SektionEins
http://www.sektioneins.de

iOS Kernel Heap Armageddon

Stefan Esser <stefan.esser@sektioneins.de>

SyScan Singapore 2012

Who am I?

Stefan Esser

- from Cologne / Germany
- in information security since 1998
- PHP core developer since 2001
- Month of PHP Bugs and Suhosin
- recently focused on iPhone security (ASLR, jailbreak)
- Head of Research and Development at SektionEins GmbH

Recap...

- public iOS kernel heap research can be summarized as
  - there is a kernel heap zone allocator
  - it comes with heap meta data
  - which can be exploited
  - here is one possible way

So what is this talk about?

- zone allocator recap
- other kernel heap managers / wrappers
- recent changes in the allocators
- cross zone attacks
- kernel level application data overwrite attacks
- generic heap massage technique

Part I

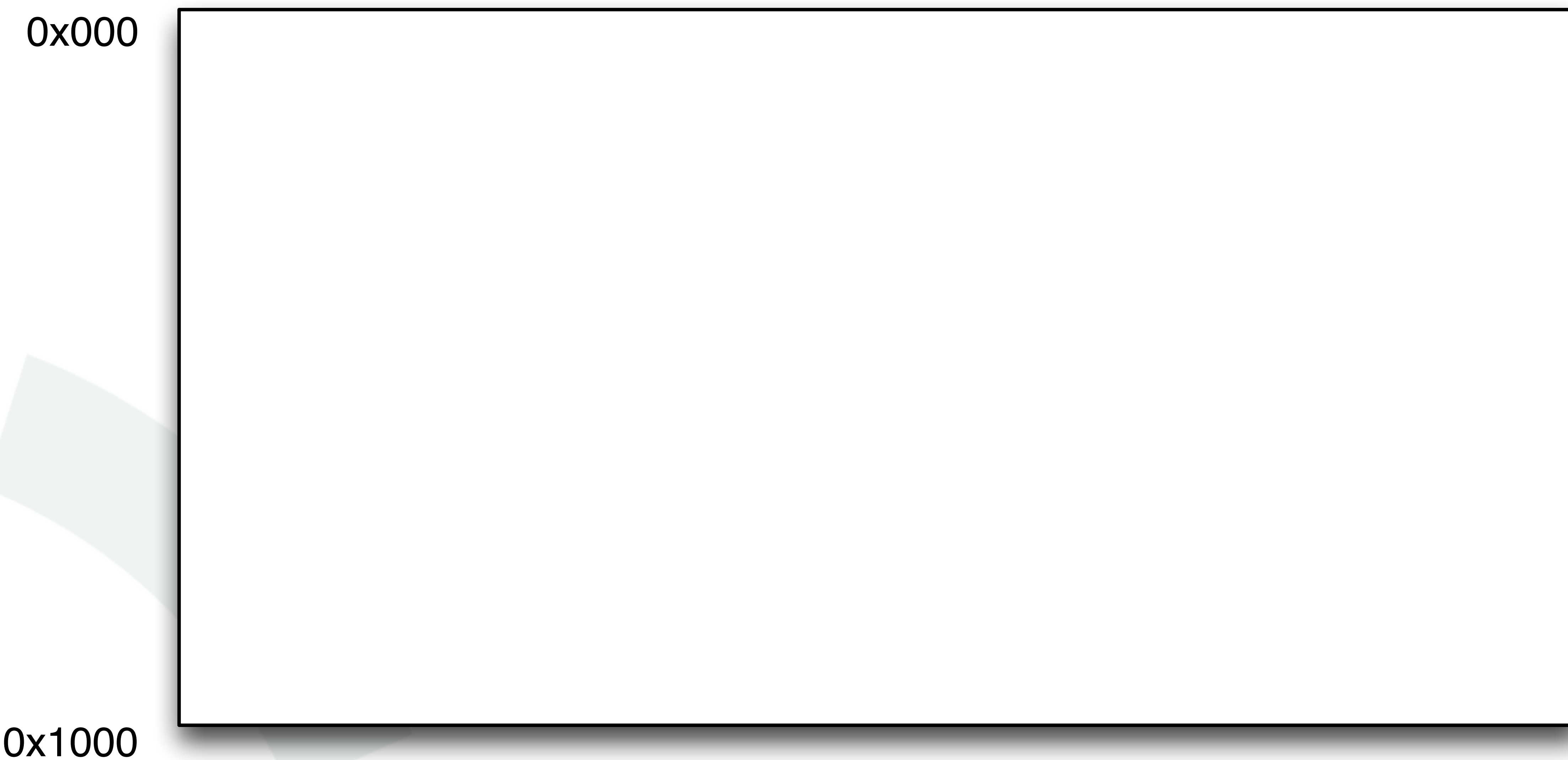
Zone Allocator Recap

Some Kernel Zones

```
$ zprint kalloc
                          elem     cur     max    cur     max   cur alloc alloc
zone name                 size    size    size  #elts   #elts inuse  size count
-------------------------------------------------------------------------------
zones                      460     84K     90K    187     200   167   20K    44
vm.objects                 148    487K    512K   3375    3542  3103    4K    27 C
vm.object.hash.entries      20     19K    512K   1020   26214   704    4K   204 C
maps                       168     11K     40K     72     243    61    4K    24
VM.map.entries              48    203K   1024K   4335   21845  3859    4K    85 C
Reserved.VM.map.entries     48     27K   1536K    597   32768   191    4K    85
VM.map.copies               48      3K     16K     85     341     0    4K    85 C
pmap                      2192    134K    548K     63     256    52   20K     9 C
...
tcp_bwmeas_zone             32      0K      4K      0     128     0    4K   128 C
igmp_ifinfo                112      3K      8K     36      73     3    4K    36 C
ripzone                    268      3K   1072K     15    4096     0    4K    15 C
in_multi                   136      3K     12K     30      90     2    4K    30 C
ip_msource                  28      0K      4K      0     146     0    4K   146 C
in_msource                  20      0K      4K      0     204     0    4K   204 C
in_ifaddr                  156      3K     12K     26      78     1    4K    26 C
ip_moptions                 52      3K      4K     78      78     1    4K    78 C
llinfo_arp                  36      0K     12K      0     341     0    4K   113 C
unpzone                    152     27K   1132K    182    7626   129    4K    26 C
fs-event-buf                64     64K     64K   1024    1024     0    4K    64
bridge_rtnode               40      0K     40K      0    1024     0    4K   102 C
vnode.pager.structures      20     19K    196K   1020   10035   655    4K   204 C
kernel_stacks            16384   1232K   1232K     77      77    33   16K     1 C
page_tables               4096   6688K    ----   1672    ----  1672    4K     1 C
kalloc.large             64898   2218K   8961K     35     141    35   63K     1
```

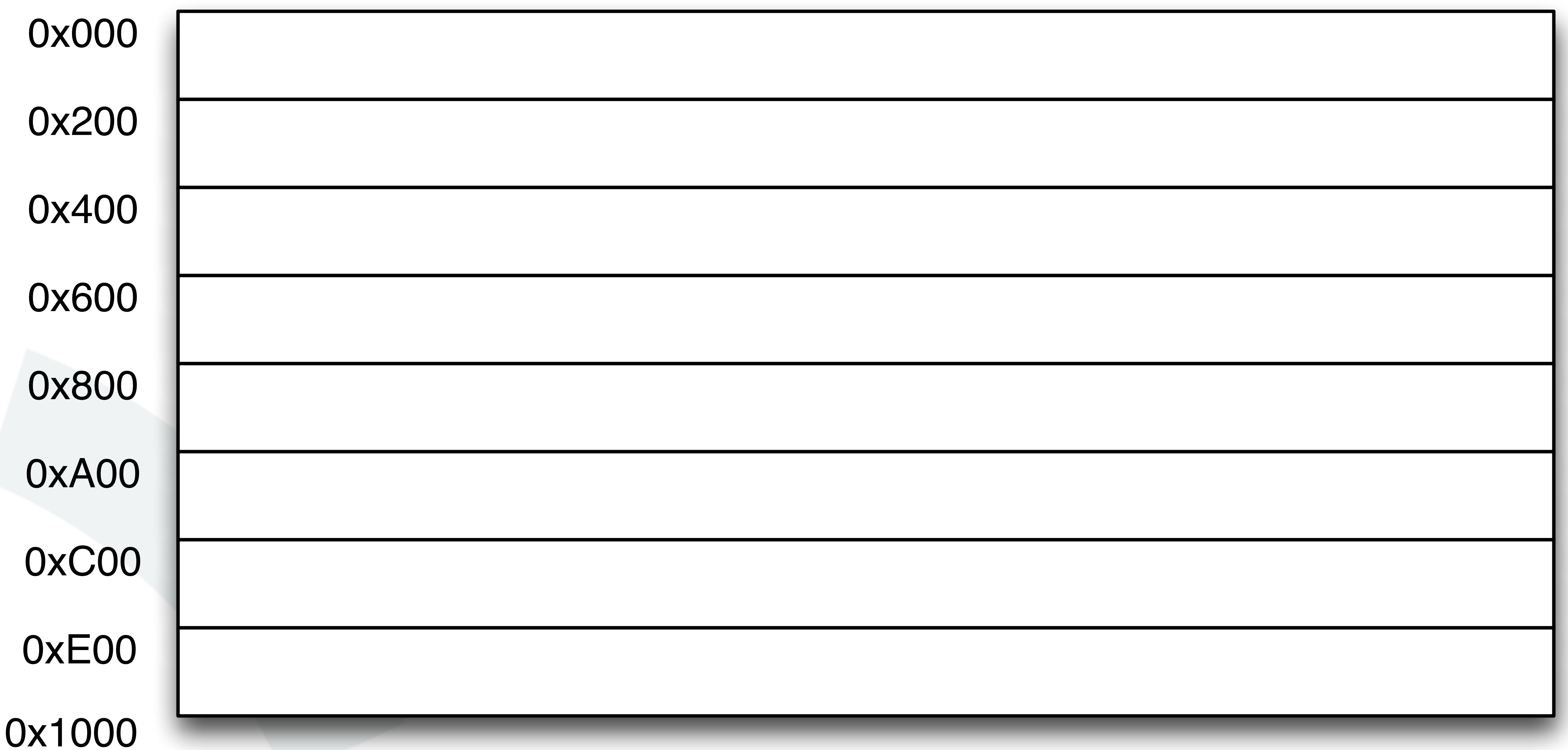
iOS Kernel Zone Allocator 101

- kernel heap is divided into so called zones
- each zone starts with a first chunk of memory (usually 1 page)



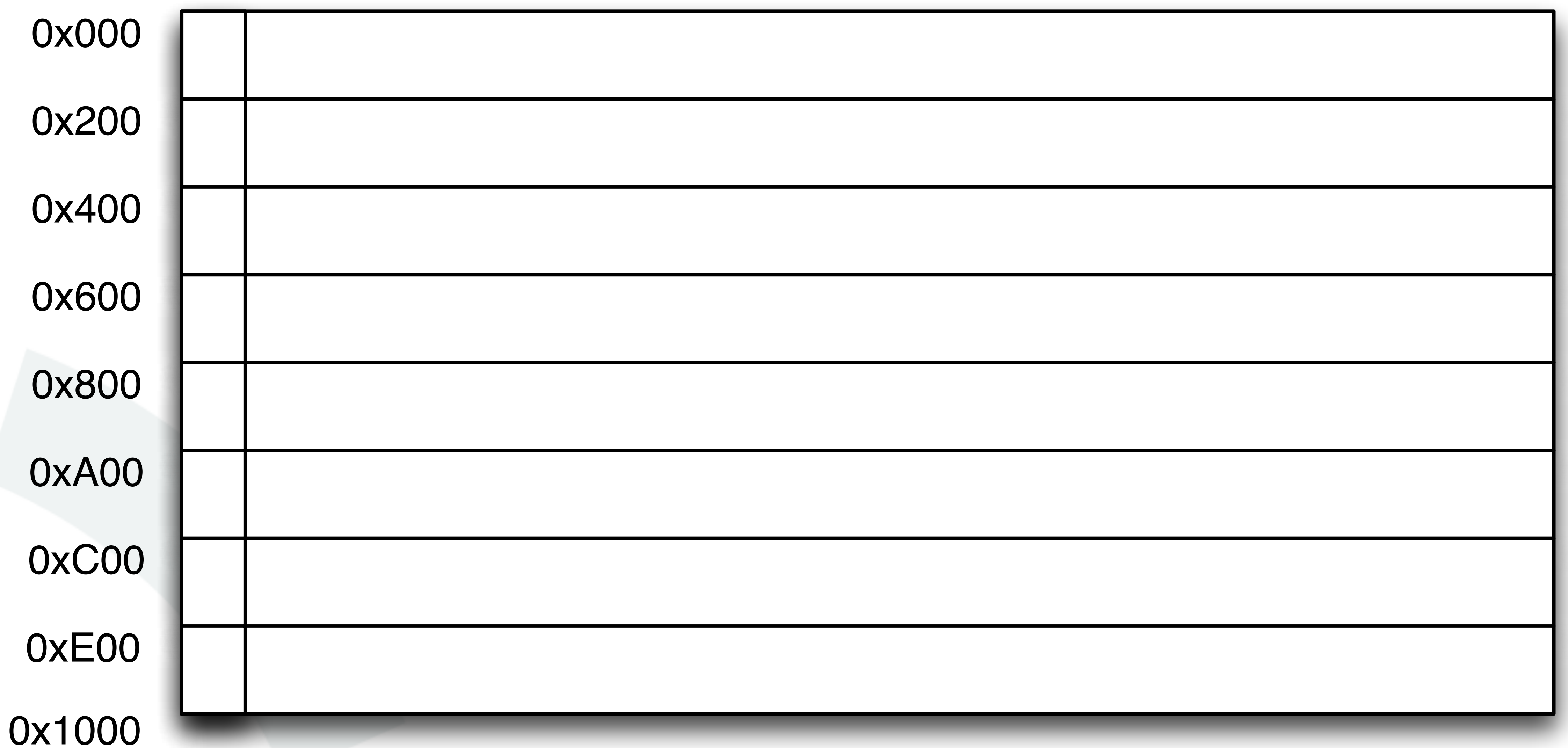
iOS Kernel Zone Allocator 101

- each zone is divided into memory blocks of the same size
- all memory allocated within a zone will have the same block size



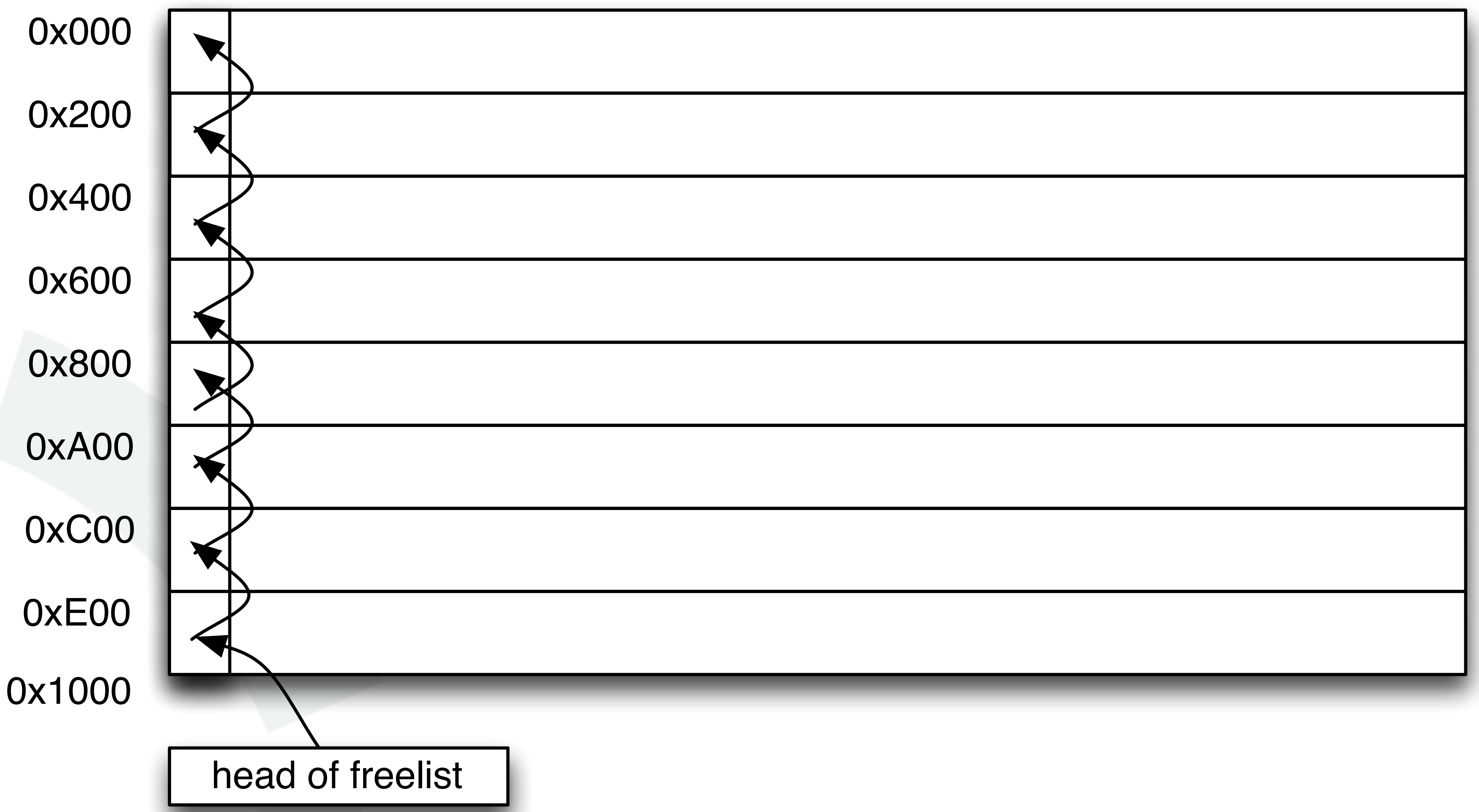
iOS Kernel Zone Allocator 101

- zone allocator keeps inbound heap meta data
- first 4 bytes of a free block is a pointer to another free block



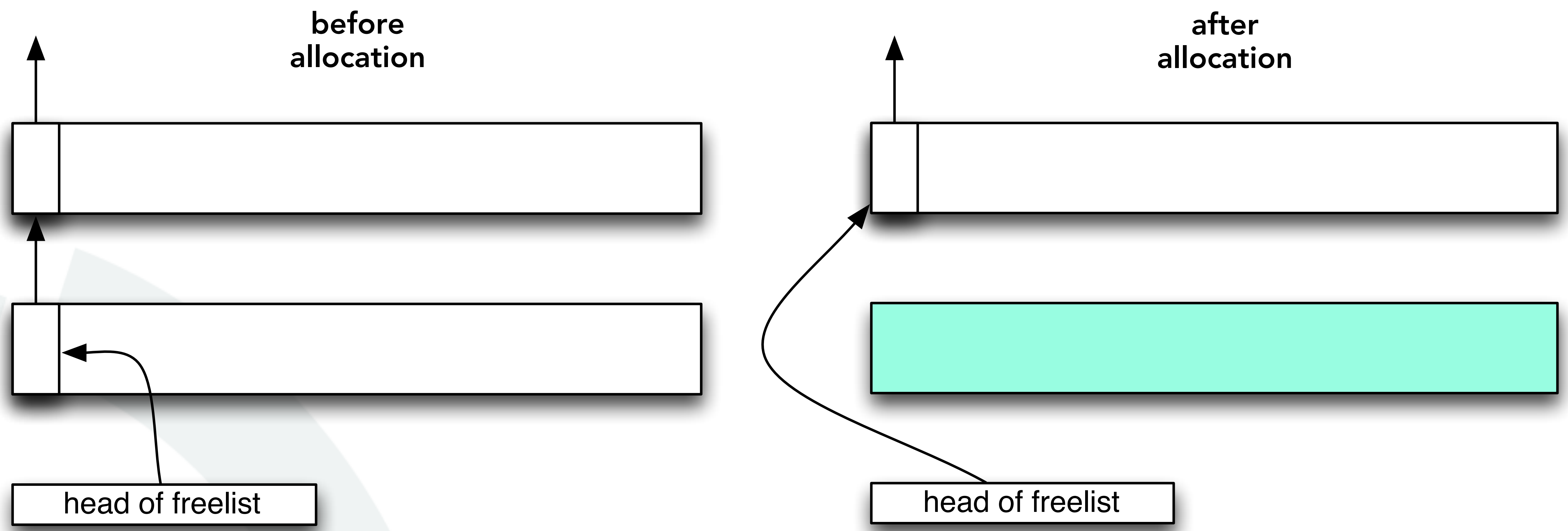
iOS Kernel Zone Allocator 101

- zone allocator keeps a single linked list of free blocks
- last memory block is first in freelist - memory is allocated backwards



