Cisco ASA Episode 3: A journey in analysing heaps and heap exploitation tools

BSidesMCR – August 2017

/Us

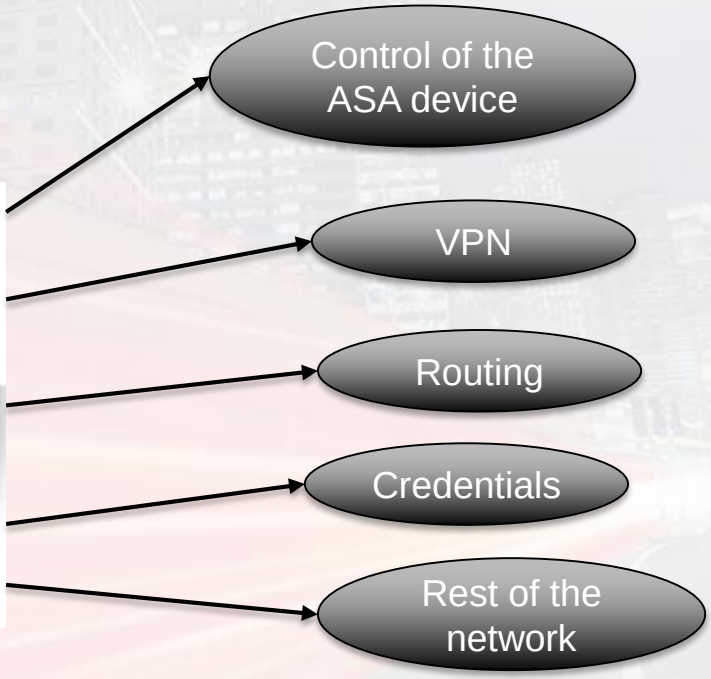
- Exploit Development Group (EDG) at NCC Group
  - Cedric Halbronn (@saidelike) – speaker today
  - Aaron Adams (@FidgetingBits)

- Vulnerability research, reverse engineering, exploit development

Context

Cisco ASA devices

- ASA stands for “Adaptive Security Appliance” (firewall, VPN gateway, router)
- ASA = Linux + `/asa` folder
  - Different than Cisco IOS which is a proprietary OS
- x86 or x86-64 (SMP, ASAv)
- `/asa/bin/lina` = main ELF binary containing all the ASA features
- Critical device – entry point to the whole network!

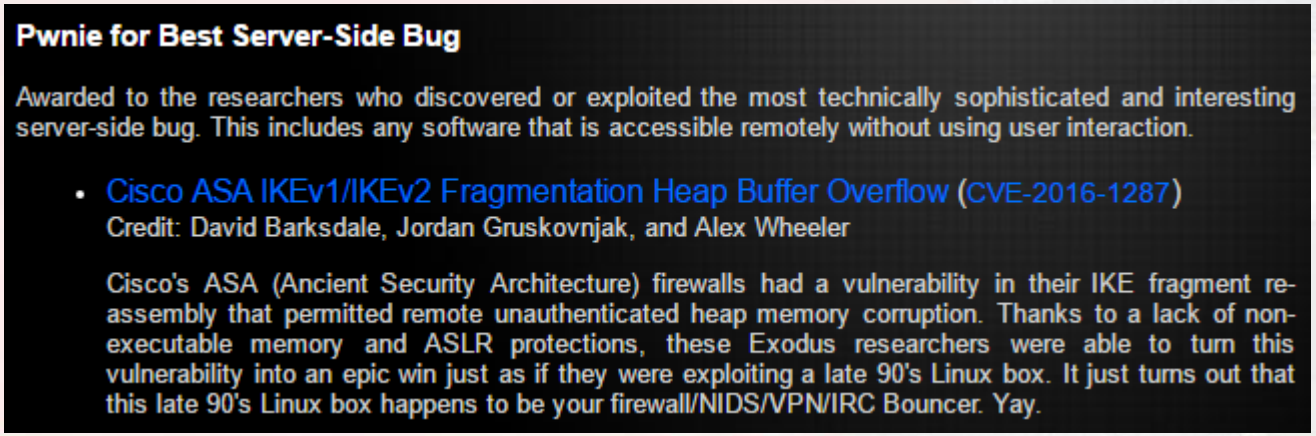


CVE-2016-1287

- Responsibly disclosed to Cisco by Exodus Intel (XI) pre-March 2016
  - “Execute My Packet”
- Targets IKE Cisco Fragmentation payload
  - Reassembled packet length integer overflow
  - Leading to heap overflow when reassembly occurs
- Pre-auth & IKE available on the Internet

- XI released a POC in April 2016
  - Targets IKEv2 / ASA 9.2.4 / 32-bit

- Awesome work! They won the Pwnie Awards 2016 contest! Yay!



