École Polytechnique Fédérale de Lausanne

Tooling and Analysis of the Scudo Allocator

by Elias Valentin Boschung

Bachelor Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Expert Reviewer
External Expert

Mao Philipp Yuxiang
Thesis Supervisor

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

June 8, 2023

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

Acknowledgments

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

Lausanne, June 8, 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.

Contents

Chapter 1

Introduction

In some previous projects at the HexHive lab, some fuzzing on android native libraries has been done, to try to find crashes that would be very hard to find without using fuzzing techniques to find specific inputs that reproducibly cause a crash. While the fuzzing was successful and some crashes could be found in different apps, the time was not sufficient to investigate each of these crashes more in depth.

While it was doable to triage the crashes in some categories, like heap related crashes, the tools to further investigate such heap related crashes on an android device were insufficient. One of the main challenges will therefore be to understand the heap allocator used on android devices, scudo, and then to create appropriate tooling to help with the analysing of those crashes. While there is a bit of documentation around scudo, there is not really a comprehensive overview of all detailed inner workings of scudo, and even less tools tailored for debugging crashes related to scudo.

The goal of this project is therefore to implement a tool that allows for improved convenience when trying to debug scudo-related crashes. Since the previous work already set up the results of the fuzzing for debugging with gdb and gef, the tool takes the form of a gef-extras plugin, that can be easily loaded into gdb with gef and adds some commands for inspecting the state of scudo.

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 variable's 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.

The *GNU Project Debugger* (GDB), is one of the most popular and powerful debuggers currently out there, especialy when it comes to C/C++ code. While already a very powerful tool on its own, it can be further enhanced with plugins using the python API.One of these plugins is *GDB Enhanced Features* (GEF), which adds more commands and features to GDB, especially popular with exploit developers, reverse engineers and also *Capture the Flag* (CTF) players. GEF itself has its own plugin system as well, using which it distributes some more optional features under the name of gef-extras. Using the system of these gef plugins, adding new commands to the gdb console is fairly simple and developers can easily mix and match the extra plugins they need.

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. Since the transfer batches that are currently in one of the freelists need to be stored somewhere as well, the first class with ClassId 0 is reserved for this use. The allocations requested by the user will never directly use this class, instead when a freelist for one of the classes needs to be refilled with new TransferBatches, these are allocated in this first class, and they are freed when they are moved to the cache of the class.

Additionally, while not yet present in the version of scudo shipped with Android 11, the latest scudo version includes a type that regroups multiple transfer batches, into a BatchGroup. Instead of the freelist for a given class containing a list of transfer batches, it now has a list of batch groups, which contain a list of transfer batches each. The overall functionality of the allocator doesn't change however, the change is just supposed to get even more performance for the frequent allocations of the primary allocator.

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.

Chapter 4

Implementation

The structure of the gef-extras plugin is heavily inspired by the existing structure of the tooling for the standard libc heap allocator used in most Linux systems, which is included in the gef plugin for gdb. The scudo plugin therefore offers multiple commands for the different structures present in scudo. These commands take generally some address to a memory region, and create an object by reading out the data from that memory address and parsing it according to the predefined structure.

The most basic command is the `scudo chunk` command, which given the address of a chunk on the heap, reads the header in front of the chunk and displays the info contained in it. The design of the output is the same as for the standard libc heap commands included in gef, with the allocation state being highlighted in color, as can be seen in Figure 4.1. In the actual implementation, a chunk is represented as a class, and like in gef, the different information is stored as properties of the class. Similarly, the data is read from memory using the `gef.memory.read` function, and then structured as a type using the ctypes library to represent the different types used in scudo. The `str` method on the class is implemented to display a short summary of the most essential info, while the `psprint` method shows the extended information display, as shown in Figure 4.1.

The next two commands concern the region info, with respectively the `scudo region` and `scudo regions` commands. Both of work with the same `ScudoRegionInfo` class, which is defined like the scudo chunk with a ctypes structure to represent the types in the scudo c code, and with properties for all the information extracted. The one special thing about the region info class is the padding, since the structure is padded to the scudo cache line size for faster access. So the plugin first builds an unpadded version of the ctypes structure, and then adds the required padding to get a multiple of the scudo cache line size to the structure to get the final version.