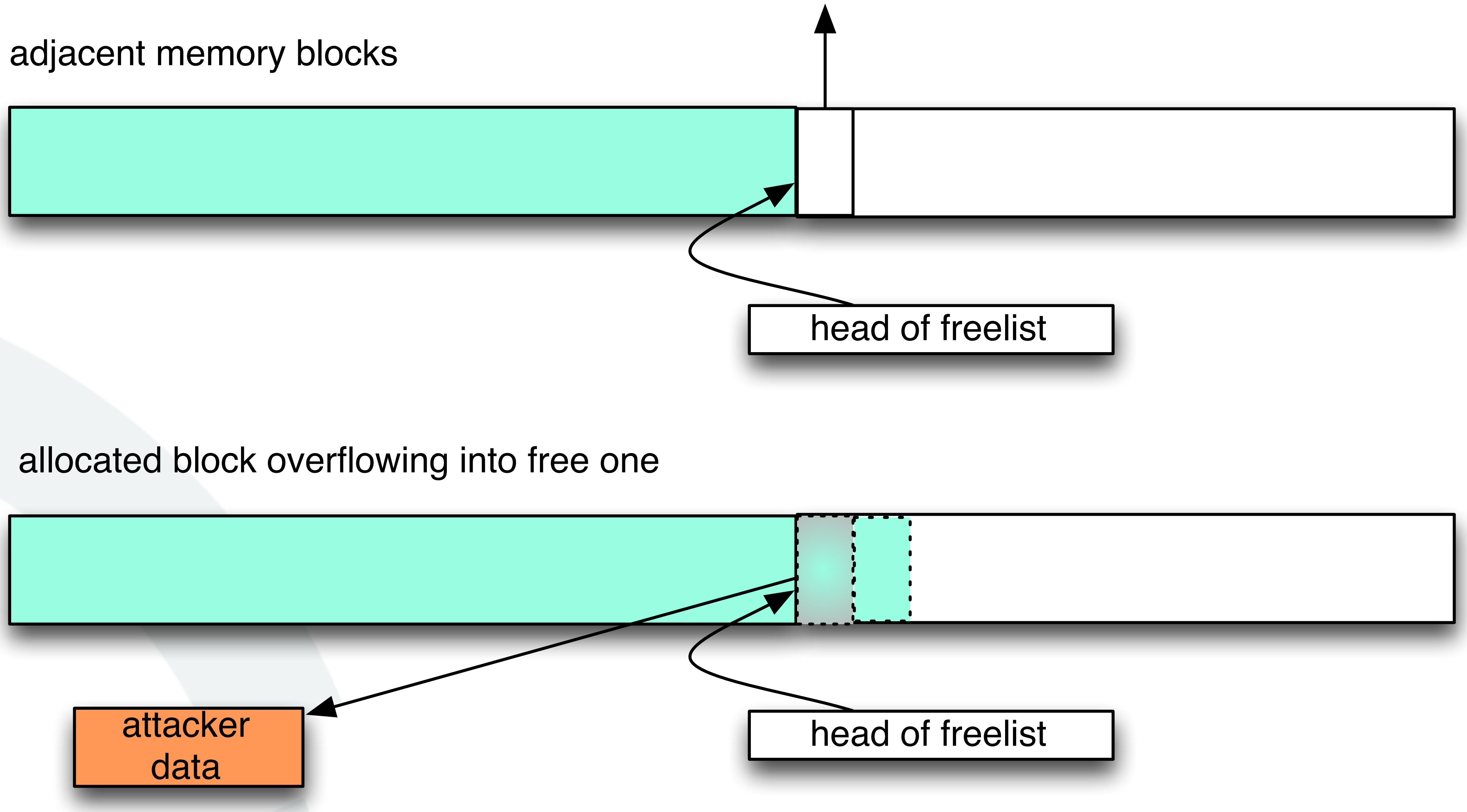
iOS Kernel Zone Allocator 101

- when memory is allocated the head of the freelist is returned
- and the pointer stored in the free memory block is made the new head



iOS Kernel Zone Allocator 101

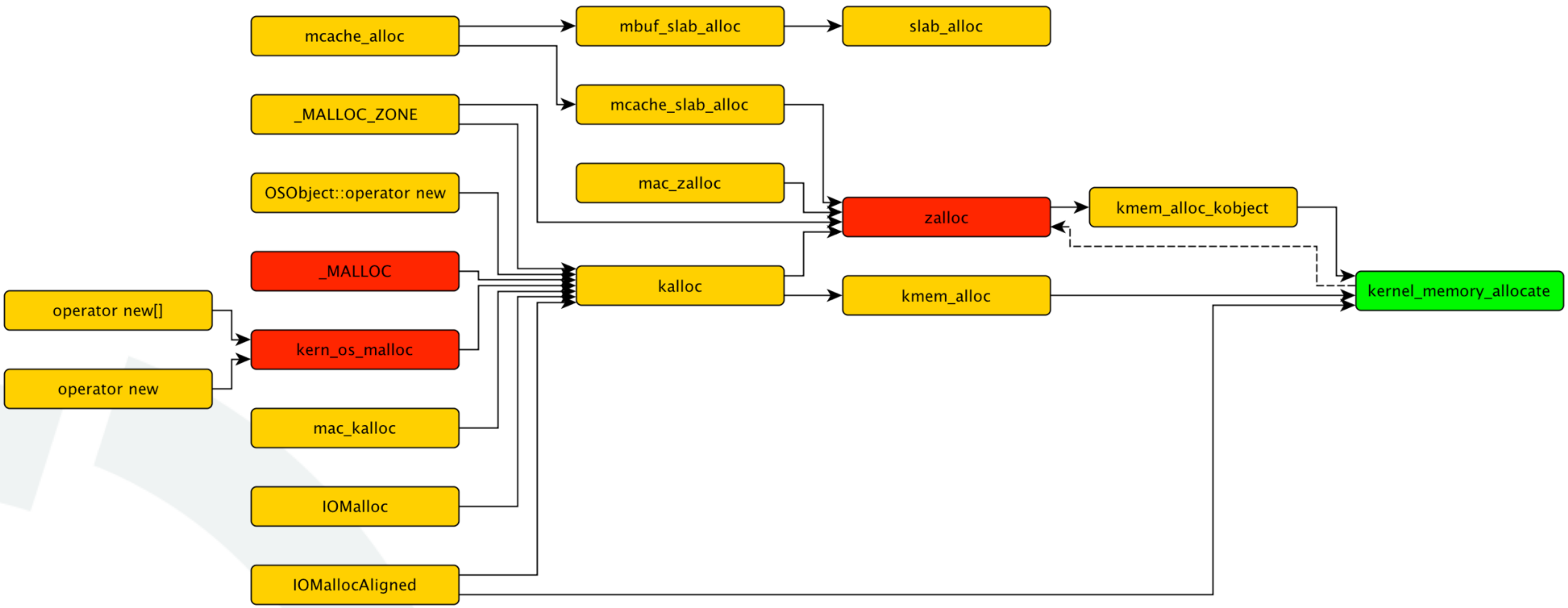
- in case of a buffer overflow the freelist pointer is overwritten
- next allocation will make attacker controlled pointer the head of freelist
- and the allocation following after will return the injected pointer



Part II

Other Heap Managers and Wrappers

Overview Managers and Wrappers



not necessary a complete overview

Let's have a look at kalloc()

kalloc()

- ***kalloc()*** is a wrapper around ***zalloc()*** and ***kmem_alloc()***
- it adds no additional heap meta data
- caller needs to keep track of allocated size

- for small requests ***zalloc()*** is used
- for bigger requests ***kmem_alloc()*** is used
- ***kalloc()*** registers several zones with names like ***kalloc.****

iOS 5 - kalloc() Zones

```
$ zprint kalloc
                          elem     cur     max     cur     max    cur  alloc alloc
zone name                 size    size    size  #elts   #elts  inuse   size count
-------------------------------------------------------------------------------
kalloc.8                     8     68K     91K   8704   11664   8187     4K   512 C
kalloc.16                   16     96K    121K   6144    7776   5479     4K   256 C
kalloc.24                   24    370K    410K  15810   17496  15567     4K   170 C
kalloc.32                   32    136K    192K   4352    6144   4087     4K   128 C
kalloc.40                   40    290K    360K   7446    9216   7224     4K   102 C
kalloc.48                   48     95K    192K   2040    4096   1475     4K    85 C
kalloc.64                   64    144K    256K   2304    4096   2017     4K    64 C
kalloc.88                   88    241K    352K   2806    4096   2268     4K    46 C
kalloc.112                 112    118K    448K   1080    4096    767     4K    36 C
kalloc.128                 128    176K    512K   1408    4096   1049     4K    32 C
kalloc.192                 192    10                                             C
kalloc.256                 256    19                                             C
kalloc.384                 384    59                                             C
kalloc.512                 512     4                                             C
kalloc.768                 768     9                                             C
kalloc.1024               1024    12                                             C
kalloc.1536               1536    10                                             C
kalloc.2048               2048     8                                             C
kalloc.3072               3072    67                                             C
kalloc.4096               4096    12                                             C
kalloc.6144               6144    42                                             C
kalloc.8192               8192    176K  32768K     22    4096     20     8K     1 C
```

- iOS 5 introduces new ***kalloc.**** zones that are not powers of 2
- smallest zone is now for 8 byte long memory blocks
- memory block are aligned to their own size their size is a power of 2

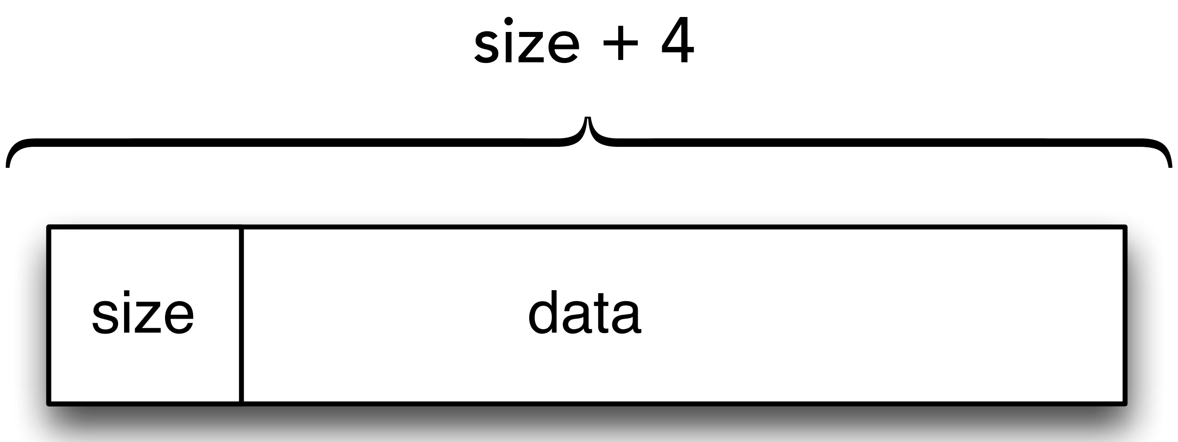
kfree()

- ***kfree()*** is a bit special
- "protection" against double frees
- keeps track of largest allocated memory block
- attempt to ***kfree()*** a larger block is a NOP

Let's have a look at _MALLOC()

_MALLOC()

- ***_MALLOC()*** is a wrapper around ***kalloc()***
- it adds the blocksize as additional heap meta data
- so the caller does not need to keep track of allocated size
- it refuses to allocate 0 byte sizes



_MALLOC() in iOS 4.x

```c
void *_MALLOC(size_t size, int type, int flags)
{
    struct _mhead   *hdr;
    size_t memsize = sizeof (*hdr) + size;

    if (type >= M_LAST)
        panic("_malloc TYPE");

    if (size == 0)
        return (NULL);

    if (flags & M_NOWAIT) {
        hdr = (void *)kalloc_noblock(memsize);
    } else {
        hdr = (void *)kalloc(memsize);
        ...
    }
    ...
    hdr->mlen = memsize;

    return  (hdr->dat);
}
```

possible integer overflow with huge size values

refuses to allocate 0 byte big blocks

```c
struct _mhead {
    size_t  mlen;
    char    dat[0];
}
```

_MALLOC() in iOS 5.x

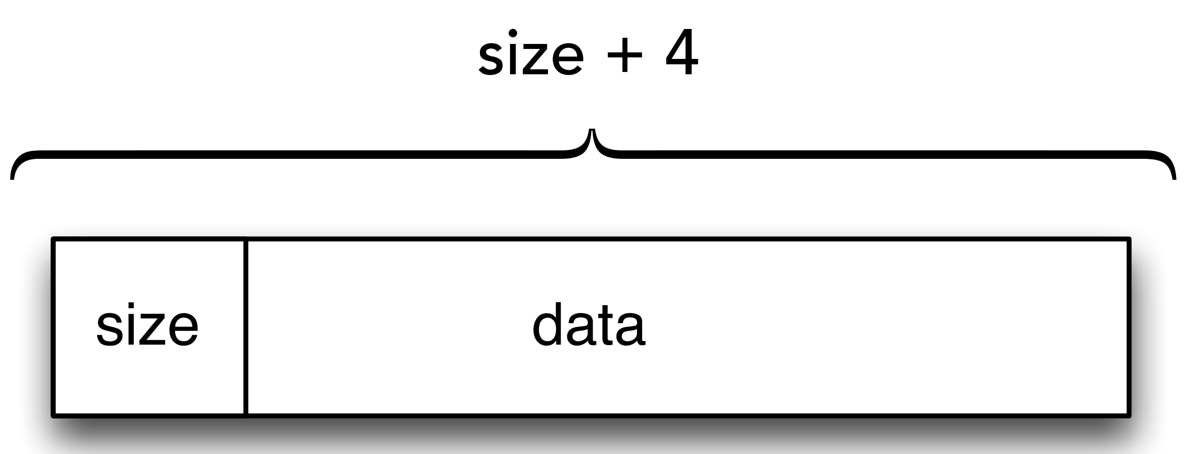
```
void *_MALLOC(size_t size, int type, int flags)
{
    struct _mhead   *hdr;
    size_t memsize = sizeof (*hdr) + size;
    int overflow = memsize < size ? 1 : 0;

    ...
    if (flags & M_NOWAIT) {
        if (overflow)
            return (NULL);
        hdr = (void *)kalloc_noblock(memsize);
    } else {
        if (overflow)
           panic("_MALLOC: overflow detected, size %llu", size);
        hdr = (void *)kalloc(memsize);
        ...
    }
    ...
    hdr->mlen = memsize;

    return  (hdr->dat);
}
```

integer overflow detection

attacker can use overflow to panic kernel M_WAIT

Overwriting _MALLOC()ed Data

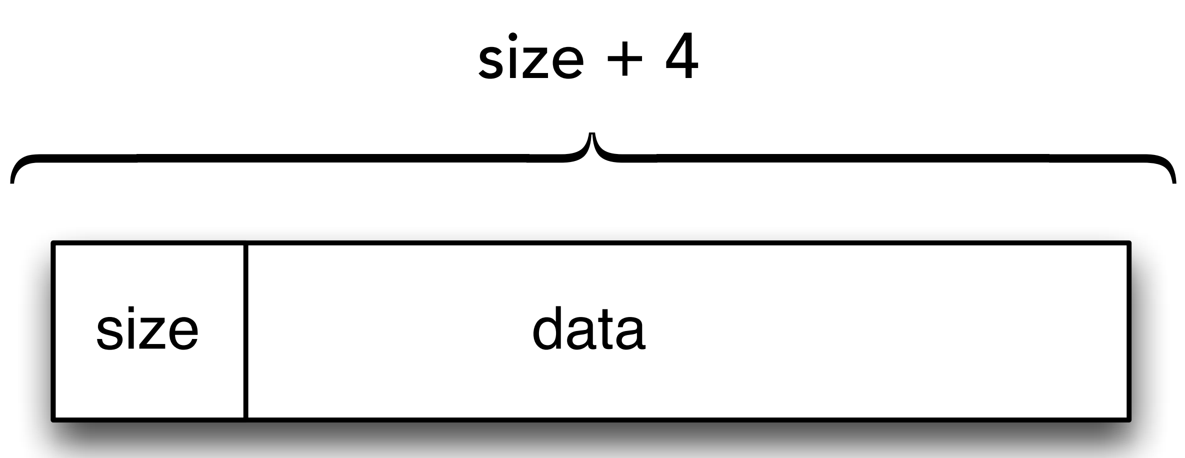
- changing the size of a memory block
- freeing the block will put it in the wrong freelist
    - smaller sizes will leak some memory
    - bigger sizes will result in buffer overflows



What about kern_os_malloc(), new and new[]

kern_os_malloc()

- ***kern_os_malloc()*** is very similar to ***_MALLOC()***
- it also adds the blocksize as additional heap meta data
- it also refuses to allocate 0 byte sizes
- new and new[] simply wrap around it
- special case: new[0] will allocate 1 byte



mcache / slab

could and might fill a whole talk by themself *and kernel_memory_allocate ???*

kernel_memory_allocate

- "master entry point for allocating kernel memory"
- allocates memory in a specific map
- allocates always whole pages
- requests for more than 1 GB fail immediately
- keeps a bunch of heap meta data inside a separate kernel zone
- no inbound meta data

Part III

Cross Zone or Cross Memory Allocator Attacks?

Cross Zone Attacks

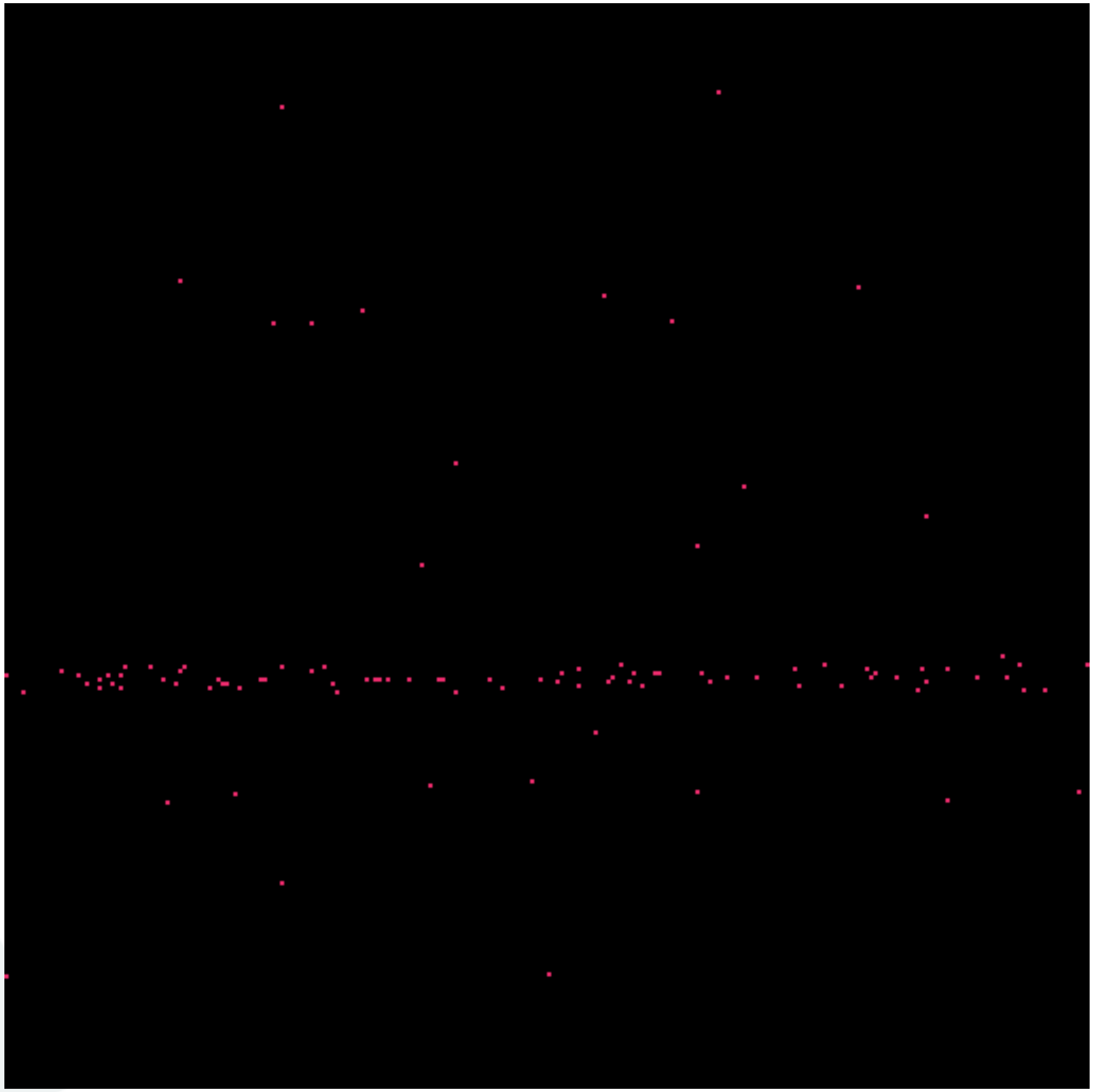
- what is the relative position of kernel zones to each other?
- what is the relative position of pages inside the same kernel zone?
- is it possible to overflow from one kernel zone into another?

