# Exploiting Android's Hardened Memory Allocator

Philipp Mao Elias Valentin Boschung Marcel Busch Mathias Payer
*EPFL, Lausanne, Switzerland*

## Abstract

Most memory corruptions occur on the heap. To harden userspace applications and prevent heap-based exploitation, Google has developed Scudo. Since Android 11, Scudo has replaced jemalloc as the default heap implementation for all native code on Android. Scudo mitigates exploitation attempts of common heap vulnerabilities.

We present an in-depth study of the security of Scudo on Android by analyzing Scudo's internals and systematizing Scudo's security measures. Based on these insights we construct two new exploitation techniques that ultimately trick Scudo into allocating a chunk at an attacker's chosen address. These techniques demonstrate — given adequate memory corruption primitives — that an attacker can leverage Scudo to gain arbitrary memory write. To showcase the practicality of our findings, we backport an n-day vulnerability to Android 14 and use it to exploit the Android system server.

Our exploitation techniques can be used to target any application using the Scudo allocator. While one of our techniques is fixed in newer Scudo versions, the second technique will stay applicable as it is based on how Scudo handles larger chunks.

## 1 Introduction

Most modern critical memory corruption vulnerabilities are heap related [36]. On Android, multiple publicly documented examples demonstrate the feasibility of exploiting a heap-based vulnerability to gain arbitrary code execution [14, 27, 30]. To protect userspace processes against heap vulnerabilities, Google has introduced the hardened Scudo allocator in Android 11 [49].

Since then, Scudo has become the default allocator for native userspace code in the Android Open Source Project. Unless explicitly modified by the vendor, all userspace processes, including apps and higher-privileged system services use Scudo.

Scudo is explicitly designed to increase the cost and complexity of heap-based exploits [3]. To protect itself from attacks, Scudo implements security measures to ensure the integrity of inline heap metadata and to prevent a predictable heap layout.

Exploitation techniques that target the allocator to escalate a heap-bound memory corruption vulnerability into an arbitrary memory write primitive or code execution have a long tradition. The security community has compiled a large compendium of such techniques for allocators like ptmalloc [5, 9, 29, 31, 43] (the glibc allocator) or jemalloc [6, 44] (Android's previous default allocator). Only by understanding these techniques can an analyst assess the criticality and exploitability of a heap-bound memory corruption vulnerability. However, Scudo has avoided scrutiny. So far, no comprehensive study on techniques targeting Scudo exists.

In this work, we explore the limitations of Scudo's protections. In particular, we find that Android's userspace architecture significantly weakens Scudo's security. All app processes and several system services are forked from a single process (the Zygote process) [21] and end up sharing the same address space layout and allocator state. The allocator state contains the secrets used to protect inline heap metadata and randomize allocation addresses. In a scenario where one Zygote-forked process attacks another such process, the allocator's secrets are shared between the attacker and the target process. This effectively bypasses any protection relying on the confidentiality of these secrets. Stripped of these security measures Scudo becomes a promising target for exploitation.

We present two exploitation techniques targeting Scudo. In the context of attacking Zygote-forked Android processes, our techniques only require a sufficiently powerful memory corruption primitive, which allows manipulating inline heap metadata, to gain arbitrary memory write. Furthermore, these techniques can be applied to any program utilizing the Scudo allocator. However, in a more generic attack scenario, an additional memory leak primitive is required.

To demonstrate that our findings apply to realistic scenarios, we backport a known vulnerability (CVE-2015-1528 [39]) to Android 14. The vulnerability is a heap under/overflow in Android's Binder deserialization. We show that in this scenario the heap underflow can be leveraged by a malicious app to achieve code execution in the system server, using our exploitation techniques.

In summary, we make the following contributions:

- Analysis and systematization of Scudo's security measures.
- Discovery of two exploitation techniques that target Scudo.
- Exploitation case study utilizing our techniques to achieve arbitrary code execution in the system server on Android 14.
- Discussion of possible mitigations, memory corruption primitives required to leverage our techniques, and Scudo's impact on Android userspace security.
- Development of a gdb plugin and python library to help in analyzing and exploiting Scudo.

We disclosed our findings to the Scudo maintainers. One of our techniques has been fixed in Android 14. While we proposed a further extension to Scudo, which mitigates our second exploitation technique, it was not merged due to performance concerns. Consequently, Scudo remains susceptible to our second technique. We will open-source our tooling for Scudo along with our exploits.

## 2 Scudo Security Measures

Scudo is a drop-in replacement for the glibc memory allocator, exposing the same API (e.g., `malloc`, `free`). As a security-hardened allocator, it implements four security measures: (i) isolation, (ii) randomization, (iii) protection, and (iv) separation. In this section, we present these security measures and discuss how they protect from exploiting heap-based vulnerabilities.

To give concrete examples of the security measures impact, we use the example program in Listing 1, which uses the Scudo allocator. The program reads the attacker's input into the `tmp` buffer (Line 8) in a loop, allocates a `0x18` sized chunk (Line 11), copies data from `tmp` into the chunk (Line 13), and then frees a specific chunk based on the value of `status` (Lines 14 - 16). The attacker controls the values of the `status` and `size` variables. This results in two security vulnerabilities, a heap-based overflow on Line 13 and a double free on Lines 14 and 15.

**Isolation.** Based on the requested allocation size, chunks are either handled as *primary* or *secondary* chunks. Primary chunks are placed into dedicated heap memory regions, while secondary chunks are allocated separately in their own memory region. To handle primary chunks, Scudo maps multiple

```
1  int main(){
2    char tmp[0x100];
3    void* class_0_secondary_chunk = malloc(0x20000);
4    void* class_1_chunk = malloc(0x8);
5    void* class_2_chunk = malloc(0x18);
6    printf("victim:%p\n", class_2_chunk);
7    while(1){
8        read(0, tmp, 0x100);
9        int status = *(int*)tmp;
10       int size = *(int*)(tmp+sizeof(int));
11       char* chunk = (char*)malloc(0x18);
12       printf("address:%p\n", chunk);
13       memcpy(chunk, tmp+sizeof(int)*2, size);
14       if(status & 0x2){free(chunk);}
15       if(status & 0x4){free(chunk);break;}
16       if(status & 0x8){free(class_2_chunk);}
17   }
18 }
```

Listing 1: A vulnerable example program with a heap buffer overflow and a double free (both values of the `size` and `status` variable are under the attacker's control). The attacker's input is read from standard input.

memory regions. Each of these regions is assigned a size range, and chunks sized within the specific range will be allocated from the corresponding region. Zero permission guard pages are used to separate these regions. In Scudo, these size ranges are referred to as *class IDs*. For primary chunks the class ID depends on the size, see Table 5 in the Appendix for a mapping between classes and size ranges. For example, a chunk of size `0x18` is assigned the class ID `2`. The class ID-specific regions only hold chunks. Allocator-internal metadata such as lists of freed chunks or the information on memory regions are stored in a separate region protected by guard pages, and libc's writable section.

In Lines 3-5 of Listing 1, chunks of different sizes are allocated, resulting in a heap memory layout shown in Listing 2. Thus, the heap buffer overflow in Listing 1 can only overwrite memory inside the `[Class 2 region]` memory region and cannot directly overwrite chunks of other size classes or allocator-internal metadata.

**Security Measure *Isolate*:**
Chunks are allocated in dedicated isolated memory regions. Furthermore, allocator-internal metadata is stored in separate memory regions from chunks.

**Randomization.** The chunks inside a region are allocated at random offsets. When a region is first mapped, several addresses where a chunk may be allocated are placed into a so-called *TransferBatch*. The order in which these addresses are returned from the TransferBatch is randomized. This randomization is achieved by shuffling the addresses in the TransferBatch using a seed stored in the allocator. This randomiza-

```
size       permission
...
0x00001000 --- [Secondary guard]
0x01001000 rw- [Class 0 secondary chunk]
0x00001000 --- [Secondary guard]
...
0xa0006000 --- [Guard and reserve]
0x00040000 rw- [Primary chunk free lists]
0x2ffbf000 --- [Guard and reserve]
0x00040000 rw- [Class 1 region]
0x2ffc0000 --- [Guard and reserve]
0x00040000 rw- [Class 2 region]
0x0ffcb000 --- [Guard and reserve]
...
0x00044000 r-- libc.so
0x00094000 r-x libc.so
0x00004000 r-- libc.so
0x00002000 rw- libc.so
0x00452000 rw- [Allocator metadata]
...
```

Listing 2: An example memory map of Scudo relevant regions. Marked in blue are regions where Scudo stores free lists and other allocator-internal metadata. Marked in orange are regions where chunks are stored. In this example, a single secondary chunk was allocated, and at least one chunk of class IDs 1 and 2 were allocated. Scudo memory regions containing chunks are surrounded by 0 permission guard pages.

tion ensures that addresses of consecutively allocated chunks cannot be predicted, effectively removing the foundation for any heap feng shui attempts. Figure 1 shows the output when running the program in Listing 1 twice for five loop iterations with the same input. As can be seen, the addresses of chunks allocated after one another are not consecutive and also differ between program executions.

