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 20, 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 20, 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

Chapter 2

Scudo Security Measures

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

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.

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

In order to benchmark the changes of the mitigation, we used some custom benchmarks to test for impact with very targeted programs, as well as a benchmarking suite for malloc implementations, mimalloc-bench, which is usually used to compare many different allocators between each other. [1]

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.

Chapter 6

Conclusion

Bibliography

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