SCUDO Hardened Allocator - Unofficial Internals Documentation

by Rodrigo Branco (BSDaemon)

v0.03[1]

Abstract

SCUDO is a user-mode memory allocator developed by Google [[1]], based on the LLVM Sanitizers' Combined allocator [[2]] and with a focus on practical security (as in: instead of trying to implement theoretical security improvements that are unproven, it tries to address practical exploitable scenarios that are relevant). SCUDO is currently used in many places, with Android 11+ [3] and Fuchsia [4] as the most prominent cases. SCUDO is currently developed as part of the LLVM Project [5].

There are a few good materials about SCUDO available, discussing some of the internals, objectives and characteristics [6] [7] [8], but to the best of my knowledge, nothing that offers a more comprehensive (and updated) view of the implementation details.

Given that SCUDO's primary objective is security, this document also covers some of the decisions made, trade-offs and limitations. A small comparison of security capabilities with other allocations is also offered, with a pragmatic approach (instead of just a table of features that do not indicate much about the real resilience against real bugs). Some of the known weaknesses of SCUDO are also discussed.

All in all, the hope is that this article provides insight into the implementation. The article also has references to other materials for the readers who want to dig deeper, or who want to see original discussions on the different parts of the allocator. Because the allocator changes frequently, I've tried to cover a very recent version (HEAD -> Latest commit on Jun 13th 2023) for the data structures and organization, but also provide defaults from the latest Android, especially where platform-specifics are not very relevant for the discussion.

Table of Contents

---

[1] A PDF version of this article, as well as supporting scripts, code, etc are available at https://github.com/rrbranco/ScudoAllocator. The article version will be updated if anything changes to address feedback, mistakes or improvements.

Introduction

SCUDO has a lot of settings/configurable parameters which we will discuss later, but first, it is important to note that historically there were two main implementations of SCUDO inside the LLVM repo:

- Standalone (used in Android/Fuchsia and currently the only remaining implementation and what we are using for the discussions of the data structures, algorithms and drawings). It is a (modern) C++ implementation that lives inside compiler-rt (on the LLVM repo), but can be compiled without any dependencies on the rest of the project. The standalone implementation became feature-complete (comparatively with the embedded/integrated version) with the RSS feature implementation patchset [9].
- Embedded/Integrated (original implementation integrated into LLVM's sanitizer) was the original implementation in C. It is not present in the LLVM repo anymore, as it seems to have been removed between versions 15.x and 16.x by this patch [21], and we won't be discussing it in this article.

Once the LLVM repo [5] is cloned, to find the SCUDO source code just navigate to the subdir: *llvm-project/compiler-rt/lib/scudo/standalone*

All source code paths provided in this write-up will be relative to the scudo/ subdir (thus llvm-project/compiler-rt-lib/scudo is assumed to be your working directly, and the article will refer to standalone/<file_name> as the path).

While for the purposes of the write-up there will be no need to compile SCUDO, the author really likes to play with source code that is studied when available. It is also helpful to have a quick command-line option to compile SCUDO for those planning to create patches for it:

```
cd llvm-project/compiler-rt/lib
clang++-11 -gmlt -fPIC -fvisibility-inlines-hidden -Werror=date-time
-Werror=unguarded-availability-new -Wall -Wextra -Wno-unused-parameter -Wwrite-strings
-Wcast-qual -Wmissing-field-initializers -pedantic -Wno-long-long
-Wc++98-compat-extra-semi -Wimplicit-fallthrough -Wcovered-switch-default
-Wno-noexcept-type -Wnon-virtual-dtor -Wdelete-non-virtual-dtor -Wsuggest-override
-Wstring-conversion -Wmisleading-indentation -fdiagnostics-color -ffunction-sections
-fdata-sections -Wall -std=c++14 -Wno-unused-parameter -O3 -m64 -Werror=conversion
-Wall -g -nostdinc++ -fvisibility=hidden -fno-exceptions -Wno-pedantic -fno-lto -O3
```

-fno-omit-frame-pointer -pthread -shared -I scudo/standalone/include/ scudo/standalone/*.cpp -o ~/libscudo.so

This will create a libscudo.so shared library in the user's home dir. It goes without saying that most of the parameters are not needed, but in the author's experience those are the flags used by the LLVM's automated test bench for any submitted patches. Notice that for just playing with SCUDO in a target software, one can just use the version that comes with clang:

```
clang -fsanitize=scudo -Wall -g -no-pie mycode.cpp -o mycode
```

For testing compilation targets that are NOT linux, standalone/platform.h has the define's used (SCUDO_FUCHSIA for Fuchsia, SCUDO_TRUSTY for Trusty, which is not a pure linux, __BIONIC__ which then defines SCUDO_ANDROID for Android). Still in the same platform file, we have the __LP64__ which defines the word size to be 64 bits (otherwise it will be a 32-bit build).

Besides the compile-time options, including platform-specific configurations, SCUDO also has other tunable parameters which can be changed in runtime by applications. In standalone/allocator_config.h are the configuration options that are defined for the primary, secondary, and overall SCUDO, which can be replaced with platform-specific definitions. Those application-accessible configuration options are the so-called SCUDO-FLAGS, and a list of them (along with a short explanation and their default values in case they are not defined) can be seen in standalone/flags.inc. **Table 1** has the full list.

Table I. SCUDO run-time flags

Flag	Default Value	Description
quarantine_size_kb	0	Size (in kilobytes) of quarantine used to delay the actual deallocation of chunks. Lower value may reduce memory usage but decrease the effectiveness of the mitigation
thread_local__quarantine_size_kb	0	Size (in kilobytes) of per-thread cache used to offload the global quarantine. Lower value may reduce memory usage but might increase the contention on the global quarantine
quarantine_max_chunk_size	0	Size (in bytes) up to which chunks will be quarantined (if lower than or equal to)
dealloc_type_mismatch	false	Terminate on a type mismatch in allocation-deallocation functions, eg: malloc/delete, new/free, new/delete[], etc
delete_size_mismatch	true	Terminate on a size mismatch between a sized-delete and the actual size of a chunk (as provided to new/new[])
zero_contents	false	Zero chunk contents on allocation
pattern_fill_contents	false	Pattern fill chunk contents on allocation
may_return_null	true	Indicate whether the allocator should terminate instead of returning NULL in otherwise non-fatal error scenarios, eg: OOM, invalid allocation alignments, etc
release_to_os_interval_ms	INT32_MIN (Android), 5000 otherwise	Interval (in milliseconds) at which to attempt release of unused memory to the OS. Negative values disable the feature.
hard_rss_limit_mb	0	Hard RSS Limit in Mb. If non-zero, once the limit is achieved, abort the process
soft_rss_limit_mb	0	Soft RSS Limit in Mb. If non-zero, once the limit is reached, all subsequent calls will fail or return NULL until the RSS goes below the soft limit
allocation_ring_buffer_size	32768	Entries to keep in the allocation ring buffer for SCUDO

To run a ‘test’ binary with SCUDO as the allocator and play with the configuration flags, one can LD_PRELOAD SCUDO and set the environment variable SCUDO_OPTIONS, like this:

```
SCUDO_OPTIONS="may_return_null=false hard_rss_limit_mb=10 soft_rss_limit_mb=1" LD_PRELOAD=./libscudo.so ./test
```

Be aware that an application might also choose to implement the __scudo_default_options() to define its runtime options. Here is an example:

```
extern "C" const char *__scudo_default_options() {
  return "may_return_null=true "
        "quarantine_size_kb=0 "
        "thread_local_quarantine_size_kb=0 "
        "zero_contents=false "
        "dealloc_type_mismatch=false ";
}
```

Some flags apparently have been deprecated, such as the abort_on_error that used to make SCUDO call abort() instead of exit() on errors. (The idea was to offer a minimal path between an error detected and the termination of the program).

SCUDO Basics

To help readers in understanding SCUDO’s implementation, it is necessary to discuss common memory allocator (heap) weaknesses and potential mitigations. I’ve used the different implementation choices/options as well as the SCUDO’s author descriptions [10] [11] to infer the stated objectives. They do not intend to be a complete discussion on heap-based vulnerabilities (which involve classes [12] such as overflows, double-frees, use-after-frees, invalid-frees, info leaks, etc) and mitigations, but to initiate the discussion specifically on SCUDO’s choices:

- No security through obscurity. (The allocator code is open-source and well-documented, including the implementation-specific parts).
- An attacker might try to force allocations and deallocations in the target software, including controlling the sizes and ordering of such operations (Heap Feng Shui [13]/Heap Massaging [14]).
- An attacker might also want to control big memory areas by spreading allocations with controlled values (Heap Spraying [15]).
- Vulnerable code might have multiple memory corruption bugs, giving the attacker many different primitives [16].