Visualization of Zone Page Allocations

- we allocated about 48MB of kernel memory through **single page zones**
- all returned memory is between 0x80000000 and 0x8FFFFFFF
- we visualize the pages returned by the kernel zone allocator

you will observe a different result when looking at allocations > 1 PAGE

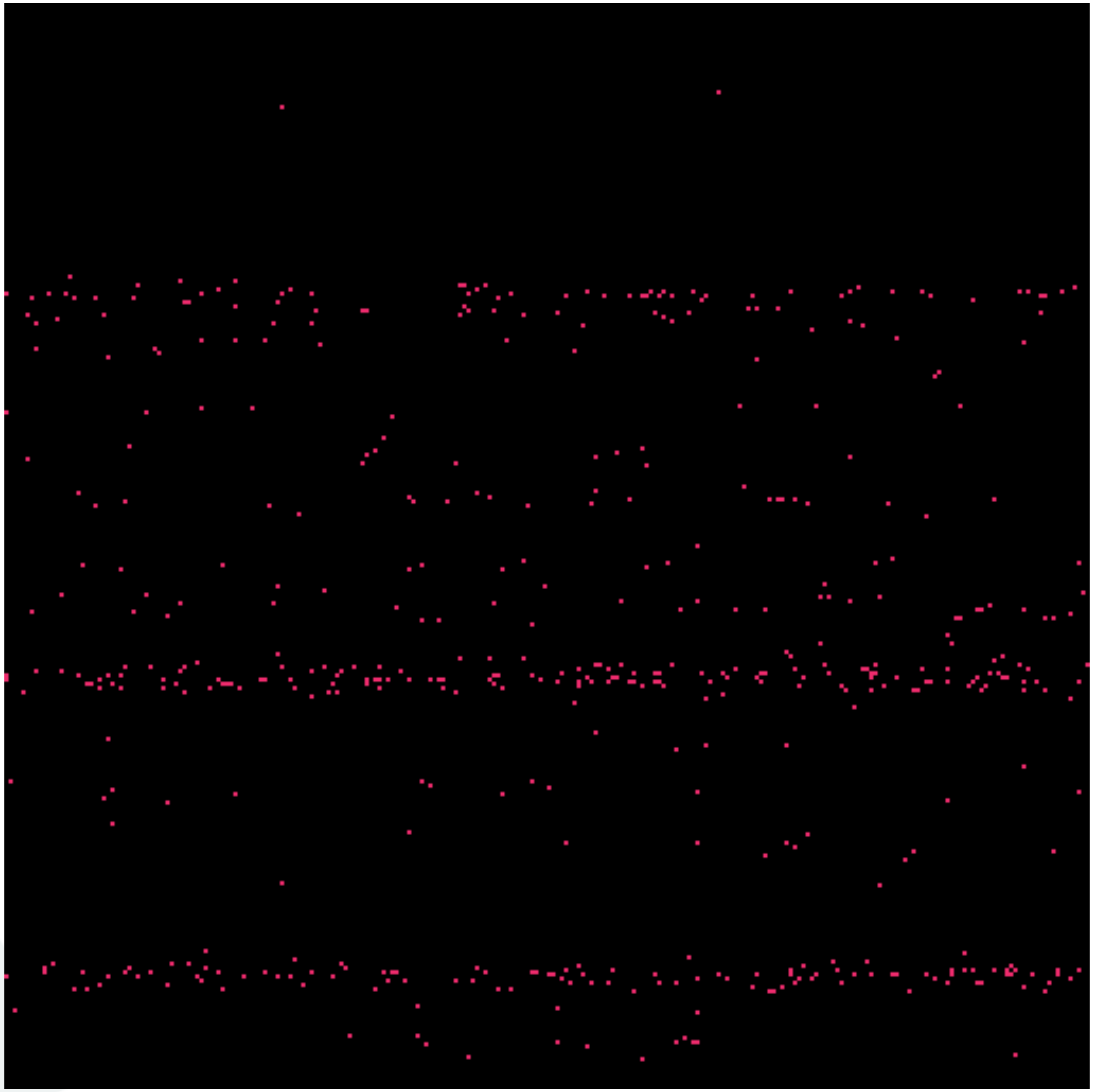
Visualization of Zone Page Allocations



after 100 allocations

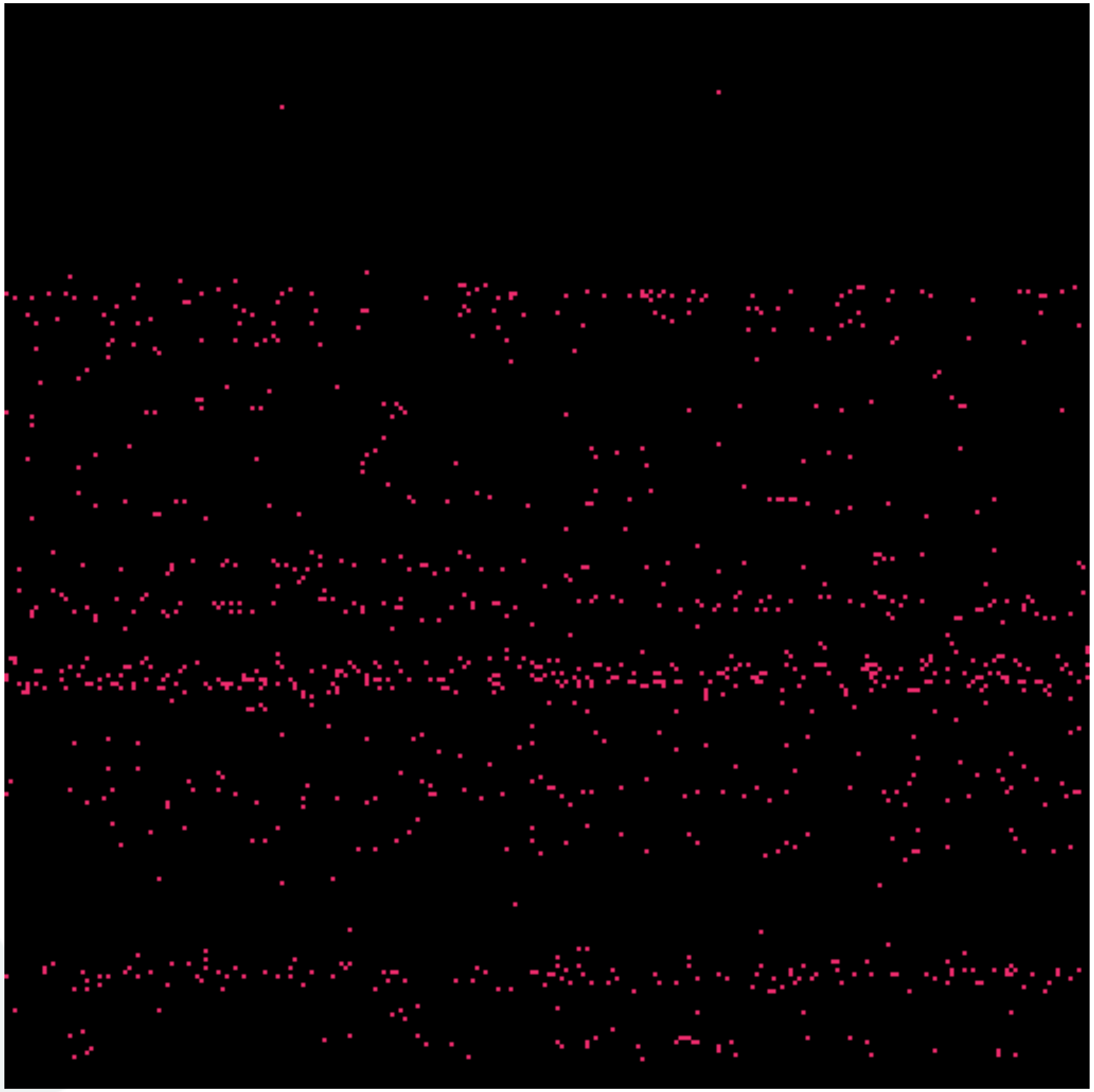
SektionEins

Visualization of Zone Page Allocations



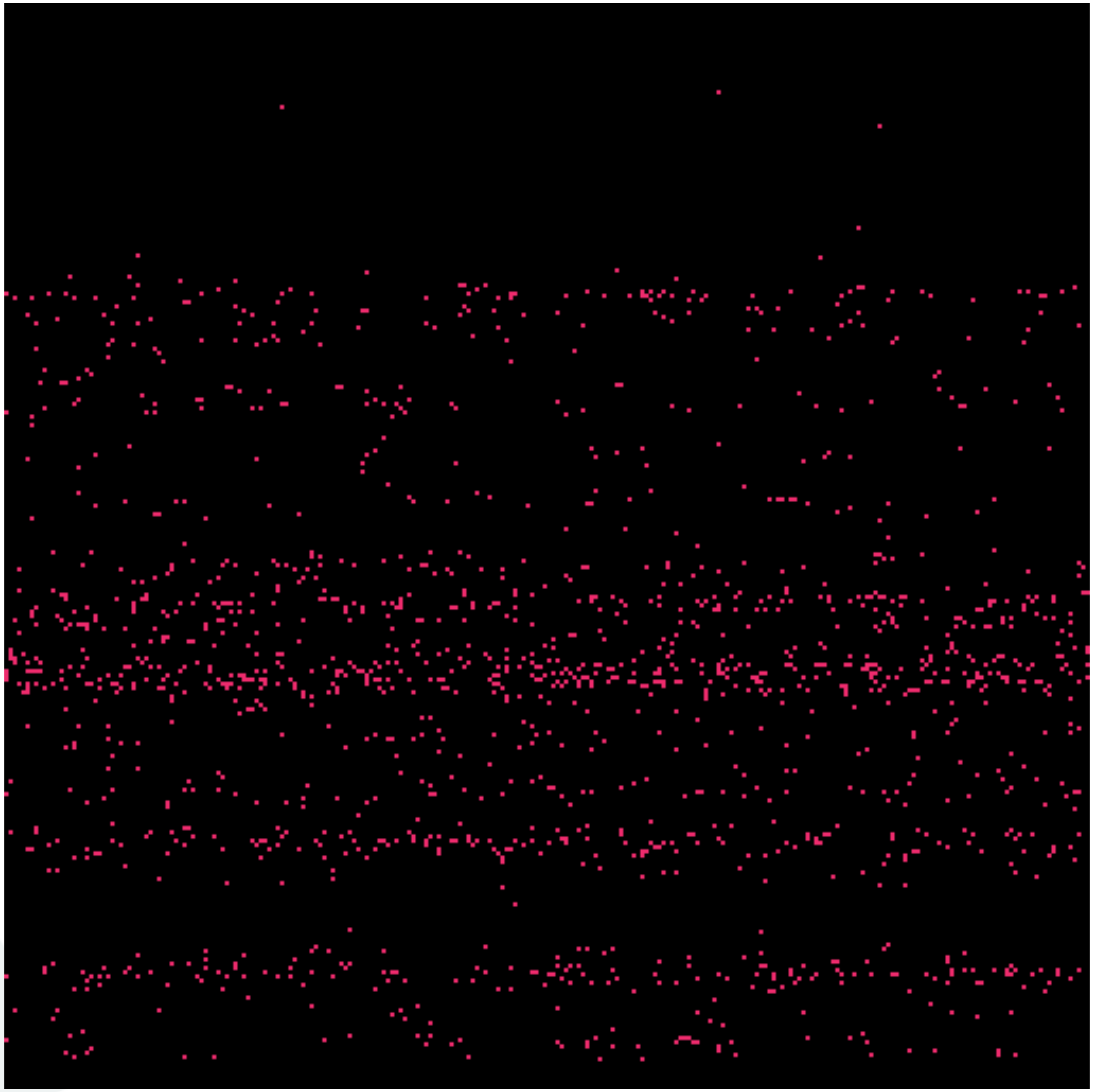
after 500 allocations

Visualization of Zone Page Allocations



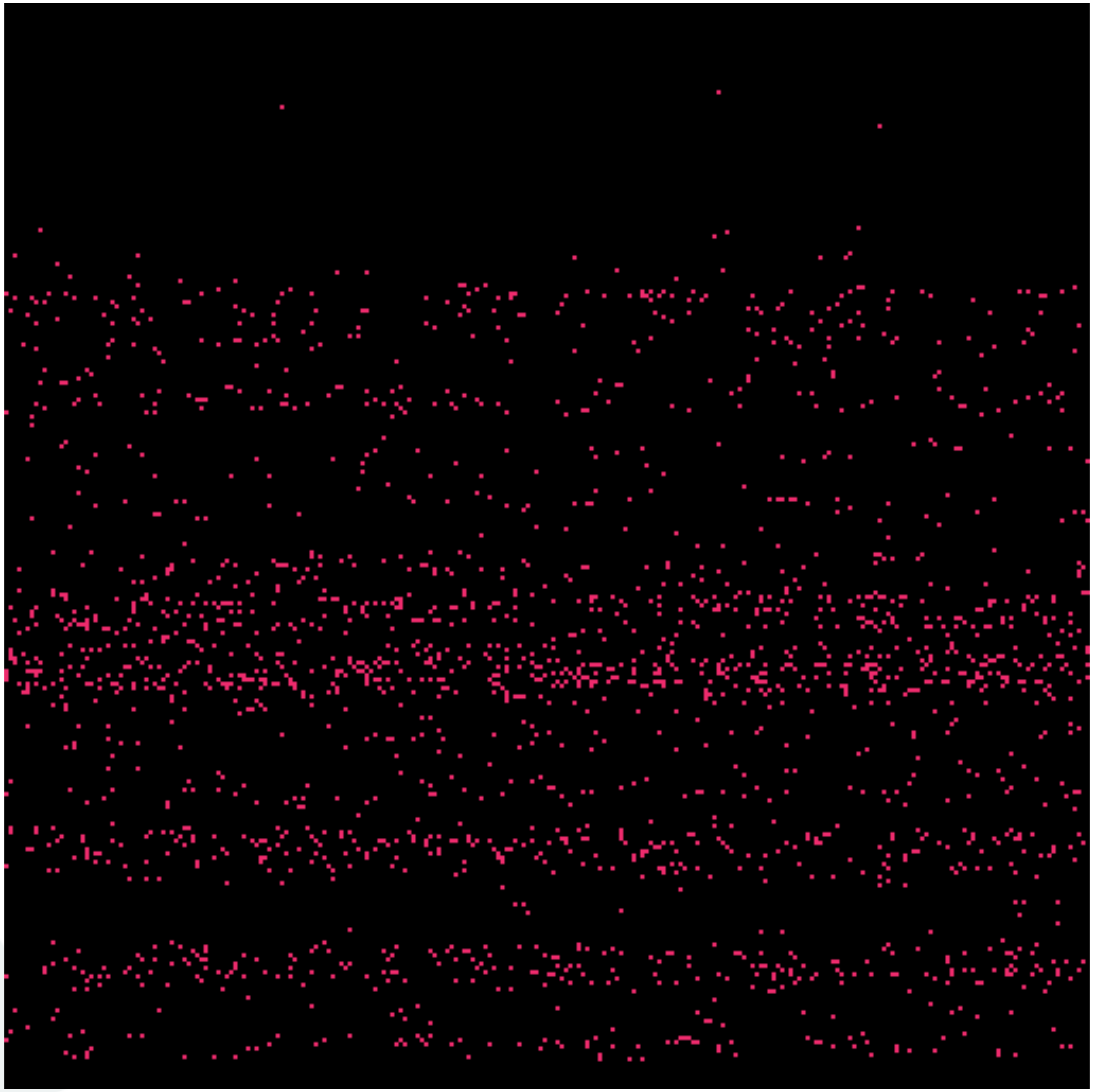
after 1000 allocations

Visualization of Zone Page Allocations



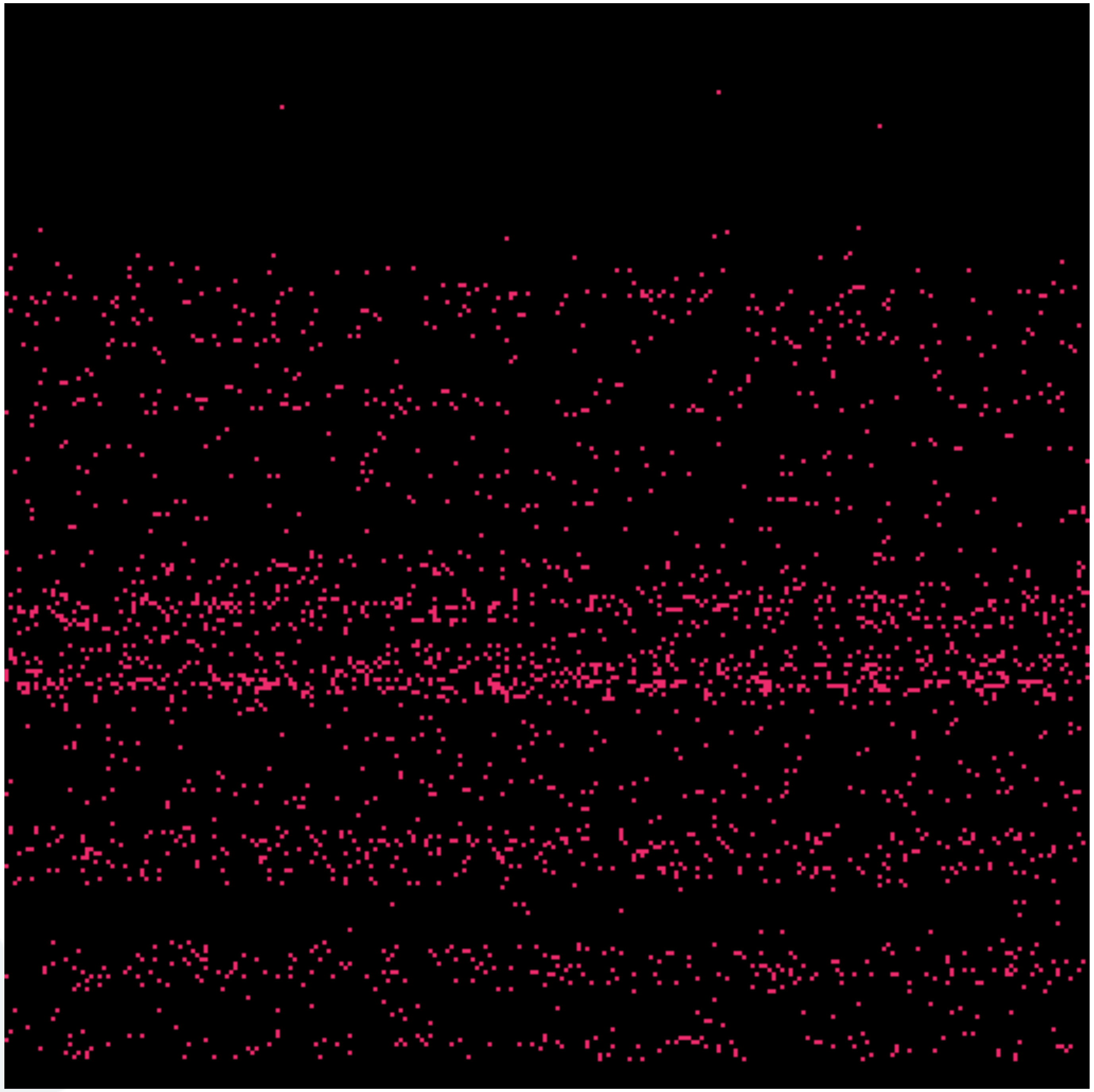
after 1500 allocations

Visualization of Zone Page Allocations



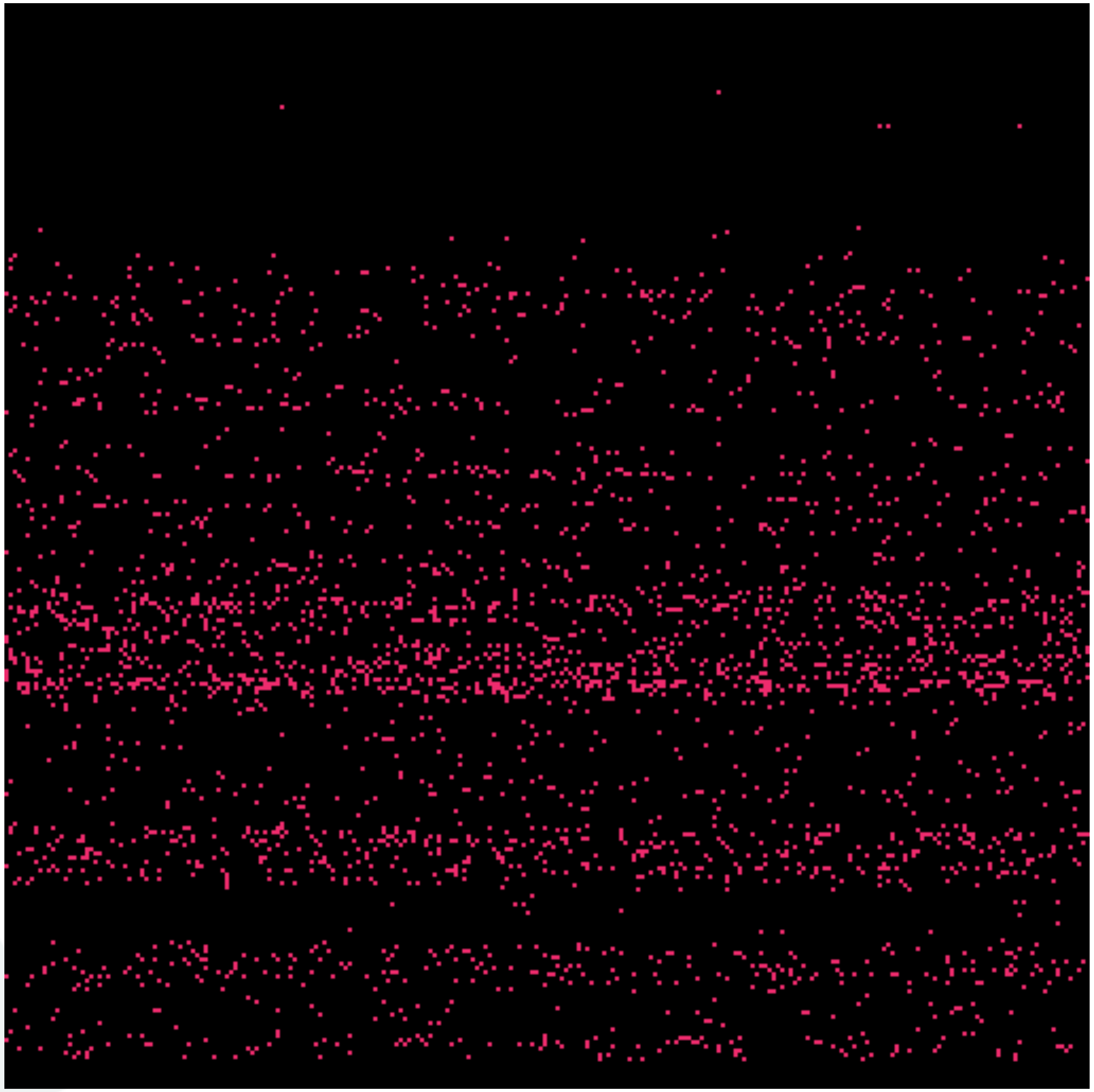
after 2000 allocations

Visualization of Zone Page Allocations



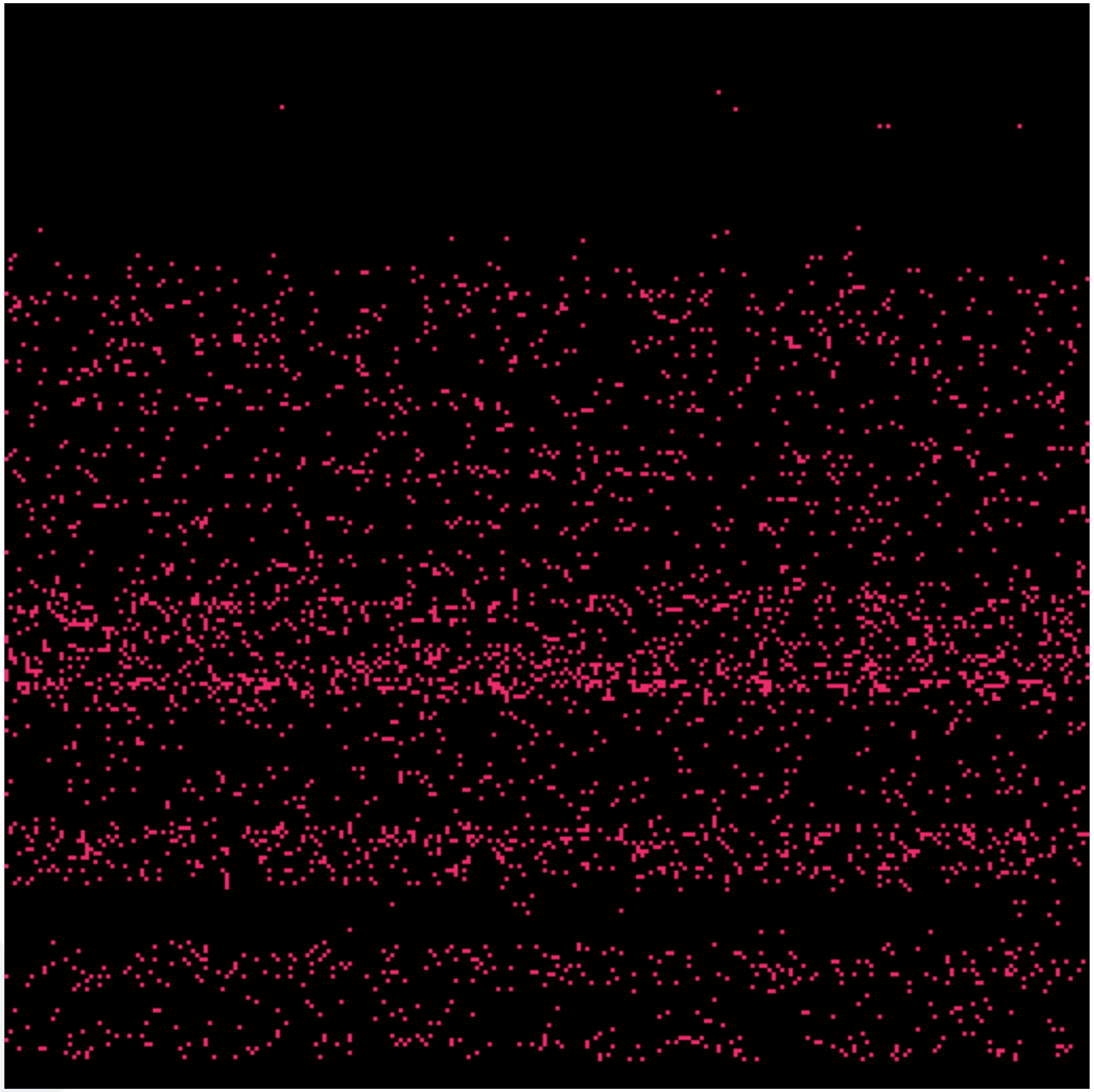
after 2500 allocations

Visualization of Zone Page Allocations



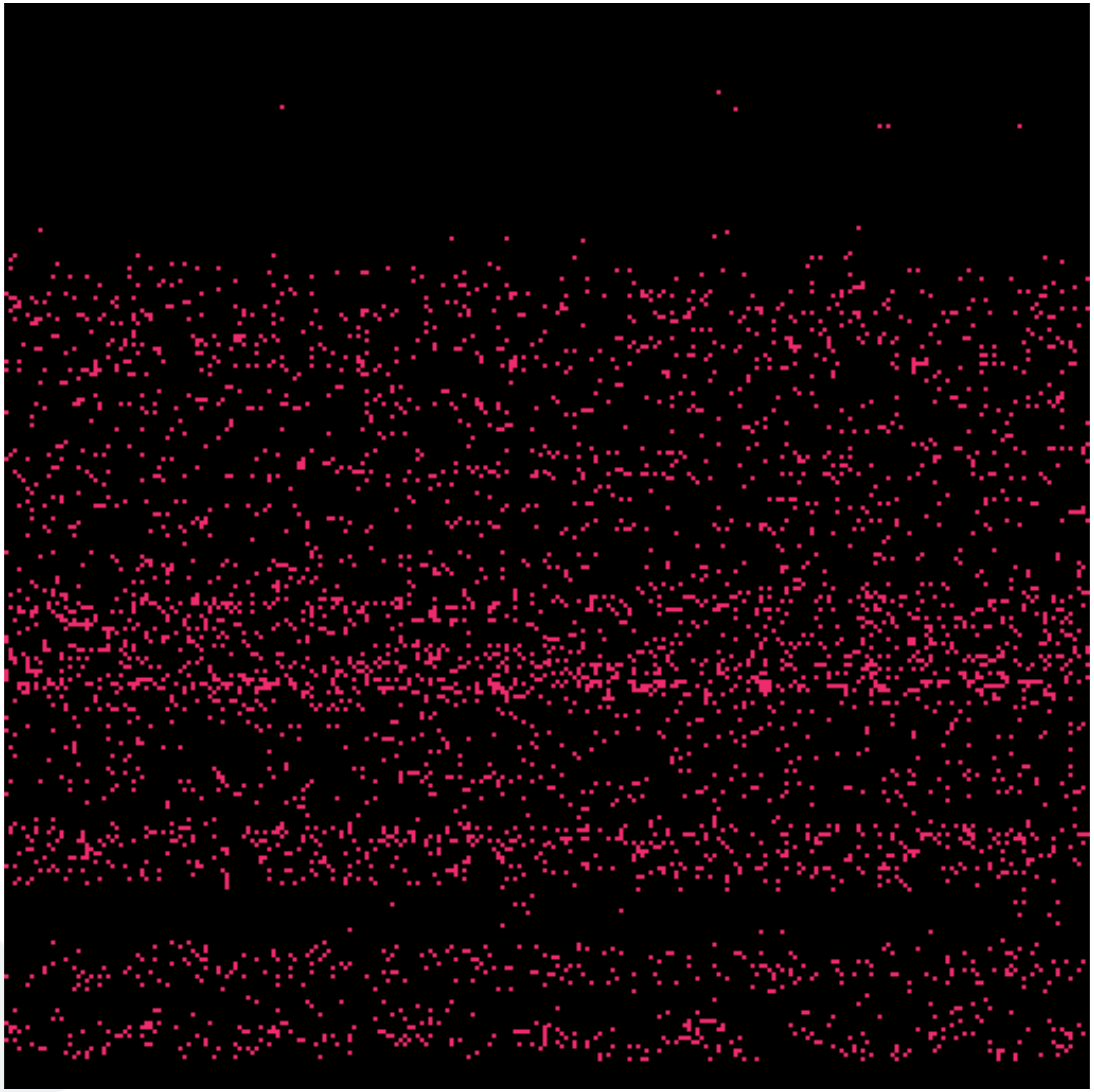
after 3000 allocations

Visualization of Zone Page Allocations