**Security Measure *Randomize*:**
Addresses of consecutive allocations are randomized.

**Protection.** When allocating a chunk, Scudo places a chunk header at address `returned pointer-0x10`. The chunk header is shown in Table 1. The relevant fields are the `ClassId`, `State`, and `Checksum`. The `ClassId` stores the chunk's class ID. The `State` field tracks if the chunk is currently in use or has been freed. To protect this header, Scudo stores a truncated CRC32 checksum of the header fields in the `Checksum` field. The checksum is computed using the chunk's address, the header, and a 32-bit cookie value, which is randomly generated when the program starts. Listing 3 shows how the checksum is computed. Any time Scudo interacts

```
> ./example < input
victim: 0x7fd4f720f650
address:0x7fd4f720e510
address:0x7fd4f720f750
address:0x7fd4f720f190
address:0x7fd4f720e4d0
address:0x7fd4f720efd0
```

```
> ./example < input
victim: 0x7fd4f7208b50
address:0x7fd4f7208390
address:0x7fd4f7209250
address:0x7fd4f7208990
address:0x7fd4f7209bd0
address:0x7fd4f7209190
```

Figure 1: The output of running the example program in Listing 1 two times to show Scudo randomizing allocation addresses. Note that for this example ASLR was disabled to show the chunk offsets in the same memory region changing between runs.

| # bits | Field |
|---|---|
| 8 | ClassId |
| 2 | State |
| 2 | OriginOrWasZeroed |
| 20 | SizeOrUnusedBytes |
| 16 | Offset |
| 16 | Checksum |

Table 1: The fields and corresponding sizes in the Scudo chunk header. The `OriginOrWasZeroed` field indicates the origin of the chunk, e.g., `malloc` or `new`. The `SizeOrUnusedBytes` field indicates the exact chunk size. `Offset` is filled with zeros.

with a chunk, it recomputes the checksum and compares it with the `Checksum` field to ensure the integrity of the chunk header. In the example program in Listing 1, if the attacker blindly overwrites the chunk header of the `class_2_chunk` with the heap overflow, Scudo will abort when freeing the chunk (Line 16) as the checksum will not match the header contents. By ensuring that the `State` field has the expected value, i.e., the chunk currently being freed is not already free, Scudo prevents double-free attacks. In Lising 1, if an attacker sends a payload that results in a `status` of `0x6`, this triggers a double free (Lines 14-15). However, Scudo immediately aborts on Line 15 as it detects the double free using the `State` field.

**Security Measure *Protect*:**
The chunk header, stored inline on the heap, is protected by a checksum.

**Separation.** Secondary chunks have the same chunk header as primary chunks, with the `ClassId` field set to `0`. Additionally, secondary chunks have an extended header beginning at `returned pointer - 0x40`, see Table 2 for an overview of the fields. This extended *secondary chunk header* stores pointers to a linked list in `next` and `prev` of allocated sec-

```
short checksum(long address, long header, int cookie){
  int intermediate = CRC32(cookie, address);
  intermediate = CRC32(intermediate, header);
  return = (short) (intermediate & (intermediate >> 16)) & 0xffff;
}
```

Listing 3: Pseudocode of how Scudo computes a chunk's checksum. `Address` points to the chunk, `header` is the chunk header without the checksum and `cookie` is a secret, set when the allocator is initialized.

| # bytes | Field | Checksum |
|---|---|---|
| 0x8 | Prev | ✗ |
| 0x8 | Next | ✗ |
| 0x8 | CommitBase | ✗ |
| 0x8 | CommitSize | ✗ |
| 0x8 | MapBase | ✗ |
| 0x8 | MapSize | ✗ |
| 0x8 | Scudo Chunk header | ✓ |

Table 2: The fields and corresponding sizes in the secondary chunk header for 64-bit programs. Only the Scudo chunk header is protected by a checksum.

ondary chunks. It also stores the mapping's base address and size, with and without the guard pages respectively in `MapBase`, `MapSize`, `CommitBase`, and `CommitSize`. Importantly, the secondary chunk header is not protected by a checksum. Instead, Scudo relies on the fact that only one chunk is stored in the mapping and that the mapping is surrounded by guard pages to protect the extended header. The chunk header and secondary chunk header are the only instances of Scudo storing metadata inline on the heap. In Listing 1, the chunk `class_0_secondary_chunk` is allocated on Line 3 and its secondary chunk header is stored at `class_0_secondary_chunk-0x40`. In Listing 2 this chunk resides in the `[Class 0 secondary chunk]` memory region.

**Security Measure *Separate*:**
Pointers stored inline are placed in separate mappings and protected by guard pages.

## 3 Threat Model

In our threat model, a malicious attacker-controlled Android app aims to escalate privileges by attacking another app or system service on the same device. The device is running an Android version using Scudo.

The attacker's goal is to gain code execution in the target process by corrupting the target's memory. Due to Android's separation of userspace processes, the attacker cannot directly manipulate the target's memory. Instead, the attacker relies on the target's exposed functionality to interact with the target's memory. Furthermore, the target contains memory corruption vulnerabilities triggerable by the attacker. There are many examples of such memory corruption vulnerabilities in apps (CVE-2019-11932 and CVE-2021-24041 [14, 42]), or system services (CVE-2015-1528, Stagefright, CVE-2020-0026, CVE-2019-2136, and CVE-2022-39907 [10, 25, 39–41]). See Figure 2 for a concrete example of our threat model.

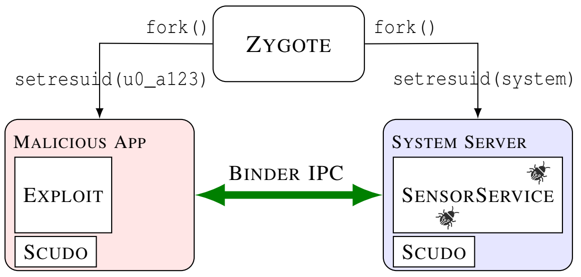


Figure 2: The malicious app is attacking the vulnerable SensorService running in the system server. The app uses Binder IPC to communicate with the SensorService. Both the app and system server are forked from Zygote.

The attacker plans to use these vulnerabilities to bypass Scudo's four security measures and leverage Scudo to achieve an arbitrary memory write for subsequent code execution. In the following sections, we discuss how in the context of our threat model each of Scudo's security measures is broken.

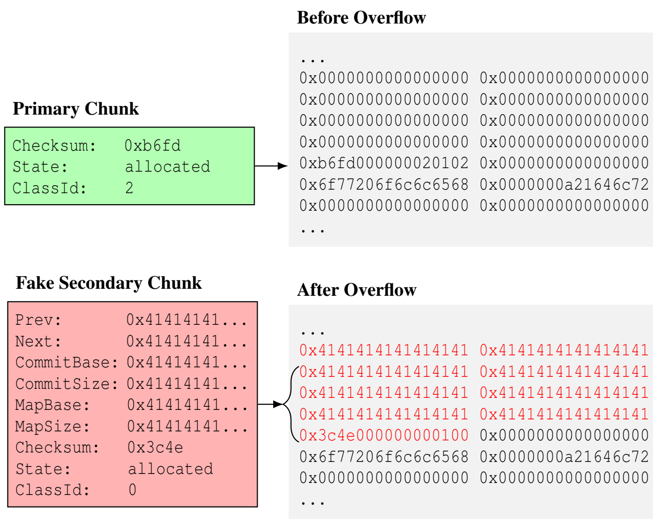
## 4 Compromising Protect and Randomize

Both security measures *Randomize* and *Protect* rely on the confidentiality of Scudo metadata and ASLR (Address Space Layout Randomization). Concretely, if an attacker can leak the contents of the TransferBatch or the seed used to shuffle the TransferBatch, the security measure *Randomize* is compromised. With the leaked information, the attacker can pinpoint exactly where Scudo will allocate future chunks. To compromise the security measure *Protect*, the attacker needs to first compromise the security measure *Randomize* or leak the address of the target chunk in another way. Additionally, the attacker also needs to obtain the cookie to calculate the checksum correctly.