With the above in mind, SCUDO provides the following security properties:

- *Less determinism*: We will discuss how SCUDO adds randomness into different parts of the implementation, but in a nutshell there are two main sources of randomization in SCUDO. We will go in detail when we introduce these different algorithms later in the write-up. One source of randomization is in the Primary Allocator in the start of every region, which helps against small out-of-bounds accesses from one region into another. The second is the shuffling of the allocated blocks in the TransferBatch. There is also the

OS-provided randomization that SCUDO's implementation leverages, which affects both the primary and the secondary allocators inside SCUDO.

- *Metadata integrity*: A common technique used is to manipulate the allocator's metadata to obtain more powerful primitives, for example transforming an overflow into an arbitrary read/write. SCUDO implements a simple (thus fast) but effective checksumming algorithm, as well as validates the state of data structures before performing operations.

For the reader interested in understanding how SCUDO implementation evolved, here is what seems to be the first patch to the Sanitizer's Allocator that leverages OS-randomization [17], and the chunk shuffling patch [18]. Also interesting is the clean-up of callbacks from the Sanitizer Allocator [19], which explains why SCUDO originally removed the support for malloc hooks, in order to prevent attackers from controlling pointers used by different hooks to gain code execution [20].

SCUDO has three main components:

1. TSDRegistryShared (we will just say local cache).
    a. Has a set of local caches.
    b. There is a local cache for every Thread Specific Data (TSD).
    c. Uses a shared and non-exclusive model -> so a set of threads own the same TSD.
2. MapAllocator, also known as the Secondary Allocator, mostly used for large allocations (larger than 64KB).
    a. When we delve into the implementation details, we will cover an exception that might cause smaller allocations to go to the secondary allocator
3. SizeClassAllocator{32|64}, also known as the Primary Allocator, used for 'small' allocations (smaller or equal to 64KB).
    a. The Primary Allocator and the local cache are both involved in allocations <= 64KB.

Given that the allocator is not initialized at the start of a program, the allocator's data structures are either mmap'ed to memory or they are included as part of the .data section. Specifically on Android, you can find SCUDO in the .data section of the 'libc.so' (standard libc). **Figure 1** has an overview of the main data structures and their relations inside the .data section of 'libc.so'.

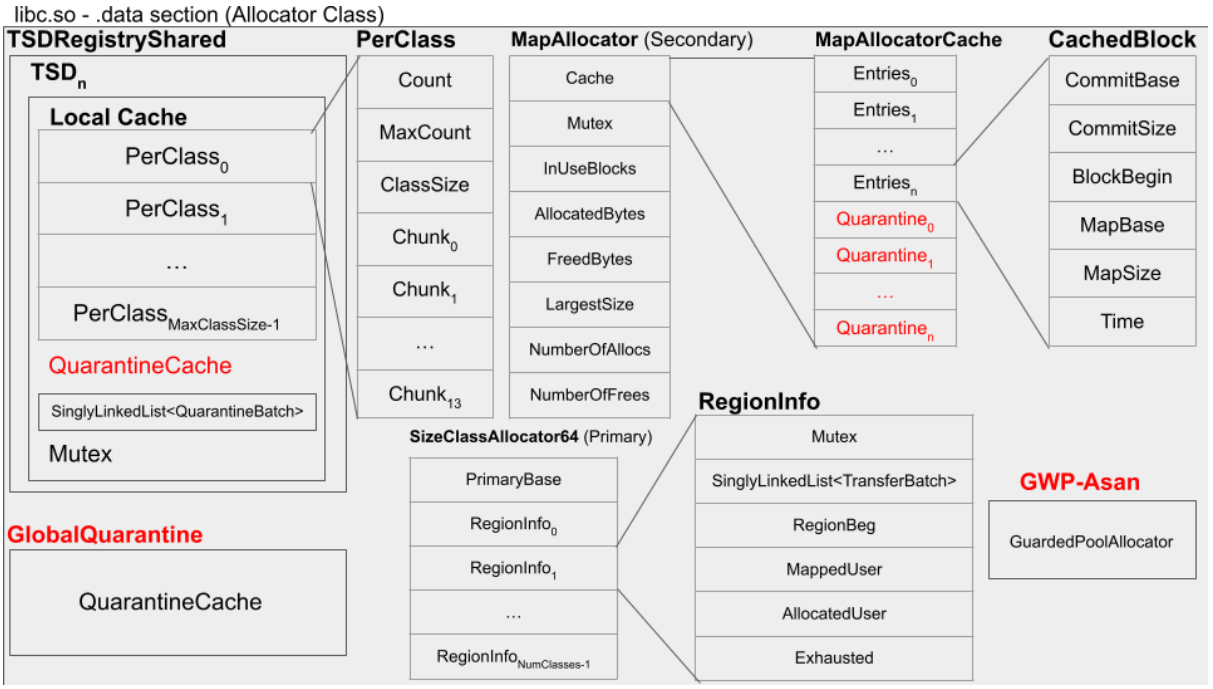

Figure 1 - SCUDO's structures in the 'libc.so' .data section

From the image, we can see that there is a third allocator in SCUDO, the GWP-ASAN (also known as GuardedPoolAllocator), but it is disabled by default. We will discuss GWP-ASAN later. The QuarantineCache (a thread local quarantine), the GlobalQuarantine, and the MapAllocator quarantine are also disabled by default in SCUDO code, and all of them are disabled as well in the compiled SCUDO for Android -- maybe some OEMs enable them, check your devices and let me know. The disabled elements are shown in red in **Figure 1**. When not specified, this article uses 64-bit, non-svelte (non-memory constrained) Android 11 default AOSP/configs/settings and tries to include the new changes from HEAD to the data structures/features discussion. Overall, this article goes through the most relevant fields in the data structures (and includes drawings with them and their relations), but does not intend to necessarily cover all of them, nor does it intend to cover small differences due to configurations, platforms or versions.

SCUDO Allocator Initialization

SCUDO keeps a 'registry' of TSDs (presumably, Thread Specific Data). The number of TSDs is platform-specific and based on the number of CPUs and up to a max configurable value (by default 8 on 64 bit). When the libc is mapped to memory for the first time and a new thread of the program is created and goes to make its first 'malloc' allocation, the Thread Local Storage (TLS) slot (which is also platform-specific and usually 6 - TLS slots are defined by a platform by defining a getPlatformAllocatorTlsSlot() function in a scudo_platform_tls_slot.h include file when compiling SCUDO) stores a pointer to one of the TSDs owned by SCUDO's TSD Registry. Every newly-created thread repeats the same process, getting assigned one of those TSDs in round-robin when they do an allocation. The TSDRegistryShared structure is defined in standalone/tsd_shared.h. As depicted in **Figure 1,** there is a mutex for each TSD, so every time an allocation or deallocation happens by a

thread, the mutex has to be acquired. The local cache is also initialized together with the TSD and simply consists of an array of PerClass structures (PerClass structure is defined in standalone/local_cache.h). The local cache is used for fast allocation/deallocation.

Each PerClass corresponds to a range of sizes that can be allocated. For 64-bit, there are 32 different possible sizes (see **Listing 1**) (defined in: standalone/size_class_map.h).

— Listing 1 - Classes for 64 bits —

```
 static constexpr u32 Classes[] = {
    0x00020, 0x00030, 0x00040, 0x00050, 0x00060, 0x00070, 0x00090, 0x000b0,
    0x000c0, 0x000e0, 0x00120, 0x00160, 0x001c0, 0x00250, 0x00320, 0x00450,
    0x00670, 0x00830, 0x00a10, 0x00c30, 0x01010, 0x01210, 0x01bd0, 0x02210,
    0x02d90, 0x03790, 0x04010, 0x04810, 0x05a10, 0x07310, 0x08210, 0x10010,
 };
```

— End of Listing 1 —

For all others (in practical terms, 32-bit), there are 64 different possible sizes (see **Listing 2**) (also defined in: standalone/size_class_map.h).