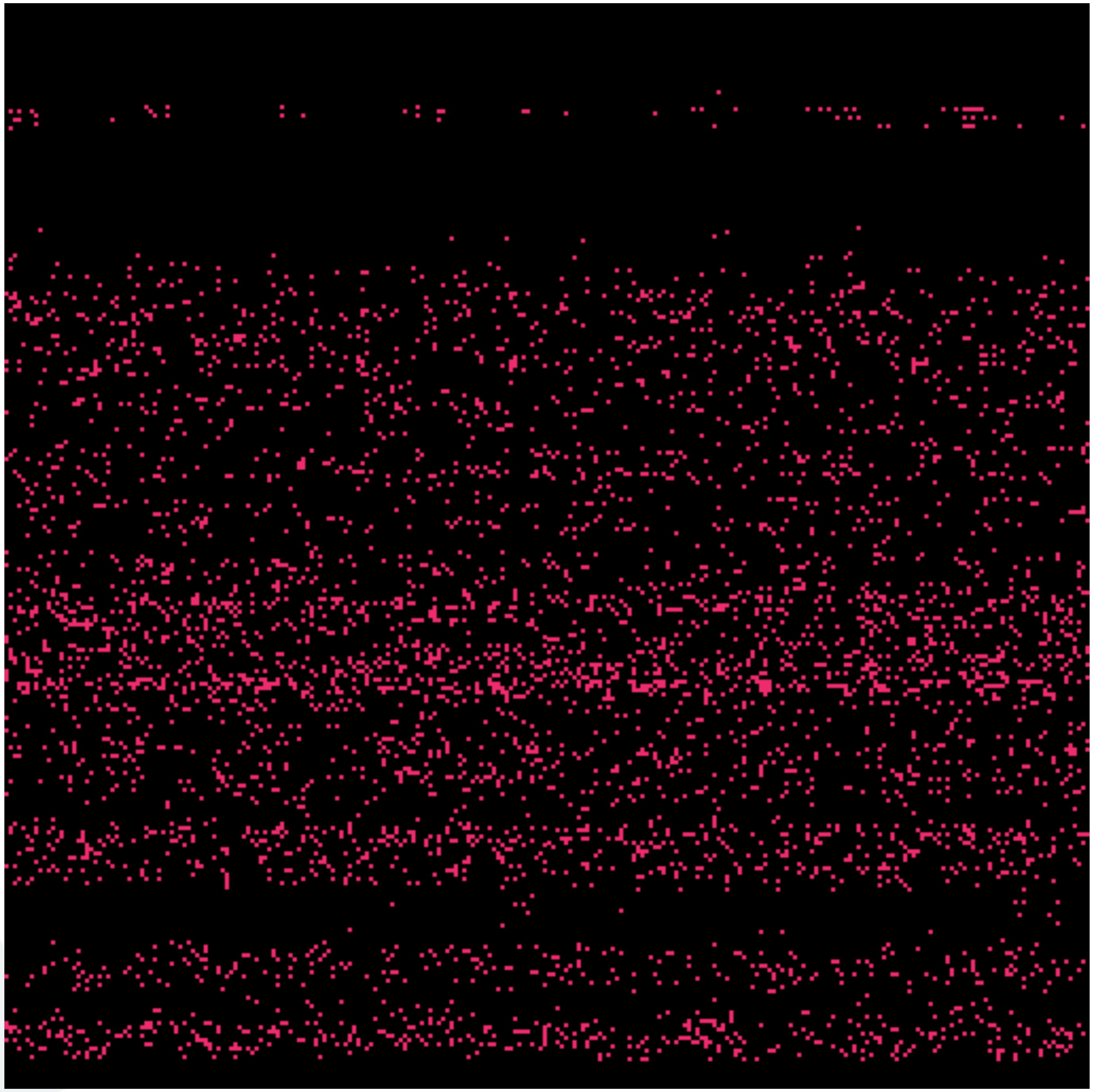
after 3500 allocations

Visualization of Zone Page Allocations



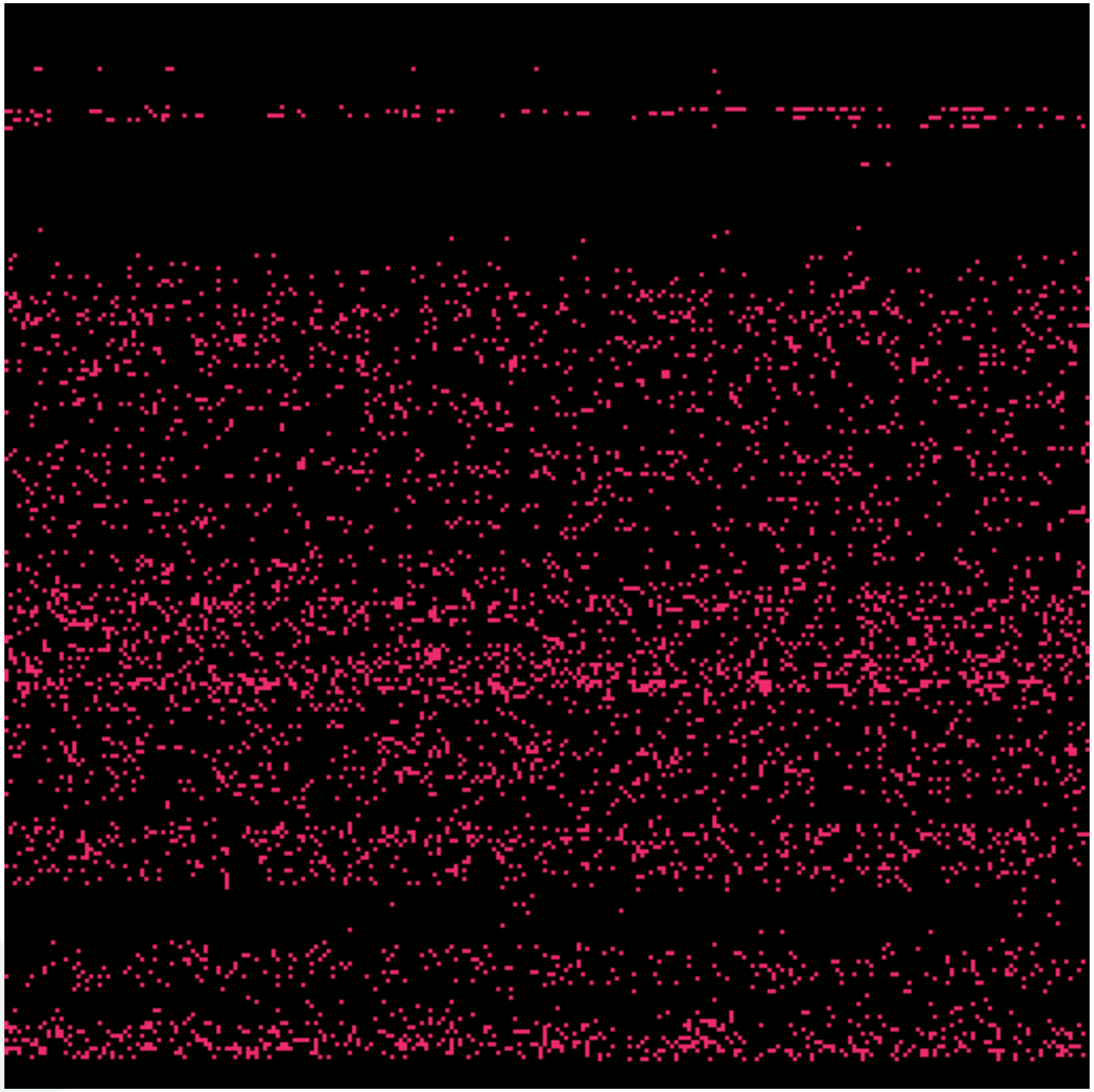
after 4000 allocations

Visualization of Zone Page Allocations



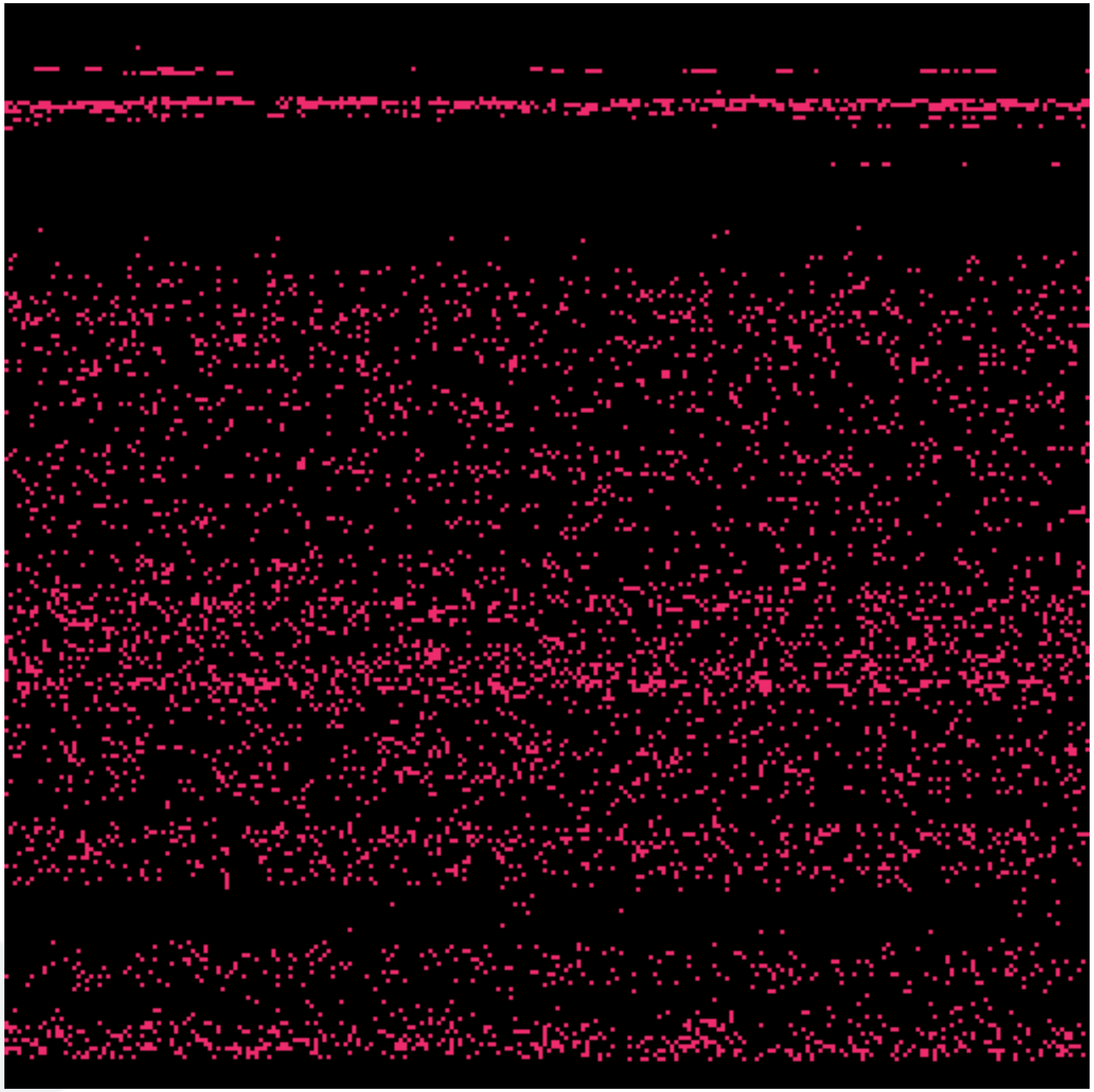
after 4500 allocations

Visualization of Zone Page Allocations



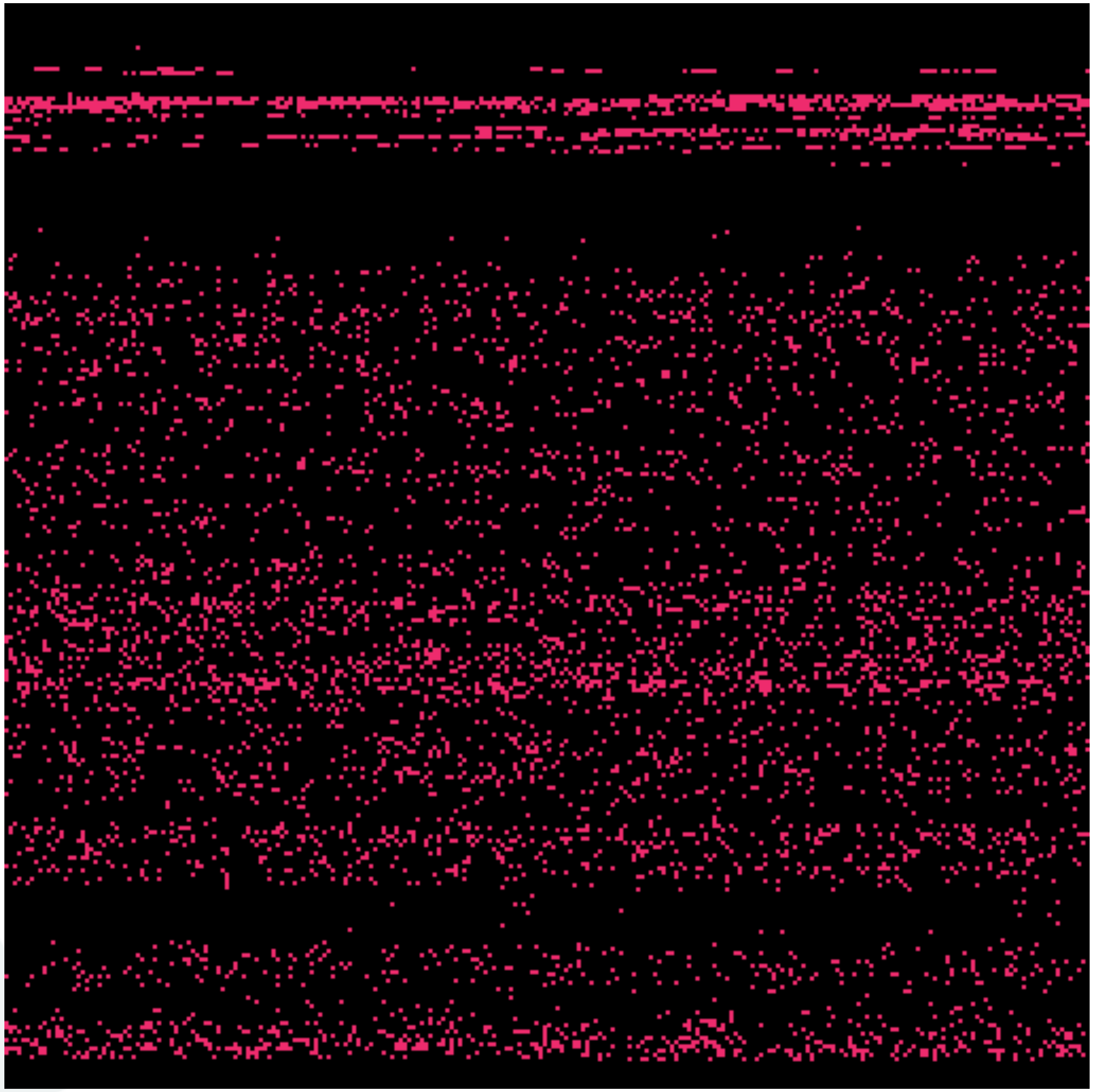
after 5000 allocations

Visualization of Zone Page Allocations



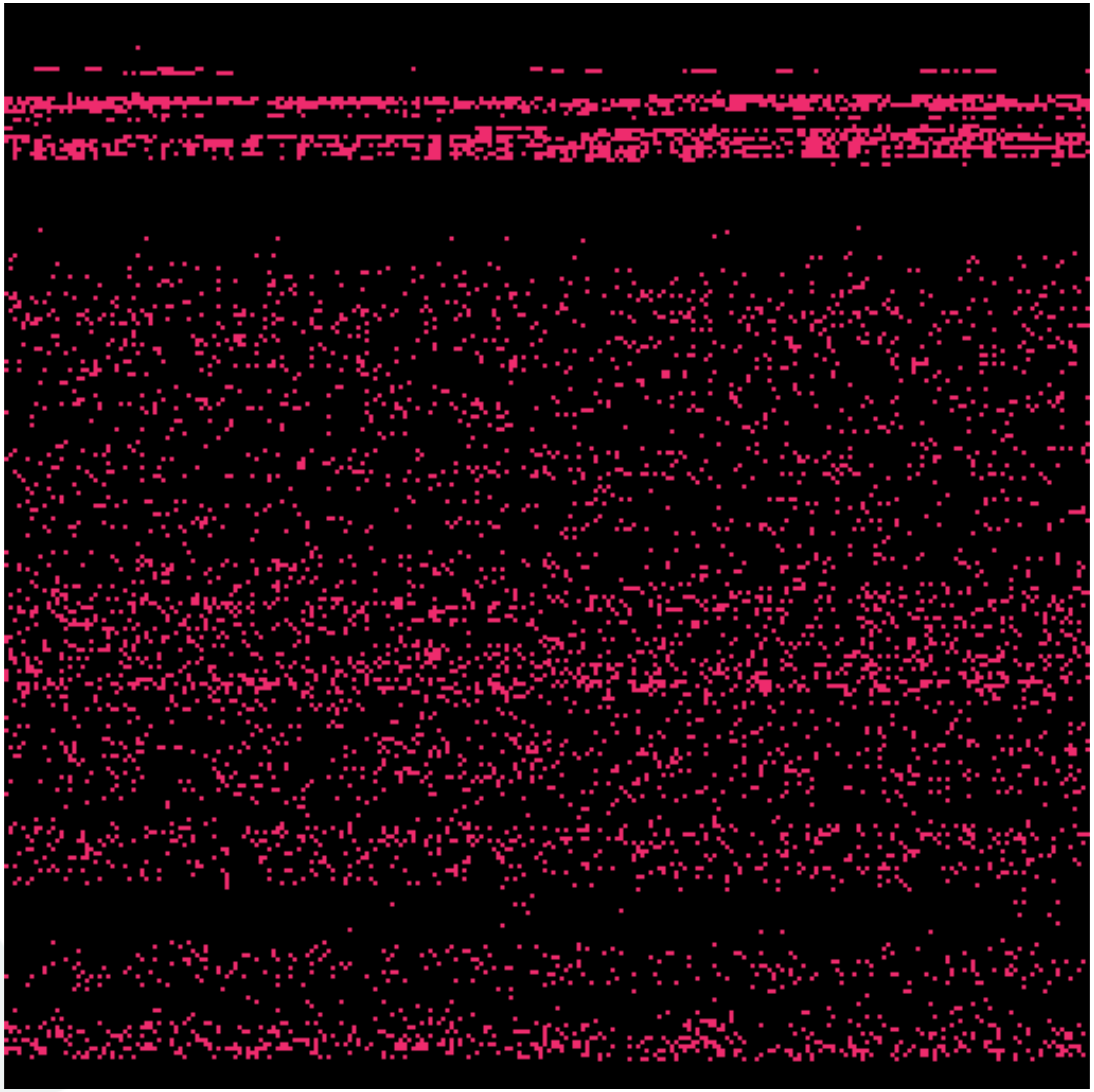
after 5500 allocations

Visualization of Zone Page Allocations



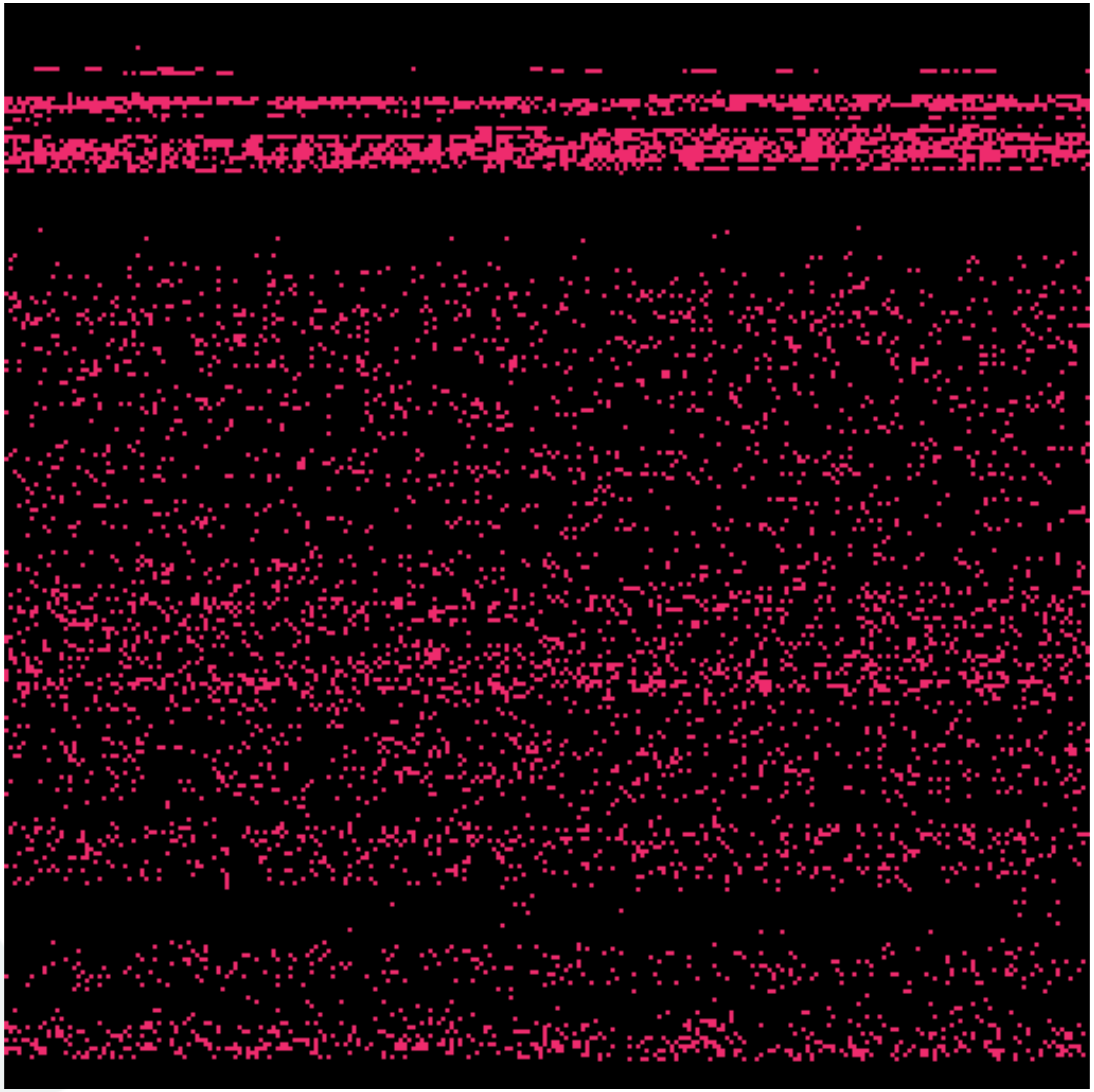
after 6000 allocations

Visualization of Zone Page Allocations



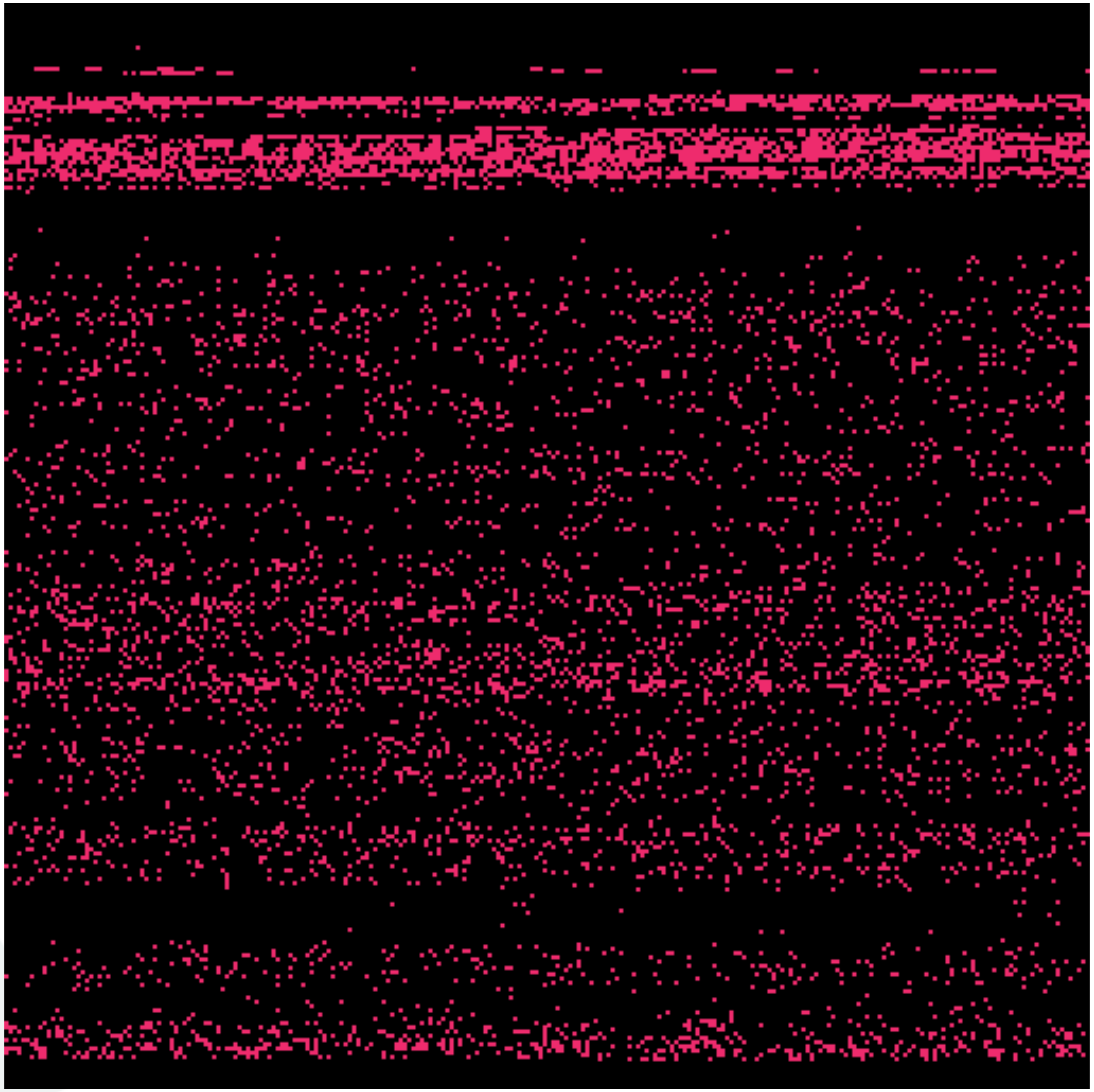
after 6500 allocations

Visualization of Zone Page Allocations



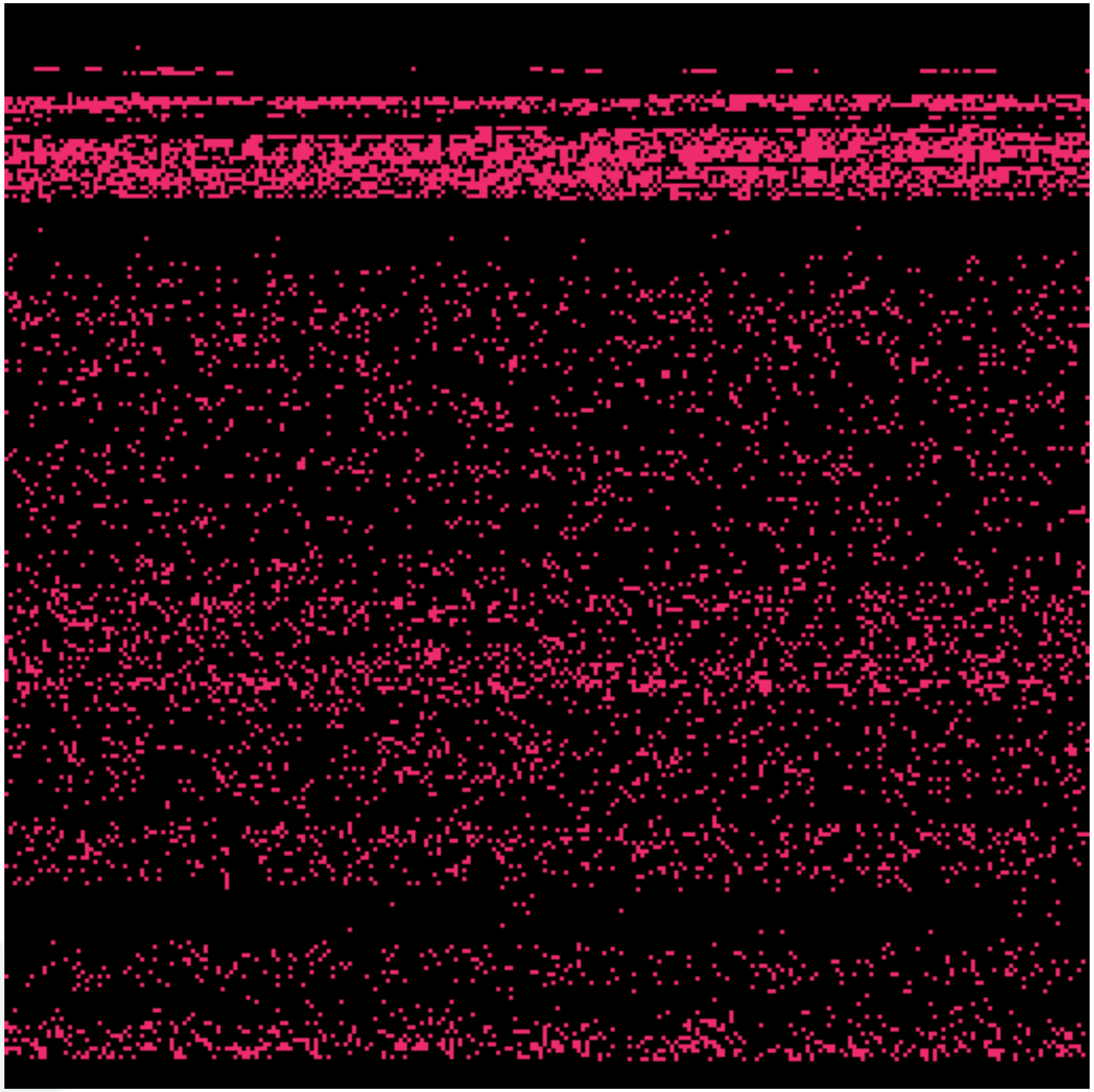
after 7000 allocations

Visualization of Zone Page Allocations



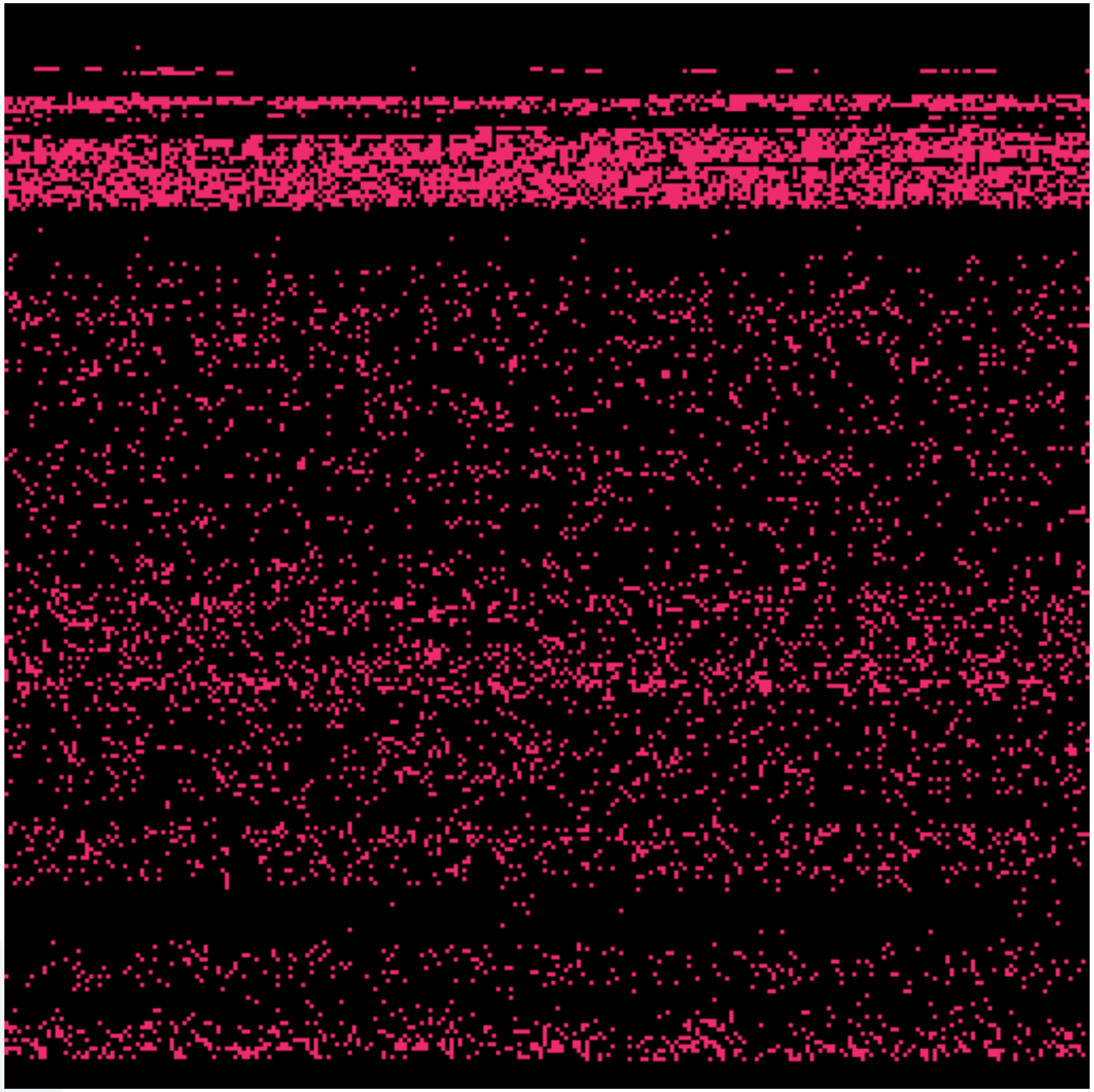
after 7500 allocations

Visualization of Zone Page Allocations



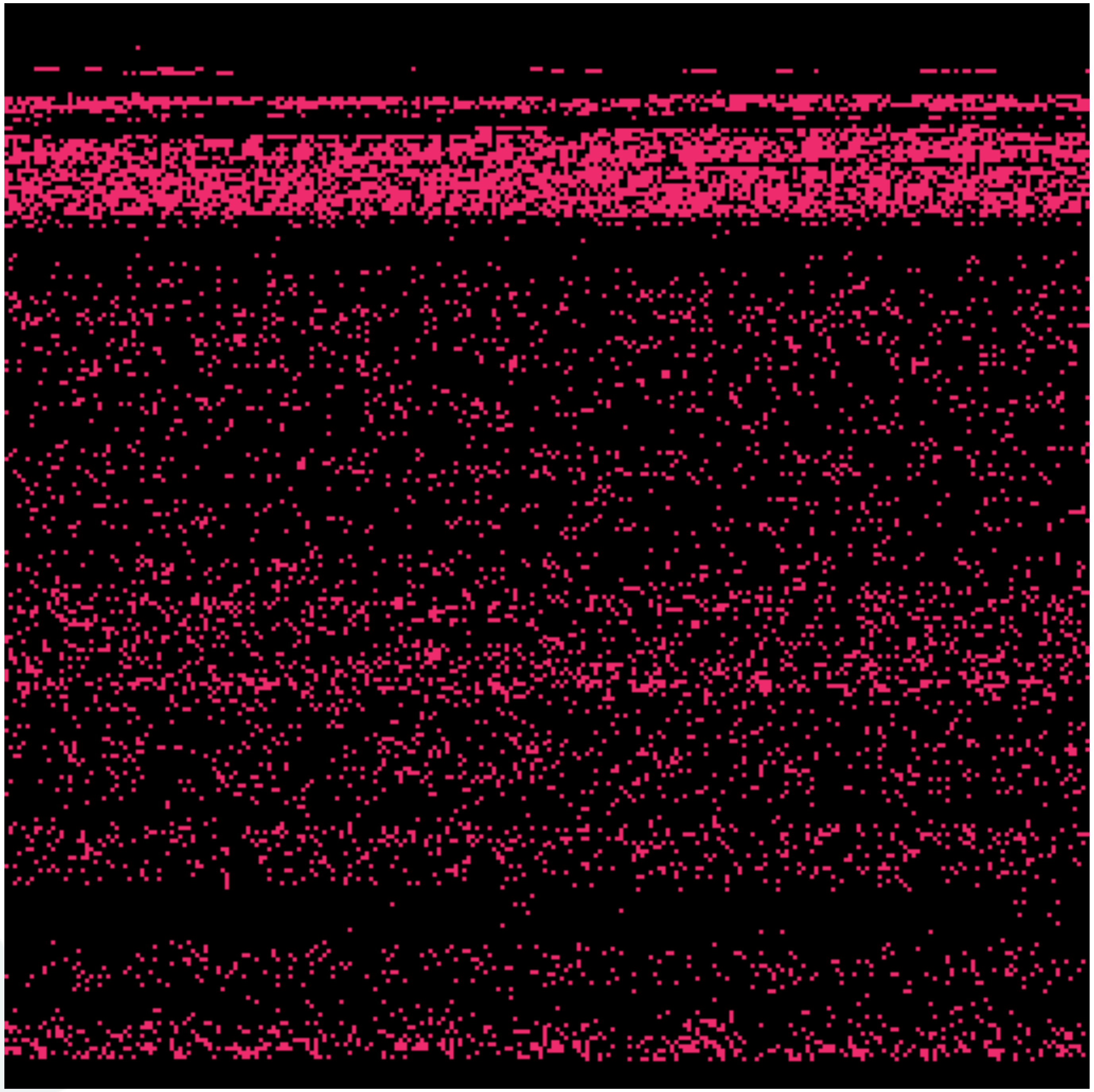
after 8000 allocations

Visualization of Zone Page Allocations



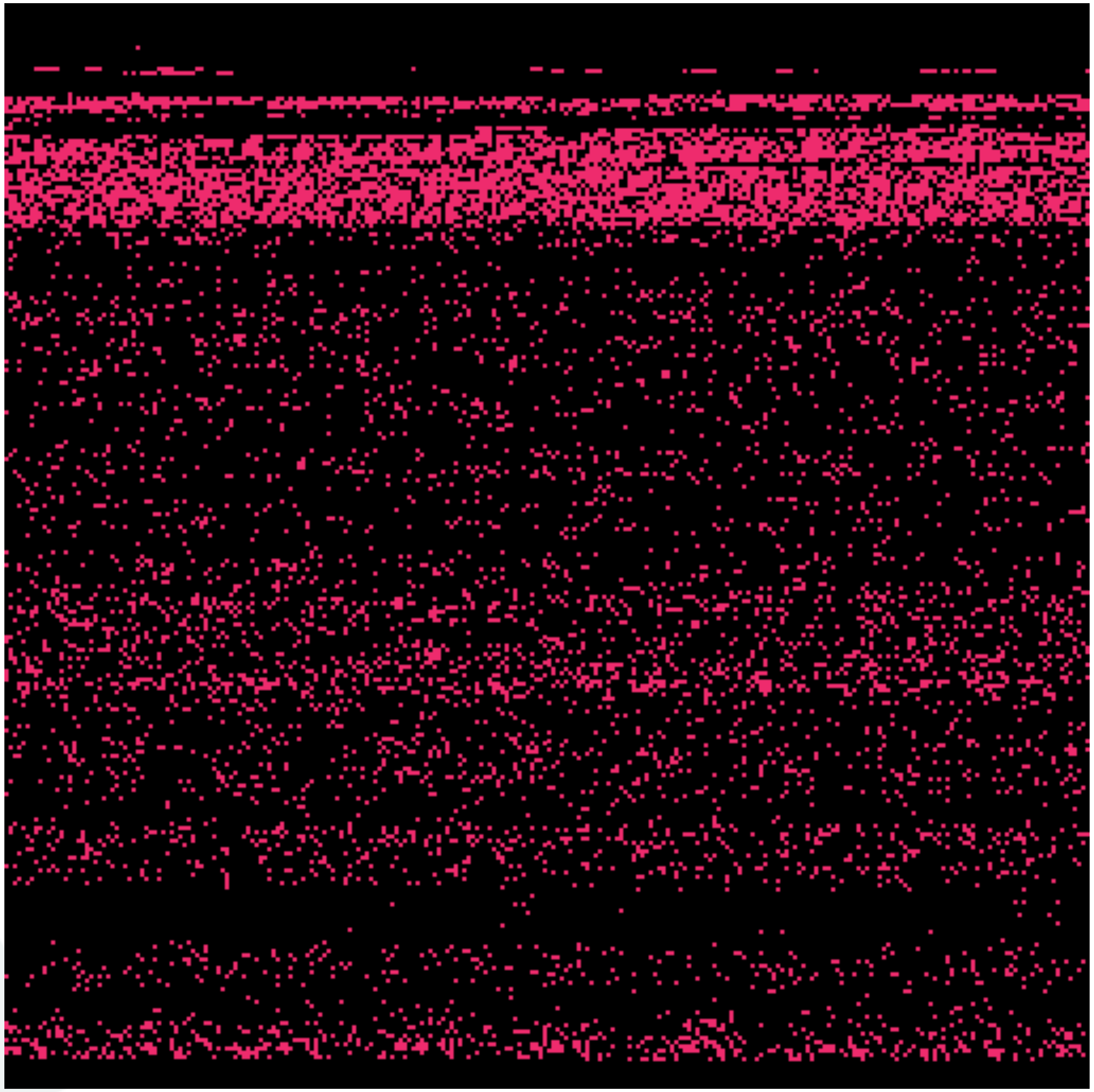
after 8500 allocations

