École Polytechnique Fédérale de Lausanne

House of Scudo

by Elias Valentin Boschung

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Mao Philipp Yuxiang
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 6, 2024

If debugging is the process of removing software bugs,
then programming must be the process of putting them in.
— Edsger Dijkstra

Dedicated to my study companion since Covid-19, my plush bear.

Acknowledgments

I thank first and foremost my family for their unconditional support throughout all of my studies until this point. I also thank my supervisor Philipp for all the tips and help in writing this project. Thanks to my friends, those who inspired and encouraged me on the path of cybersecurity, as well as those who helped me relax when I was stressed. Thanks also to the HexHive lab for the opportunity to do this Master Semester Project, and also for providing the template for this report, which is available at `https://github.com/hexhive/thesis_template`.

Lausanne, January 6, 2024 Elias Valentin Boschung

Abstract

Heap bugs are a very frequent issue in C programming, due to the quite high complexity of memory management for programmers. Attackers can exploit such heap bugs like overflows, double free's and similar to abuse the underlying mechanics of the heap allocator and get even bigger control over heap memory such as arbitrary writes. For the common libc allocator there are a lot of known techniques to achieve that, a few of which are known as heap exploitation houses. Scudo is an allocator used on Android 11+ that is supposed to be specifically hardened to prevent or at least complicate such attacks. [5] However, there exists no documentation of any tests of the resilience of Scudo to a dedicated attacker.

In this project we looked for exploits using these heap vulnerabilities and targeting the mechanics of Scudo specifically. We discovered two exploits that use different heap vulnerabilities and mechanics of the Scudo allocator to trick it into returning a chunk at an address chosen by the attacker. These two exploits therefore enable an arbitrary write primitive similar to the primitives provided by some libc exploitation houses.

In addition to the attacks, we propose a mitigation that prevents either of the discovered exploits by targeting a mechanic of the Scudo allocator that is used in both of our exploits and which further hardens Scudo. We show that the impact of our mitigation on the allocator's performance should be negligible in most common scenarios.

Contents

Chapter 1

Introduction

Heap vulnerabilities are a frequent and dangerous attack surface and have been for a long time, as even Microsoft already stated back in 2013 that they were the most common type of vulnerability that they addressed through security updates. [4] These vulnerabilities can be used by an attacker to get remote code execution, essentially allowing the attacker to take control of the device. Attackers often leverage the underlying mechanisms of the heap allocator itself. For the widely used glibc allocator there exist generic techniques that, given some preconditions, give a pattern to escalate a heap vulnerability and to get remote code execution. like the unsafe unlink exploit or the list of libc heap exploitation houses. Such heap vulnerabilities are also relevant in the Android environment, as can be seen by the example of a remote code execution found in WhatsApp back in 2019, which was caused by a double free. [1]

To harden applications on Android against such attacks, it does not use the popular glibc allocator, but since Android 11 uses a new heap allocator called Scudo. [5] Scudo was developed specifically with security in mind, to reduce the possibilities of an attacker to leverage the allocators mechanisms to get an arbitrary write primitive. Scudo uses a checksum to secure critical metadata stored in-line, and it also tries to isolate some of the metadata, specifically all the free list related metadata, to its own region, to reduce the risk of possible corruption with a simple overflow. Since it is designed completely differently, there is currently no existing study on an attack that uses the known techniques for the glibc allocator to exploit Scudo.

Therefore, in this project we try to find techniques to exploit mechanisms from the heap allocator itself, similar to the way such techniques exist for the glibc heap allocator. We start by defeating the checksum with a single leak of the metadata of a chunk, including the corresponding checksum. Then we show two exploits, that build on defeating the checksum to forge different headers and tricking Scudo to process them without any errors, which will allow us to gain an arbitrary write. In addition to these two exploits, we also build some additional tools to facilitate further research and exploiting of Scudo, such as a python library to break the checksum and some templates for

different exploit approaches to Scudo.

Finally, we detect that the two attacks we found both rely on a common mechanism, being how Scudo handles larger chunks compared to smaller ones. In order to protect against these attacks, we propose a mitigation that adds some additional checks to address that specific exploitation of the mechanism, and thus prevents these attacks. We evaluate our mitigation and find that the overhead on benchmarks is negligible. We therefore open a pull request to llvm, the official development team behind the Scudo allocator.

Chapter 2

Threat model

In our threat model, the attacker has access to some basic heap vulnerabilities, such as overflows, double free's and arbitrary free's. Additionally, if address space layout randomization is used (ASLR), the attacker also has access to an ASLR leak through a channel, allowing him to know where some memory regions are loaded. These heap vulnerabilities can give the attacker some control over a restrained space of heap memory. The goal of the attacker however is to get an arbitrary write primitive, them to get full control of the program. We don't consider any vulnerabilities in application specific data, so the attacker from our threat model should only exploit the allocator's mechanisms and not rely on any application data, such as data or code pointers stored on the heap.