— Listing 2 - Classes for non 64 bits —

```
 static constexpr u32 Classes[] = {
    0x00020, 0x00030, 0x00040, 0x00050, 0x00060, 0x00070, 0x00080, 0x00090,
    0x000a0, 0x000b0, 0x000c0, 0x000e0, 0x000f0, 0x00110, 0x00120, 0x00130,
    0x00150, 0x00160, 0x00170, 0x00190, 0x001d0, 0x00210, 0x00240, 0x002a0,
    0x00330, 0x00370, 0x003a0, 0x00400, 0x00430, 0x004a0, 0x00530, 0x00610,
    0x00730, 0x00840, 0x00910, 0x009c0, 0x00a60, 0x00b10, 0x00ca0, 0x00e00,
    0x00fb0, 0x01030, 0x01130, 0x011f0, 0x01490, 0x01650, 0x01930, 0x02010,
    0x02190, 0x02490, 0x02850, 0x02d50, 0x03010, 0x03210, 0x03c90, 0x04090,
    0x04510, 0x04810, 0x05c10, 0x06f10, 0x07310, 0x08010, 0x0c010, 0x10010,
 };
```

— End of Listing 2 —

If we would annotate the code to define the ranges that the PerClass covers, we would have something like in **Listing 3**.

— Listing 3 - Classes for 64 bits and their allocation ranges —

```
/* 64-bit classes */
static constexpr u32 Classes[] = {
        0x00020,                // 32 decimal (so it covers from 0 to 32 byte allocation)
        0x00030,                // 48 decimal (so it covers from 33 and 48 byte allocation)
        …
        0x10010,                // 64KB + 16 bytes (65552)
};
```

— End of Listing 3 —

Notice that SCUDO does allow for size-0 allocations. Also, the sizes are the sizes needed to fulfill the allocation (the requested size to malloc plus all the data structures). The individual blocks allocated contain chunks, which have a chunk header (see **Figure 2**) and have

alignment requirements (for example, on ARM64 it is 16 bytes). With all that, only allocations from 0 to 16 bytes actually end up in the first PerClass (on 64 bits), since the rest of the 16 bytes of space are used by internal requirements. That is also the reason why the last class size for 64 bits (as in **Listings 1 and 3**) is 0x10010 (0x10000 is 64 KB and when we add the 0x10 which is 16 bytes of data structures, gives us the total 65552 bytes). The $PerClass_0$ in $Region_0$ in the PrimaryAllocator is special and is used to store TransferBatch objects (which we will discuss later).

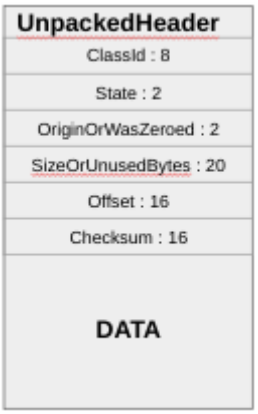

Figure 2 - Chunk Header

Still during the libc.so mapping, the allocator class is initialized. As shown in **Figure 1,** the primary allocator class name is SizeClassAllocator64 (on 64 bits) and SizeClassAllocator (on 32 bits). When running on a 64-bit system, SCUDO reserves (but does not commit) a large region of memory (2 ^ 28 * NumClasses bytes). NumClasses is defined in standalone/size_class_map.h and is different per-platform. On Android we have 256MB * 32 (the number of PerClasses), for a total of 16GB, which is only possible because 64-bit address space is so large. The reserved range is divided equally into regions, which we will cover in the **Primary Allocator** subsection.

Local Cache - Allocation

The PerClass structure has a Count, a MaxCount, the ClassID associated with the bin size (the ClassSize field) and an array of chunks. SCUDO uses compact pointers for the chunks, so in 64-bit systems they are stored as 32 bits (the base address is pointed to the region in the primary allocator). As previously stated, the PerClass structure is defined in standalone/local_cache.h. The Chunks array in each PerClass has the number of elements = 2 * TransferBatch::MaxNumCached (which is defined per supported system and is a hint for the maximum number of bytes cached per class).

In a malloc, when Count != 0 and Count != MaxCount (so the array has entries in it), for an allocation smaller or equal to 64KB (so, an allocation that uses the Primary Allocator), SCUDO has to:

1) Get a pointer from the chunk (and unpack it).

2) Fill some fields of the chunk header (such as the checksum).
3) Return the pointer, decrementing the counter.

The procedure is very simple (and as a result, very fast), which is the purpose of the local cache. There are a few special cases, such as when the Count == 0 in the PerClass (meaning that the bin of ClassSize is empty). In that scenario, the PerClass has to be repopulated, which from the local cache perspective means that it has to request a TransferBatch from the PrimaryAllocator (as shown in **Figure 3**).

The TransferBatch structure is not that different from a PerClass. A TransferBatch contains compact pointers (on 64 bits) to the chunk headers within all the blocks in the PrimaryAllocator. A TransferBatch is requested for a specific ClassSize, and is copied into the PerClass for that size, populating it. The Count=N returned depends on the Count of the TransferBatch (see **Figure 3**). The counter can then be decremented and a pointer to the data portion of the chunk can be returned to the requesting function (for example, malloc).

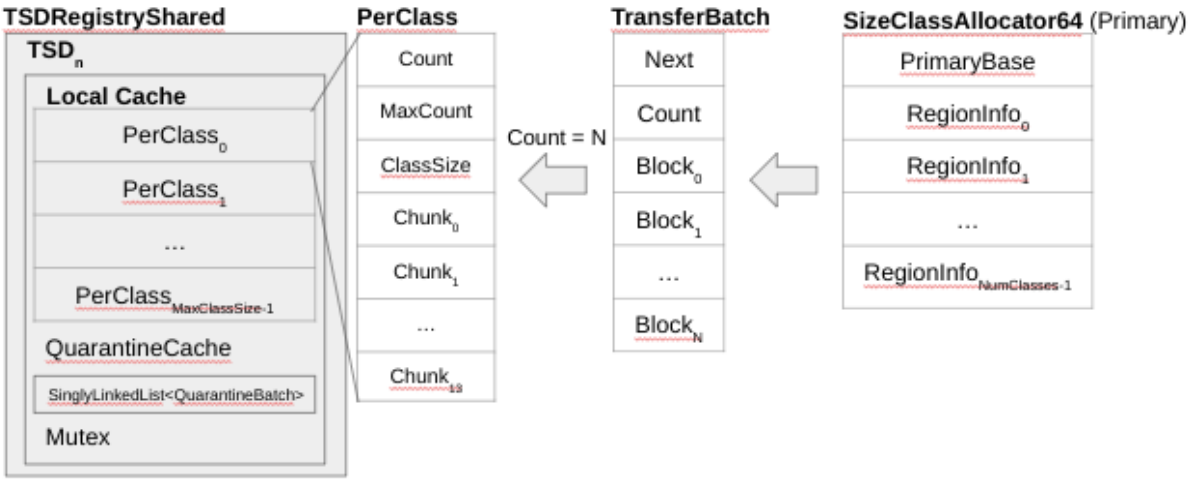

Figure 3 - PerClass - Allocation

Local Cache - Deallocation

The deallocation is the reverse operation. When Count < MaxCount, empty chunk slots exist, and the chunk headers have to be updated (like marking their state as free). The pointer is packed (in 64 bits) and put back on the list; finally, Count is incremented. The special case on the deallocation is when Count == MaxCount, meaning that the PerClass is full. A full PerClass requires a ‘Drain’. Draining is the reverse operation of repopulation, seen before in the allocation path. Draining is depicted in **Figure 4.**

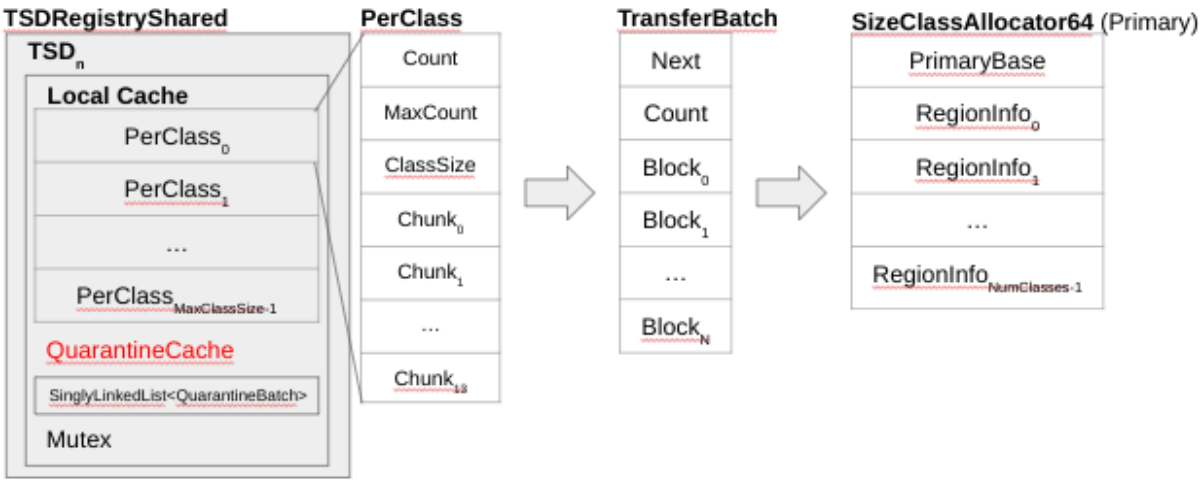

Figure 4 - PerClass - Deallocation

As shown on **Figure 4,** the local cache allocates a TransferBatch (the same data structure discussed previously), copies all the chunks to the blocks in the TransferBatch and sends the TransferBatch to the PrimaryAllocator corresponding region's freelist (still to be discussed). With that, the local cache PerClass is now empty, allowing the deallocated chunk to be pushed.