In our threat model both security measures *Randomize* and *Protect* are immediately compromised. On Android, all apps as well as several system services are forked from the same Zygote process. This reduces the startup time and reduces memory consumption by sharing RAM pages used for framework code and resources [21] but comes at a devastating cost to security [32]. As a consequence, most Android userspace processes share the same ASLR layout, including Scudo regions. Furthermore, the Zygote process allocates several chunks initializing the Scudo allocator i.e., setting the cookie and TransferBatch randomization seed. After forking all of this allocator state is preserved. A malicious app can predict exactly where chunks of other Zygote-forked processes will be allocated by using its allocator as an oracle, which breaks security measure *Randomize*. To break security measure *Protect*, the malicious app can simply read out its Scudo cookie and forge valid checksums for any chunk header.

Security measures *Randomize* and *Protect* are compromised in an attack scenario in which a malicious Android app is attacking another Zygote-forked process.

Going forward, we assume that the attacker has compromised the security measures *Randomize* and *Protect*.



```
Checksum:   0xb6fd
State:      allocated
ClassId:    2
```

```
...
0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000
0xb6fd000000020102 0x0000000000000000
0x6f77206f6c6c6568 0x0000000a21646c72
0x0000000000000000 0x0000000000000000
...
```

```
Prev:       0x41414141...
Next:       0x41414141...
CommitBase: 0x41414141...
CommitSize: 0x41414141...
MapBase:    0x41414141...
MapSize:    0x41414141...
Checksum:   0x3c4e
State:      allocated
ClassId:    0
```

```
...
0x4141414141414141 0x4141414141414141
0x4141414141414141 0x4141414141414141
0x4141414141414141 0x4141414141414141
0x4141414141414141 0x4141414141414141
0x3c4e000000000100 0x0000000000000000
0x6f77206f6c6c6568 0x0000000a21646c72
0x0000000000000000 0x0000000000000000
...
```

Figure 3: A heap buffer overflow, which overwrites a chunk header and changes a primary chunk's `ClassId` to `0`, defeats security measure *Separate*. Note that the figure shows lower addresses at the top, growing downwards.

## 5 Arbitrary Write

The holy grail of heap exploitation is to coerce the allocator into allocating a chunk at an attacker's chosen address. This can lead to code execution for example by allocating a chunk on the stack and writing a ROP (Return Oriented Programming) chain to the chunk. Since the security measures *Randomize* and *Protect* are bypassed, only the measures *Isolate* and *Separate* stand in the way of an arbitrary write. In classical ptmalloc heap exploitation, an arbitrary write is usually achieved by manipulating inline pointers. However, as shown in Section 2 the security measure *Separate* separates inline pointers from the rest of the heap. The only instance of Scudo storing pointers inline is in the secondary chunk header, which is stored in memory regions separated from the rest of the heap.

To go further we assume the attacker to have access to a memory corruption primitive which allows manipulating the header of a victim chunk. This memory corruption primitive is limited to Scudo's primary heap regions. An example of such a primitive is a heap buffer overflow that overwrites the chunk header of a subsequent victim chunk. After bypassing the security measure *Protect*, the attacker can freely set any fields of the overflown chunk header (Table 1) and calculate a valid checksum. When the victim chunk is freed, the checksum verification will succeed and Scudo will parse attacker-controlled metadata. An example of such a primitive is shown in Listing 1. The vulnerable program allows an attacker to overwrite the chunk header of `class_2_chunk` using the heap buffer overflow. Since the attacker has broken the *Randomize* security measure, the attacker can keep allocating chunks in a loop until the overflowing chunk is just below the `class_2_chunk`. Subsequently the `class_2_chunk` can be freed (Line 16) by setting `status` to `0x8`.

Both, manipulating the `State` and `ClassId` fields, are interesting for the attacker. By manipulating the `State` field, a double free can be turned into a UAF (Use After Free). In this scenario, the chunk is first freed, the attacker then overwrites the header and changes the `State` field from `freed` back to `allocated`. The chunk is then freed again and ends up in the free list twice, setting up the UAF. However, the UAF is only interesting if attacking the application's data is in scope since the UAF does not give access to any Scudo metadata. As opposed to ptmalloc where a UAF may allow overwriting pointers to other free chunks.

More interesting for the attacker is manipulating the `ClassId` field. By changing the `ClassId` of a primary chunk to `0`, the class ID of secondary chunks, the attacker effectively places the secondary chunk header inline on the heap rendering the security measure *Separate* ineffective. In the scenario of an overflow, the secondary chunk header is fully under the attacker's control. Figure 3 illustrates this phenomenon. Before the overflow, the victim chunk is simply a primary chunk with `ClassId 2`. After the overflow, the victim chunk is replaced by a fake secondary chunk with a bogus secondary chunk header. Freeing the overflown victim chunk causes a segfault as Scudo attempts to read the linked list entry at `0x4141414141414141`.

To compromise security measure *Separate* an attacker needs access to a memory corruption primitive which allows the creation of faked secondary chunks.

In the following sections, we present two exploitation techniques, Forged CommitBase and Safe Unlink. Both techniques manipulate this newly created and inlined secondary chunk header in different ways to achieve an arbitrary memory write, thus breaking security measure *Isolate*.

## 5.1 Forged CommitBase

The Forged CommitBase technique manipulates the `CommitBase` header field of the inlined secondary header to achieve an arbitrary memory write.

The `CommitBase` field of the secondary chunk header stores a pointer to the start of the secondary chunk (including the secondary chunk header). After freeing the secondary chunk, the `CommitBase` is stored in the free list of secondary chunks. When this secondary chunk is used to serve an allocation request, Scudo uses the `CommitBase` stored in the free list to determine where this chunk is located.

By cleverly setting the `CommitBase` of the faked secondary header to the desired target address, the attacker can bring Scudo to allocate a secondary chunk at the desired address, breaking security measure *Isolate*.

Figure 4 shows the sequence of events taking place in this exploit and the relevant fields of the faked secondary chunk. At ① the attacker overwrites the primary chunk's header and places the fake secondary chunk on the heap, using a memory corruption primitive (for example an overflow). The `CommitBase` is set to the target address (`0x7fffffffd840`). As discussed previously, `Prev` and `Next` are pointers to entries in a linked list. For the free to succeed, these pointers need to be valid. Fortunately, Scudo checks if the pointers are null. If they are, the unlinking step is skipped. At ② the fake secondary chunk is freed, and the `CommitBase` address is placed into the secondary chunk free list. At ③ a secondary chunk is requested, which Scudo serves from the secondary free list. Since Scudo uses the address stored in the free list, the newly allocated chunk is located on the stack (at `0x7fffffffd840`). Note that the attacker is free to choose any `CommitBase` address.

For this exploit to succeed, at least one secondary chunk needs to be allocated at the time of freeing. Otherwise, the counter of secondary chunks in use is flipped to -1 and Scudo will crash on the next secondary chunk allocation.

> Security measure *Isolate* can be bypassed by manipulating the `CommitBase` field of a secondary chunk header.

In the next section we present an alternative technique, which achieves an arbitrary write by manipulating different secondary chunk header fields.

## 5.2 Safe Unlink

In contrast to the previous technique, our second technique (Safe Unlink) leverages the unlinking taking place when a secondary chunk is freed to obtain an arbitrary memory write.

In glibc heap exploitation, the "unsafe unlink" attack [45] for newer libc versions exploits a linked list unlink to achieve arbitrary memory write. This technique is almost directly applicable to the unlinking taking place when the secondary chunk is freed. Just like newer glibc versions, Scudo diligently checks the integrity of the linked list, see Listing 8 in the Appendix. To leverage this safe unlink, the attacker needs to create a fake linked list. While the glibc exploitation technique relies on an application-specific pointer, for Scudo, we will leverage allocator metadata to fake a linked list with two entries. One entry is the secondary chunk header, the other entry is placed inside the `PerClass` structure. The `PerClass` structure is a free list storing pointers to free chunks of a specific class ID. Listing 4 shows the structure of the `PerClass` free list. It holds the number of chunks, the maximum number of chunks, and a list of pointers to free chunks.

By cleverly forging primary chunks overlapping the fake secondary header and freeing these chunks, the attacker can place pointers to the fake secondary header into the `PerClass` structure. Figure 5 shows the attacker-created fake linked list before and after unlinking. After unlinking, an address pointing to the free list will be inserted into the free list itself.