When using the glibc allocator, or some other unhardened allocators, these basic heap vulnerabilities can directly lead to an arbitrary write primitive, by using pointers stored in the controllable heap memory. In glibc this can for example be achieved by using a tcache forward pointer corruption, which exploits the fact that the cache linked list is stored inside the free chunks themselves. Since these free chunks may be controlled by a use after free or an overflow for example, the cache can be corrupted and an arbitrary address inserted. An example of such a corruption can be found on [6].

However, Scudo is built with such security issues in mind, therefore the use of pointers in the same memory regions as the chunks with data is avoided as much as possible. Therefore, the attacker needs to trick some other, more complicated allocator mechanisms in order to gain the full arbitrary read or write primitive.

We do not consider any attacks on the quarantine, since it is not used in the configuration of Scudo used on Android devices. Even on the llvm website, the development team behind Scudo, they mention that quarantine may have a quite high impact on performance and is therefore disable by default.

Chapter 3

Scudo Security Measures

3.1 Basics

The Scudo hardened allocator divides the heap allocations it is asked to do into two big types of allocations, based on the size of the chunk to be allocated. The smaller chunks are handled by the primary allocator inside Scudo while the big chunks are handled by the secondary allocator, which stores some additional metadata differently.

3.2 Protected Inline Metadata

Since Scudo was designed to be hardened against some heap vulnerabilities, metadata that is stored inline with the data is avoided wherever possible. If it still needs to store some metadata next to chunk data, it tries to secure it in some way.

For the header data that is stored before any of the primary or secondary chunks, this is done by using a simple checksum with a random global secret, that is verified before any action with the chunk is processed by the Scudo allocator. The secondary allocator stores some additional metadata in front of that protected header. While this additional metadata is not protected by a checksum, all secondary chunks are protected by guard pages before and after the chunk, so the secondary header metadata can't be modified by using an overflow either.

The primary allocator stores all information related to its free list separated from the chunk data. The first level of the free list is stored in a thread-specific memory region, while the rest of the global free list is stored in a segregated region that is only used for allocator internal data.

3.3 Primary Memory Regions

The primary allocator separates its chunks into multiple size classes, depending on the size of the chunk. Each of these classes has its own dedicated memory region, increasing again the segregation between different chunk data. To further complicate the task of an attacker, the chunks inside each memory region are allocated in a randomized order, so that the attacker is unable to predict which chunk will next be allocated based on a sequence of previously allocated chunks without knowing the random seed used.

Chapter 4

Breaking the Cookie

To check the integrity of chunk headers, Scudo uses a simple checksum with a random 32-bit cookie which is used as seed to the hash algorithm. It may use one of two hashing algorithms, either CRC32 if hardware support is available for it, or the BSD checksum algorithm. The following pseudocode demonstrates how the checksum is calculated.

```
// Inputs
uint32 cookie = randomInt()
uint64 address
uint64 header

// Output
uint16 checksum

// Logic
If USES_HARDWARE_CRC32:
  uint32 intermediate = CRC32(cookie, address)
  intermediate = CRC32(intermediate, header)
  checksum = (intermediate & (intermediate >> 16)) & 0xffff
Else:
  uint16 seed = cookie & 0xffff
  uint16 intermediate = BSD(seed, address)
  intermediate = BSD(intermediate, header)
  checksum = intermediate
```

This checksum protects against blind heap overflows, since in a blind heap overflow the attacker has no way to forge the correct checksum. Therefore, if the attacker tries to modify any values in the

header, Scudo will try to verify the checksum of the header before taking any action with the chunk, and will crash the program since the checksum will not match the data in the header.

However, these checksum/hash algorithms are not designed to be cryptographically secure. While the cookie is 32 bits long, the checksum is only 16 bits, and it is therefore possible to find a cookie collision and produce the same checksums the original cookie would have produced. Indeed, it is enough to get a single heap leak of a complete chunk header together with its address in order to recover the cookie and forge valid chunk headers. Recovering the cookie can be done by simply brute-forcing the cookie value until we find the same checksum that we leaked. Such a brute-force of the cookie was already used by the un1fuzz hobby research group, and another researcher used a z3 solver to achieve the same thing. [7] [2]

Using a C program brute-forcing the cookie takes around 1–2 seconds, therefore we implemented a C library that takes care of the checksum calculation and brute-forcing of the cookie given the necessary data. To facilitate integration with the pwntools python tooling often used for exploit development, we also provide python bindings for this library.

Chapter 5

Exploits

5.1 Change Class ID

This exploit uses the breaking of the checksum to modify some of the inline header metadata, to trick the allocator into processing a chunk differently. It is used in the two other exploits, which then take further advantage of the processing of the chunk that is different from the intended way.

5.1.1 Summary

Requirements:

- Header leak with corresponding address
- Free where we control the *0x10* bytes in front of the address

Result: Move a primary chunk to another size class, making potentially larger chunks in a small chunk region and multiple chunks overlap.