When pushing a new batch onto a freelist, the Primary64 allocator might release ranges of blocks to the OS using the MADV_DONTNEED flag (MADV_DONTNEED informs the kernel that the pages are not expected to be accessed in the near future, so the kernel can free resources associated with it). We better explain the 'why' behind this choice in the Secondary Allocator section.

The most complicated pieces of SCUDO (but still, following the overall simplicity in its design) are the Primary and Secondary allocators: they are responsible for managing the underline chunks.

Primary Allocator

The primary allocator (SizeClassAllocator64 Class) has a few RegionInfoClass (as shown in **Figure 5**).

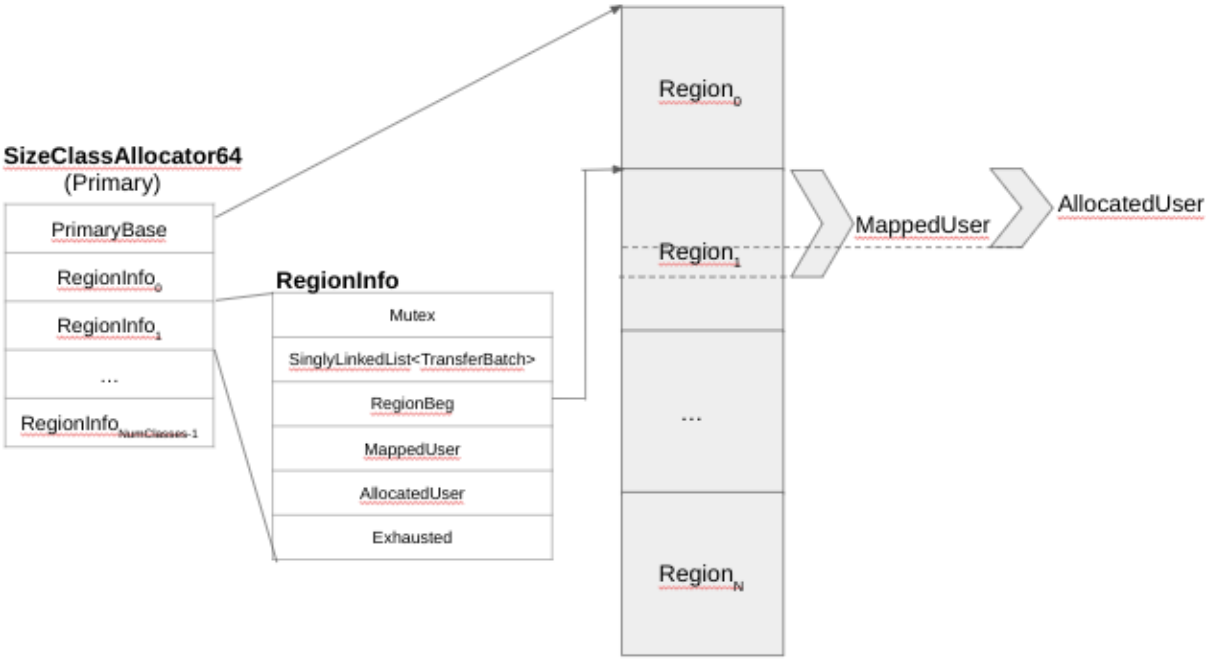

Figure 5 - Primary Allocator Data Structures

The RegionInfoClass is the 'glue' between the PerClass and the SizeClass (having a 1<->1 correspondence with them). The singly-linked list TransferBatch structure in the RegionInfo shown on **Figure 5** is the FreeList implementation. Commonly, allocators manage individual chunks to be returned by malloc, but on SCUDO, the primary allocator manages sets of chunks and stores them in a TransferBatch, making the FreeList a singly-linked list of TransferBatches and not chunks.

As said before, $Region_0$ is special, since it is used to store the TransferBatch objects. The MappedUser and AllocatedUser integers represent, respectively, the number of bytes committed in the region (backed by physical pages) and how many bytes within the committed region that are actually allocated for blocks. Over time, the difference (MappedUser - AllocatedUser) divided by blocksize is the amount of memory that is committed but still available to allocate blocks.

Because the regions never grow, when a specific class size is exhausted, the region is marked as exhausted (using the Exhausted field in RegionInfo data structure on **Figure 5**). When a region is exhausted, the repopulation discussed previously for the local cache fails, which results in a local cache miss. That is when the logic for both Primary and Secondary allocators moves to the next (bigger) ClassSize to retry the repopulation to address a given allocation. SCUDO calls such logic the Combined Allocator (implemented in standalone/combined.h). If every region is exhausted, SCUDO will fall back to the secondary allocator, and that is how small allocations might end-up getting serviced by the secondary allocator. This is the reason why previously we've mentioned that the secondary allocator is *usually*, but not always, used for bigger allocations.

If we once again consider the case of the first allocation by a thread, as explained, it gets assigned a TSD: If the PerClass is empty, it has to repopulate. The Primary Allocator in that case has to return a TransferBatch for a certain ClassSize (ClassID), so it has to look up the

region that corresponds to it, to get the region's freelist (as discussed, the singly-linked list of TransferBatch). If it is not empty, it just has to return the first entry. Initially it should be empty though, so both MappedUser and AllocatedUser will be equal to 0 (as shown in **Figure 6**). SCUDO commits memory in 256 KB chunks (64 times PageSize, which is 4 KB on most platforms - PageSize is obtained via getPageSize(), which is platform-specific - Android/Linux implementation is in standalone/linux.cpp, Fuchsia's in standalone/fuchsia.cpp and Trusty's in standalone/trusty.cpp). That essentially means that once memory is committed, the MappedUser is incremented by that size (256) and the requested ClassSize blocks will start being allocated (**Figure 7**).