Visualization of Zone Page Allocations



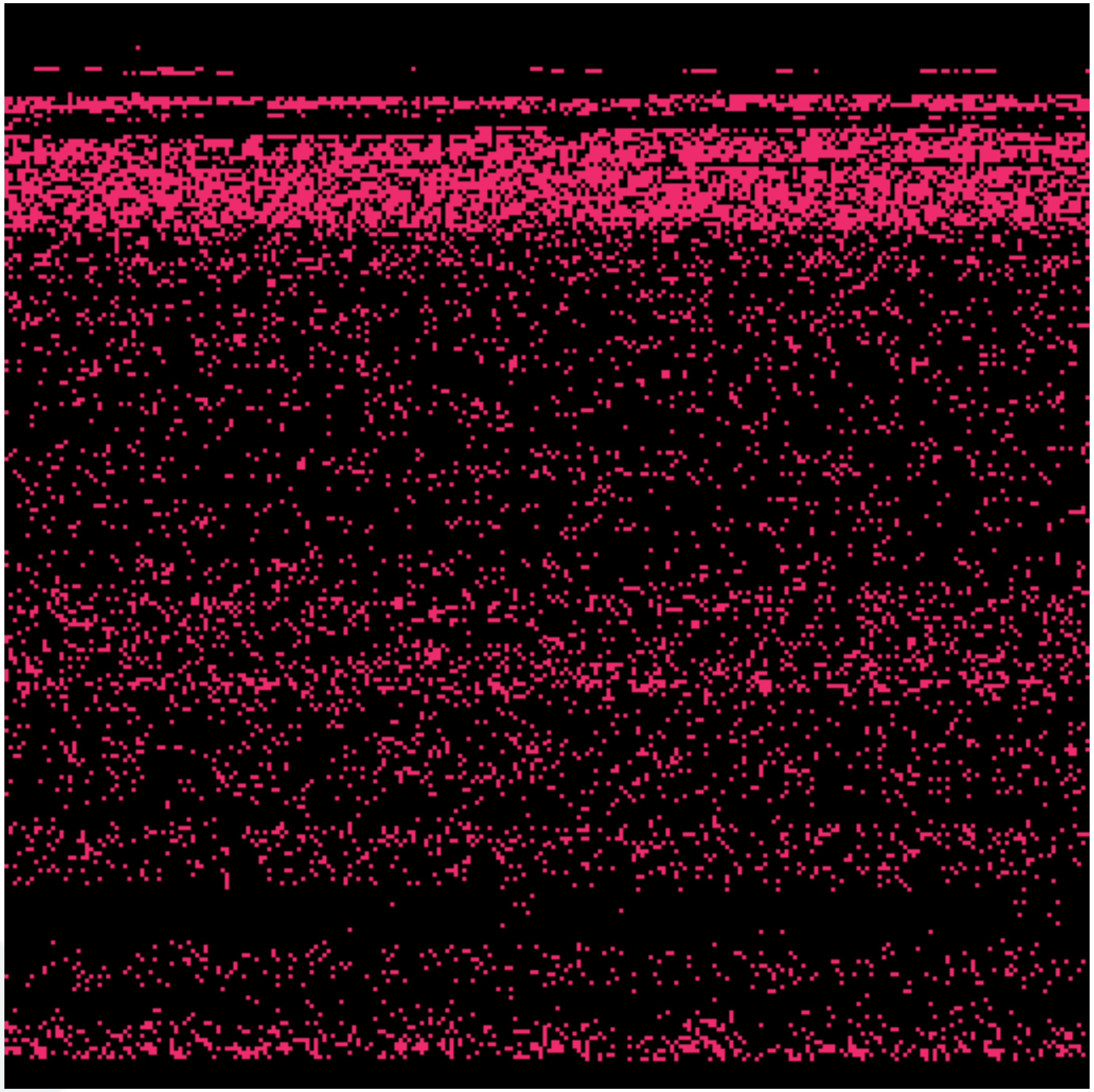
after 9000 allocations

Visualization of Zone Page Allocations



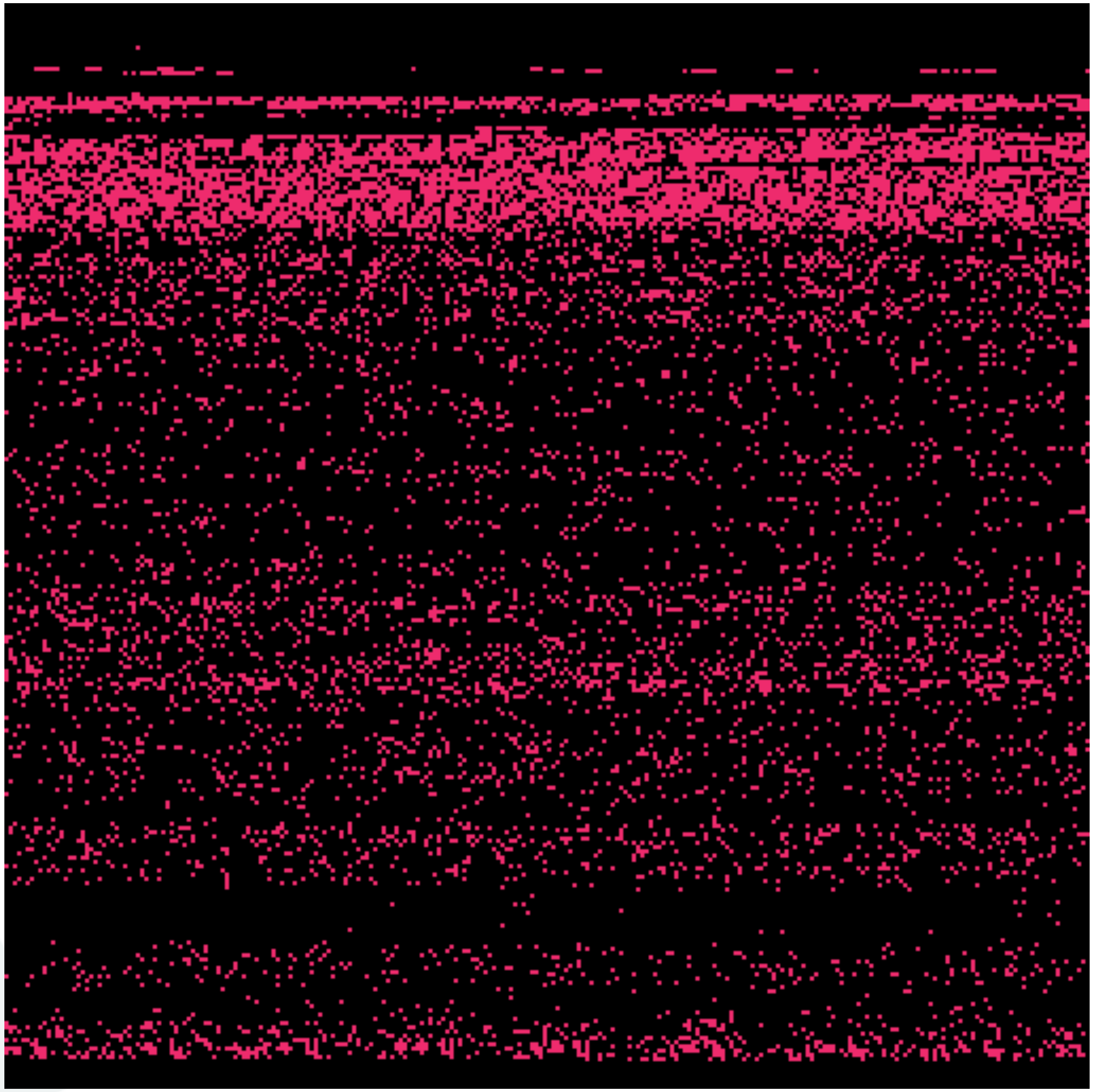
after 9500 allocations

Visualization of Zone Page Allocations



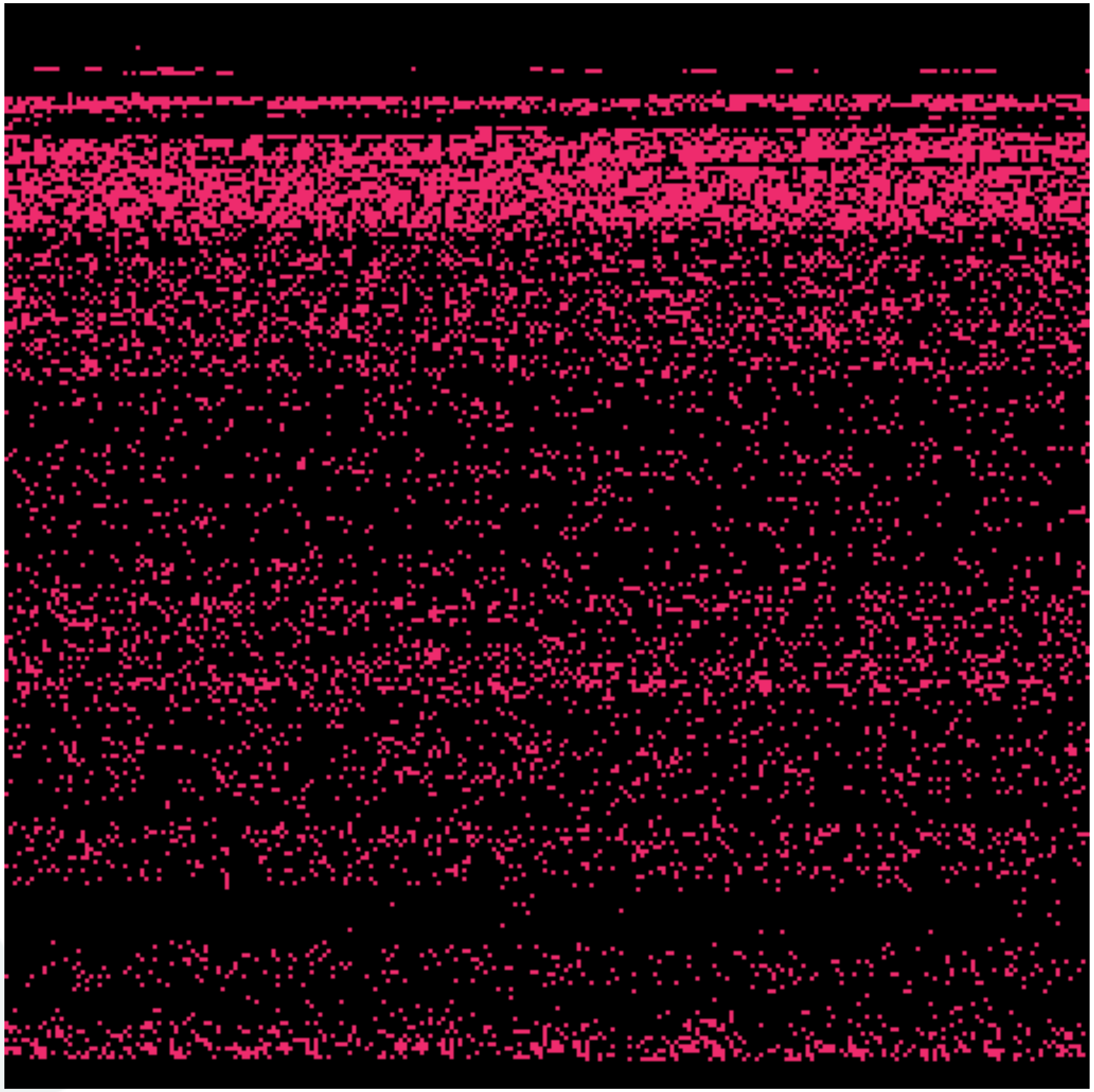
after 10000 allocations

Visualization of Zone Page Allocations



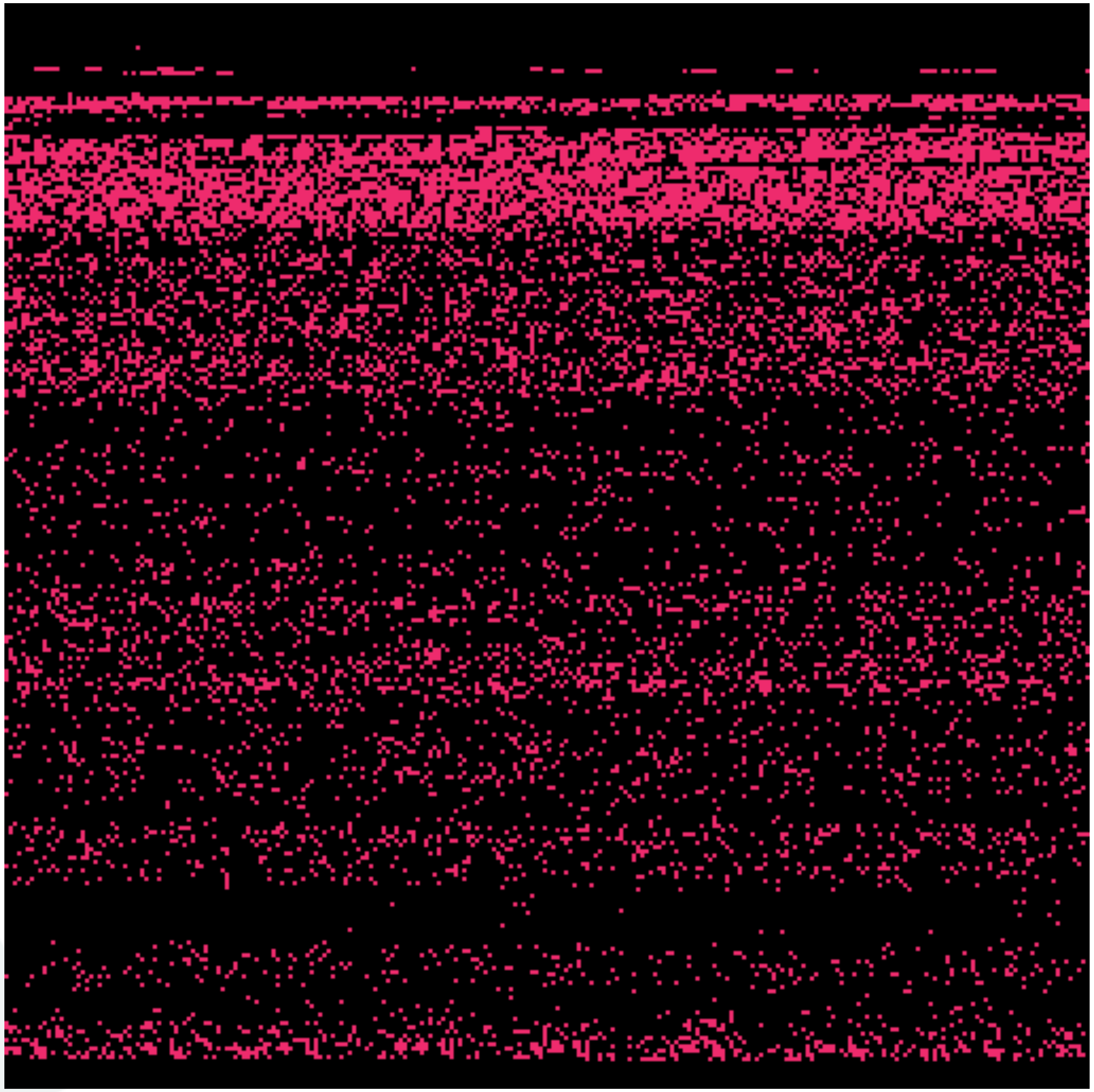
after 10500 allocations

Visualization of Zone Page Allocations



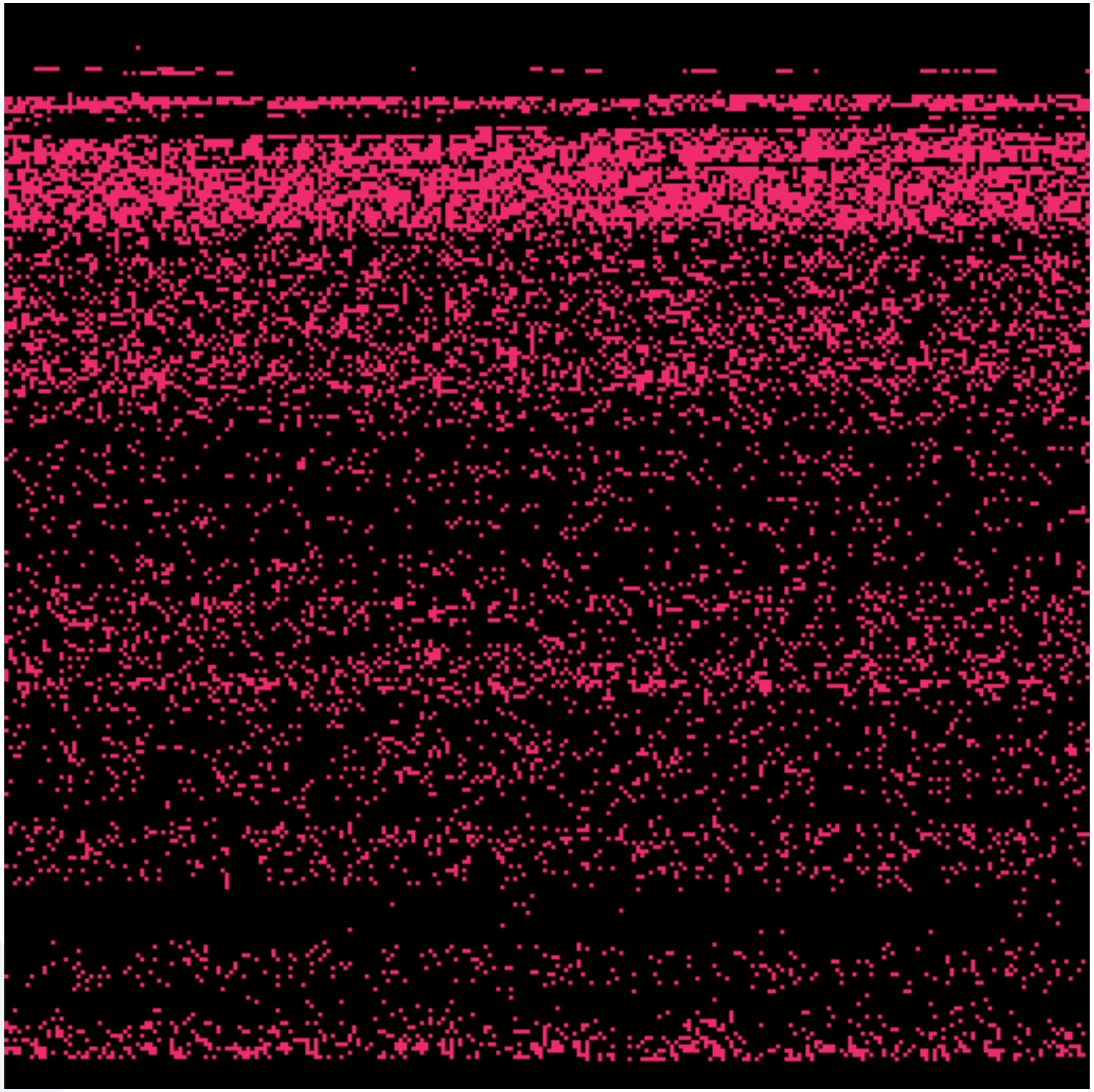
after 11000 allocations

Visualization of Zone Page Allocations



after 11500 allocations

Visualization of Zone Page Allocations



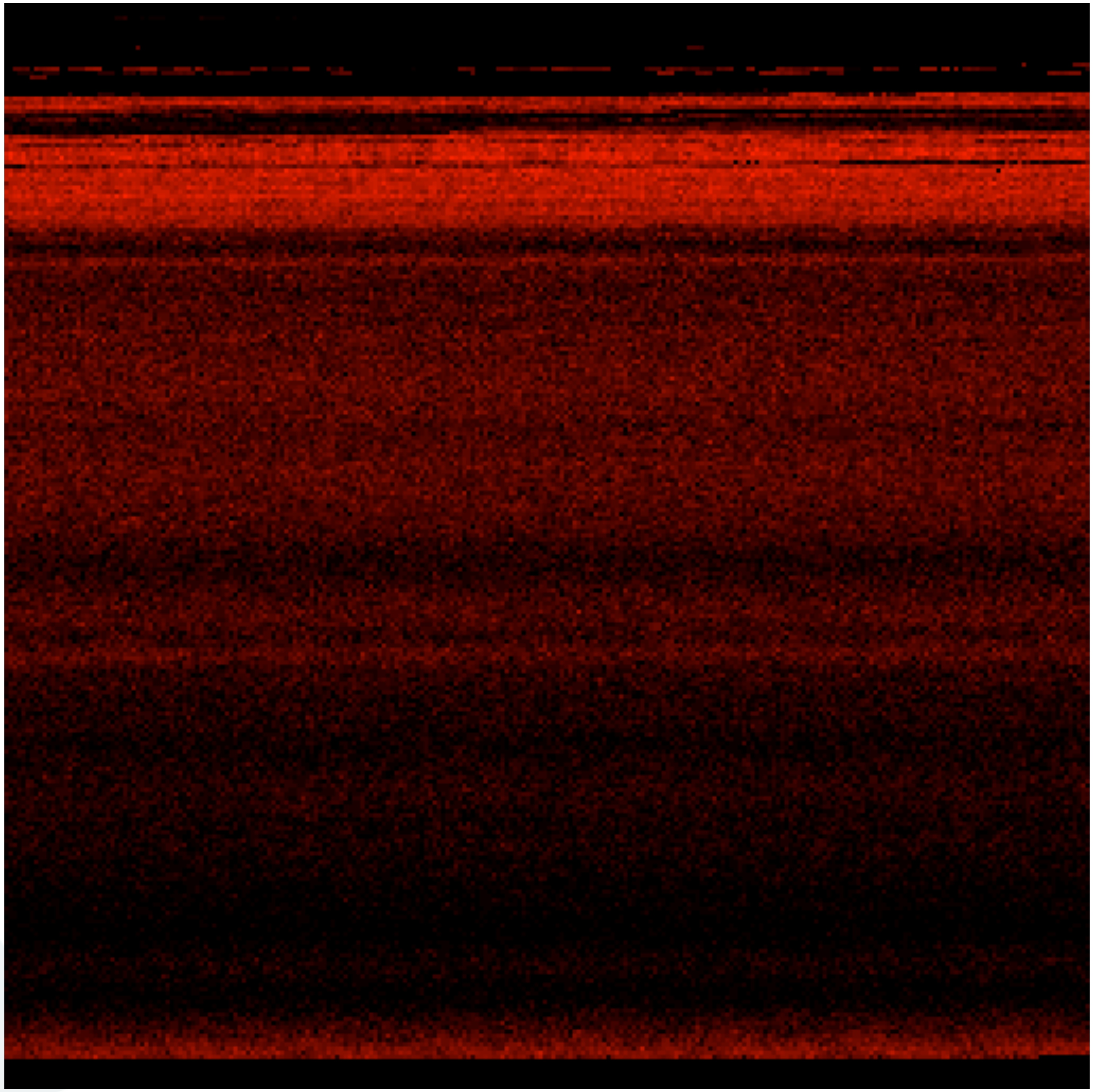
after 11800 allocations

SektionEins

Zone Page Allocation Distribution

- zone page allocator seems to be random
- but several clusters in the beginning of the address space and end
- but that was only one run
- so lets do an average across 25 reboots

Zone Page Allocation Distribution (across reboots)



after 11800 allocations

Zone Page Allocation Distribution

- accross 25 reboots there was a single common page among all the allocations
