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 5, 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 5, 2024 Elias Valentin Boschung

Abstract

Heap exploits are a very frequent issue in C programming, due to the quite high complexity of memory management for programmers. Attackers can use such heap exploits 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. However, Scudo is an allocator used on Android 11+ that is supposed to be specifically hardened to prevent or at least complicate such attacks. [3]

In this project we tried to look for exploits using these heap vulnerabilities and targeting the mechanics of Scudo specifically. The goal was to find ways to trick the Scudo allocator despite its defenses and get techniques similar to the exploitation houses for the more common libc allocator.

We indeed discovered two exploits that by exploiting different heap vulnerabilities and mechanics of the Scudo allocator trick it into returning a chunk at an address chosen by the attacker, thus essentially achieving an arbitrary write primitive.

Additionally to the two exploits, we propose a mitigation that targets a mechanic of the Scudo allocator that is exploited in both of our exploits. We also 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, as they allow attackers to exploit the underlying functions of the heap allocator itself, that works the same way independent of the application's data. This can also be seen by the fact that there exist generic techniques that, given some preconditions, give a pattern to escalate a heap vulnerability and to get for example remote code execution. There exist a lot of these techniques for the widely used glibc allocator, like the libc:free_hook, unsafe unlink 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]

However, Android devices don't use the popular glibc allocator, but since Android 11 use a new heap allocator called Scudo. 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. For example Scudo uses a checksum to secure some critical metadata, 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. Also since it is designed completely differently, the known techniques for the glibc allocator don't work in the same way for Scudo. [3]

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. Additionally 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 in normal use, the overhead should be low enough to be acceptable. We therefore open a pull request to llvm, the official development team behind the Scudo allocator.

Chapter 2

Threat model

In order to search for exploits in Scudo, we first define a threat model, that defines what capabilities our imaginary attacker may have. Thanks to this threat model we will know how far we need to search and in what direction our exploits should be oriented.

In our threat model, our attacker may have access to some basic heap vulnerabilities, such as overflows, double free's and arbitrary free's. These 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 read or write primitive, allowing them to control the whole heap memory. We don't consider any vulnerabilities in application specific data, so the attacker from our threat model should only exploit some allocator mechanisms and not rely on any application data.

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 [4].

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.

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, which is designed to optimize performance as much as possible by adapting to finer-grained size differences between these smaller chunks. All the big chunks are handled by the secondary allocator, which is less optimized, since the most frequent chunks are the small ones and thus the primary allocator has a bigger impact on performance than the secondary allocator.

3.2 Combined Chunk Header

ClassId	8 bits
State	2 bits
OriginOrWasZeroed	2 bits
SizeOrUnusedBytes	20 bits
Offset	16 bits
Checksum	16 bits

Figure 3.1: Layout of the Scudo chunk header

The chunks allocated by both the primary and the secondary allocator store some metadata in a combined header. This header is then stored just in front of the actual content of the chunk, such that it can easily be checked whenever the chunk gets accessed. An overview of the contents of this header can be seen in Figure 3.1. The ClassId identifies the class the chunk belongs to in case it

was allocated by the primary allocator, with a ClassId of 0 meaning that the chunk was allocated by the secondary allocator. The header also contains information about the state of the chunk, which can be allocated, quarantined or available, as well as the origin of the allocation, e.g., malloc or new, which can be used to detect errors when the type of deallocation does not match the type of the allocation. Furthermore, the header includes the size for the primary chunks or the number of unused bytes for the secondary chunks and an offset, which is the distance from the beginning of the returned chunk to the beginning of the actual backend allocation. Finally, the header contains a checksum, which is generated using a cookie (a random number generated during the initialization of the allocator), the heap address of the chunk, and the actual content of the header. This checksum is used for guarding against programming errors as well as attackers, and it is checked each time the header is loaded.