5.1.2 Requirements

The change class ID exploit needs some flaws in the target binary to be applicable. First we need to somehow leak a header value along with its address, in order to brute-force the cookie and forge new header checksums like explained earlier. Then the attacker also needs to have a free operation occurring, at an address in front of which we can write some bytes, specifically *0x10* bytes in front of the free target. Finally, there needs to be some primary chunk allocations after the free, which are located in the same class ID as our target we want to move the chunk to.

5.1.3 Explanation

For the retrieval of the cookie refer to the previous section for a thorough explanation on how it works.

Header field	Example Value before modification	Example Value after modification
ClassID	1	10
State	1	1
OriginOrWasZeroed	0	0
SizeOrUnusedBytes	0x18	0x18
Offset	0	0
Checksum	abcd	7654

Figure 5.1: Illustration of the modification to chunk header

Once the attacker has the cookie, he needs to find a vulnerable free operation, where he can write data to the *0x10* bytes in front of the free'd address. This can be achieved in different ways, such as an overflow while writing to a chunk of a small size, causing the attacker to be able to write

over the header of one of the next chunks. Another possible way to achieve the primitive can be controlling a pointer that is freed, and setting that pointer to a memory region that was set up to look like a fake chunk, thus freeing the completely fake chunk.

Though the exact offset from the first chunk which contains the overflow to the target header is random due to Scudo's defense mechanisms, the attacker can still manage to overwrite it, since the offset is a multiple of a fixed size that depends on the size class that is attacked. The target header might still be allocated before the chunk which the attacker can overflow, but with some random luck and if the attacker can execute the program as often as he wants, at some point the target header should be allocated at one of these multiples after the overflow.

The attacker then forges a new valid header with its corresponding checksum thanks to the brute-forced cookie. In this case the attacker probably wants to get a large chunk in the small chunk region, thus he will replace the header with one specifying a larger class ID instead of the original one.

When the chunk with the forged header gets free'd, it will be put into the cache of the larger class ID, and when a chunk from that same class ID gets allocated again, Scudo will allocate a larger sized chunk in the small chunk region.

5.2 Exploit CommitBase

In this exploit, the attacker modifies the unprotected metadata of a fake secondary chunk, to put an address of the attacker's choosing into the cache for secondary chunks. This injected address is then used to directly allocate a new secondary chunk at the address chosen by the attacker. The exploit relies on the Change Class ID exploit to get a fake secondary chunk, since real secondary chunks are protected against overflows with guard pages.

5.2.1 Summary

- Header leak with corresponding address
- At least one secondary chunk allocated
- Free on any address where we control *0x40* bytes in front
- Allocation of secondary chunk of known (approximate) size

Result: We can choose an arbitrary location for the secondary chunk allocated

5.2.2 Requirements

The forge secondary exploit has a certain number of requirements that the target binary needs to fulfill.

First we need a header leak of any chunk from the heap as well as its address, in order to brute-force the cookie. Afterward we can forge any header checksum from the cookie we calculated from this leak.

Second we need a free for which we can control a certain number of bytes before the free, more specifically we need to control *0x40* bytes in front of the address of the free. This is the size of the secondary header plus the size of the padded primary header.

Third we need there to be at least one secondary chunk allocated before the free, and we need there to be another secondary allocation afterward, of which we know the approximate size. This is the allocation where we will get our arbitrary access.

5.2.3 Explanation

The breaking of the cookie works exactly the same as explained previously, and the first part of this exploit also reuses a previously introduced exploit. Indeed, the first part is basically a change class ID exploit, except that we don't change the class ID to a different primary one, but to the special class ID 0, that is used to designate chunks from the secondary allocator.

The second part exploits the unsecured secondary header and the secondary cache. When freeing a secondary chunk, the info from its secondary header is put into the cache, and when allocating a new secondary chunk the cache is checked for a similarly sized space. Since we can change the secondary header of our fake secondary chunk, we can change the CommitBase and MapBase to any address where we want to allocate a chunk, and CommitSize and MapSize to our desired size. When setting the size we need however to pay attention to the fact that the cache is only checked for similar sized chunks, so the size we set can only be a certain number of pages larger than the next allocation where we want our chunk to be allocated. The exact allowed size difference may depend on the specific configuration of Scudo.

Following that, the next secondary chunk should be located at our chosen address. The special requirement that there was at least one secondary chunk allocated previously, is due to the way the free of a secondary chunk works. Indeed, freeing a fake secondary chunk that was not correctly allocated previously somewhat corrupts the tracking of allocated secondary chunks. While the list of allocated chunks itself is stored as a linked list with the chunks itself, and as such the freeing of the fake secondary chunk doesn't touch that list, it still modifies the number of items recorded for the list. Since this number is stored as an unsigned int, if there was no previously allocated