The info included in the RegionInfo structure includes the list of transfer batches for that region, the number of popped and pushed blocks, the amount of memory mapped and allocated inside the region, the random state generated for the region as well as information about the last release of

```
gef➤  scudo chunk 0x7fd5f75d7b70
Chunk(addr=0x7fd5f75d7b70, size=0x32, state=Quarantined, classid=3)
Origin: Malloc
Chunk size: 50 (0x32)
Offset: 0 (0x0)
Checksum: 0xb3c8
Quarantined

gef➤  scudo chunk 0x7fe2f75d4410
Chunk(addr=0x7fe2f75d4410, size=0x3e8, state=Allocated, classid=0)
Origin: Malloc
Chunk size: 1000 (0x3e8)
Offset: 0 (0x0)
Checksum: 0xfe18
Allocated
```

Figure 4.1: Example of the scudo chunk command in gef

blocks to the operating system.

The data of this `RegionInfo` structure is stored in an array, with an entry for each of the regions/classes. The array is part of the primary allocator class, and so the plugin finds it by using the offset from the `Allocator` symbol to get the first entry in the array. The `scudo regions` command then simply displays a list of short information about each region, as seen in Figure 4.2.

```
gef➤  scudo regions
Region(base=0x7ffff7b750c0, region_begin=0x7fd2f75ce000, mapped=0x40000, allocated=0x2000, num_batches=1)
Region(base=0x7ffff7b75180, region_begin=0x7fd3f75d7000, mapped=0x0, allocated=0x0, num_batches=0)
Region(base=0x7ffff7b75240, region_begin=0x7fd4f75c9000, mapped=0x0, allocated=0x0, num_batches=0)
Region(base=0x7ffff7b75300, region_begin=0x7fd5f75cd000, mapped=0x40000, allocated=0x1e00, num_batches=1)
Region(base=0x7ffff7b753c0, region_begin=0x7fd6f75cc000, mapped=0x0, allocated=0x0, num_batches=0)
Region(base=0x7ffff7b75480, region_begin=0x7fd7f75c8000, mapped=0x0, allocated=0x0, num_batches=0)
Region(base=0x7ffff7b75540, region_begin=0x7fd8f75c8000, mapped=0x0, allocated=0x0, num_batches=0)
Region(base=0x7ffff7b75600, region_begin=0x7fd9f75d2000, mapped=0x0, allocated=0x0, num_batches=0)
```

Figure 4.2: Example of the scudo regions command in gef

The `scudo region` command on the other hand displays more detailed information about a specific region, which can be specified either by giving the address to the RegionInfo structure, by giving the max allocation size of the region, or by the index in the array, which corresponds to its ClassId. An example of the display can be seen in Figure 4.3, which shows the special region which holds the transfer batches.

The next two commands concern the free list of the primary allocator, as they display the information about the batch groups and the transfer batches. Both of these commands are very simple, since both of them basically contain a list of another structure. Also since there isn't one universal point where all of them are stored, the commands need to be provided with the address

```
gef➤  scudo region --index 3
Region(base=0x7ffff7b75300, region_begin=0x7fd5f75cd000, mapped=0x40000, allocated=0x1e00, num_batches=1)
Free list:
        Number free: 1
        First free: 0x7fd2f75cf980
        Last free: 0x7fd2f75cf980

Region stats:
        Popped blocks: 10
        Pushed blocks: 0

Random state: 2791513300

Release to OS:
        Pushed blocks at last release: 0x0
        Ranges released: 0x0
        Last released bytes: 0x0
        Last release at ns: 37685423746953
```