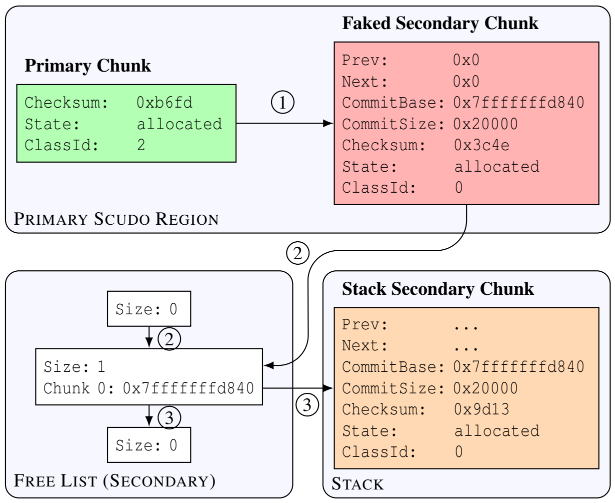


Figure 4: The attacker overwrites a primary chunk's header and modifies the `CommitBase`. After freeing the chunk, a stack address is placed into the secondary chunk free list. Allocating from the secondary chunk free list then allocates a chunk on the stack. (`0x7fffffffd840` is a stack address in this example.)

```
struct PerClass {
  short Count;
  short MaxCount;
  void* Chunks[MaxCount];
};
```

Listing 4: The `PerClass` free list, which stores class-specific free chunks. `Count` tracks the number of entries in the list. `MaxCount` is the maximum number of chunks that may be stored in the list. `Chunks` is an array of pointers, pointing to the address of the free chunks headers.

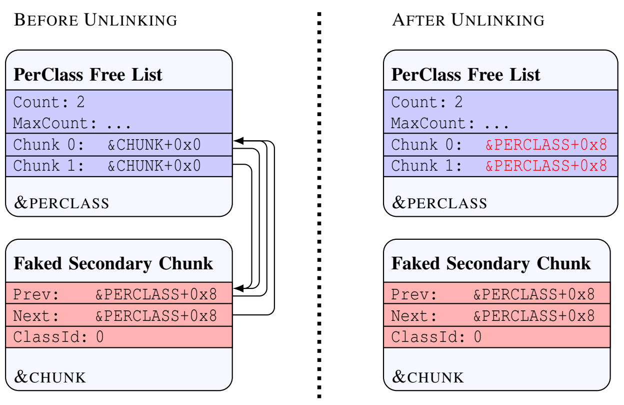


Figure 5: The attacker created linked list with two entries. The linked list is created by freeing fake chunks that overlap the secondary chunk header and adjusting the `Next` and `Prev` pointers to point into the `PerClass` free list. The attacker knows the address of the `PerClass` free list due to mitigations *Randomize* and *Protect* being broken.

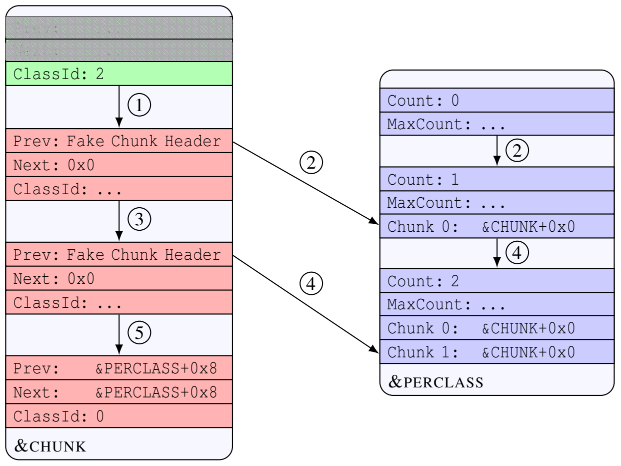


Figure 6: The steps to forge a fake linked list between the secondary chunk header and the `PerClass` free list. `&CHUNK` is the address of the fake secondary chunk header. `&PERCLASS` is the address of the `PerClass` free list.

Allocating from this `PerClass` free list returns a chunk overlapping the free list. The attacker thus gains control over the free list and can control the addresses of future allocations.

In order to create the fake linked list, the attacker needs more powerful memory corruption primitives. Besides being able to forge a secondary chunk, the attacker is also able to trigger two frees at address `fake secondary chunk header +0x10`. An example of such a primitive is a controlled free in which the attacker can corrupt a pointer and then have that pointer freed. Furthermore, the attacker can trigger the memory corruption primitive multiple times, i.e., overwriting the fake secondary header three times.

Figure 6 shows the steps needed to set up the fake linked list. At ① the attacker writes a primary chunk header with a chosen `ClassId` and `allocated` state to the address where the fake secondary header starts. This fake primary chunk header overlaps with the `Next` field of the fake secondary header. The fake primary chunk is then freed at ②. Effectively, the address of the fake secondary's `CommitBase` entry (`fake secondary chunk header + 0x10`) is passed to free. Consequently, the address of the fake secondary chunk header is placed into the `PerClass` structure for the chosen `ClassId`. (Note that Scudo tracks chunks in the `PerClass` free list by the address of the chunk's header.) The attacker then repeats the previous steps (③ and ④). Now the address of the fake secondary chunk header is twice at consecutive positions in the `PerClass` free list. Finally at ⑤, the attacker sets up the fake secondary chunk header to complete the fake linked list. Both `Next` and `Prev` are modified to point to the first instance of the fake secondary chunk address in the `PerClass` structure. With this the fake linked list, as seen in Figure 5, has been set up. The attacker knows the location of the `PerClass` structure because of measures *Randomize* and *Protect* being broken. The `CommitBase`, `CommitSize`, `MapBase`, and `MapBase` fields of the chunk header are not relevant to this exploit. Only the chunk header needs to be overwritten to have `ClassId 0`.

After the secondary chunk is freed, the attacker can allocate a chunk overlapping the `PerClass` structure, effectively allowing the attacker to insert any addresses into the free list, breaking the *Isolate* mitigation.

Security measure *Isolate* can be bypassed by manipulating the `Prev` and `Next` fields of a secondary chunk header along with cleverly freeing fake chunks into a `PerClass` free list.

## 6 Exploitation Case Study

We demonstrate our findings by reintroducing an n-day vulnerability and exploiting the system server on an Android Virtual Device running Android 14 using our techniques.

The Android system server is the first process forked from the Zygote process. It starts all system services, either starting the service in a separate process or starting a new thread running the service inside the system server. The system server is an interesting target for escalating privileges from an app. Firstly, each service running inside the system server is exposed over Binder IPC to normal apps. Binder is the Android-specific IPC mechanism, which facilitates communication between Android apps and Android services. In total, the system server exposes around 42 services [18]. Secondly, the system server runs as the high-privileged system user, just slightly less powerful than root. Third, the system server restarts after crashing giving the attacker multiple exploitation attempts. Finally, the system server is forked from Zygote, and thus Scudo's security measures *Randomize* and *Protect* are ineffective in our attack scenario as described in Section 4.

To provide the attacker app with a memory corruption primitive to defeat security measures *Isolate* and *Separate*, we backport CVE-2015-1528 [39] to Android 14. CVE-2015-1528 is a heap underflow or overflow in the Binder data deserialization due to missing sanity checks. Listing 5 shows the relevant code and the code changes reintroducing the vulnerability. The `native_handle_create` function allocates the `native_handle` object whose size depends on the `numFds` and `numInts` arguments. Both of these arguments are read from the attacker-controlled Binder data. Since the sanity check on the arguments is removed, an attacker can trigger a heap underflow by setting `numFds` to a negative number, which will cause the first argument of `read` in `readNativeHandle` to point behind the allocated chunk. Likewise, by setting `numInts` to a negative number, a heap overflow is triggered in the loop which reads file descriptors from the Binder data. The change in the loop removes an early exit if reading the file descriptor from the Binder data fails.

At Black Hat USA 2015, Gong [27] used this vulnerability to exploit the system server. In the exploit, Gong coerced jemalloc to allocate a chunk on the stack. Almost ten years and one secure allocator later we will show how the same vulnerability remains exploitable in Scudo.

```
—/libcutils/native_handle.c

native_handle_t* native_handle_create(int numFds, int numInts)
{
-    if (numFds < 0 || numInts < 0 || numFds > kMaxNativeFds
-      || numInts > kMaxNativeInts){return NULL;}
     native_handle_t* h = malloc(
       sizeof(native_handle_t) + sizeof(int)*(numFds+numInts));
     if (h) {
       h−>version = sizeof(native_handle_t);
       h−>numFds = numFds;
       h−>numInts = numInts;
     }
     return h;
}

—/libs/binder/Parcel.cpp

native_handle* Parcel::readNativeHandle() const
{
     int numFds, numInts; status_t err;
     err = readInt32(&numFds);
     if (err = NO_ERROR) return 0;
     err = readInt32(&numInts);
     if (err = NO_ERROR) return 0;

     native_handle* h = native_handle_create(numFds, numInts);
     //may lead to a buffer overflow if numInts is negative
     for (int i=0 ; err==NO_ERROR && i<numFds ; i++) {
          h−>data[i] = dup(readFileDescriptor());
-         if (h->data[i] < 0) {
-         for (int j = 0; j < i; j++) {
-         close(h->data[j])}
-         native_handle_delete(h);
-         return nullptr;}
+         if (h->data[i] < 0) err = BAD_VALUE;
     }
     //may lead to a buffer underflow if numFds is negative
     err = read(h−>data + numFds, sizeof(int)*numInts);
     if (err = NO_ERROR) {
          native_handle_close(h);
          native_handle_delete(h);
          h = 0;
     }
     return h;
}
```

Listing 5: The code changes to reintroduce CVE-2015-1528.