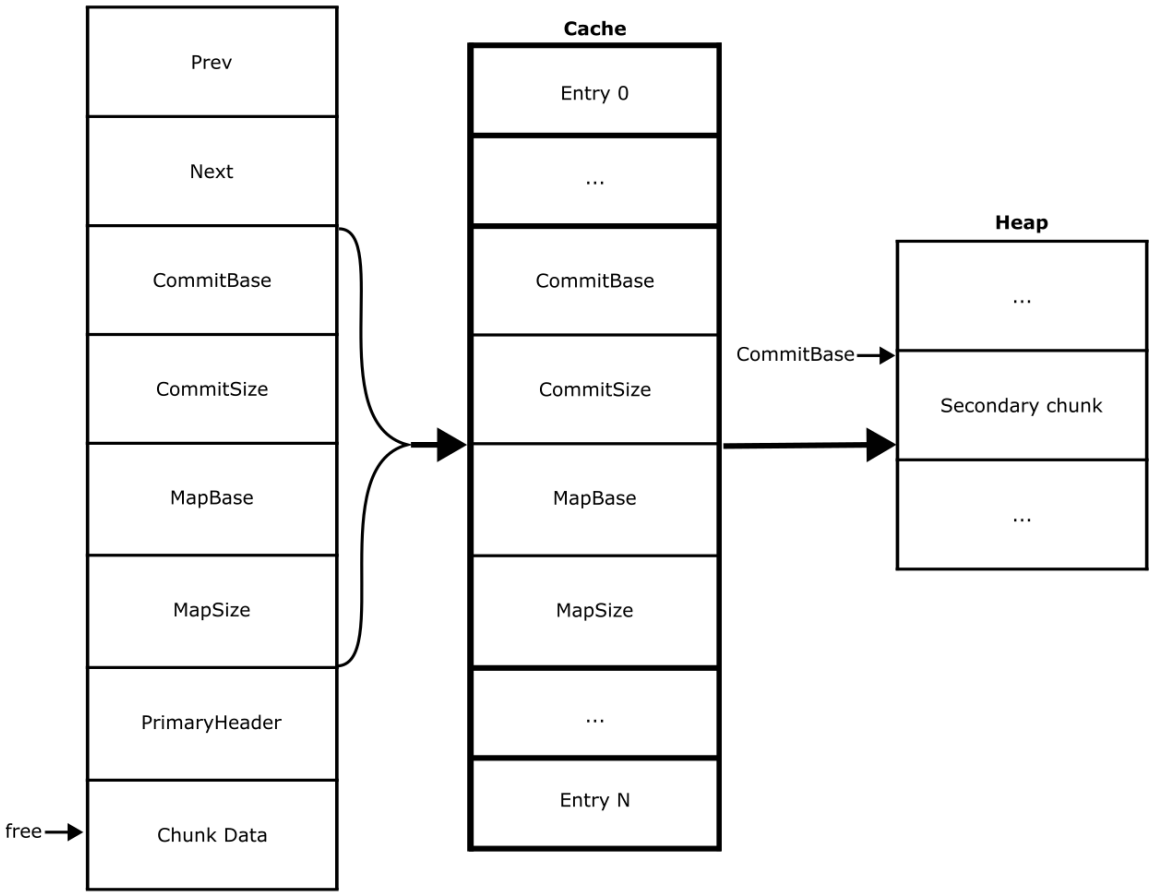

Figure 5.2: Illustration of how an attackers address gets from the header into an actual chunk on the heap

secondary block, our free of the fake secondary chunk would cause a negative overflow, and on the next allocation the program would crash, so before we could get our arbitrary chunk.

5.3 Safe Unlink Secondary

This exploit uses a trick similar to the libc safe unlink exploit, that tricks the linked list deletion function into inserting a pointer to Scudo's free list into the free list itself. The attacker achieves this by building a fake linked list, and is rewarded with the ability to write into the internal Scudo structures, which can be escalated into an arbitrary write.

5.3.1 Summary

Requirements:

- Header leak with corresponding address
- Scudo library base leak
- Three consecutive frees on the same address
- Control over the *0x10* bytes in front of the address before the first two frees
- Control over the *0x40* bytes in front of the address before the third free

Result: Next allocation will be a chunk in the PerClass structure itself (thread-local free list), allowing control of future allocations

5.3.2 Requirements

The safe unlink exploit has a certain number of requirements that the target binary needs to fulfill.

First we need a header leak of any chunk from the heap as well as its address, in order to brute-force the cookie. Afterward we can forge any header checksum from the cookie we calculated from this leak.

Second we need a free for which we can control a certain number of bytes before the free, more specifically we need to control *0x40* bytes in front of the address of the free. This is the size of the secondary header plus the size of the padded primary header.

Third we want to forge a fake linked list, for which we need a leak of the Scudo library base, and two more frees of the chunk in the previous step and being able to control the *0x10* bytes in front of the address to modify the header.