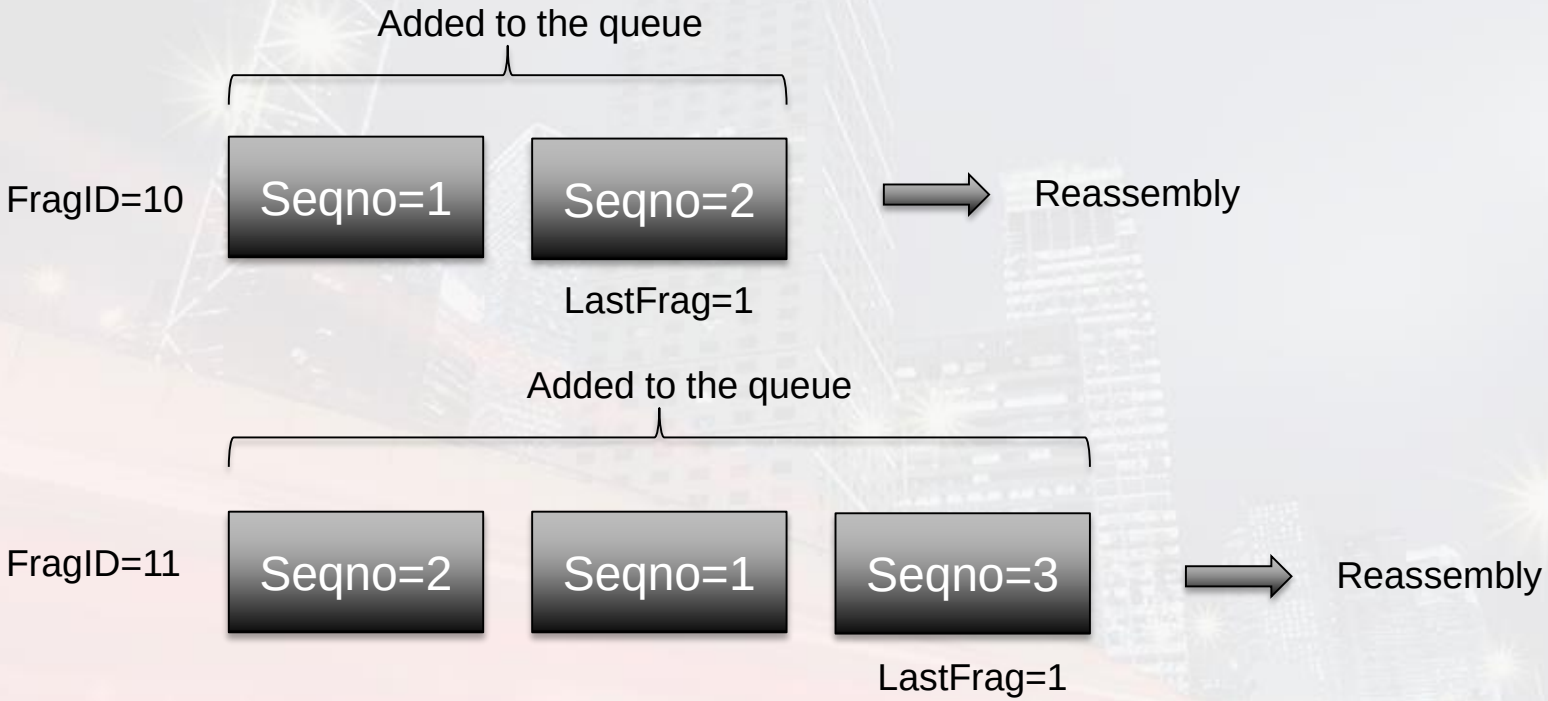
Presentation’s goals

- Exploiting a heap overflow requires understanding heap internals so we can abuse them!
- How we ported the exploit to IKEv1 to both 32-bit and 64-bit

- 3 gdb plugins to analyse heaps and assist in exploit dev

- Participate in an awesome journey into Cisco ASA heap internals and flaws
  - Presentation focus on 32-bit
  - Lots of concepts apply to 64-bit too