- the 26th reboot made it go away
- because of the randomness adjacent memory pages are very unlikely
- it is not possible to say anything about the relative position of pages
- overflowing out of a page will most likely crash

Cross Memory Allocator Attacks

- most of the allocation functions deeply down use the zone allocator
- if allocation functions share the same zone then cross attacks are possible
- everything based on kalloc() is affected

- e.g. ***new , kern_os_malloc, _MALLOC, kalloc***

Part IV

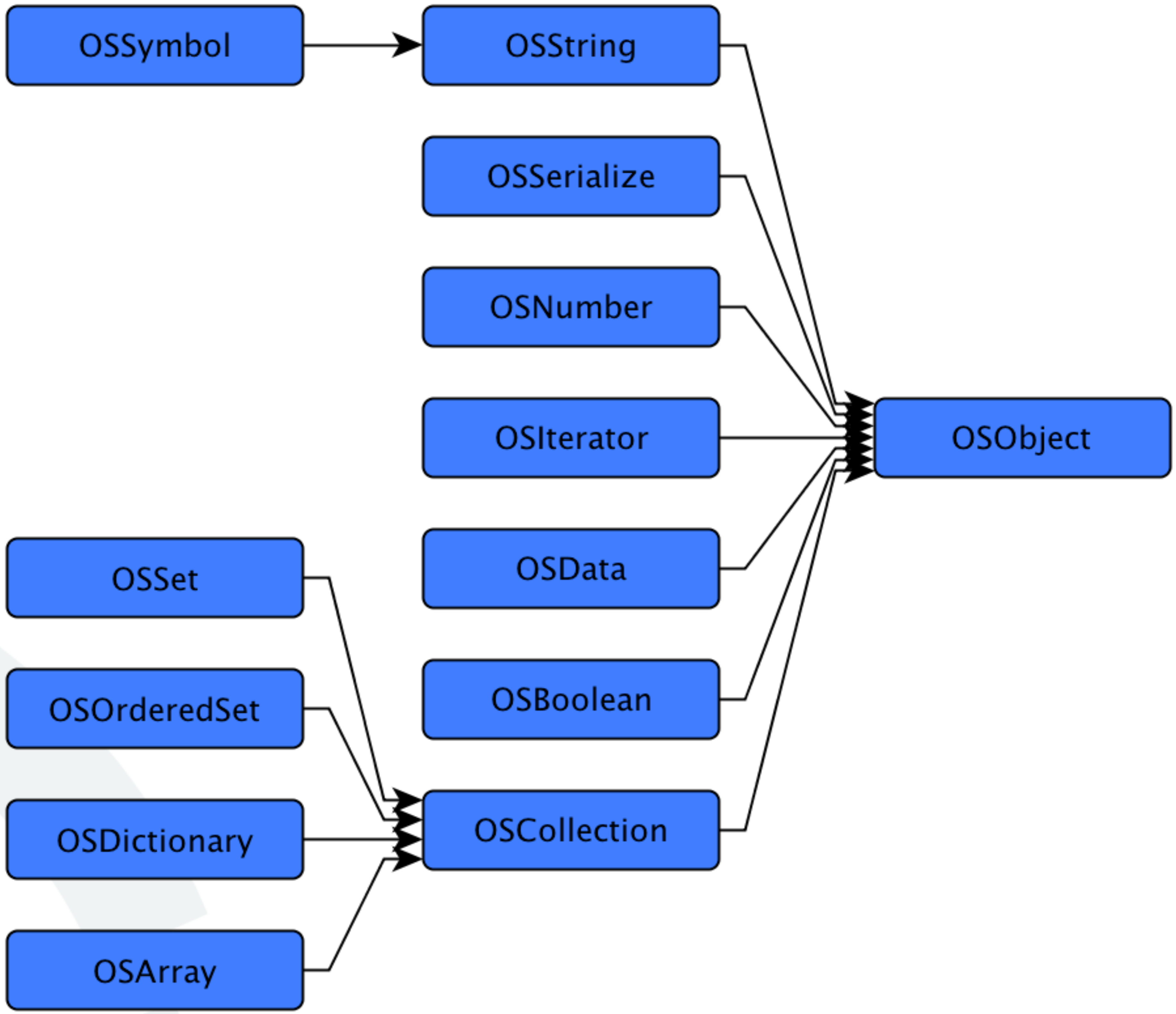
Kernel Heap Application Data Overwrites

(a kernel c++ object case study)

iOS Kernel C++

- iOS kernel's libkern supports a subset of C++
- allows kernel drivers to be C++
- and indeed only used by kernel drivers - mostly IOKit
- brings C++ vulnerability classes to the iOS kernel
- libkern C++ runtime comes with a set of base object

iOS Kernel C++ Base Objects



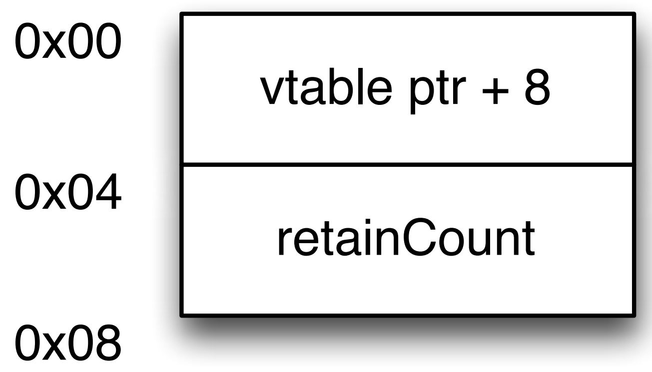
OSObject Memory Layout

0x00	vtable ptr + 8
0x04	retainCount
0x08	

```
802A1570 ; `vtable for'OSObject
802A1570 __ZTV8OSObject
802A1570      DCD 0
802A1574      DCD 0