5.3.3 Explanation

As with most Scudo exploits, the first step is to get the cookie, which can be done from any single header leak, as explained in the dedicated section. This will be used later, to craft a fake secondary chunk by changing the class ID saved in the header to 0. This causes Scudo to look at the forged secondary chunk header and to parse some of the data contained in it, like the pointers to the previous and next in use secondary chunks. In this exploit we target the linked list pointers and the logic to remove an element from the linked list. An overview of what our forged linked list should

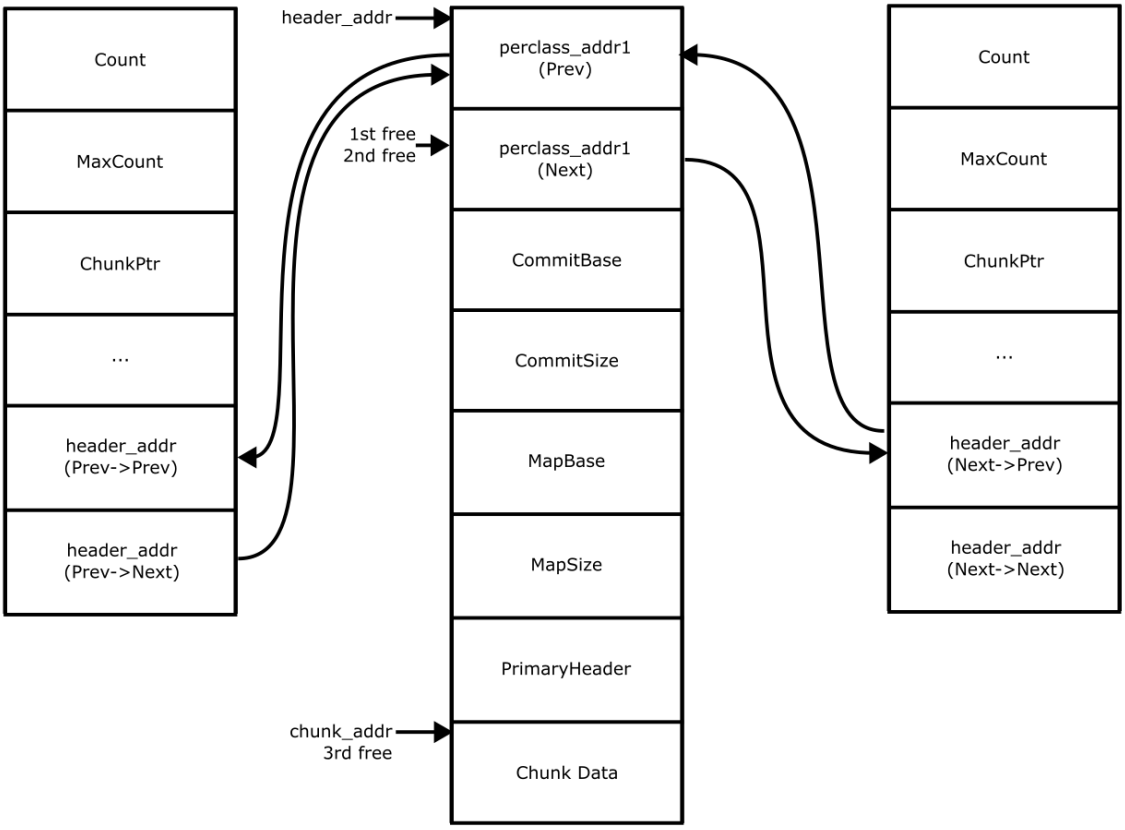

Figure 5.3: Illustration of forged linked list and the frees we need (left and right memory views correspond to the same memory)

look like can be seen in Figure 5.3. In that figure are also shown the locations of the three different frees that are needed and that are explained in more detail below.

Then we need to have two locations in memory that point to the chunk we will tamper with in the next section. The way we achieve this is by having two frees on that same chunk, leaving its address twice in the PerClass structure next to each other. The PerClass structure constitutes the first level, thread specific free list of chunks. It is handled as a simple array of chunk addresses, and therefore freeing the same chunk twice leaves two pointers to that chunk right next to each other in the PerClass structure. In order to not trigger double free detection, we just need to modify the header of the chunk between the two frees to set its status to Allocated again. Since Scudo completely trusts the header and doesn't try to detect a double free in any other way, this is sufficient to get the double free without Scudo throwing any error.

Once we have set up the two addresses in the PerClass array, we need to know its address still. To calculate the address of the PerClass array(s) we need to have a leak of some Scudo address, like