## 6.1 SensorService

Unlike Gong, who targeted the WindowsManagerService in the system server, we will target the SensorService. Listing 6 shows the relevant code in the SensorService's `onTransact` function. The `onTransact` function is the service's callback to handle incoming Binder requests. Both the `data` and `code` argument to the function are fully under the attacker's control. By setting the code of the Binder request to `CREATE_SENSOR_DIRECT_CONNECTION`, the attacker can trigger the vulnerable `readNativeHandle` function. After the vulnerable function, the descriptors in the newly created `native_handle` object are tagged with fdsan. Fdsan is a file descriptor sanitizer implemented to detect use-after-close or double-closes [4]. Since Android 11, fdsan aborts the process if an issue is discovered. Concretely, if we pass `numFds` greater than zero, we need to ensure no duplicate integers are present as otherwise fdsan will abort in `native_handle_close_with_tag`. The `createSensorDirectConnection` function contains the actual implementation to handle the binder request, we can exit early from this function by setting the `format` variable, read from the Binder data to an invalid value. Finally, the allocated `native_handle` object is freed.

In summary, we can get a chunk of any size allocated, trigger a controlled heap underflow or overflow originating from that chunk, and have the chunk freed right afterward. Finally, all of these primitives are accessible via Binder by an unprivileged app.

## 6.2 Exploitation Over Binder

To gain code execution in the system server context, our malicious app sends two Binder requests to the SensorService. The first Binder request leverages the heap underflow to place a stack address into the secondary chunk-free list as described in Section 5.1. The second Binder request allocates this secondary chunk and writes a ROP chain to the stack.

**Forging a secondary chunk** Table 3 shows the data sent in the first Binder request. The first five fields

```
status_t BnSensorServer::onTransact(uint32_t code,
      const Parcel& data, Parcel* reply, uint32_t flags)
{
  switch(code) {
    case CREATE_SENSOR_DIRECT_CONNECTION: {
      CHECK_INTERFACE(ISensorServer, data, reply);
      String16& opPackageName = data.readString16();
      const int deviceId = data.readInt32();
      uint32_t size = data.readUint32();
      int32_t type = data.readInt32();
      int32_t format = data.readInt32();
      native_handle_t *resource = data.readNativeHandle();
      if (resource == nullptr) {
        return BAD_VALUE;
      }
      native_handle_set_fdsan_tag(resource);
      sp<ISensorEventConnection> ch =
          createSensorDirectConnection(...);
      native_handle_close_with_tag(resource);
      native_handle_delete(resource);
      reply->writeStrongBinder(IInterface::asBinder(ch));
      return NO_ERROR;
    }
  ...
}
```

Listing 6: The relevant parts of the SensorService's `onTransact` function [17]. Marked in red is the function call that will trigger the heap memory corruption.

read by the system server from the Binder request are not relevant to our exploit and only serve to make the `createSensorDirectConnection` function exit early (`opPackageName` until `format`). `NrFds` is set to `-5-9*2` (`-23`). `-5` moves the underflow start just before the allocated chunk header and `-9*2` moves the underflow start to the beginning of the faked secondary chunk header. The remaining data is then written to the heap in `readNativeHandle` (Occurs in the `read` function, which reads `sizeof(int)*numInts` to the heap). The remaining data contains the secondary chunk header (`Prev` until `MapSize`), the chunk header to overwrite the original header (`FakeHeader` and `zeros`) and filler data (`filler`) such that the chunk is allocated from a specific primary chunk class.

Both `Next` and `Prev` are set to zero to avoid unlinking. The `CommitBase` is set to the target stack address and the `CommitSize` is set to the size of a secondary chunk, we use `0x20000`. To correctly craft the overwritten chunk header (`FakeHeader`) with `ClassId 0`, we need both the address of the chunk and the cookie. The cookie can be directly read from the memory of our own app. Although we know exactly at which addresses Scudo will allocate chunks, using the allocator of our own app as an oracle, predicting the `native_handle`'s allocated address is complicated by the non-determinism of the system server (at least 40 threads each handling Binder requests). We found that allocating a primary chunk of `ClassId 32` (the largest class for primary

| Type | Value | Name |
|---|---|---|
| String | "wow" | opPackageName |
| int | 20 | deviceId |
| int | 20 | size |
| int | 20 | type |
| int | 20 | format |
| int | -23 | nrFds |
| int | 0x3f56 | numInts |
| long | 0x0 | Prev |
| long | 0x0 | Next |
| long | 0x7ffdd0b564a8 | CommitBase |
| long | 0x20000 | CommitSize |
| long | 0x7ffdd0b564a8 | MapBase |
| long | 0x20000 | MapSize |
| long | 0x507a000000008100 | FakeHeader |
| long | 0x0 | zeros |
| char* | 0xfd00 * "A" | filler |

Table 3: Example of the first Binder request sent to exploit the system server. `0x7ffdd0b564a8` is the target stack address.

chunks) allowed us to correctly predict the chunk's address around one out of ten times. Both `numInts` and `filler` serve to set the size of the allocated `native_handle` object.

The `CommitBase` in our fake secondary chunk header points to the main thread's stack. The main thread in the system server runs in an infinite loop polling for messages. When writing to the stack, we will overwrite the stored return address of the `android:Looper:pollOnce` function. Our target stack address is around `0x20000` (our `CommitSize`) below where the return address is stored. Note that we do not directly point our chunk at the return address. Doing so would cause `ReadNativeHandle` to inadvertently write over the stack's maximum address causing a segfault.

For the first Binder request, we do not need to worry about fdsan because `nrFds` will be negative and the loop which reads file descriptors from the Binder data in `readNativeHandle` iterates zero times.

After receiving this Binder request, the `BnSensorServer::onTransact` function is called. Then, the `readNativeHanlde` function is called, which in turn calls `native_handle_create`. Our `native_handle` chunk is allocated, and the heap underflow is triggered, replacing the original primary chunk with our fake secondary chunk. Right afterward, this chunk is freed and the target stack address is placed into the secondary free list. The details of creating the fake secondary chunk and placing the stack address into the secondary chunk free list can be found in Section 5.1.

**Writing to the stack** Table 4 shows the second Binder request sent to the system server. `numInts` is set to the size of the ROP chain divided by four. `nrFds` is set to

| Type | Value | Name |
|---|---|---|
| String | "wow" | opPackageName |
| int | 20 | deviceId |
| int | 20 | size |
| int | 20 | type |
| int | 20 | format |
| int | 0x7fe2 | nrFds |
| int | 30 | numInts |
| long[15] | ... | ROPChain |

Table 4: Example of the second Binder request sent to exploit the system server.

the difference between the allocation size `0x20000` and the size of the ROP chain divided by four. When allocating memory in `native_handle_create` for this `0x20000`-sized `native_handle` object, the chunk is allocated from the secondary free list and placed on the main thread's stack. In `readNativeHandle`, the ROP chain (`ROPChain`) is written to the stack starting at the stored return address of the `android:Looper:pollOnce` function. After the ROP chain has been written to the stack, there is a race between the main thread returning from the `android:Looper:pollOnce` function and the SensorService calling `native_handle_close_with_tag`. If the SensorService wins, the process is aborted by fdsan, due to duplicate integers being passed as file descriptors to fdsan. Instead, if the main thread wins, the first gadget of the ROP chain clobbers the `nrFds` field of the stack-allocated `native_handle`, setting it to a negative number and thus avoiding fdsan attempting to close any file descriptors. After clobbering `nrFds`, the ROP chain simply prints to logcat. Figure 7 shows the logcat output after successful exploitation. The exploit succeeds after around ten attempts.

## 7 Discussion

In this section, we discuss our presented exploitation techniques focusing on mitigations, generalization, and trade-offs.