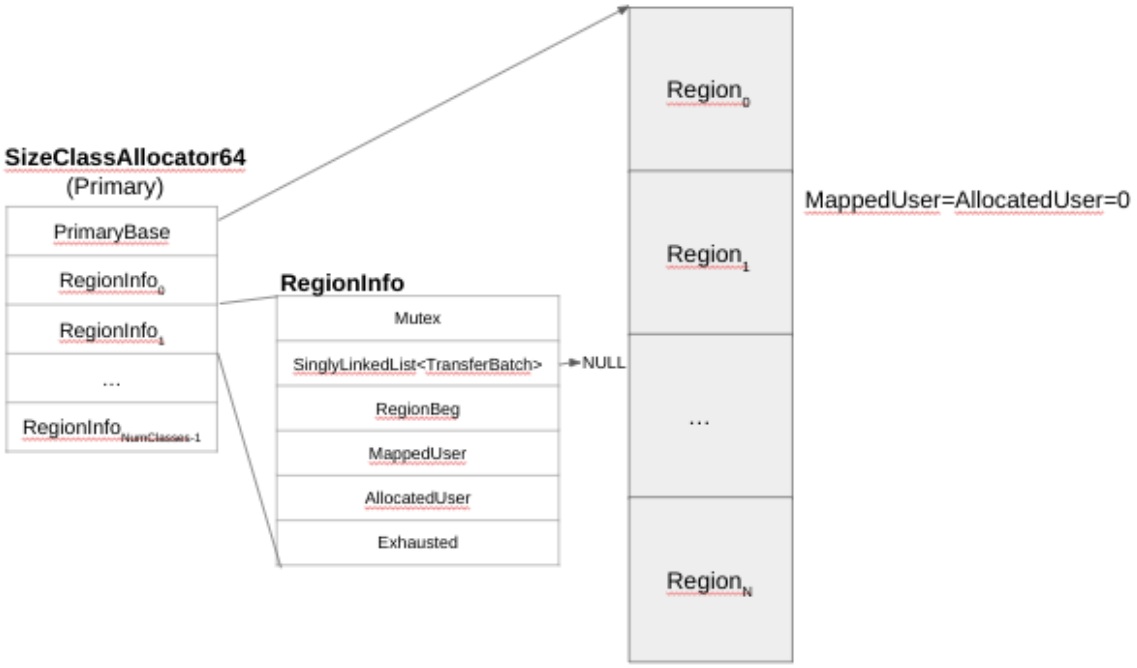

Figure 6 - Primary Allocator - Initial Region State

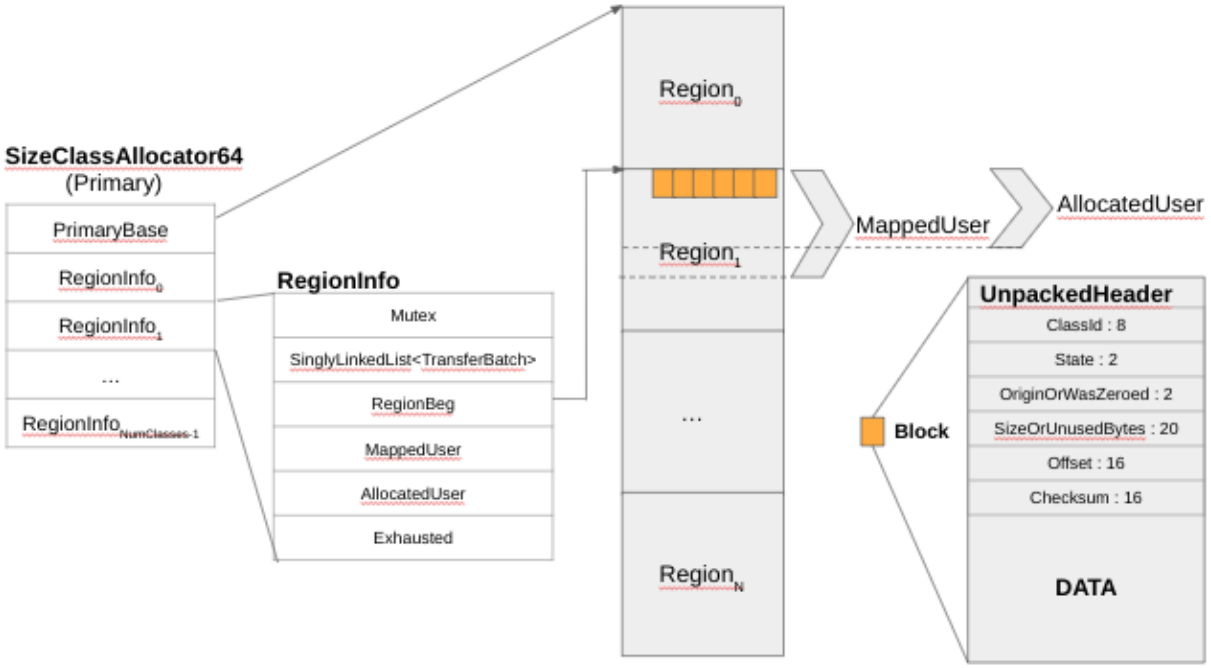

Figure 7 - Primary Allocator - Allocated Blocks

As was briefly mentioned earlier, one of the sources of randomization in SCUDO is the start of the blocks in a given Region. **Figure 7** shows that the blocks are ***NOT*** at the region start. There is a random 1 to 16 pages offset into every region that defines where the blocks start to be allocated. The randomization helps against bugs when the overwrite moves from one region to another (for example, due to a small index out-of-bounds access).

Once the memory is committed and MappedUser is incremented, the block is ready for SCUDO's use. A small nuance is that from the PrimaryAllocator perspective, the blocks in the regions are called blocks, but other places in the source code, they are called chunks. In the write-up we were not pedantic on the terms, but they are not the same. While there is only 1 chunk for every block, the chunk header might ***NOT*** start at the base of the block, due to alignment requirements. To address the case in which the chunk does not start at the base of the block, there is a special 4-byte block marker (aptly named BlockMarker and defined in standalone/combined.h as 0x44554353 - or 'SCUD', in little-endian hex) followed by an offset to the chunk header. We are not pedantic on the terminology because normally the chunk header is at the base of the block. Right after the block header resides the block data (also shown on **Figure 7**). After all blocks are allocated, the TransferBatch is allocated (shown in **Figure 8**).

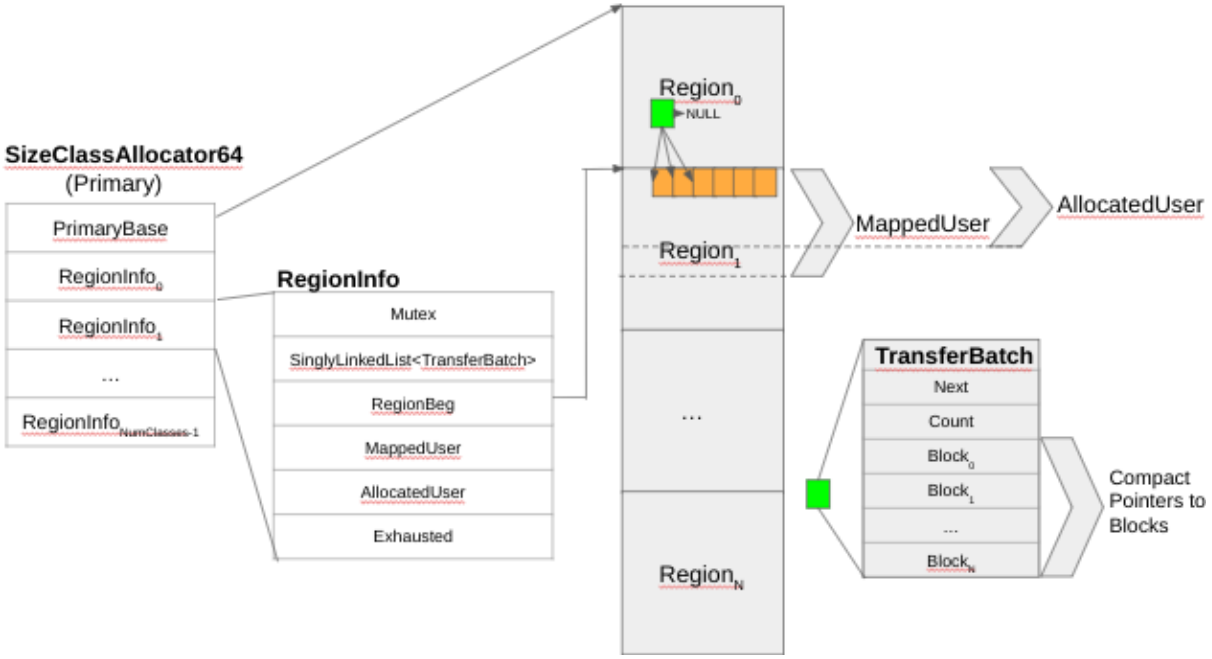

Figure 8 - Primary TransferBatch

As mentioned before, the $Region_0$ and $PerClass_0$ are special because they are used to store those TransferBatch objects. The reason for this is that they are SCUDO's own management data that are dynamically allocated and have to be stored somewhere. The TransferBatch was covered in the **Local Cache** section. As previously mentioned, the second source of randomness within the Primary Allocator is related to the TransferBatch, because the blocks are shuffled before they are added to it, as shown on **Figure 9** (the implementation is in the function Shuffle, at standalone/common.h and part of the class). When the TransferBatch is returned to the local cache (it is simply copied over), the order in which mallocs of a ClassSize happen preserves that randomization, meaning that the blocks returned are randomized. That is the most important source of randomization in SCUDO.