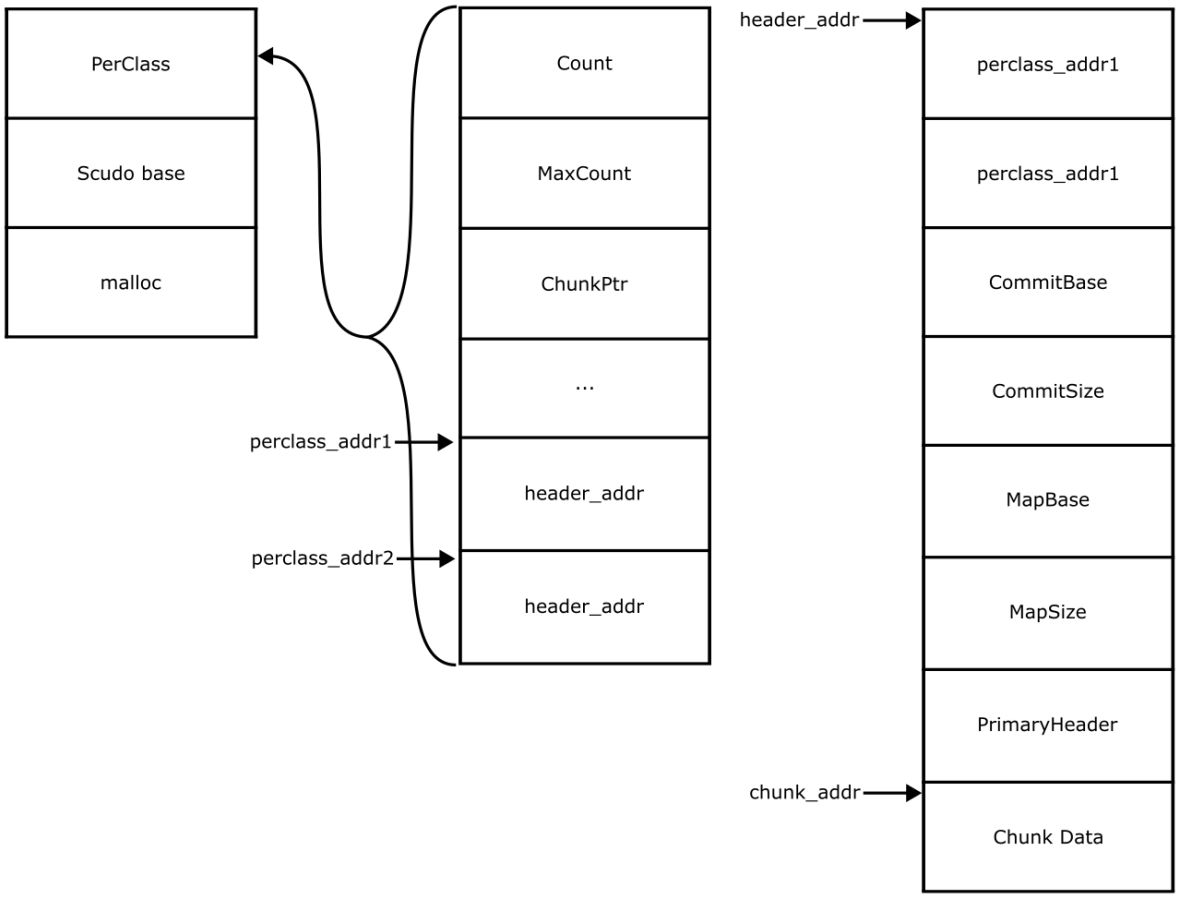

Figure 5.4: Illustration of different important memory regions

the base address where Scudo is loaded or the address of the malloc function. We can calculate the offset needed from there based on the Scudo version, and then advance by *0x100* times the value of the class ID where we free'd the chunks. Then we just need to guess the number of chunks present in the PerClass structure of that class ID. We will refer to the addresses that we calculate in this step as the *perclass_addr1* and *perclass_addr2* from here on forward for simplicity. These two addresses both point to a location in the PerClass array, at which our two addresses of our chunks are located.

Finally, we need to prepare the header of the final free that will trigger the safe unlink. For that we need to configure a fake secondary header in front of the primary header, as well as modifying the latter to set its class ID to 0. We set the prev (*chunk_addr - 0x40*) to *perclass_addr2 - 0x8* and the next (*chunk_addr - 0x38*) to *perclass_addr1*. With the PerClass setup, we can set both of them to the address where the first of the two addresses is stored. Next follow the CommitBase, CommitSize and MapBase, MapSize. We can set them to the values we want, we can set them to whatever since we don't actually use them. They have a size of *0x8* bytes each. Finally, we just need to set the primary header with the class ID set to 0 and the checksum recalculated.

When the free of that chunk then happens, it is handed to the secondary allocator since we set the class ID to 0. The secondary allocator tries to remove the chunk from the linked list of in use chunks, even when our fake chunk never was in it. So it tries to set the Next→Prev pointer to Prev and the Prev→Next pointer to Next, which leads it to write the addresses of our locations in the PerClass structure (*perclass_addr1* and *perclass_addr2*) to those same addresses, which will allow us to allocate a chunk in the PerClass structure itself.

Chapter 6

Mitigation

Both our exploits depend on the same mechanism of how secondary chunks are handled, by setting the ClassId of a chunk to 0 before freeing it and therefore tricking Scudo into handling a small chunk as if it was a big chunk. In the process of freeing the fake big chunk, Scudo then parses metadata before the chunk header that is controlled by the attacker. Therefore, in our mitigation we implement an additional system to verify before freeing a large chunk that it was actually allocated as a large chunk.

