École Polytechnique Fédérale de Lausanne

Tooling and Analysis of the Scudo Allocator

by Elias Valentin Boschung

Bachelor Thesis

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

June 6, 2023

Follow the white rabbit...
— The Matrix

Dedicated to my pet bunny.

The dedication is usually a short inspirational quote.

Define your dedication in `\dedication{...}` and show them with `\makededication`.

Acknowledgments

This is where you thank those who supported you on this journey. Good examples are your significant other, family, advisers, and other parties that inspired you during this project. Generally this section is about 1/2 page to a page.

Consider acknowledging the use and location of this thesis package.

Define your acknowledgments in `\acknowledgments{...}` and show them with `\makeacks`.

Lausanne, June 6, 2023 Elias Valentin Boschung

Abstract

The Scudo GEF Plugin tool enables inspection of the heap memory of an android app which uses native C libraries.

While there is a lot of existent tooling to debug errors related to heap memory in C code, it is only for the standard libc allocator. However, Android uses its own allocator since Android 11, the scudo hardened allocator. Since scudo uses its own structures and way to allocate memory, those tools can not be used for debugging android C libraries.

In this project the goal was to analyze the way scudo allocates memory and then write some tooling to debug it. The tool developed takes the form of an extras plugin into the popular GEF plugin for GDB, which in turn is a popular debugger for C programs.

The abstract serves as an executive summary of your project. Your abstract should cover at least the following topics, 1–2 sentences for each: what area you are in, the problem you focus on, why existing work is insufficient, what the high-level intuition of your work is, maybe a neat design or implementation decision, and key results of your evaluation.

Contents

Chapter 1

Introduction

The introduction is a longer write-up that gently eases the reader into your thesis [1]. Use the first paragraph to discuss the setting. In the second paragraph you can introduce the main challenge that you see. The third paragraph lists why related work is insufficient. The fourth and fifth paragraphs discuss your approach and why it is needed. The sixth paragraph will introduce your thesis statement. Think how you can distill the essence of your thesis into a single sentence. The seventh paragraph will highlight some of your results The eighth paragraph discusses your core contribution.

This section is usually 3–5 pages.

Chapter 2

Background

Most standard android apps are written in Java, which is the main development language for writing android apps. However, for more advanced uses there exists support for including libraries written in standard C code, called native libraries referring to the underlying structure of android which is actually a modified version of Linux. These native libraries can be used together with Java code, allowing C functions to be called from the java part of the code.

In C, the memory is divided into three big types that are handled differently. There is the data segment, which contains the global and static variables that are defined in a program, and the data segment can be further divided into the initialized global and static variables, and the uninitialized or initialized to zero global and static variables. Then there is the stack, which is used for local variables in a function as well as for arguments and some additional info about called functions. The size of variables on the stack have to be fixed-size and the variables lifetime is limited to the scope of the function. The third type of memory is the one important for this thesis, the heap memory. The heap memory is the most flexible of the three types of memory, as it is not limited to a specific lifetime like the stack and variables on the heap can be resized. Heap memory can be allocated by the programmer by calling the $malloc$ function, and unlike the stack or data segment, it has to be explicitly freed by the programmer again, by calling $free$. To resize a variable on the heap, the $realloc$ function can be used. Since the variables have to be freed manually, the programmer has full control over the lifetime of a variable on the heap, and it can be used for variables that are needed outside the scope of a single function. Furthermore, the heap has a virtually infinite amount of space, and is generally also used for very big memory allocations. However, due to this big flexibility, the handling of heap memory is also quite error-prone, by forgetting to free allocated memory or allocating a chunk of the wrong size of the heap and trying to access memory outside the allocated part.

Due to the ability to resize the heap variables and their flexible lifetime, the heap cannot simply allocate contiguous chunks of memory in the order that variables were allocated, as there would have to be a lot of moving when variables were resized and there would be a lot of space lost when some chunks in the middle of the heap were freed. Instead, there is a lot of bookkeeping done around which chunks of memory are allocated, which are free and the heap allocator tries to decide in a smart way which chunks to allocate when and where to get as much performance as possible.