802A1578      DCD sub_801E7C0C+1
802A157C      DCD __ZN8OSObjectD0Ev+1
802A1580      DCD __ZNK8OSObject7releaseEi+1
802A1584      DCD __ZNK8OSObject14getRetainCountEv+1
802A1588      DCD __ZNK8OSObject6retainEv+1
802A158C      DCD __ZNK8OSObject7releaseEv+1
802A1590      DCD __ZNK8OSObject9serializeEP11OSSerialize+1
802A1594      DCD __ZNK8OSObject12getMetaClassEv+1
802A1598      DCD __ZNK15OSMetaClassBase9isEqualToEPKS_+1
802A159C      DCD __ZNK8OSObject12taggedRetainEPKv+1
802A15A0      DCD __ZNK8OSObject13taggedReleaseEPKv+1
802A15A4      DCD __ZNK8OSObject13taggedReleaseEPKvi+1
802A15A8      DCD __ZN8OSObject4initEv+1
802A15AC      DCD __ZN8OSObject4freeEv+1
```

OSObject Retain Count

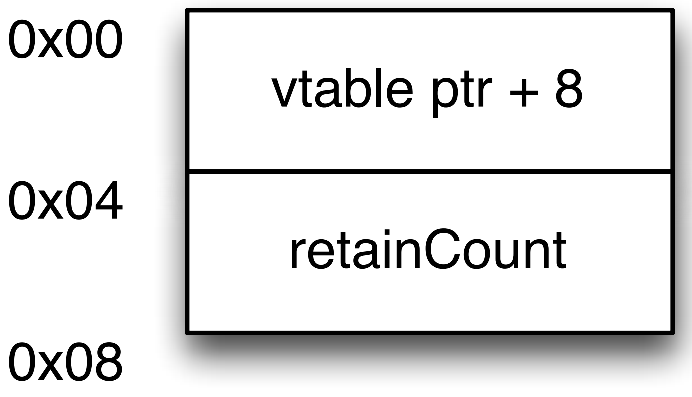
- reference counter for objects
- 32 bit field - but only lower 16 bit are the reference counter
- upper 16 bit used as collection reference counter
- reference counting stops at 65534 -> memory leak



SektionEins

Overwriting an OSObject in Memory

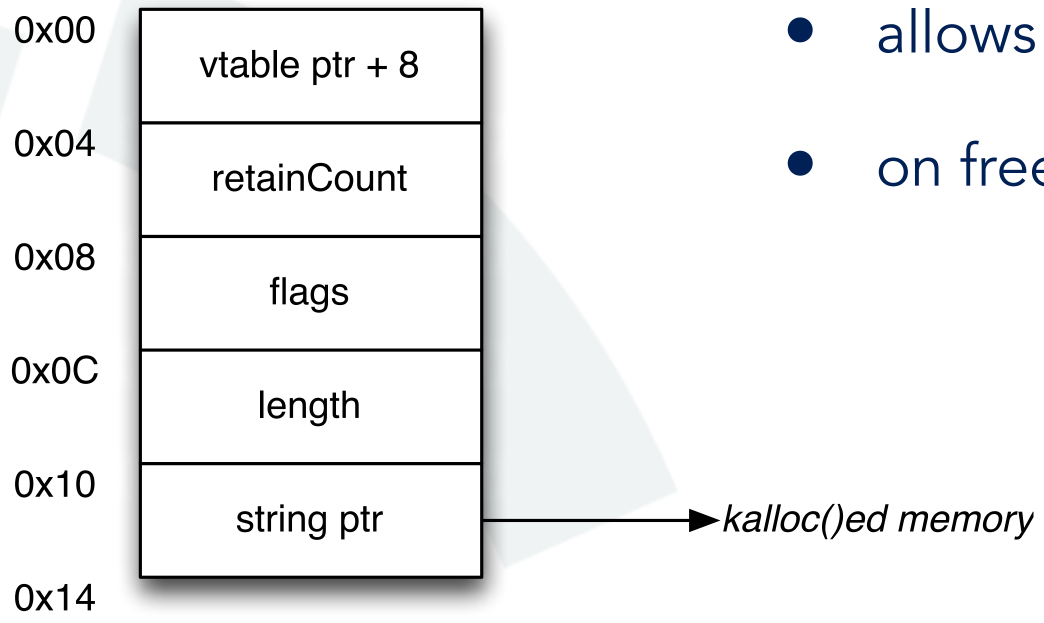
- overwriting or corrupting the ***vtable ptr***
  - everything the kernel will do with the object will trigger code exec



