits and share the most significant 12 bits. Then, when the location is shifted, the following math occurs: `(12 left-most Bits of P) XOR (0)`. This leaks the most significant 12 bits of the pointer.

For example, let's take the storage location of 0x987654987 and the Fd of 0x987654321. In the example image below, the storage location has already been shifted.



From the formula, it is easy to see *why* we have leaked the 12 left-most bits of the Pointer, as it just XOR's with 0x0.

Another key observation can be made: *set 1* of the bits from the Fd pointer is exactly the same as the *set 2* bits on the storage location! Why is this a key observation? With XOR, if two of the values are known then the third can be recovered because XOR is reversible. Because we know both the output and the storage location we can calculate the bits for the Fd pointer! This is done by XORing the two known values together. An example of this is shown below:



Full Algorithm

By generalizing the explanation above, it is possible to recover both the storage location and the Fd pointer of a mangled pointer. There is one big assumption though:

- P and L need to be on the same page for this technique to work (S/O to RJ Crandall for the correction on this. Throughout this blogpost, the assumed page size is 12 bits, just for simplicity). The more bits that are in common, the more information that can be leaked. Additionally, even if the memory is not on the same page, this technique can still be used to work a fair amount of the time (depending on the locations of both the Fd and Storage Location Ptr, the percentage of accuracy will vary). However, that is out of the scope for this post.

Although the above statement is not always the case, this can happen through careful heap allocations and freeing (heap feng shui).

The process described above can be continued until the end of the pointer is hit. Here is the algorithm:

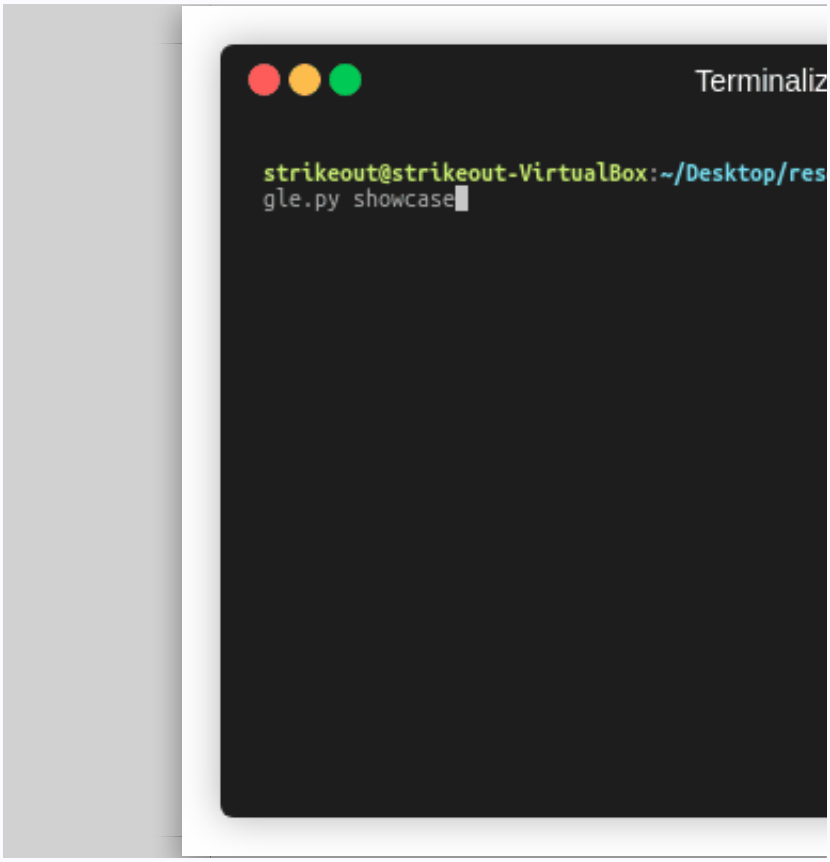
1. First, we KNOW the top 12 bits of the heap. Because of the shift, the bits of the storage location pointer are given to us for free, for round 1. Later, the bits from the output of step 3 can be used.
2. Next, we XOR this known set of bits with the next most significant (to the right) 12 bits of the mangled pointer. This outputs the value of the 12 bits of the Fd Pointer. This is seen in the picture above.
3. Repeat until we are at the end of the mangled pointer. The 'known bits' (from step 1) for the next operation are just the output from step 3, shown above.

At the end of this, the output contains both the address of the Fd Pointer and the Storage location.

Note: It should be known that the storage locations final 12 bits are not recoverable with this operation, because they are shifted out. However, because these bits are deterministic, they can be added back in for the storage location after the algorithm has been completed. For mangling a Fd pointer (at this storage location) the last 12 bits are not needed though, as they are shifted out anyway.

Mangle

Using the algorithm discussed above, it is possible to unmangle these pointers in real time! This is not just some magic trick that theoretically works; this can actually be used! I wrote up **working scripts** that can do all of the mangling and demangling described above (including the leakless demangling). Below is a demo of the CLI portion of the tool:



To access this tool, go to mdulin2/mangle to use or play around with. The repository also includes a LibC and Loader that have the new Malloc source code, a modified Malloc Playground with a nice Python interface via pwntools and examples to understand how to bypass the mangling mitigations. This has both a CLI (via Fire) and an importable interface to it. Eventually, I hope something similar will be included into GEF and pwndbg by default.

Conclusion

Overall, binary exploitation is no where near dead. But, this is a big increase in the security of Malloc. As an overview, it has the following consequences:

- Fd Pointer Leaks are no longer free; they require some heap feng shui in order to get a full leak.
- All fake chunks must be aligned (reducing the amount of places where a fake chunk can be at)
- Relative overwrites now require a significant amount of brute forcing (likely a full byte)
- Fake Fd pointers need to be mangled with the storage location, which requires a heap leak
- Heap leaks on Fd pointers now need to be demangled to be usable
- Fake chunks in non-heap locations now require a heap leak

Hope you enjoyed the article and learned something interesting about Malloc today. Major S/O to @seiranib for taking the time to review this article before publishing. Feel free to reach out to me (contact information is in the footer) if you have any questions or comments. Cheers from **Maxwell "ꓘ" Dulin**.

Afternote

Throughout the article several simplications were made. The goal was to make this understandable by the average person; hence, some items were left out or not explained for the 'full truth'. Here is a list of them in order to come clean and not have the redditors go crazy on this article:

- The value *12* used in the algorithm (for mangling) is actually PAGE_SHIFT. However, in the normal case, this is just 12.
- When using the *prev_size* field, it is only used when the chunk below is free and NOT in the tcache or fastbin. This is because tcache and fastbin (although in a bin) do not remove the in_use bit of a chunk, even though they are free.
- In theory, the brute forcing (of relative byte) can be as good as 1/16 if you have extremely good control over the allocations. But, that comes with other complications and tricks beyond the scope of this article.
- Fastbin and Tcache bins have many differences that

were not explained in this article as it was just not necessary. For more information on bins, read this great article by Azeria Labs.
- Other researchers in this article go over similar content. I independently did this research and came to similar conclusions; they just published their research first.

Maxwell Dulin