While the general concepts of these memory types are universal for the C language, there are some differences in the concrete implementation, especially of the heap allocator. While most Linux programs use the more standard libc malloc, which is well documented and for which tooling exists to investigate the state of the heap, android uses its own allocator since Android 11, which is called the scudo hardened allocator.

Explain gdb, gef and gef extra plugins and how they come together/interact.

The background section introduces the necessary background to understand your work. This is not necessarily related work but technologies and dependencies that must be resolved to understand your design and implementation.

This section is usually 3–5 pages.

Chapter 3

Scudo Internals

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.

The chunks allocated by both the primary and the secondary allocator are prepended by the same header, which holds some information about that chunk. This combined header contains the ClassId, which identifies the class the chunk belongs to if 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 amount 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.

The primary allocator structures the heap into Classes (also named Regions), the amount of which depend on the configuration. These classes are identified by a ClassId, starting from 0. The first class, with ClassId 0, is however a special class, that works differently than the other classes. The other classes have a given size of the chunks that can be allocated in that class, which increases with the ClassId, with the exact numbers depending again on the configuration. So the chunk that is actually returned by the primary allocator might be bigger than what was asked for, but this helps against having to do expensive bookkeeping to avoid fragmentation inside the classes. So when an allocation request is made to scudo, and it is small enough to fit into one of the primary allocator regions, then the primary allocator first finds the smallest class that still contains chunks big enough for the requested size. It then checks in the cache associated to that class, to find a chunk that is already available. However, if the cache is empty, meaning that there are no readily available free chunks, the primary allocator tries to refill the cache with some new chunks. For that purpose, it looks up the freelist of TransferBatches for the given class, and if it is not empty it just takes all the chunks of the first TransferBatch in the freelist and moves them to the cache of the class. One TransferBatch holds a certain number of chunks defined by the configuration, which is the same for all classes. In case the freelist is out of TransferBatches, the allocator will allocate some new TransferBatches to fill up the freelist. The amount of TransferBatches allocated at one such time depends on the configuration, but it is typically smaller the bigger the class gets, as one TransferBatch represents more actual memory the bigger the class is. The allocator will also map more space for a specific class if the amount of TransferBatches to be created does not fit into the existing mapped space.

The secondary allocator is a bit less sophisticated, since it does not need to be able to do as many allocations and deallocations as the primary allocator, and the optimizations of the primary allocator would not be that efficient for the bigger and therefore generally more variable sizes the secondary allocator has to handle. Instead, the secondary allocator just keeps a simple cache of the previously freed chunks, and if there is no matching chunk in its cache upon a request for a new allocation, it maps some new memory for that chunk. For bookkeeping, the secondary allocator simply keeps a doubly linked list of all chunks currently in use and keeps track of the details of each allocated chunk in a special header that is prepended to the combined header.

Explain how scudo allocates memory in the primary and secondary allocator.

Introduce and discuss the design decisions that you made during this project. Highlight why individual decisions are important and/or necessary. Discuss how the design fits together.

This section is usually 5–10 pages.

Chapter 4

Implementation

Explain how the plugin is specifically structured, how the commands, the data structures and the constants are defined.

The implementation covers some of the implementation details of your project. This is not intended to be a low level description of every line of code that you wrote but covers the implementation aspects of the projects.

This section is usually 3–5 pages.

Chapter 5

Evaluation

Give some examples of the crashes analyzed, how the commands of the plugin can be used to figure out what is happening.

In the evaluation you convince the reader that your design works as intended. Describe the evaluation setup, the designed experiments, and how the experiments showcase the individual points you want to prove.

This section is usually 5–10 pages.

Chapter 6

Related Work

Mention some of the links as provided for the beginning of the project.

The related work section covers closely related work. Here you can highlight the related work, how it solved the problem, and why it solved a different problem. Do not play down the importance of related work, all of these systems have been published and evaluated! Say what is different and how you overcome some of the weaknesses of related work by discussing the trade-offs. Stay positive!

This section is usually 3–5 pages.

Chapter 7

Conclusion

In the conclusion you repeat the main result and finalize the discussion of your project. Mention the core results and why as well as how your system advances the status quo.

Bibliography

[1] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization”. In: *IEEE International Symposium on Security and Privacy*. 2020.