To be able to do this verification, we allocate a new special memory region, which can by default hold 1001 addresses. Whenever a new big chunk is allocated, its address is stored in the first free space in this special memory region, where a free space is defined by the address being 0. If the 1000 first addresses in this special memory region are all already used, a new region of 1001 addresses is allocated, and the address of this new region is stored as the 1001st address in the first region, creating space for 1000 additional addresses.

Whenever a big chunk is requested to be free'd, this special memory region is first searched for its address. If it is found, the space in the special memory region is set to 0 to indicate it is free again, and the rest of the logic to free the big chunk continues normally. If however the address is not found in the special memory region, the free is ignored as it is assumed to be either a corrupt chunk or an attacker trying to trick Scudo into freeing a fake forged chunk.

Two options were added in the Allocator Config to configure the details of the mitigation and to completely disable the mitigation if it is deemed too expensive in some configurations.

6.1 Evaluation

6.1.1 Effectiveness

After implementing the mitigation, we first checked to see if our exploits were indeed fixed by our mitigation. Since our fake chunks we are trying to free to trick the allocator mechanisms were in both cases originally small chunks, the mechanisms are never even triggered since our mitigation detects that the fake chunks are not supposed to be large chunks. Therefore, Scudo does abort the processing of the chunk before it gets to parsing the attacker controlled metadata.

6.1.2 Performance

Table 6.1: Custom benchmark results

benchmark name	runtime (in s)	with mitigation (in s)
single-block-used	5.8379	5.8170
many-blocks-used	7.0976	9.8458
random-blocks-order	3.8206	4.3806
random-half-blocks-order	3.7602	4.5997
many-many-blocks-used	6.4697	9.4870

In order to benchmark the changes of the mitigation, we used some custom benchmarks to test for impact with very targeted programs, results of which are show in Table 6.1, as well as a benchmarking suite for malloc implementations, mimalloc-bench, which is usually used to compare many different allocators between each other. [3] We slightly modified mimalloc-bench to include our patched Scudo version, and ran the benchmarks between the latest Scudo version and our patched version.

While in the worst case (many secondary allocations with many secondary chunks being allocated at the same time) the performance impact is quite high, in usual usage it should not matter much. This seems to also be confirmed by the benchmark tests of mimalloc-bench we ran, some of which are shown in Table 6.2. For the full results check Table A.1 in the Appendix.

Table 6.2: Excerpt of mimalloc-bench results

test	alloc	time	rss	user	sys	page-faults	-reclaims
cfrac	sys	8.49	3171	8.49	0.0	0	434
cfrac	scudo_fixed	10.03	4645	10.03	0.0	0	612
cfrac	scudo	9.96	4672	9.96	0.0	0	611
espresso	sys	6.36	2540	6.35	0.01	1	473
espresso	scudo_fixed	7.05	4852	7.03	0.01	0	659
espresso	scudo	7.02	4860	7.0	0.02	0	654
barnes	sys	3.57	58405	3.54	0.03	0	16570
barnes	scudo_fixed	3.42	59704	3.39	0.03	0	16645
barnes	scudo	3.77	59739	3.75	0.02	0	16645

Chapter 7

Conclusion

In this project, we used our previous work and acquired knowledge on the Scudo allocator and the gef plugin we built for it, to try to find some exploits that work on the Scudo allocator, similar to the glibc exploitation houses. The source code of the different exploitation scripts, the used exploitable programs, as well as some utilities to facilitate exploit development are available at `https://github.com/SIELA1915/house-of-scudo`.

We developed a python library, that takes advantage of the speed of native C code to take on the task of brute-forcing the cookie, given one header leak. Then we found and implemented two exploits, which are based on the common technique of forging headers using the brute-forced cookie. The Safe Unlink Secondary exploit was coincidentally fixed by a change on March 6th 2023, with commit $a6e3bb9$. Since the CommitBase exploit however was not fixed, we implemented a possible mitigation, that would fix both of our exploits. We then evaluated our mitigation, by checking it indeed fixes both of our exploits and by benchmarking the performance impact of our mitigation. Since the performance impact was negligible in most common scenarios, we submitted the mitigation to the official llvm repository in the form of a pull request on December 13th 2023.

Bibliography

[1] ashujaiswal109. *CVE-2019-11932 - Hack Android Devices by using Just a GIF Image*. https://www.exploit-db.com/docs/48632. [Online; accessed 8-June-2023]. 2019.

[2] Dr Silvio Cesare. *Breaking Secure Checksums in the Scudo Allocator*. https://blog.infosectcbr.com.au/2020/04/breaking-secure-checksums-in-scudo_8.html. [Online; accessed 05-January-2024]. 2020.

[3] daanx and Contributors. *Mimalloc-bench*. https://github.com/daanx/mimalloc-bench. [Online; accessed 13-December-2023]. 2019.

[4] swiat for Microsoft. *Software Defense: mitigating heap corruption vulnerabilities*. https://msrc.microsoft.com/blog/2013/10/software-defense-mitigating-heap-corruption-vulnerabilities. [Online; accessed 05-January-2024]. 2013.