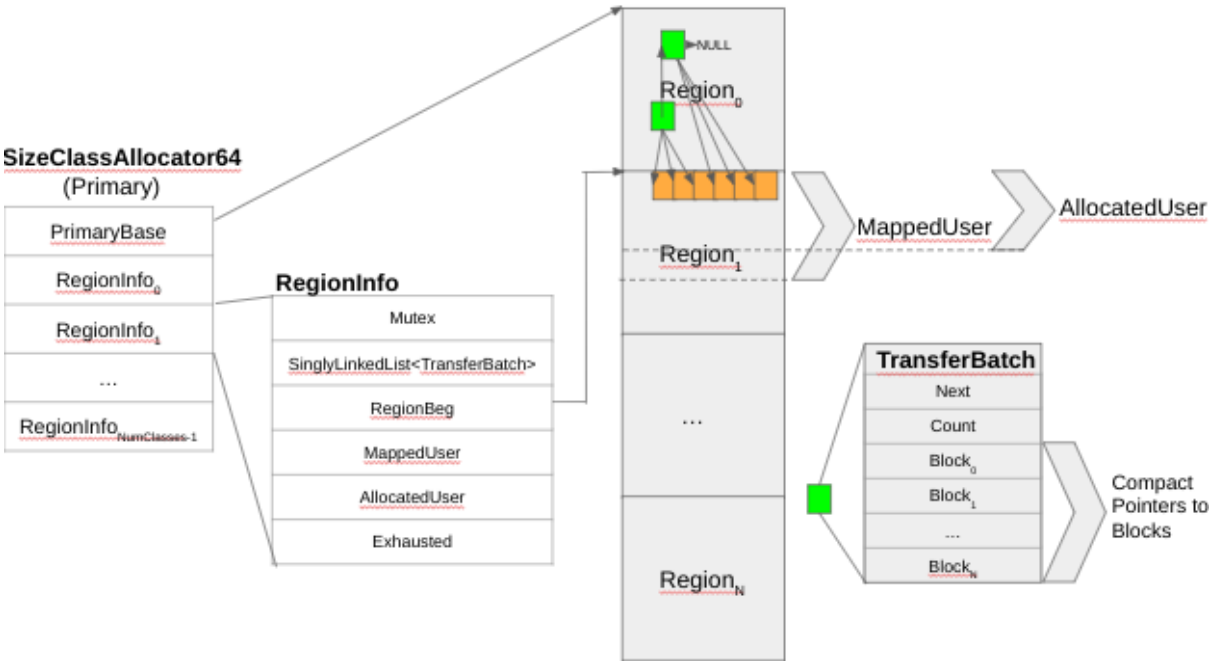

Figure 9 - TransferBatch Shuffle

When the freelist is repopulated, Scudo does not only create one TransferBatch to copy on the freelist, but instead up to 8 (on 64-bit and depending on a configurable value). After the freelist is available, an entry is sent to the local cache, which is then repopulated.

Secondary Allocator

As mentioned before, the secondary allocator is used for allocations bigger than 64KB or when all the primary regions that could satisfy a request are exhausted (in-use).

The object class for the Secondary Allocator is the MapAllocator and it is defined in standalone/secondary.h (**Figure 1** has the MapAllocator data structures represented).

Figure 10 shows the MapAllocator relation with an allocated block (InUseBlocks pointer).

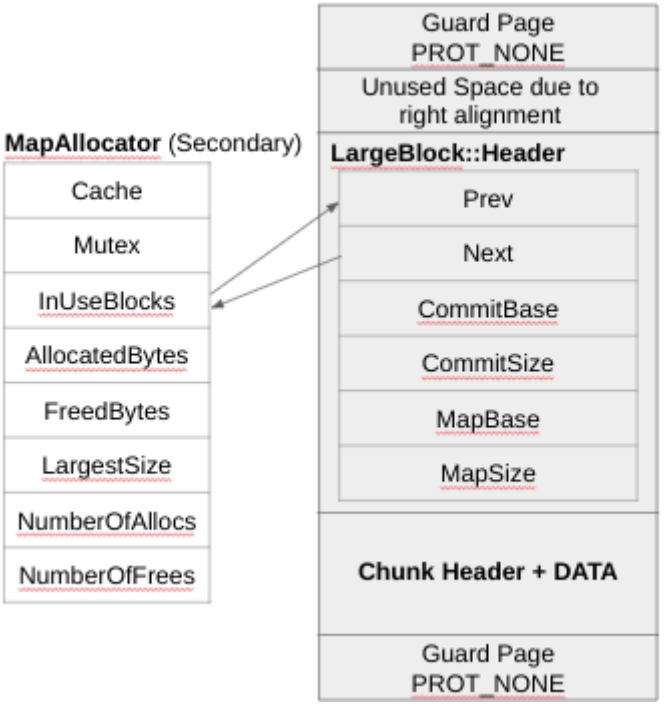

Figure 10 - Secondary Allocator and InUseBlocks

A doubly-linked list permits the implementation of a function malloc_iterate() [22], which is used to iterate over all allocated chunks within the system (there is also a way to iterate in the Primary Allocator because the blocks are of a certain size and the headers have a state to say if they are available or allocated). Allocations by the Secondary Allocator include, as shown in **Figure 10**:

- Guard Page
- Unused Space due to right-alignment of the next items (the data is right-aligned and after the alignment there is another guard page)
- LargeBlock::Header (defined in standalone/secondary.h), which consists of the following fields (also shown in **Figure 10)**:
    - Prev/Next (doubly-linked list pointers).
    - CommitBase for the pointer to the committed region (the end of the first guard page to the beginning of the next).
    - CommitSize (basically the size of the committed region which is the region between guard pages).
    - MapBase points to the first guard page.
    - MapSize the size between the end of the second guard page and the beginning of the first (so the whole mmap'ed size).
- Chunk Header + Data
- Guard Page

All of the above gets mmap'ed, with the guard pages having PROT_NONE and the rest PROT_RW (in the Mitigations section we discuss how ARM MTE impacts this). The secondary allocator also has a cache and a quarantine. **Figure 1** shows the connection between the MapAllocator Cache field and the MapAllocatorCache data structure. The MapAllocatorCache stores by default 32 entries, called CacheBlock. A CacheBlock (shown

in **Figure 11**) has the same basic fields as the LargeBlock::Header (plus a time field, minus the doubly-linked pointers). When SCUDO frees (or deallocates) a large block (anything managed by the Secondary Allocator), two thresholds are going to be checked: the max cache size for the Secondary Allocator (2MB) and the maximum number of entries (32). If either criteria is met, then it is put back into the secondary allocator cache. The time is a monotonically increased timestamp. When SCUDO goes to store something in the MapAllocator cache, it iterates over all the entries and calls MADV_DONTNEED on any entry that is greater than 1s. The idea is to cover a common scenario in which the application starts to quickly make allocations bigger than 64 KB (thus using the Secondary Allocator). This is really fast, but if suddenly the application changes the pattern and is not allocating and deallocating fast anymore (it is taking more than 1s), the system is able to try to release the frames (physical pages backing those allocations).

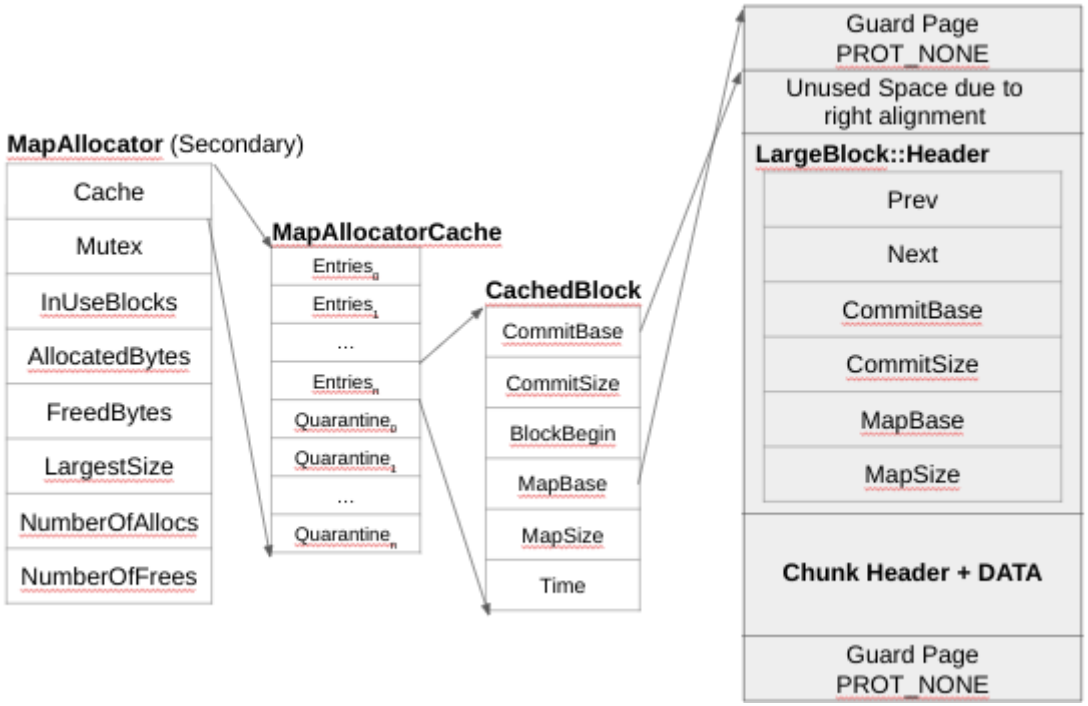

Figure 11 - MapAllocator and MapAllocatorCache

The Secondary Allocator quarantine is very similar to the primary allocator quarantine. The quarantine does not try to optimize by amortizing the cost of frees like the traditional delayed freelist. Instead, it is a mechanism to try to prevent use-after-frees [12]. ARM MTE (Memory Tagging Extension) [23] supersedes the quarantine.