Cisco fragmentation



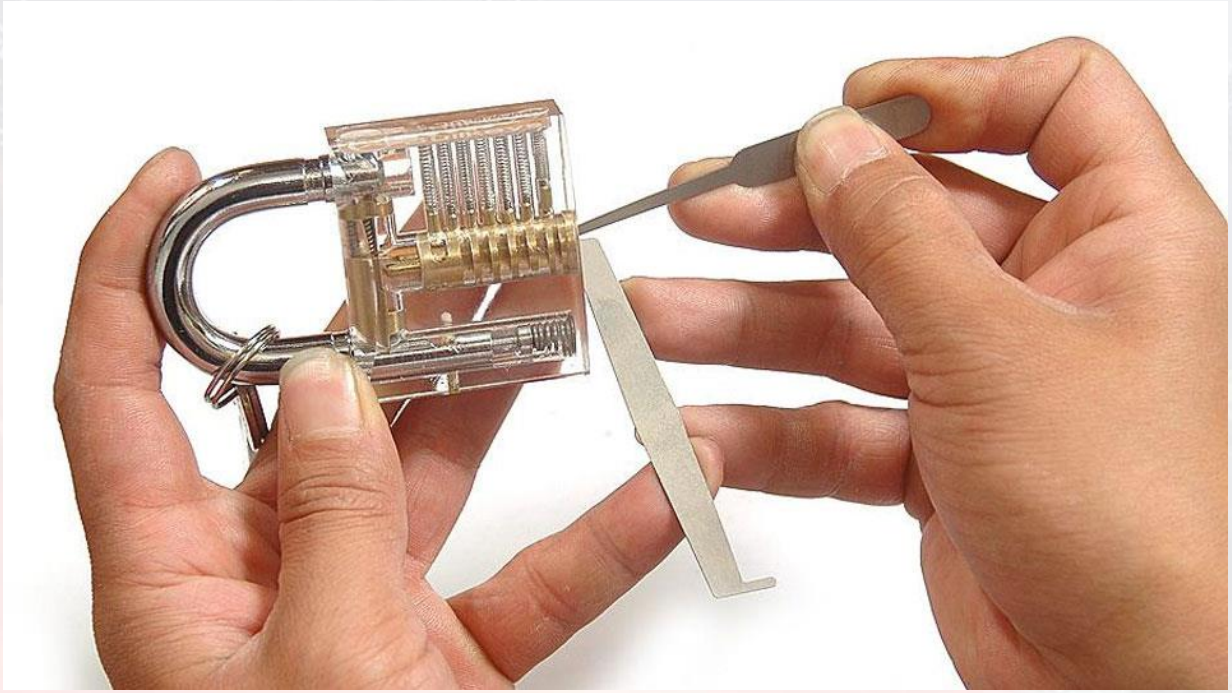
- Fragments with the same FragID are added to a queue
- They all have a different Seqno
- When the last fragment is received (with LastFrag=1), it triggers reassembly

