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

December 30, 2023

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, December 30, 2023 Elias Valentin Boschung

Abstract

As a followup to the Bachelor Project for creating tooling for the Scudo allocator, this project focuses on using the tool and the acquired knowledge of Scudo to try to find exploits of heap errors, similar to existing exploitation techniques for the standard libc allocator.

Using the fact that the checksum used to secure the header of a chunk is breakable with a single header leak, we introduce two exploits that use a free operation where we can control preceding bytes to get control over the next chunk allocation locations.

One of these two exploits has already been fixed in the latest Scudo version, while we propose a mitigation for the second one.

Contents

Chapter 1

Introduction

Bugs related to heap memory are one of the most frequent problems encountered by C developers since in bigger projects it can very quickly get hard to track all the resources that are open and which were already closed. It is very tedious to track all of such errors down, and just a single oversight might create a bug that opens a huge security vulnerability. Additionally, nowadays, many apps are so complex that they use various third-party libraries to do different tasks since re-implementing everything on your own would be a waste of time. However, one hardly has the time to check the whole source code of every library one uses, so one has to trust the library developers to have done their work correctly. An example of where such trust was the cause of a major vulnerability is a remote code execution in WhatsApp that was found in 2019, where a simple double free in a third-party GIF library allowed an attacker to get arbitrary remote code execution. [1]

Since checking all the code of all third-party libraries is practically not feasible, it is important to be able to debug heap bugs when they are found, in order to understand how they happen, the consequences of an exploit using the bug might have, and to be able to fix the bug rapidly. To encourage and assist developers in this process, there needs to be some freely available tools that make dynamic debugging of the heap easier.

Since Android devices are used daily by millions of users, with many thousands to millions of different apps, the attack surface for exploits is huge. That might well be one of the reasons that Android uses the new and mostly unknown heap allocator called Scudo. [4] Scudo tries to mitigate some of the potential vulnerabilities due to heap bugs directly in the allocator while keeping a high performance for memory allocations.

In a previous project, we analyzed the inner workings of the Scudo allocator and developed a gef plugin to debug programs using the Scudo allocator. [2] Using that plugin and the general knowledge acquired during that project, we aim to find some exploits that are possible with the Scudo allocator and figure out how to circumvent some of the specific mitigations already integrated in the Scudo

allocator. If we find some exploits, we implement a possible mitigation for it and propose it to the official Scudo developers, as a pull request on the llvm github repository.

Chapter 2

Scudo Security Measures

2.1 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.

2.2 Combined Chunk Header

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

Figure 2.1: 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 2.1. 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.

2.3 Secondary Chunk Header

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

Figure 2.2: 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.

2.4 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.

Talk about chunk header integrity checks, checksum, class id, cache

Chapter 3

Breaking the Cookie

To check the integrity of chunk headers, Scudo uses a CRC32 checksum whenever hardware CRC32 is available, otherwise it instead uses a software BSD checksum. Both of these checksum algorithms are similar in that they aren't cryptographically secure. The cookie used to calculate the checksum is 32 bits long and used as the starting value for the CRC32 (or BSD) checksum. However the actual checksum stored in a chunk header is only 16 bits long, so in order to shorten the checksum, the XOR of the lower 16 bits with the higher 16 bits is taken.

Due to the checksum only being 16 bits, instead of the full 32 bits from the cookie, it is 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 bruteforce 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 bruteforce 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 bruteforce a cookie given the header leak data from within python.

The python library exposes four different methods, *bruteforce* to bruteforce the cookie value from the address, checksum and header data taken from a header leak, *calc_checksum* to calculate a checksum from the address and header data as well as the cookie value, and the same methods followed by *_bsd* to do the same for the BSD checksums which are used when no hardware CRC32 is available.

Chapter 4

Exploits

4.1 Change Class ID

4.1.1 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.

4.1.2 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 bruteforce 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.

4.1.3 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 defence 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 bruteforced 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.

4.2 Safe Unlink Secondary

4.2.1 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 free's
- 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

4.2.2 Requirements

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

First of all we need a header leak of any chunk from the heap as well as it's address, in order to bruteforce the cookie. Afterwards 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‘ resepectively) 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 free's 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 lib base, in order to get the location of the thread-local free list (PerClass list).