[5] LLVM Project. *Scudo Hardened Allocator*. https://llvm.org/docs/ScudoHardenedAllocator.html. [Online; accessed 8-June-2023]. 2020.

[6] shellphish and Contributors. *tcache_poisoning.c*. https://github.com/shellphish/how2heap/blob/master/glibc_2.36/tcache_poisoning.c. [Online; accessed 03-January-2024]. 2023.

[7] un1fuzz. *Scudo's Internals*. https://un1fuzz.github.io/articles/scudo_internals.html. [Online; accessed 8-June-2023]. 2022.

Appendix A

Full mimalloc-bench results

Table A.1: Full mimalloc-bench results

test	alloc	time	rss	user	sys	page-faults	-reclaims
cfrac	sys	8.49	3171	8.49	0.0	0	434
cfrac	scudo_fixed	10.03	4645	10.03	0.0	0	612
cfrac	scudo	9.96	4672	9.96	0.0	0	611
espresso	sys	6.36	2540	6.35	0.01	1	473
espresso	scudo_fixed	7.05	4852	7.03	0.01	0	659
espresso	scudo	7.02	4860	7.0	0.02	0	654
barnes	sys	3.57	58405	3.54	0.03	0	16570
barnes	scudo_fixed	3.42	59704	3.39	0.03	0	16645
barnes	scudo	3.77	59739	3.75	0.02	0	16645
redis	sys	9.26	7929	0.42	0.04	3	1240
redis	scudo_fixed	9.81	9587	0.44	0.05	0	1482
redis	scudo	8.85	9648	0.4	0.06	0	1480
larsonN-sized	sys	3.0	374957	257.39	4.62	0	106747
larsonN-sized	scudo_fixed	128.05	168812	94.15	152.25	15	82785
larsonN-sized	scudo	129.12	168765	94.66	150.81	20	82673
mstressN	sys	5.05	1526414	37.09	20.58	0	3288327
mstressN	scudo_fixed	13.81	803629	119.19	108.42	0	6181076

Table A.1: Full mimalloc-bench results

test	alloc	time	rss	user	sys	page-faults	-reclaims
mstressN	scudo	13.7	800176	115.88	105.06	0	6171572
rptestN	sys	4.64	162499	28.06	10.43	2	87482
rptestN	scudo_fixed	22.79	143832	60.92	40.67	3	528002
rptestN	scudo	21.94	145304	58.56	39.31	0	526599
lua	sys	8.45	61516	7.81	0.57	179	144312
lua	scudo_fixed	8.44	71195	7.76	0.68	2	179348
lua	scudo	8.52	71162	7.8	0.71	0	179306
alloc-test1	sys	5.44	13859	5.43	0.01	0	9342
alloc-test1	scudo_fixed	5.87	15688	5.85	0.01	0	11476
alloc-test1	scudo	5.73	15740	5.72	0.01	0	10898
alloc-testN	sys	6.12	17233	87.82	0.04	7	11993
alloc-testN	scudo_fixed	10.77	17896	168.24	0.06	7	16705
alloc-testN	scudo	12.23	17858	192.0	0.06	6	19608
sh6benchN	sys	0.76	335016	14.53	1.21	0	83493
sh6benchN	scudo_fixed	39.32	371601	448.45	175.71	17	137682
sh6benchN	scudo	38.37	371660	450.64	179.58	15	137678
sh8benchN	sys	1.47	424444	47.6	0.45	0	110235
sh8benchN	scudo_fixed	94.2	277936	749.79	4333.51	17	220871
sh8benchN	scudo	95.0	278531	768.26	4356.15	25	223608
xmalloc-testN	sys	1.02	126016	256.71	1.76	11	109729
xmalloc-testN	scudo_fixed	85.24	88415	23.92	191.14	0	45575
xmalloc-testN	scudo	82.89	88232	23.99	196.19	0	50801
cache-scratch1	sys	1.82	3692	1.82	0.0	0	239
cache-scratch1	scudo_fixed	1.94	3909	1.94	0.0	0	276
cache-scratch1	scudo	1.88	3871	1.87	0.0	0	273
cache-scratchN	sys	0.09	4001	4.49	0.0	0	392
cache-scratchN	scudo_fixed	0.1	4606	4.64	0.0	0	527

Table A.1: Full mimalloc-bench results

test	alloc	time	rss	user	sys	page-faults	-reclaims
cache-scratchN	scudo	0.1	4685	4.74	0.0	0	530
glibc-simple	sys	5.43	1977	5.43	0.0	0	216
glibc-simple	scudo_fixed	7.14	3623	7.13	0.0	0	365
glibc-simple	scudo	7.5	3696	7.5	0.0	0	380
glibc-thread	sys	0.68	12809	109.56	0.03	70	4955
glibc-thread	scudo_fixed	2.66	15507	109.55	0.04	9	4254
glibc-thread	scudo	2.9	15431	109.55	0.05	15	4257