**Mitigations** Independent from our research, the safe unlink exploitation technique as described in Section 5.2 has been fixed in Android 14. The fix changes the `PerClass` free list to store offsets, relative to the primary chunk heap region, instead of pointers. This makes building the fake linked list impossible by freeing chunks. Note that it is still possible to try and construct a safe unlink exploit by targeting application-specific objects to build the linked list. However, we did not find a suitable target inside of Scudo that could be used to build the fake linked list after this fix.

To prevent attackers from creating fake inlined secondary chunks, as described in Section 5, we propose an extension to Scudo which tracks allocated secondary chunks in an isolated memory region. Any time a secondary chunk is freed, our mitigation would check that a secondary chunk was allocated before at the address to be freed. We opened a pull request on the LLVM repository (where the Scudo source is hosted) to add the proposed mitigation to Scudo [11]. However, the request was not merged due to performance concerns. Without fundamentally changing how secondary chunks are handled or accepting the performance penalty, Scudo will remain vulnerable to these types of attacks.

**Generalizing our Exploitation Techniques** Our exploitation techniques presented in Section 5 apply to any program using Scudo. Both our techniques require the attacker to know the exact address of the victim chunk, whose class ID will be changed.

In Section 4, we show how the typical threat model on Android renders the security measures *Randomize* and *Protect* ineffective. With *Randomize* broken the attacker knows the addresses and order of allocations for all chunks. However, this may not be enough for a sufficiently complex program to exactly predict the address of the victim chunk. Chunks may be freed and reallocated in a non-deterministic matter. For example for our case study in Section 6, we can only predict the address of our victim chunk one out of ten times. Thus, to apply our techniques to complex targets, either the target program restarts after crashing or the attacker has another way to probe the heap state, such as a program-specific memory leak vulnerability.

In a generic scenario, the attacker needs a powerful memory leak primitive akin to an arbitrary read to break both security measures *Randomize* and *Protect*. To break *Randomize*, the attacker needs to use this primitive to read the seed used to shuffle the TransferBatch or directly leak chunk addresses. To break *Protect*, the attacker needs to either read the cookie directly or leak both a chunk's header and its address. Using this header and address, the attacker can brute force a valid cookie using the code in Listing 3. The brute force attack is viable since the cookie is only 16 bits long. After obtaining a valid cookie, the attacker can forge valid Scudo chunk headers.

**Required Memory Corruption Primitives** Both our presented exploits (Forged CommitBase and Safe Unlink) require a memory corruption primitive that allows forging a fake secondary chunk. The fake secondary chunk can be forged either by overwriting an existing chunk's header or by passing a controlled pointer to free, pointing to a fake secondary chunk.

Both a heap underflow and overflow are examples of primitives that can overwrite an existing chunk's header. For our case study, we choose a heap underflow primitive. When targeting the system server's SensorService, we are only able to control one chunk. The heap underflow allows us to reliably overwrite only what is needed, i.e., that chunk's header.

```
...
30963 31252 E SensorService: Ashmem direct channel requires a memory region to be supplied
30963 30981 D CompatibilityChangeReporter: Compat change id reported: 218533173; UID 10142; state: ENABLED
30963 30981 D CompatibilityChangeReporter: Compat change id reported: 262645982; UID 10142; state: DISABLED
30963 31379 W Parcel : Attempt to read object from Parcel 0x7ebd2a8acaa0 at offset 104 that is not in the object list
30963 30963 I H3Ll0 : FR0m_5y5T3M_53rV3r
```

Listing 7: The logcat output of the system server after successful exploitation, marked in orange is the logcat print triggered by the ROP chain.

Instead, if we used a heap overflow, we would have needed to overwrite chunks used by other Binder threads, increasing the complexity and reducing the reliability of our exploit.

An alternative to overwriting existing chunks is freeing fake attacker-created chunks. For this primitive the attacker needs to be able to control a pointer passed to free, which points to an attacker-controlled memory location. In glibc heap exploitation, the "House of Spirit" [46] leverages this primitive to insert the modified pointer into the free list, assuming the attacker was previously able to write a chunk header at that pointer's location. The "House of Spirit" is only feasible if the attacker can already write to the start and end of the target memory address. However, in Scudo, the primitive can be used to achieve an arbitrary write primitive. This primitive becomes even more appealing for Scudo exploitation because it is not influenced by random allocations, as the attacker chooses the location of the chunks.

In summary, the ideal memory corruption primitive to exploit Scudo is an attacker-controlled free. For the heap underflow or overflow, the preference heavily depends on how the target application handles the underflown or overflown chunk.

**Manipulating Scudo Chunk Header Fields** The Scudo chunk header has six fields, see Table 1. In Section 5 we discussed how manipulating header fields is only possible if the `Checksum` field is set properly and how the `State` field can be manipulated to induce double frees and use-after-frees. For our exploitation techniques, we overwrite the `ClassId` field to create a fake secondary chunk. However, for exploitation scenarios where our techniques are not applicable, there exists an alternative way of manipulating the `ClassId` field to achieve a heap overflow. Instead of changing the `ClassId` to `0`, which transforms the primary chunk to a secondary chunk, an attacker can change the primary chunk's size by replacing the original class ID with a larger one. Once that chunk is freed, it is placed into the `PerClass` free list of that larger primary chunk class ID. After said chunk is allocated from the free list, the chunk will overlap other smaller chunks, leading to a heap overflow in the Scudo memory region of the original chunk's class ID. For the remaining header fields, `OriginWasZeroed`, `SizeOrUnusedBytes`, and `Offset`, we did find a way to manipulate them in a useful manner.

**ARM MTE** ARM MTE (Memory Tagging Extension) [16] is a hardware security feature. Scudo has added support for MTE early on and in October 2023 the first Android devices supporting MTE running Scudo were released [13].

MTE uses the top bits of pointers to tag memory regions. A new instruction allows assigning a tag to a memory region. After a memory region has been tagged, the region can only be accessed with pointers whose top bits match the assigned tag. Tag mismatches result in a segmentation fault.

Allocators can leverage MTE to detect illegal memory accesses to the heap. By assigning tags to the body of allocated chunks, allocators can probabilistically prevent heap overflows or use-after-frees.

If enabled, Scudo tags the body of primary chunks on allocation and deallocation. The body of secondary chunks is not tagged. The chunk headers are assigned predictable tags (0 and 2 for the chunk headers of primary and secondary chunks respectively, 1 for the secondary chunk header).

With these tagged headers our exploitation techniques are mostly mitigated. Bypassing the *Separate* security measure, as described in Section 5, without crashing due to a tag mismatch is now limited to two specific scenarios. Overwriting the header of an existing primary chunk to create a fake secondary chunk is only possible if there is a chunk just before the overwritten chunk header and that chunk has tag 1, matching the tag assigned to the secondary chunk header. Alternatively, a fake secondary chunk may be forged by creating it on the border between two memory regions tagged with 1 and 2. The secondary chunk header is written to the lower memory region tagged with 1 and the chunk header to the region tagged with 2. Then an arbitrary free, freeing the address just after the forged chunk header passes MTE checks.

Bypassing security measure *Isolate* is only possible with the forged CommitBase technique as the safe unlink technique requires overlaying primary chunk headers (tagged with 0) over the fake secondary chunk header (tagged with 1). The forged CommitBase technique is further hampered by the fact that it needs to point to an address, whose first 0x30 bytes are tagged 1. Otherwise, Scudo crashes as it tries to write the secondary chunk header to the target address with a pointer tagged 1.

In conclusion, ARM MTE almost completely mitigates our exploitation techniques.

**Zygote Forking** To the best of our knowledge, Android is the only significant production deployment of Scudo. As discussed in Section 4, both security measures *Randomize* and *Protect* are rendered useless for Android userspace processes that are forked from Zygote. Without these security measures, Scudo's security becomes similar to that of the standard glibc allocator (predictable allocations and inline metadata that may be manipulated), while still incurring a performance overhead (calculating the checksum). This issue affects any Android userspace memory allocator and can only be solved by moving away from Zygote-forked userspace processes. Table 7 in the Appendix shows the userspace processes running on our stock emulator. 33 userspace processes are Zygote-forked (around 30% of all userspace processes), out of which seven processes run as a higher-privileged user. Note that the remaining processes are system apps, which are usually assigned special privileges.

**Quarantine** An optional Scudo feature is the quarantine which delays freed chunks from being allocated again right away. This can make Use-After-Frees harder to exploit but incurs a heavy performance penalty [34]. Related work has shown how the additional complexity and metadata introduced by the quarantine can be exploited for an arbitrary write primitive [24, 50]. Since the quarantine is disabled by default and disabled on Android, we omit it from this work.