Figure 4.3: Example of the scudo region command in gef

of the batch group or transfer batch to output. Since the batch groups are a new addition, the command doesn't work on a real Android 11 phone, however it works when using the standalone compiled version from the llvm repository. It shows detailed information about the batch group, including the address of the next batch group, since the batch groups form a linked list, the number of pushed blocks, and the information about the doubly linked list of transfer batches it contains. While this detailed view can be seen in Figure 4.4, when specifying a number as argument with `--number X`, it will instead display a short version for this amount of batch groups in the linked list, with the first one being the one specified by address.

```
gef➤  scudo batchgroup 0x7fd2f75cf980
BatchGroup(base=0x7fd2f75cf980, num_batches=7, first_batch=0x7fd2f75cf600, pushed_blocks=80)
Next batch group: 0x0
Compact base addr: 0x3feafba
Max cached per batch: 10
Pushed blocks: 80
Pushed blocks at last checkpoint: 0
Batches:
        Number batches: 7
        First batch: 0x7fd2f75cf600
        Last last: 0x7fd2f75cf900
```

Figure 4.4: Example of the scudo batch group command in gef

The `scudo transferbatch` command simply outputs the addresses of every batch it contains, as well the address of the next transfer batch, which can be seen in Figure 4.5. Since the transfer batches also form a linked list similarly to the batch groups, the number argument can be specified in the same way to get an overview of that number of transfer batches.

The next command concerns the cache, which is actually the first place the primary allocator tries to look for free chunks to allocate for the user. While again the content of each per class structure is fairly simple with a list of the chunks it contains and the maximal number of chunks it

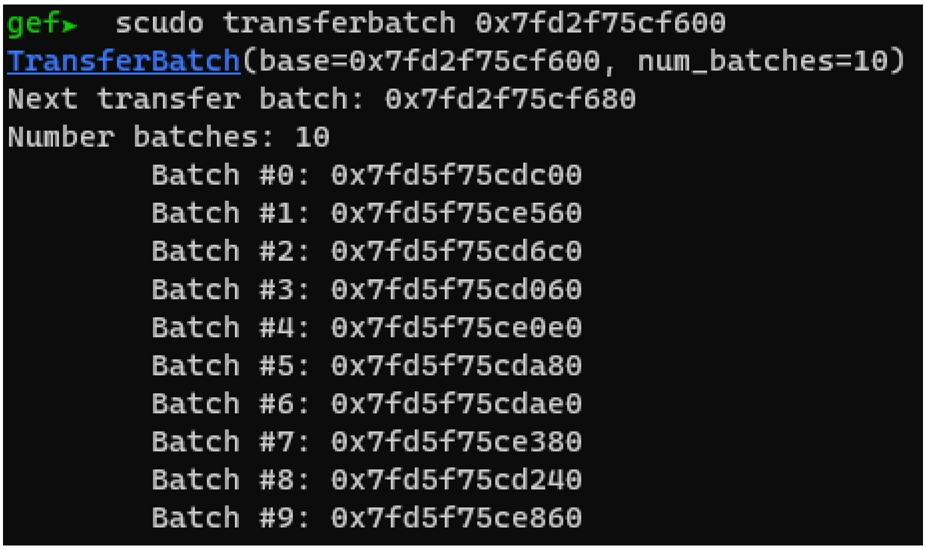

Figure 4.5: Example of the scudo transfer batch command in gef

can contain, there was a particular challenge about implementing this command, which was to find the location of the actual array. Since scudo was designed for multi threading, there actually exists a cache for each thread, and as such the per class array of each thread is not simply stored at an offset from the Allocator symbol. However, there luckily is a trick to find the different per class arrays of the different threads, as the cache of each thread contains some statistics about the allocated and free blocks in that cache. In order to have some global statistics over all the different threads, scudo stores some global stats, that include a linked list of all the local stats of the different threads. So in order to get the address of the per class array of a thread, the plugin gets the linked list of statistics from its offset to the Allocator symbol and then follows the linked list the number of times to reach the statistics for the correct thread. Since the statistics are stored right after the per class array in the cache, the plugin then simply subtracts the size of the per class array to get the address of the first entry in the class array. For that reason the `scudo perclass` command also takes two arguments, the first being the thread id and the second being the class id, which uniquely define the entry to get. If the ids are omitted a default value of 0 is taken. An example of the output can be seen in Figure 4.6.

The last command is the only one concerning the secondary allocator specifically, and it reads out the special header that is prepended to the normal chunk header for any secondary chunk. The normal scudo chunk command still works for secondary chunks as well though. The large block command gives information about the commit and map base addresses and sizes, as well as the next and previous large blocks in the linked list. The command can either be provided by an address to read out the corresponding large block, or if the address is omitted it will take the first block from the list of blocks in use. An example can be seen in Figure 4.7. A `--number X` argument can optionally be specified to get once again a list of short infos about that amount of large blocks.