CVE-2016-1287 heap overflow

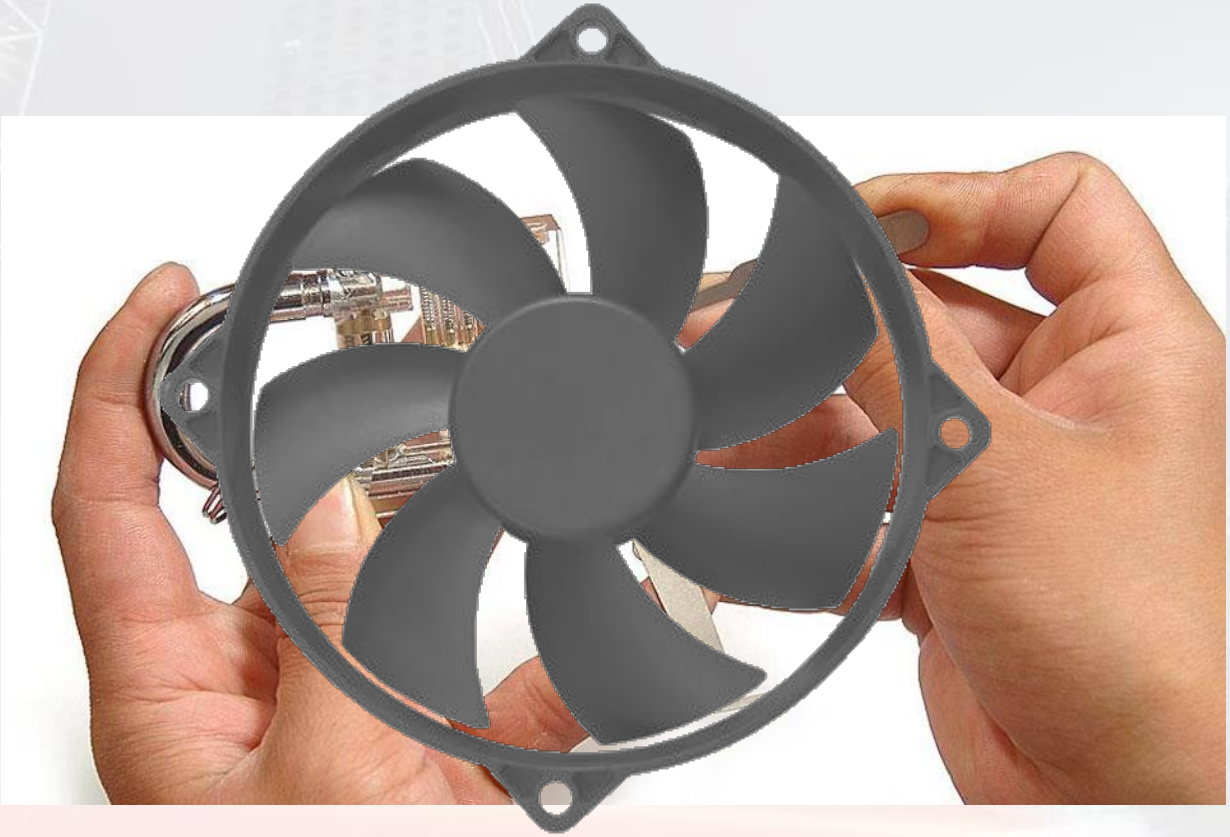
- Possible to fake the size of each fragment
  - Code substracts 8 when adding fragment's data
- Triggers a heap overflow when reassembly occurs

- What we try to achieve is hard
  - We don't control `malloc()`/`free()`
  - But can still gain control by making the right requests at the right time
  - We need to understand heaps

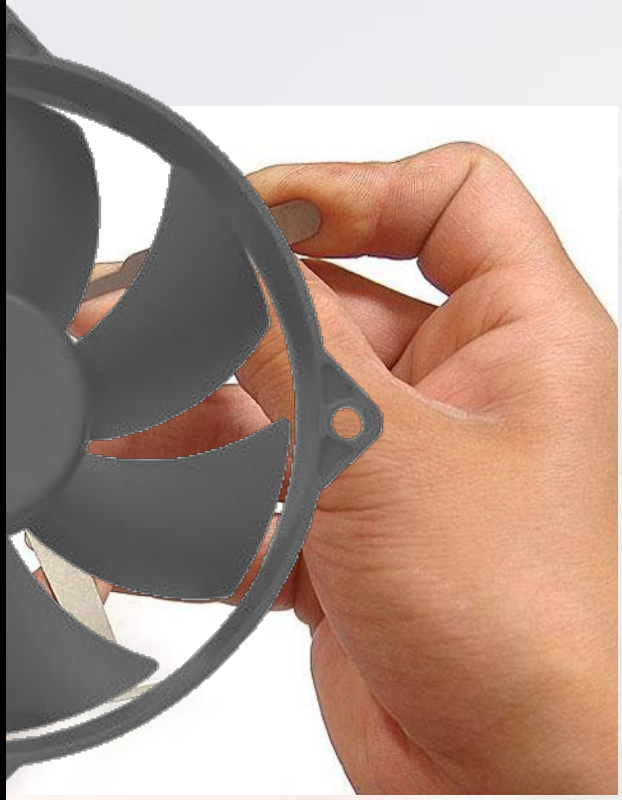
“At the moment…”



“At the moment…”