**Scudo deployment** Although Scudo is the default allocator in Android's libc, vendors may choose to utilize jemalloc or implement their allocator. To understand if vendors choose to deploy Scudo we analyzed the firmware of recently released phones. We picked 15 devices, whose firmware is easily available, and analyzed the symbols in the shipped libc binary. Table 6 in the Appendix lists the analyzed devices. Overall out of the 15 devices, 6 devices use Scudo. Of the remaining 9 use jemalloc. From this sample of firmware, it is clear that Scudo is deployed in production but has not replaced jemalloc.

**Case Study** We backported and exploited the system server on the Android emulator running an x86 image[1]. Most Android production devices are ARM-based, however, we designed a data-only exploit. The only part of the exploit that needs to be changed for an ARM device is the ROP chain. Furthermore, the emulator provides high fidelity for the Android userspace [22].

## 8 Related Work

The security community has recently started showing an interest in Scudo. Un1fuzz [50] gives an overview of Scudo internals and presents two exploits against Scudo quarantine. Cesare demonstrates how to compute the cookie with z3 after leaking the chunk header and the chunk header's address [15]. More recently multiple blogs have been published detailing the inner workings of Scudo [7, 12, 20]. Concurrently to us, Ecob discovered the forged CommitBase exploit and presented his findings at Bsides Canberra [24]. In this work, we systematize Scudo's security mechanisms, put these mechanisms in the context of the Android userspace, present two exploits and demonstrate our findings against a real target.

Besides Scudo, allocators have long been a target. Researchers have demonstrated exploits against the glibc allocator [5, 9, 29, 31, 43] and jemalloc [6, 44]. These works serve as an inspiration to us and we hope to extend the community's compendium of exploitation techniques with our Scudo specific techniques.

HeapHopper [23] and ArchHeap [54] are systems to automatically discover heap exploitation primitives. These systems mainly focus on dlmalloc or ptmalloc. ArchHeap included Scudo in its evaluation but failed to discover any exploitation primitives.

Other works have focussed on manipulating the heap's layout [28, 33, 51]. The systems proposed by these works analyze the target program to identify heap manipulation primitives. In our work we focus solely on Scudo and leave identifying the heap manipulation primitives out of scope.

There has been a large body of work on building secure allocators [2, 8, 19, 35, 38, 47, 48] or securing existing allocators [1, 26, 37, 52, 53]. Unlike these works, we focus on dissecting the security measures of an existing, widely deployed allocator.

## 9 Conclusion

We investigated the security of Scudo, Android's hardened memory allocator. We have found that a large part of Scudo's security measures are rendered ineffective by Android's userspace architecture in the context of our attacker model. Given a memory corruption vulnerability, we demonstrate two exploits which manipulate Scudo into allocating a chunk at an attacker's chosen address.

To demonstrate that our findings are indeed practical, we backported an n-day memory corruption vulnerability to Android 14 and exploited the highly privileged system server from the context of an unprivileged app, achieving a privilege escalation.

In the process of researching Scudo we have developed a gdb plugin, which allows inspecting Scudo chunks and free lists, and a python library to forge Scudo chunk headers. We open-source all these tools along with the code and artifacts of our exploitation case study at `https://github.com/HexHive/scudo-exploitation`.

[1]Android emulator image: system-images;android-34;google_apis;x86_64

## Acknowledgments

We thank the anonymous reviewers and artifact evaluation committee for their feedback on the paper This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2_186974, and DARPA HR001119S0089-AMP-FP-034

## References

[1] Sam Ainsworth and Timothy M Jones. Markus: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 578–591. IEEE, 2020.

[2] Periklis Akritidis et al. Cling: A memory allocator to mitigate dangling pointers. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[3] android developers.googleblog.com. System hardening in android 11. https://android-developers.googleblog.com/2020/06/system-hardening-in-android-11.html, 2020. Accessed: January 2024.

[4] android.googlesource.com. fdsan. https://android.googlesource.com/platform/bionic/+/master/docs/fdsan.md, 2024. Accessed: January 2024.

[5] anonymous. Once upon a free()... http://phrack.org/issues/57/9.html, 2001. Accessed: January 2024.

[6] huku argp. Pseudomonarchia jemallocum. http://www.phrack.org/issues/68/10.html#article, 2012. Accessed: January 2024.

[7] Jacob Bech. Advancing cybersecurity: Introduction to the scudo allocator. https://vectorize.re/blog/internals/introduction-to-scudo/, 2023. Accessed: January 2024.

[8] Emery D Berger and Benjamin G Zorn. Diehard: Probabilistic memory safety for unsafe languages. *Acm sigplan notices*, 41(6):158–168, 2006.

[9] blackngel. Malloc des-maleficarum. http://phrack.org/issues/66/10.html, 2009. Accessed: January 2024.

[10] blog.thalium.re. The fuzzing guide to the galaxy: An attempt with android system services. https://blog.thalium.re/posts/fuzzing-samsung-system-services/#cve-2022-39907, 2023. Accessed: March 2024.

[11] Elias Boschung. Mitigation pull request. https://github.com/llvm/llvm-project/pull/75295, 2023. Accessed: January 2024.

[12] Rodrigo Branco. Scudo hardened allocator — unofficial internals documentation. https://www.l3harris.com/newsroom/editorial/2023/10/scudo-hardened-allocator-unofficial-internals-documentation, 2023. Accessed: January 2024.

[13] Mark Brand. First handset with mte on the market. https://googleprojectzero.blogspot.com/2023/11/first-handset-with-mte-on-market.html, 2023. Accessed: March 2024.

[14] Valerio Brussani. Whatsapp 2.19.216 remote code execution. http://packetstormsecurity.com/files/154867/Whatsapp-2.19.216-Remote-Code-Execution.html, 2019. Accessed: January 2024.

[15] Dr. Silvio Cesare. Breaking secure checksums in the scudo allocator. https://blog.infosectcbr.com.au/2020/04/breaking-secure-checksums-in-scudo_8.html, 2020. Accessed: January 2024.

[16] community.arm.com. Memory tagging extension: Enhancing memory safety through architecture. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety, 2019. Accessed: January 2024.

[17] cs.android.com. Android sensorservice source. https://cs.android.com/android/platform/superproject/main/+/main:frameworks/native/libs/sensor/ISensorServer.cpp, 2024. Accessed: January 2024.

[18] cs.android.com. Android system server source. https://cs.android.com/android/platform/superproject/main/+/main:frameworks/base/services/java/com/android/server/SystemServer.java;l=250?q=systemserver.&sq=&ss=android%2Fplatform%2Fsuperproject%2Fmain, 2024. Accessed: January 2024.

[19] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th USENIX security symposium (USENIX security 17)*, pages 815–832, 2017.

[20] Kevin Denis. Behind the shield: Unmasking scudo's defenses. https://www.synacktiv.com/publications/behind-the-shield-unmasking-scudos-defenses, 2023. Accessed: January 2024.

[21] developer.android.com. Overview of memory management. https://developer.android.com/topic/performance/memory-overview#SharingRAM, 2024. Accessed: January 2024.

[22] developer.android.com. Run apps on the android emulator. https://developer.android.com/studio/run/emulator, 2024. Accessed: January 2024.

[23] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Heaphopper: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 99–116, 2018.

[24] Zac Ecob. Scudo allocator exploitation. In *Bsides Canberra*, Australia, Sidney, September 2023.

[25] en.wikipedia.org. Stagefright (bug). https://en.wikipedia.org/wiki/Stagefright_(bug), 2023. Accessed: March 2024.

[26] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. PTAuth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054. USENIX Association, August 2021.

[27] Guang Gong. Exploiting heap corruption due to integer overflow in android libcutils. https://www.blackhat.com/docs/us-15/materials/us-15-Gong-Fuzzing-Android-System-Services-By-Binder-Call-To-Escalate-Privilege-wp.pdf, 2015. Accessed: January 2024.

[28] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 763–779, 2018.

[29] jp. Advanced doug lea's malloc exploits. http://phrack.org/issues/61/6.html, 2003. Accessed: January 2024.

[30] Mateusz Jurczyk. Mms exploit part 1-5. https://googleprojectzero.blogspot.com/2020/07/mms-exploit-part-1-introduction-to-qmage.html, 2020. Accessed: January 2024.

[31] Michel "MaXX" Kaempf. Vudo malloc tricks. http://phrack.org/issues/57/8.html, 2001. Accessed: January 2024.

[32] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. In *2014 IEEE Symposium on Security and Privacy*, pages 424–439. IEEE, 2014.