While the whole plugin is contained in a single file to follow the gef standard, the organization

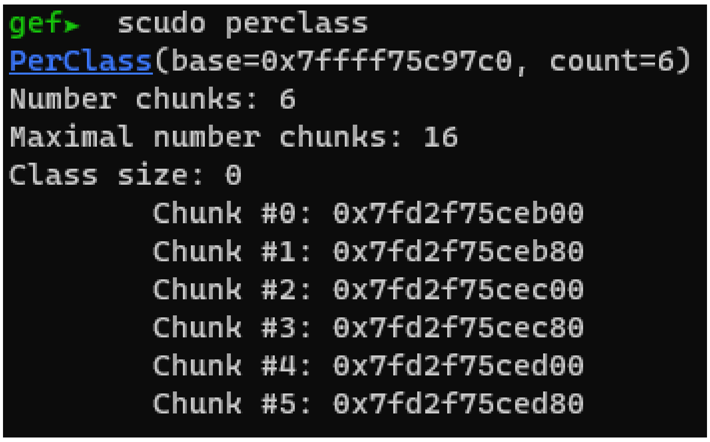

Figure 4.6: Example of the scudo per class command in gef

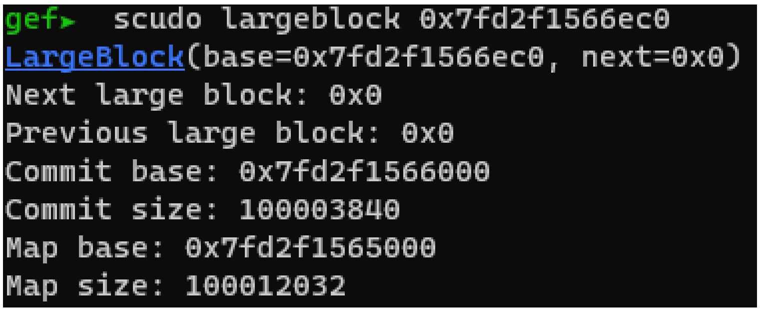

Figure 4.7: Example of the scudo large block command in gef

with classes and using the ctypes structures allows for a flexibility in adapting and evolving the plugin. As such it would be quite easy to adapt the plugin for any minor changes in the scudo structure should it evolve in the future, or adapted to support older versions of scudo, with one such version already being implemented for the version of scudo that was shipped with Android 11. The main changes for this older version of scudo included that the batch groups did not exist yet, some minor reordering of some fields in some of the structures, which also caused some of the offsets from the Allocator symbol to be changed.

Chapter 5

Evaluation

During the main development process the plugin was tested with the standalone version of scudo compiled with debug symbols from the llvm repository. In order to get the plugin loaded by gef, the folder containing the plugin file needs to be specified as gef-extras plugin dir. The easiest way to do this is to run `gef config gef.extra_plugins_dir /path/to/plugin/dir`. In order to use multiple different gef extra plugins that may be located at different paths, multiple paths can be specified separated by semicolons. Also, the configuration can be saved using `gef save`, which will create a gef.rc file that will be loaded in order to keep the settings between restarts of gdb.

Since the main development was done on the standalone version, the first step to evaluate the plugin was to actually test it on a real android phone. The test device was a Samsung Galaxy S10, which was already rooted using Magisk and had gdb and gef installed using the Termux app. Access to the phones console and copying of the plugin to the phone was done using adb. In order to do the evaluation directly using WSL from a Windows workstation, the usbipd driver was used to pass the phone connected using USB to WSL. Without going into detail, the commands for attaching the phone to WSL to be entered in an Administrator Command Prompt are the following, replacing X-X with the busid shown in the first command for the android phone:

```
usbipd wsl list
usbipd wsl attach --busid X-X
```

While the plugin installation and loading worked without problem on the phone, it rapidly became evident that the plugin did not work as intended and instead returned some random looking numbers for most commands. So after some investigation, it was discovered that the scudo version shipped with Android 11, which was installed on the testing device, differed in multiple points from the most recent version on the llvm repository. The plugin was therefore copied and adapted into a version for Android 11, which notably does not contain the batch groups, as well as differing in some minor changes in different structures.