3.3 Secondary Chunk Header

Prev	64 bits
Next	64 bits
CommitBase	64 bits
CommitSize	64 bits
MemMap	128 bits

Figure 3.2: Secondary chunk header on linux

The secondary header stores some additional information in a header which is prepended to the normal chunk header. First it stores pointers to the next and previous large chunk headers, which forms the doubly linked list of allocated secondary blocks. These pointers are used in the Safe Unlink Secondary exploit which is explained in a later section. Then it stores the CommitBase and CommitSize, which are the address of the chunk itself and its size. Lastly it stores some mapping information which can depend on the underlying operating system, but in the case of a Linux OS it simply corresponds to the MapBase and CommitBase. These are similar to CommitBase and CommitSize, but they may be slightly different due to alignment requirements. All these values are, contrary to the combined chunk header, not protected by a checksum, so they could be changed without trouble if we have write access to them.

3.4 Secondary Cache

The secondary allocator has its own cache, in which it simply stores the free'd secondary chunks in a simple list, and before allocating a new chunk it searches the list for a chunk whose size is at most a certain number of pages larger than the requested size. The values stored in this cache list, are

based on the values of the secondary chunk header at the moment the chunk is free'd. Interestingly, no integrity checks are done on the values saved in the cache, which is used in the CommitBase exploit which is explained in a later section.

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. Independent of which of the two algorithms is used, it is first applied to the cookie and the address, and then again applied to the previous result and the header with the checksum bytes in the header set to 0. If CRC32 is used, the result is then xor'd with itself shifted right by 16 bits, and truncated to the 16 least significant bits, whereas if the BSD checksum is used the cookie is already truncated to the 16 least significant bits and all calculations are done over 16 bits. The resulting checksum is therefore in either case 16 bits long. 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. Also 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. To accomplish that we can simply brute-force the cookie value until we find the same checksum that we leaked.

Using a C program this takes around 1–2 seconds, however due to the poor performance of loops in python, the brute-force does not natively work in python. Therefore, we created a small python library written in native C code, which can be used to calculate checksums and brute-force a cookie given the header leak data from within python. The python library exposes methods to brute-force the cookie and to calculate a new checksum given a cookie for both the CRC32 and the BSD algorithms.

Chapter 5

Exploits

5.1 Change Class ID

5.1.1 Summary

- 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.

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, but the most likely might be 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. Though the exact offset from the first chunk which contains the overflow and the target header is random due to Scudo's defense mechanisms, the attacker can still manage to overwrite it with some spraying and random luck if he can execute the program as often as he wants.

The attacker then calculates 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 Safe Unlink Secondary

5.2.1 Summary

- 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

5.2.2 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 need two interesting places in memory (at addresses ‘add1‘ and ‘add2‘ respectively) that have the address of the free address from the previous step stored. We will store *add2* at *add1+0x8* and *add1* at *add2*. Therefore, ‘add1‘ and ‘add2‘ should be in some interesting location allowing elevation of our access. The easiest way to get this is by having 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. With this method we however also need a leak of the Scudo library base, in order to get the location of the thread-local free list (PerClass list).