Security Mitigations in SCUDO

Throughout the write-up mitigations have been briefly discussed, but the focus was on the algorithms and data structures of the management implementation. The following is a summary and additional details on aspects of implementation that are relevant to security.

Integrity and Consistency Checks

First of all, it was mentioned that SCUDO has integrity checks on the metadata. Every block has a chunk header right before its data (and right after the large block header in the case of the Secondary Allocator). As shown in **Figure 2**, these are the fields and their uses:

- ClassID (8 bits) (same ClassID that maps to the ClassSize discussed before)
- State (2 bits) stores the state of the allocation (free, allocated, quarantined)
- OriginOrWasZeroed (2 bits) is used when a chunk is allocated to have one of 4 possible origins. While for SCUDO something is either an allocation or a deallocation, a C/C++ program may have different ways of triggering these actions, and SCUDO tracks and checks them for consistency when deallocating, explained a bit better later in this section:
  i. malloc
  ii. memalign
  iii. new operator
  iv. newarray operator
- SizeOrUnusedBytes (20 bits) is used to keep track during re-allocations. For example, if a chunk is realloc(), SCUDO won't have to migrate to another block size if it still has slack space on the current block, it can just grow the chunk up to the requested size.
- Offset (16 bits) holds the distance to the chunk from the beginning of the block (if the chunk header is in the beginning of the block, the offset is 0).
- Checksum (16 bits) is a CRC32, with the seed set to a 32-bit random cookie that is stored within the top level allocator class (class Allocator in standalone/combined.h). The 32-bit cookie in the static allocator class is set to a random value during initialization. Once SCUDO calculates the CRC32, it XORs the low and high 16 bits together and that is the stored checksum (16 bits). The algorithm is performant (due to fast HW implementations for CRC32), but is vulnerable to collisions. If an attacker is able to leak a heap address and the header of a block, it is possible to leverage the collisions to produce a valid checksum, as demonstrated by Silvio Cesare [8]. This is a good example of a security/performance trade-off. SCUDO also supports something different than CRC32, called BSDChecksum. It is a 16-bits checksum (implemented in standalone/checksum.h by the function computeBSDChecksum()), but this is not enabled on most modern platforms.

Whenever a header is accessed, SCUDO uses two helper functions loadHeader/storeHeader (both defined in standalone/chunk.h and that are used on free/deallocate for example). This is where SCUDO verifies the checksum. For ARM MTE, the actual tags for the 16-byte granules within the header are different tags than the ones used for the data portion (even the large block header will be a different tag). As a result, this is a convenient place to validate the checksum, and if the checksum is invalid, SCUDO calls Abort (and it never returns).

We already talked about the 4 origins that are tracked, which alludes to an interesting bug class. The origin is checked during deallocation for mismatch. For example, if a C/C++ program uses the new array operator and tries to free the memory by calling free() instead of the delete array operator, the application will abort.

There are also chunk state validity checks, so if the chunk state is not valid, it results in abort(). Some examples of invalid states:

1. During free(), the header state that is passed has to be in an allocated state, otherwise, abort(). This prevents double frees.
2. The chunk state must be Quarantined during recycling.

Randomization

We already discussed the two instances of randomization, but just for completeness:

1. The order of blocks within newly-created TransferBuffers is shuffled which results in chunks in the same class size being non-contiguous in memory. There still exists spatial locality for newly-allocated sets of blocks by the primary allocator.
2. The beginning of each region for each ClassSize in the primary allocator starts at a random 1 to 16 page offset. A weakness of this is the fact that primary allocations forced to the secondary allocator miss such randomization. Randomizations of mmap() by the kernel still affect the secondary allocations, but this is not SCUDO-defined and is target-system dependent so the problem is only relevant for systems that don't have such randomization.

Quarantines

We have mentioned the Quarantines in SCUDO, but they are disabled by default. Quarantines provide a weak use-after-free mitigation and are superseded by ARM MTE. Essentially each TSD (that is pointed by each thread) also has a local quarantine cache. The quarantine caches are just singly-linked lists of QuarantineBatch data structures (defined in standalone/quarantine.h). As seen in **Figure 12**, they are very similar to PerClass and TransferBatch.

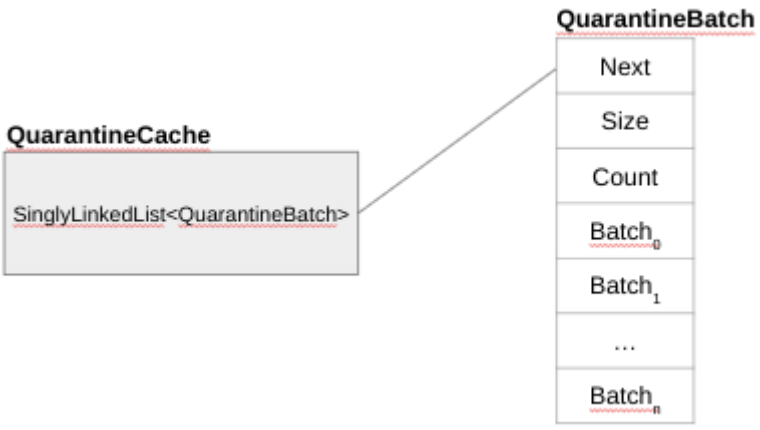

Figure 12 - QuarantineCache

The quarantine is implemented in the deallocation patch, so it applies only when a chunk is deallocated. There is a configuration option, quarantine_max_chunk_size (as seen in **Table 1**, and defined as 256KB on Android) that limits the maximum size of a chunk to be quarantined. There is also a threshold for the local quarantine (the configuration option thread_local__quarantine_size_kb in the **Table 1**, which represents not the number of entries in the quarantine, but the number of bytes in it, so the sum of the sizes of all chunks). If this threshold is surpassed, then SCUDO acquires the lock for the GlobalQuarantine and will transfer (merge) all those QuarantineBatch to the GlobalQuarantine. The GlobalQuarantine also has a configurable threshold (the configuration option quarantine_size_kb in the **Listing 1**). When this threshold is crossed, the many QuarantineBatch get merged (to save space) and they recycle by simply going through each batch, calling deallocate on the chunk, and following the usual deallocate patch to return it to the Primary or Secondary allocators.

GWP-ASAN (GuardedPool Allocator)

The GuardedPool Allocator is also disabled by default and it is similar to the sanitizers allocator (it is implemented in standalone/combined.h). When the GuardedPoolAllocator is first initialized, it reserves a large region of memory, similarly to the Primary Allocator. But the GuardedPool Allocator interleaves every slot (by default 4 KB) with guard pages to detect overflows/underflows (as shown in **Figure 13**).