4.2.3 Explanation

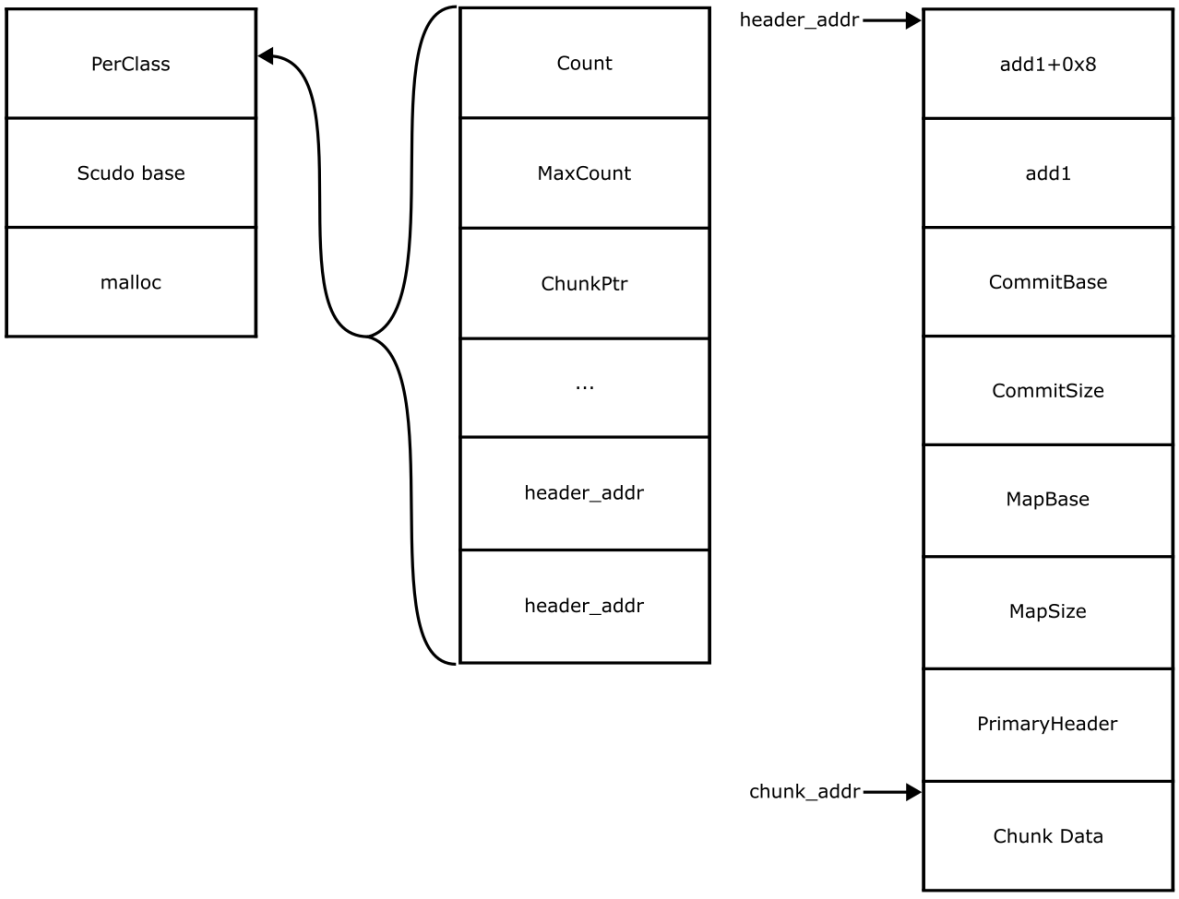

Figure 4.1: Illustration of different important memory regions

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 it's 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 setup the two addresses in the perclass array, we need to know it's 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 it's 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.

4.3 Exploit CommitBase

4.3.1 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

4.3.2 Requirements

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

First of all we need a header leak of any chunk from the heap as well as it's address, in order to bruteforce the cookie. Afterwards 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 afterwards, of which we know the approximate size. This is the allocation where we will get our arbitrary access.

4.3.3 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 5

Mitigation

The Safe Unlink Secondary exploit was already fixed in the latest scudo version, and we only detected it due to partly working with an older version of scudo. However it worked in certain configurations up to the version *XXX*.

The CommitBase exploit on the other hand was still applicable to the latest scudo version as of this project. So after discovering it and having proven it works, we also thought about a possible mitigation, which was proposed to the official scudo/LLVM team in form of a pull request on December 13th, 2023.

This proposed mitigation tracks in a separate memory region the addresses of all currently allocated secondary chunks and only adds free'd secondary chunks to the cache if they were recorded in this dedicated memory region.

The tracking in this mitigation happens through a simple list of addresses, which is looped over at allocation/deallocation. Each block of contiguous memory has a at compile time configurable number of addresses stored, as well as one additional address pointing to the next block in case there are more addresses.

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

5.1 Benchmark

Table 5.1: 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 5.1, 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 match. This seems to also be confirmed by the benchmark tests of mimalloc-bench we ran, results of which are shown in Table 5.2.

Table 5.2: 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

Table 5.2: mimalloc-bench results

test	alloc	time	rss	user	sys	page-faults	-reclaims
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
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

Table 5.2: mimalloc-bench results

test	alloc	time	rss	user	sys	page-faults	-reclaims
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 6

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 found two exploits, which are based on a common technique, which relies on breaking the cookie used to generate checksums, for which we implemented a python library taking advantage of the speed of C code to bruteforce the cookie given one header leak. One of the two exploits was coincidentally fixed by a change in the latest scudo versions, while for the other exploit no fix was available. We therefore implemented a possible mitigation, which we benchmarked to test for the performance impact and then submitted to the official llvm repository in the form of a pull request.

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] Elias Valentin Boschung. *Tooling and Analysis of the Scudo Allocator*. https://github.com/SIELA1915/scudo-gdb-tooling/blob/main/report/thesis.pdf. [Online; accessed 29-December-2023]. 2023.

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

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