5.2.3 Explanation

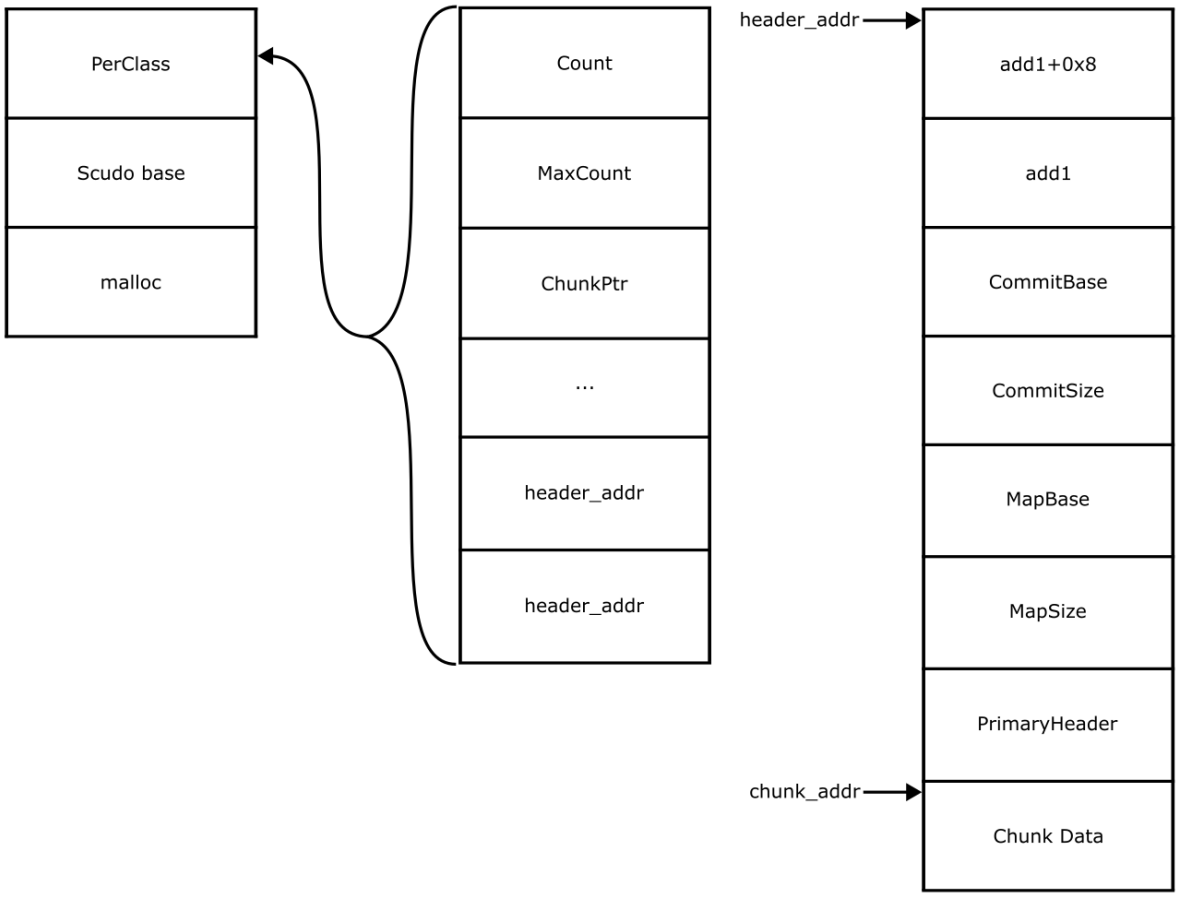

Figure 5.1: Illustration of different important memory regions

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.

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. This setup is especially interesting since getting the allocation of a chunk in the PerClass structure could allow us to control all following allocations and to allocate chunks at arbitrary addresses. The only obstacle is that we need to modify the chunk header before the second free to mark the chunk as allocated again.

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 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.

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 the first of our locations *-0x8* and the next (*chunk_addr - 0x38*) to the second of our locations. 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 to those same addresses, which will allow us to allocate a chunk in the PerClass structure itself.

5.3 Exploit CommitBase

5.3.1 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

5.3.2 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.

5.3.3 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 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.

Chapter 6

Mitigation

Both our exploits depend on the same mechanism of how big 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. 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 where 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. Indeed, our exploit scripts still ran without error, however when displaying what was supposed to be our chosen address for our chunk allocation, with both our exploits the returned address didn't match our chosen address anymore. 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.

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. [2] 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, results of which are shown in Table 6.2.

Table 6.2: 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
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

Table 6.2: mimalloc-bench results

test	alloc	time	rss	user	sys	page-faults	-reclaims
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
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

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

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

[4] 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.