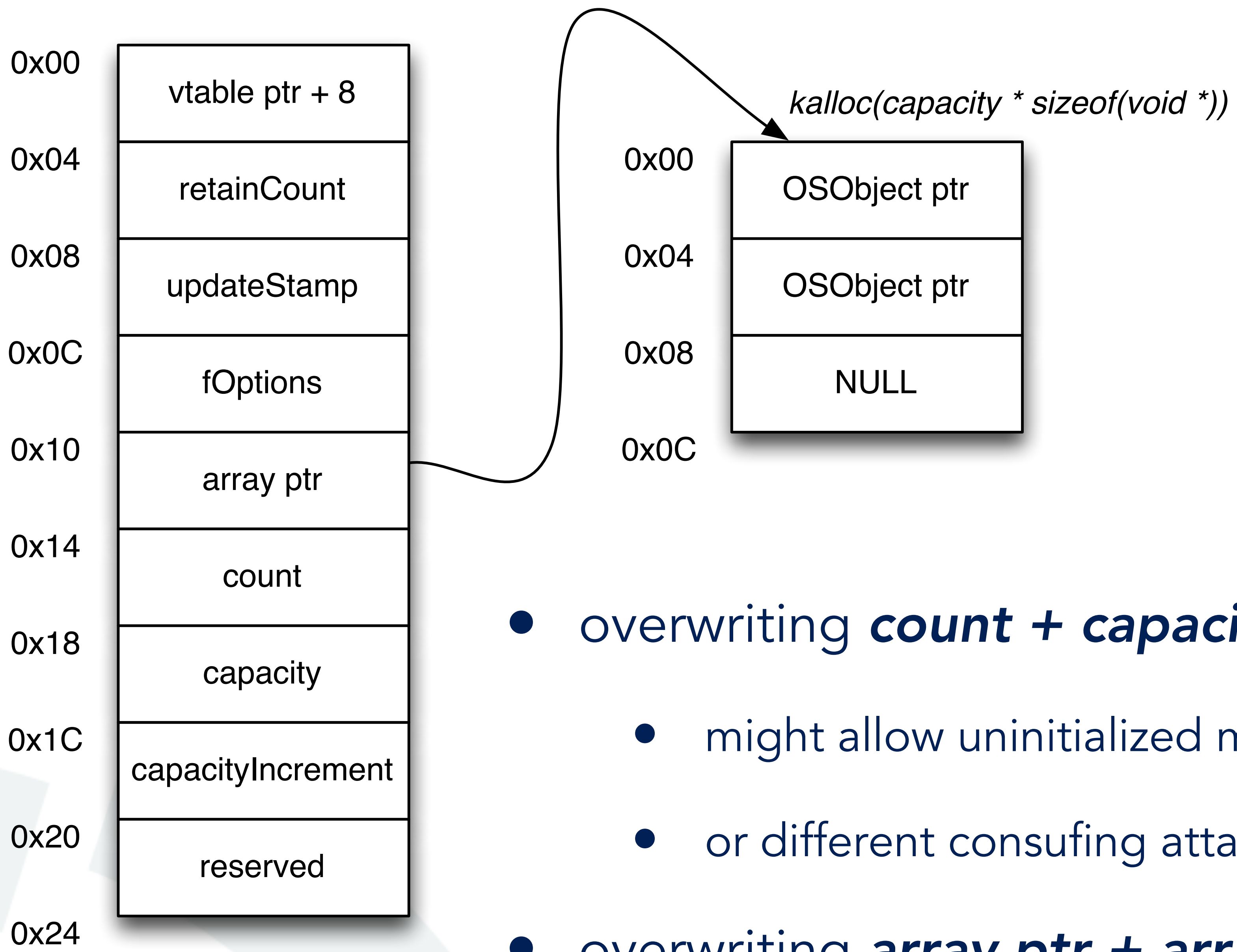
- overwriting the ***retain count***
  - might allow freeing the object early
  - and code execution through dangling references
  - use after free

OSString Memory Layout and Overwriting It

- overwriting ***flags*** controls if string is freed or not
- overwriting ***length***
  - might allow kernel heap information leaks
  - on free memory end up in wrong **kalloc** zone
- overwriting ***string ptr***
  - allows kernel heap information leaks
  - on free arbitrary pointer ends up in kalloc zone



OSArray Memory Layout and Overwriting It



- overwriting *updateStamp + fOptions* = not interesting

- overwriting ***count + capacity + capacityIncrement***
  - might allow uninitialized memory access
  - or different consufing attacks agains **kalloc** zones
- overwriting ***array ptr + array itself***
  - allows supplying arbitrary **OSObject** ptrs
  - any action the kernel performs on these will result in code exec

Part V

"Generic" Technique to control the iOS Kernel Heap

“Generic” Technique to control iOS Kernel Heap

- **Heap Spraying**
  - fill up kernel heap with arbitrary data
- **Heap Feng Shui** or **Heap Massage** or **Heap Setup** or **Heap Layout Control**
  - bring the kernel heap into a known state
  - by carefully crafted allocations and deallocations
- public iOS kernel exploits use **vulnerability specific** (de-)allocations
- we want a **more generic** solution

Heap Spraying

- allocate repeatedly
- allocate attacker controlled data
- allocate large quantities of data in a row
- usually fill memory with specific pattern

Heap Feng Shui / Heap Massage / ...

- allocate repeatedly (to close all memory holes)
- allocate arbitrary sized memory blocks
- poke allocation holes in specific positions
- control the memory layout
- fill memory with interesting meta / application data

Once Technique to rule them all...

Audience meet OSUnserializeXML()

OSUnserializeXML()

- deserialization of iOS kernel base objects
- used to pass objects from user space to kernel space (IOKit API)
- data in XML .plist format
- numbers, booleans, strings, data, dictionaries, arrays, sets and references

```
<plist version="1.0">
<dict>
    <key>IsThere</key>
    <string>one technique to rule them all?</string>
    <key>Answer</key>
    <true />
    <key>Audience</key>
    <string>meet OSUnserializeXML()</string>
</dict>
</plist>
```

How does the parser work? (I)

- parser starts at the beginning
- objects are identified by searching for starting tag
- and then parsing the inner value first
- **`<plist>`** tags will be ignored by the parser

```
<plist version="1.0">
<dict>
    <key>IsThere</key>
    <string>one technique to rule them all?</string>
    <key>Answer</key>
    <true />
    <key>Audience</key>
    <string>meet OSUnserializeXML()</string>
</dict>
</plist>
```

How does the parser work? (II)

- dictionaries are starting with the **<dict>** tag
- parser repeatedly reads key and value objects
- until closing **</dict>** tag

```
<plist version="1.0">
<dict>
    <key>IsThere</key>
    <string>one technique to rule them all?</string>
    <key>Answer</key>
    <true />
    <key>Audience</key>
    <string>meet OSUnserializeXML()</string>
</dict>
</plist>
```

How does the parser work? (III)

- after having seen a new object it is stored in a linked list
- parser stores each object in a 44 byte **`object_t`** struct
- memory is allocated via ***kern_os_malloc()*** which includes a header

```
typedef struct object {
    struct object   *next;          // next in collection
    struct object   *free;          // for freelist
    struct object   *elements;      // inner elements
    OSObject    *object;
    OSString    *key;               // for dictionary
    int     size;
    void        *data;              // for data
    char        *string;            // for string & symbol
    long long   number;             // for number
    int     idref;
} object_t;
```

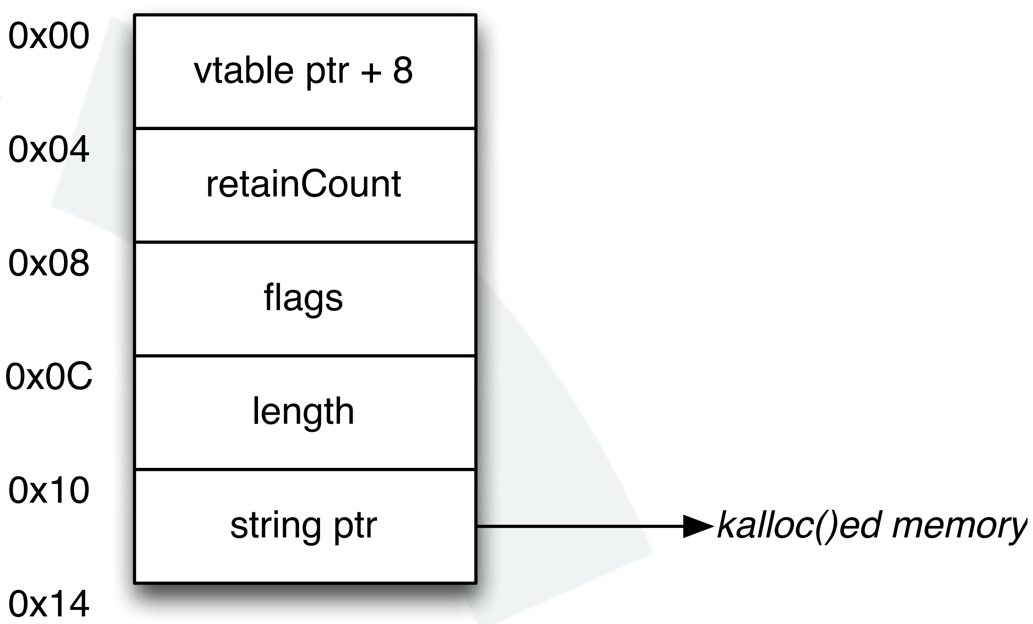
How does the parser work? (IV)

- parser now starts to fill the **elements** field of the **<dict>** object
- next expected object is a key indicated by the **`<key>`** tag
- to extract the key the parser determines length until next **<** character
- **`length + 1`** bytes are allocated via ***kern_os_malloc()*** plus a header

```
<plist version="1.0">
<dict>
    <key>IsThere</key>
    <string>one technique to rule them all?</string>
    <key>Answer</key>
    <true />
    <key>Audience</key>
    <string>meet OSUnserializeXML()</string>
</dict>
</plist>
```

How does the parser work? (V)

- **`key`** parser object is then converted to an internal **`OSString`** object
- ***new*** operator will allocate 20 bytes for **`OSString`** object via ***kalloc()***
- **`OSString`** contructor will create a copy of the string with ***kalloc()***
- string in parser key object will be freed with ***kern_os_free()***



Allocations so far:

```
// Dict
kern_os_alloc(44)     = kalloc(44+4)

// Key
kern_os_alloc(7+1)    = kalloc(7+1+4)
kern_os_alloc(44)     = kalloc(44+4)
kalloc(20)
kalloc(7+1)
kern_os_free(x, 7+1) = kfree(x, 7+1+4)
```

How does the parser work? (VI)

- next expected object is the dictionary value
- in this case it is a string defined by the **<string>** tag
- because it is a string it is handled in the same way as a key
- **length + 1** bytes are allocated via ***kern_os_malloc()*** plus a header
- string is copied into it

```
<plist version="1.0">
<dict>
    <key>IsThere</key>
    <string>one technique to rule them all?</string>
    <key>Answer</key>
    <true />
    <key>Audience</key>
    <string>meet OSUnserializeXML()</string>
</dict>
</plist>
```

How does the parser work? (VII)

- **`string`** parser object is then converted to an internal **`OSString`** object
- ***new*** operator will allocate 20 bytes for **`OSString`** object via ***kalloc()***
- **`OSString`** contructor will create a copy of the string with ***kalloc()***
- string in parser key object will be freed with ***kern_os_free()***

<u>Allocations so far:</u>

```
// Dict
kern_os_alloc(44)       = kalloc(44+4)

// Key
kern_os_alloc(7+1)      = kalloc(7+1+4)
kern_os_alloc(44)       = kalloc(44+4)
kalloc(20)
kalloc(7+1)
kern_os_free(x, 7+1)    = kfree(x, 7+1+4)

// Value
kern_os_alloc(31+1)     = kalloc(31+1+4)
kern_os_alloc(44)       = kalloc(44+4)
kalloc(20)
kalloc(31+1)
kern_os_free(x, 31+1) = kfree(x, 31+1+4)
```

How does the parser work? (VIII)

- once all **elements** are created the closing **</dict>** tag will create the dict
- the parser objects will be kept in a freelist and reused for further parsing

```
// Dict
kern_os_alloc(44)       = kalloc(44+4)

// Key "IsThere"
kern_os_alloc(7+1)      = kalloc(7+1+4)
kern_os_alloc(44)       = kalloc(44+4)
kalloc(20)
kalloc(7+1)
kern_os_free(x, 7+1)    = kfree(x, 7+1+4)

// Value
kern_os_alloc(31+1)     = kalloc(31+1+4)
kern_os_alloc(44)       = kalloc(44+4)
kalloc(20)
kalloc(31+1)
kern_os_free(x, 31+1) = kfree(x, 31+1+4)

// Key "Answer"
kern_os_alloc(6+1)      = kalloc(6+1+4)
kern_os_alloc(44)       = kalloc(44+4)
kalloc(20)
kalloc(6+1)
kern_os_free(x, 6+1)    = kfree(x, 6+1+4)
```

```
// Boolean Value
kern_os_alloc(44)       = kalloc(44+4)

// Key "Audience"
kern_os_alloc(8+1)      = kalloc(8+1+4)
kern_os_alloc(44)       = kalloc(44+4)
kalloc(20)
kalloc(8+1)
kern_os_free(x, 8+1)    = kfree(x, 8+1+4)

// String Value
kern_os_alloc(23+1)     = kalloc(23+1+4)
kern_os_alloc(44)       = kalloc(44+4)
kalloc(20)
kalloc(23+1)
kern_os_free(x, 23+1) = kfree(x, 23+1+4)

// The Dict
kalloc(36)
kalloc(3*8)
```

Memory Sizes Cheat Sheet

	in memory size	kalloc zone size	additional alloc
OSArray	36	40	+ capacity * 4
OSDictionary	36	40	+ capacity * 8
OSData	28	32	+ capacity
OSSet	24	24	+ sizeof(OSArray)
OSNumber	24	24	
OSString	20	24	+ strlen + 1
OSBoolean	12	16	cannot be generated by OSUnserializeXML()

Heap Spraying (Remember?)

- allocate repeatedly
- allocate attacker controlled data
- allocate large quantities of data in a row
- usually fill memory with specific pattern

Allocate Repeatedly

- there is no possibility to loop in a plist
- but we can make as many allocations as we want with e.g. arrays

```
<plist version="1.0">
<dict>
    <key>ThisIsOurArray</key>
    <array>
        <string>again and</string>
        <string>again and</string>
        <string>again and</string>
        <string>again and</string>
        <string>again and</string>
        <string>again and</string>
        <string>...</string>
    </array>
</dict>
</plist>
```

Heap Spraying

- **allocate repeatedly ✔**
- allocate attacker controlled data
- **allocate large quantities of data in a row ✔**
- usually fill memory with specific pattern

Allocate Attacker Controlled Data

- by putting data into a **<data>** tag we can fill memory with any data
- because data is either in **base64** or **hex** format we can have NULs
- **<data>** is more convenient than **<string>** because it reads in chunks of 4096

```
<plist version="1.0">
<dict>
    <key>ThisIsOurData</key>
    <array>
        <data>VGhpcyBJcyBPdXIgRGF0YSB3aXRoIGEgTlVMPgA8+ADw=</data>
        <data format="hex">00112233445566778899aabbccddeeff</data>
        <data>...</data>
    </array>
</dict>
</plist>
```

Heap Spraying

- allocate repeatedly ✔
- allocate attacker controlled data ✔
- allocate large quantities of data in a row ✔
- usually fill memory with specific pattern ✔

Heap Feng Shui / Heap Massage / ...

- allocate repeatedly ✔
- allocate arbitrary sized memory blocks /
- poke allocation holes in specific positions
- control the memory layout
- fill memory with interesting meta / application data

Fill Arbitrary Sized Memory Blocks with App Data

- allocating arbitrary sized memory blocks is easy with **<string>** or **<data>**
- arbitrary sized memory blocks with app data required different approach
- we can achieve by having **size / 4 <array>** elements (or dictionaries)

```
<plist version="1.0">
<dict>
    <key>ThisArrayAllocates_4_Bytes</key>
    <array>
        <true />
    </array>
    <key>ThisArrayAllocates_12_Bytes</key>
    <array>
        <true /><true /><true />
    </array>
    <key>ThisArrayAllocates_28_Bytes</key>
    <array>
        <true /><true /><true ><true /><true /><true /><true />
    </array>
</dict>
</plist>
```

Heap Feng Shui / Heap Massage / ...

- **allocate repeatedly ✔**
- **allocate arbitrary sized memory blocks ✔**
- poke allocation holes in specific positions
- control the memory layout
- **fill memory with interesting meta / application data ✔**

Poking Holes into Allocated Data

- deallocation of arbitrary sized memory is possible with **<dict>**
- reusing the same dictionary key will delete the previously inserted value
- in this example the middle value ZZZ...ZZZ is freed

```
<plist version="1.0">
<dict>
    <key>AAAA</key>
    <data>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</data>
    <key>BBBB</key>
    <data>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</data>
    <key>CCCC</key>
    <data>ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ</data>
    <key>DDDD</key>
    <data>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</data>
    <key>EEEE</key>
    <data>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</data>
    <key>CCCC</key>
    <true />
</dict>
</plist>
```

Heap Feng Shui / Heap Massage / ...

- allocate repeatedly ✔
- allocate arbitrary sized memory blocks ✔
- poke allocation holes in specific positions ✔
- control the memory layout ✔
- fill memory with interesting meta / application data ✔

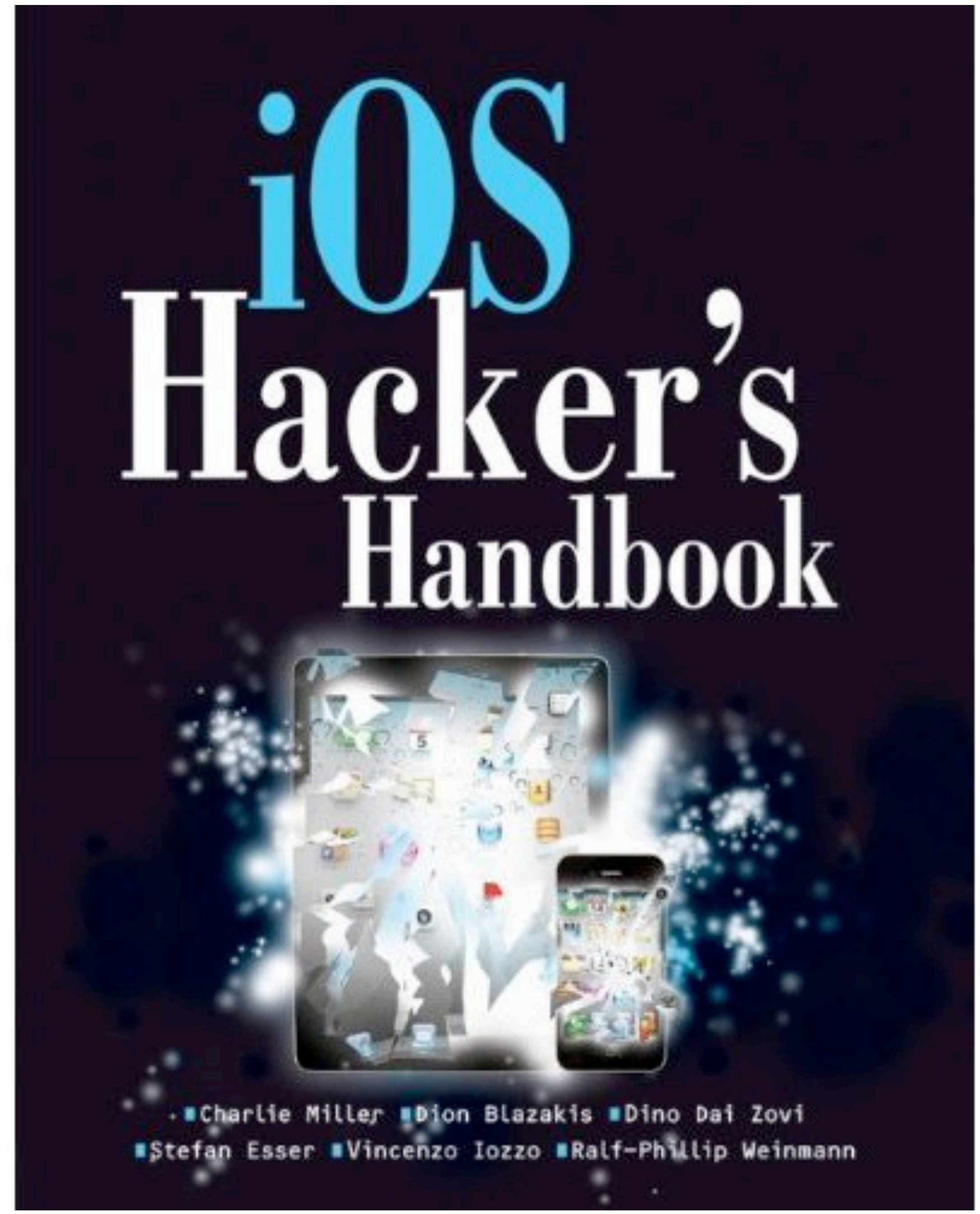
Extra: Keeping Data Allocated

- several places inside the kernel will keep the objects allocated for you
- but if the data is immediately freed you can leak the memory
- just abuse the **retainCount** freeze at 0xFFFE by creating many references

```
<plist version="1.0">
<dict>
    <key>AAAA</key>
    <array ID="1" CMT="IsNeverFreedTooManyReferences">...</array>
    <key>REFS</key>
    <array>
        <x IDREF="1"/><x IDREF="1"/><x IDREF="1"/><x IDREF="1"/>
        <x IDREF="1"/><x IDREF="1"/><x IDREF="1"/><x IDREF="1"/>
        <x IDREF="1"/><x IDREF="1"/><x IDREF="1"/><x IDREF="1"/>
        ...
        <x IDREF="1"/><x IDREF="1"/><x IDREF="1"/><x IDREF="1"/>
    </array>
</dict>
</plist>
```

Questions

?


iOS
Hacker's
Handbook
Charlie Miller Dion Blazakis Dino Dai Zovi
Stefan Esser Vincenzo Iozzo Ralf-Philipp Weinmann