[33] Runhao Li, Bin Zhang, Jiongyi Chen, Wenfeng Lin, Chao Feng, and Chaojing Tang. Towards automatic and precise heap layout manipulation for general-purpose programs. In *NDSS*, 2023.

[34] llvm.org. Scudo hardened allocator. https://llvm.org/docs/ScudoHardenedAllocator.html, 2024. Accessed: January 2024.

[35] Vitaliy B Lvin, Gene Novark, Emery D Berger, and Benjamin G Zorn. Archipelago: trading address space for reliability and security. *ACM SIGARCH Computer Architecture News*, 36(1):115–124, 2008.

[36] Matt Miller. Root cause of microsoft rce cves by patch year. https://twitter.com/epakskape/status/984481101937651713/photo/1, 2018. Accessed: February 2024.

[37] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 international symposium on Memory management*, pages 31–40, 2010.

[38] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584, 2010.

[39] nvd.nist.gov. Cve-2015-1528. https://nvd.nist.gov/vuln/detail/CVE-2015-1528, 2015. Accessed: January 2024.

[40] nvd.nist.gov. Cve-2019-2136. https://nvd.nist.gov/vuln/detail/CVE-2019-2136, 2019. Accessed: March 2024.

[41] nvd.nist.gov. Cve-2020-0026. https://nvd.nist.gov/vuln/detail/CVE-2020-0026, 2020. Accessed: March 2024.

[42] nvd.nist.gov. Cve-2021-24041. https://nvd.nist.gov/vuln/detail/CVE-2021-24041, 2021. Accessed: March 2024.

[43] Phantasmal Phantasmagoria. The malloc maleficarum. https://seclists.org/bugtraq/2005/Oct/118, 2005. Accessed: January 2024.

[44] Shmarya Rubenstein. A tale of two mallocs: On android libc allocators. In *INFILITRATE 2018*, United States, Miami Beach, April 2018.

[45] how2heap shellphish. unsafe_unlink. https://github.com/shellphish/how2heap/blob/master/glibc_2.38/unsafe_unlink.c, 2020. Accessed: January 2024.

[46] how2heap shellphish. house_of_spirit. https://github.com/shellphish/how2heap/blob/master/glibc_2.31/house_of_spirit.c, 2022. Accessed: January 2024.

[47] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2389–2403, 2017.

[48] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: A tunable secure allocator. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 117–133, 2018.

[49] source.android.com. Scudo. https://source.android.com/docs/security/test/scudo, 2024. Accessed: January 2024.

[50] un1fuzz.github.io. Scudo internals, attacking scudo's quarantine. https://un1fuzz.github.io/index.html, 2020. Accessed: January 2024.

[51] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. Maze: Towards automated heap feng shui. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1647–1664, 2021.

[52] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, Jung-Won Lim, Sanidhya Kashyap, and Taesoo Kim. Preventing use-after-free attacks with fast forward allocation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2453–2470, 2021.

[53] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. Pumm: Preventing use-after-free using execution unit partitioning. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 823–840, 2023.

[54] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1111–1128, 2020.

## Appendix

| ClassId | Size Start | Size End |
|---|---|---|
| 1 | 0x0 | 0x10 |
| 2 | 0x11 | 0x20 |
| 3 | 0x21 | 0x30 |
| 4 | 0x31 | 0x40 |
| 5 | 0x41 | 0x50 |
| 6 | 0x51 | 0x60 |
| 7 | 0x61 | 0x80 |
| 8 | 0x81 | 0xa0 |
| 9 | 0xa1 | 0xb0 |
| 10 | 0xb1 | 0xd0 |
| 11 | 0xd1 | 0x110 |
| 12 | 0x111 | 0x150 |
| 13 | 0x151 | 0x1b0 |
| 14 | 0x1b1 | 0x240 |
| 15 | 0x241 | 0x310 |
| 16 | 0x311 | 0x440 |
| 17 | 0x441 | 0x660 |
| 18 | 0x661 | 0x820 |
| 19 | 0x821 | 0xa00 |
| 20 | 0xa01 | 0xc20 |
| 21 | 0xc21 | 0x1000 |
| 22 | 0x1001 | 0x1200 |
| 23 | 0x1201 | 0x1bc0 |
| 24 | 0x1bc1 | 0x2200 |
| 25 | 0x2201 | 0x2d80 |
| 26 | 0x2d81 | 0x3780 |
| 27 | 0x3781 | 0x4000 |
| 28 | 0x4001 | 0x4800 |
| 29 | 0x4801 | 0x5a00 |
| 30 | 0x5a01 | 0x7300 |
| 31 | 0x7301 | 0x8200 |
| 32 | 0x8201 | 0x10000 |
| 0 | 0x10001 | ... |

Table 5: Chunk sizes and the corresponding class ID on Android 14.

```
void remove(T *X) {
    T *Prev = X->Prev;
    T *Next = X->Next;
    if (Prev) {
      CHECK_EQ(Prev->Next, X);
      Prev->Next = Next;
    }
    if (Next) {
      CHECK_EQ(Next->Prev, X);
      Next->Prev = Prev;
    }
    Size = Size -1;
}
```

Listing 8: Excerpt from the Scudo source code, which unlinks the secondary chunk from the linked list of allocated secondary chunks. Scudo checks the integrity of the linked list with the `CHECK_EQ` macro. The `CHECK_EQ` macro aborts if the arguments are not equal.

| **Device** | **Date** | **Allocator** |
|---|---|---|
| Samsung S24 | 2/2024 | Scudo |
| Samsung S23 Ultra | 1/2024 | jemalloc |
| Samsung M14 5G | 12/2023 | jemalloc |
| Samsung A34 5G | 10/2023 | Scudo |
| Samsung Galaxy Z Fold 5 | 2/2024 | jemalloc |
| Google Pixel 8 | 2/2024 | Scudo |
| Google Pixel Fold | 2/2024 | Scudo |
| Xiaomi Redmi Note 13 5G | 2/2024 | jemalloc |
| Xiaomi Redmi 12 5G | 12/2023 | jemalloc |
| Xiaomi Redmi 13C 5G | 1/2024 | jemalloc |
| Xiaomi Redmi Note 12 4G | 11/2023 | jemalloc |
| Vivo y35 | 1/2024 | Scudo |
| Vivo y73 | 1/2024 | Scudo |
| Oppo A96 5G | 6/2023 | jemalloc |
| Oppo Reno 8 Pro | 1/2024 | jemalloc |

Table 6: The analyzed firmware to understand if vendors deploy Scudo. The Date column denotes the firmware's release date. Out of the 15 devices, 6 use the Scudo allocator. The remaining 9 use jemalloc.

| User | Name |
|---|---|
| system | system_server |
| u0_a160 | com.android.systemui |
| webview_zygote | webview_zygote |
| network_stack | com.android.networkstack.process |
| bluetooth | com.google.android.bluetooth |
| secure_element | com.android.se |
| radio | com.android.phone |
| u0_a172 | com.google.android.ext.services |
| u0_a158 | com.google.android.apps.nexuslauncher |
| u0_a169 | com.google.android.permissioncontroller |
| u0_a129 | com.google.android.gms.persistent |
| u0_a142 | com.google.android.inputmethod.latin |
| u0_a129 | com.google.android.gms |
| u0_a127 | com.google.android.as |
| u0_a131 | com.google.android.googlequicksearchbox:interactor |
| u0_a130 | com.google.android.apps.messaging:rcs |
| system | com.android.emulator.multidisplay |
| u0_a131 | com.google.android.googlequicksearchbox:search |
| u0_a130 | com.google.android.apps.messaging |
| u0_a129 | com.google.android.gms.unstable |
| u0_a185 | com.google.android.providers.media.module |
| u0_a129 | com.google.process.gservices |
| u0_a92 | android.process.media |
| u0_a154 | com.google.android.gm |
| u0_a184 | com.google.android.rkpdapp |
| u0_a175 | com.google.android.adservices.api |
| u0_a129 | com.google.process.gapps |
| u0_a81 | android.process.acore |
| u0_a144 | com.google.android.apps.photos |
| u0_a173 | com.google.android.devicelockcontroller |
| u0_a124 | com.google.android.settings.intelligence |
| u0_a151 | com.google.android.contacts |
| u0_a162 | com.google.android.apps.wallpaper |

Table 7: The 33 userspace processes which are forked from Zygote on the Android 14 emulator (`system-images;android-34;google_apis;x86_64`). Overall there are 107 userspace processes out of which 33 (30%) are Zygote-forked. 7 (6%) Zygote-forked processes are running as higher-privileged users. Note that the remaining processes are mostly privileged system apps. Compromising such an app from a normal app still escalates the attacker's privileges. Any additional, user-installed app will also be Zyote-forked.