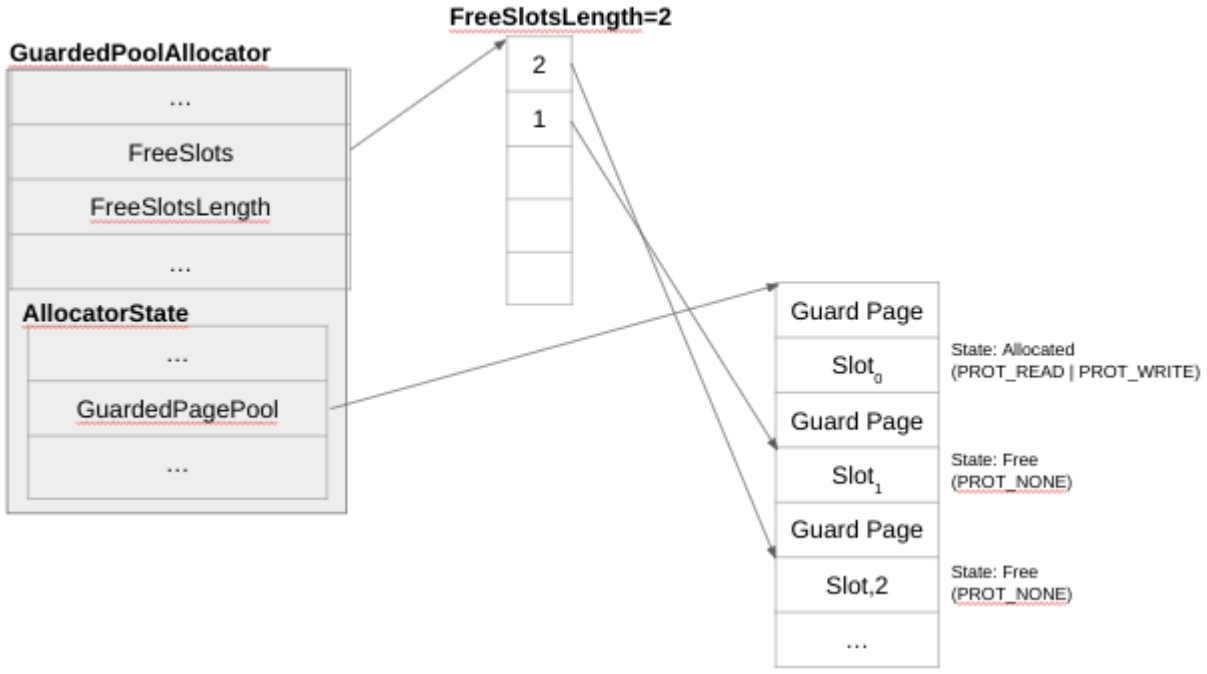

Figure 13 - GuardedPoolAllocator

The GuardedPoolAllocator also implements a freelist, which is just an array of indexes to available slots (FreeSlots field). For every deallocation/free, the GuardedPoolAllocator sets the slot back to PROT_NONE. This mechanism permits the GuardedPoolAllocator to catch (with high probability) use-after-free. The reason why the mechanism is not deterministically

catching all use-after-frees is because there is randomization in which slot is picked within the FreeSlots array.

The GuardedPoolAllocator, if it is enabled, is only used in sampling (otherwise it would have a high overhead). The sampling Rate, defined in the GWP_ASAN configuration (outside of SCUDO, via the SampleRate option) is by default 5000. Therefore, the probability that the allocation will skip SCUDO entirely and go to the GuardedPoolAllocator is 1/5000.

Memory Tagging (MTE)

ARM introduced the Memory Tagging Extension (MTE) in their ISA 8.5a. SCUDO implemented support for it early on; indeed it was one of the reasons for SCUDO's existence [23]. When SCUDO is committing new pages, it uses the new flag in mmap/mprotect (PROT_MTE). This flag gives the underlying memory the ability to store 4-bit tags for every 16 bytes of memory (granule is 16 bytes). Memory and pointers are tagged randomly during allocation and reallocation (there are helper functions to wrap the ARM instructions to ask for a random tag, implemented in standalone/memtag.h). ARM ISA supports the exclusion of certain tags in order to guarantee that the adjacent block tag is different from the new one requested. The tag received has to be applied to the unused byte of the 64-bit pointers to form the final pointer returned (the tagged pointer). Just using the random tag (without the exclusion set mask) would give a 7% chance for a tag collision (for 4-bit tags). All of the Chunk::Headers and LargeBlock::Headers have a different tag than the tag used for the remainder of the chunk, too. Chunks are also retagged on deallocation to catch UAFs.

The OriginOrWasZeroed (in the chunk header that was discussed before but the OrWasZeroed was not explained) exists because when a chunk is deallocated and MTE is enabled, the allocator gets for 'free' (no performance overhead) initializing the memory when it is retagged during the allocation (so when the chunk gets reused later on, if it needs to be zero-filled because it came from an origin like a calloc in which it has to be zeroed, they do not need to re-zero again).

RSS Limits

This is a simple feature, and it is not exactly a security mitigation, even though it could help against heap spraying in some niche cases. What it does is limit the amount of memory allowed to be allocated. As the **Table 1** covered, such a limit can be defined so that SCUDO starts failing allocations (soft_rss_limit_mb) or that SCUDO will abort the process once the limit is reached (hard_rss_limit_mb).

Conclusions

SCUDO is a simple memory allocator that has a very pragmatic approach to security mitigations. It is actively maintained, widely deployed (and thus relevant) and hopefully interested readers will help in keeping this (or similar posts) up-to-date with new capabilities,

configurations and choices. Feedback to this write-up is welcome, especially if any relevant SCUDO feature is missing, or if anything has changed or needs correction.

Acknowledgements

I would like to thank an anonymous researcher (whom I never had the pleasure to meet) for an excellent internal presentation on SCUDO - his methodical way of showing the data structures is reflected in many of the drawings. I would also like to thank Bastian Kersting for the collaboration on patches/work on SCUDO in the past. It goes without saying that all the contributors to SCUDO, especially its creator Kostya Kortchinsky, are greatly appreciated. Finally, to the reviewers Tracy Mosley (who also did the final editing), Dan Rosenberg, Jorge Villasmil and Jeremy Fetiveau for their patience and feedback.

References

[0] This article's repo. Link: https://github.com/rrbranco/ScudoAllocator
[1] Scudo Hardened Allocator. Link: https://llvm.org/docs/ScudoHardenedAllocator.html
[2] LLVM's Sanitizer CombinedAllocator. Link: https://github.com/llvm/llvm-project/blob/main/compiler-rt/lib/sanitizer_common/sanitizer_allocator_combined.h
[3] Android Announcement of SCUDO usage. Link: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html
[4] Fuchsia usage of SCUDO as the default sanitizer. Link: https://fuchsia.googlesource.com/fuchsia/+/refs/heads/master/zircon/third_party/ulib/musl/BUILD.gn#6
[5] LLVM Project Github. Link: https://github.com/llvm/llvm-project
[6] Introduction to Scudo Memory Allocator (translated from Chinese). Link: https://blog.csdn.net/feelabclihu/article/details/122264427
[7] Scudo's Internals. Link: https://un1fuzz.github.io/articles/scudo_internals.html
[8] Breaking Secure Checksums in the Scudo Allocator. Link: https://blog.infosectcbr.com.au/2020/04/breaking-secure-checksums-in-scudo_8.html
[9] Add Soft/Hard RSS Limits to Scudo Standalone. Link: https://reviews.llvm.org/D126752
[10] High Level Overview of Scudo. Link: http://expertmiami.blogspot.com/2019/05/high-level-overview-of-scudo.html
[11] What is the Scudo Hardened Allocator. Link: http://expertmiami.blogspot.com/2019/05/what-is-scudo-hardened-allocator_10.html
[12] Heap Vulnerability Classes. Link: https://openwall.info/wiki/_media/people/jvanegue/files/spw15_heap_models_vanegue_latest.pdf
[13] Heap Feng Shui. Link: https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf

[14] Heap Massaging. Link: http://actes.sstic.org/SSTIC07/Rump_sessions/SSTIC07-rump-Richarte-Heap_Massaging.pdf
[15] Heap Spraying. Link: https://en.wikipedia.org/wiki/Heap_spraying
[16] A Mathematical Modeling of Exploitations and Mitigation Techniques Using Set Theory. Link: http://spw18.langsec.org/papers/Kawakami-Exploit-modeling-using-set-theory.pdf
[17] Scudo Patch Non-fixed. Link: https://reviews.llvm.org/rL267094
[18] Scudo Patch Primary Randomization. Link: https://reviews.llvm.org/rL279793
[19] Scudo Patch Remove Callbacks. Link: https://reviews.llvm.org/D20742
[20] Advancing exploitation: a scriptless 0day exploit against Linux desktops. https://scarybeastsecurity.blogspot.com/2016/11/0day-exploit-advancing-exploitation.html
[21] Make -fsanitize=scudo use standalone. Migrate tests. Link: https://reviews.llvm.org/D102543
[22] Android Native Memory Allocator Verification. Link: https://android.googlesource.com/platform/bionic/+/HEAD/docs/native_allocator.md
[23] Memory Tagging and how it improves C/C++ memory safety. Link: https://arxiv.org/pdf/1802.09517.pdf