In order to evaluate the actual functioning of the plugin, it was tested with some of the crashes provided by the supervisor. The starting of the corresponding apps and causing the crashes was already set up automatically, opening gdb before starting to run the program in order to allow for the setting of any breakpoints if needed.

5.1 Smartbill Backtrace 5

One of the crashes analyzed to test if the plugin could be used for real debugging work was in the app `com.skt.smartbill`, the backtrace numbered BT5 in the provided documents. In order to analyze the crash, the first thing done was to just let the whole crash run in order to get a backtrace of what had caused the crash, which looked like this (shortened for readability):

```
abort  from libc.so
scudo::die () from libc.so
scudo::ScopedErrorReport::~ScopedErrorReport () from libc.so
scudo::reportMisalignedPointer () from libc.so
scudo::Allocator<scudo::AndroidConfig, &>::deallocate () from libc.so
Clear_OBJECT_IDENTIFIER  from libUbikeyJni.so
Delete_OBJECT_IDENTIFIER  from libUbikeyJni.so
Convert_ASN1_to_X509_NAME  from libUbikeyJni.so
??  from libUbikeyJni.so
Convert_ASN1_to_X509_TBS_CERT  from libUbikeyJni.so
Convert_ASN1_to_X509_CERT  from libUbikeyJni.so
CERT_Load  from libUbikeyJni.so
Java_com_ubikey_jni_UbikeyJni_jGetUser  from libUbikeyJni.so
main  at harness_debug.cpp:247
```

So from the backtrace the goal was to move back in the stack trace trying to pinpoint the error. Therefore the next step was to set a breakpoint somewhere in `Clear_OBJECT_IDENTIFIER`, so to find out where in the function the crash happened, the frame can be moved up with `f 5` to go to the frame of `Clear_OBJECT_IDENTIFIER`. Then using `x/10i $pc-0x10`, the context around where the crash happened can be discovered, and gdb helpfully shows the line offset from the start of the function, in this case 44. Therefore, when restarting the crash again, we now set a breakpoint with `b *Clear_OBJECT_IDENTIFIER+44`, and then continue running the program with `c`. While this now lets us stop right before the problematic call to `free()`, the inconvient is that there are plenty of other calls to `Clear_OBJECT_IDENTIFIER` before the one causing the crash. However, by using the plugin to inspect the chunk that is about to be freed with `scudo chunk $x0`, it is very easily visible that at the first two breaks, the chunks passed to free are correctly allocated. Once the execution arrives at the third break though, the address passed to free is a garbage address, that is not even valid. While this could also have been identified without the plugin, by going even higher in the backtrace stack, we arrive in the `Convert_ASN1_to_X509_NAME` function at offset 488, where we can see with the plugin that we are trying to call `Delete_OBJECT_IDENTIFIER` with a chunk that is not allocated. So from that point why we already found the reason for the crash, we can finalize our investigation by looking at the `disassemble` of that function. With some investigation we can figure out where on the stack the address of the chunk we're trying to free comes from. So we can restart the crash again, this time breaking at the start of `Convert_ASN1_to_X509_NAME` and by stepping through the lines of the program while observing the stack, we can see with the plugin that the chunk is allocated at some point, but that it gets freed before we try to free it again. So the crash is clearly a classic double free error, which can also be confirmed by decompiling the whole library with Ghidra, where we can see that the object is deleted, but the pointer is not set to `NULL`, but the later code checks if the pointer is null, and otherwise frees it again, resulting in this double free crash.

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

The plugin works on real Android phones as well as on the standalone scudo version and it can easily be loaded inside gef. The commands of the plugin allow the inspection of the most important structures of scudo, which can be used in investigating real crashes. While the source code of scudo was already openly accessible and there were some (though quite few) documentations about scudo, there was no easy way to investigate crashes related to scudo on real Android apps with native libraries. While the plugin handles the most important structures, it also builds a base which could be extended to analyze further structures used by scudo, like quarantine related ones.