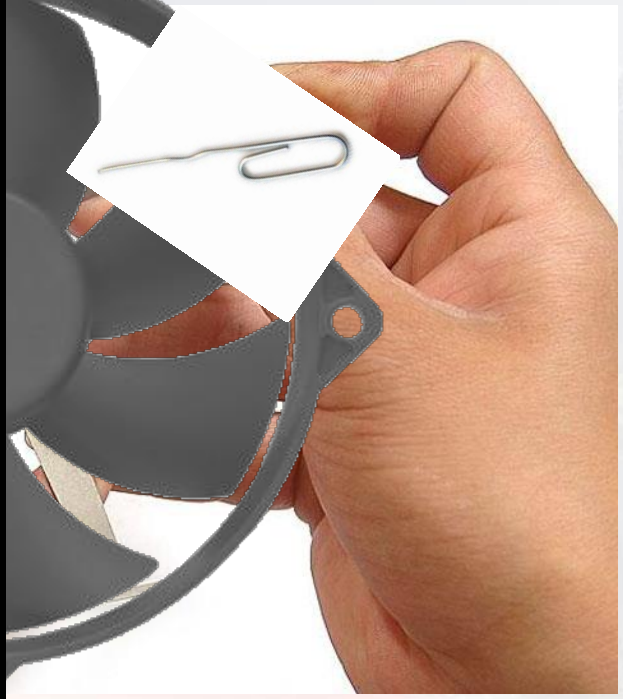


“At the moment…”



“At the moment…”

“At the moment…”



“…It is a bit like picking a lock through an industrial fan, in the dark, with a paper clip ☺” by Chris Anley

Agenda

- First steps into Cisco ASA heaps
- Cisco ASA's modifications
- 3 tools to rule them all

First steps into Cisco heaps

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.

dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use

dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

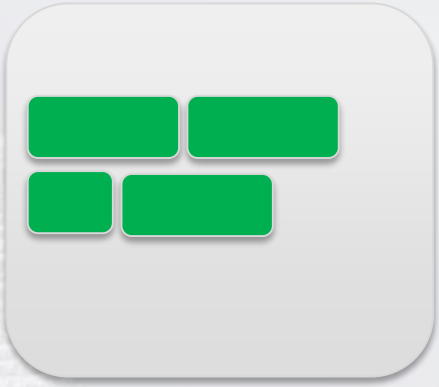
- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use

dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use

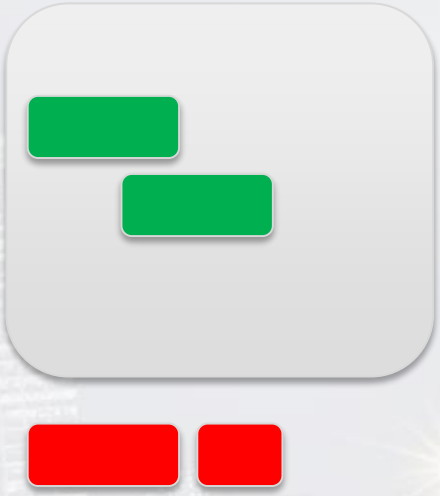


dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use
  - And tracks free chunks only (performance)

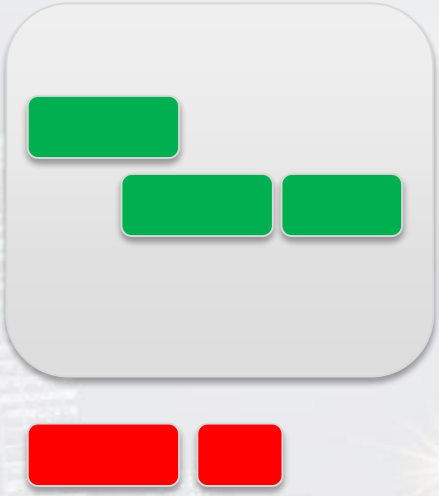


dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use
  - And tracks free chunks only (performance)

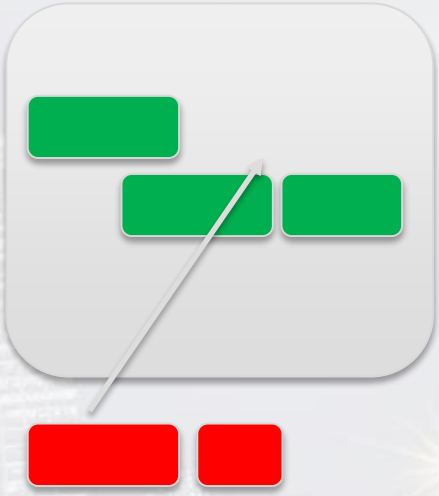


dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use
  - And tracks free chunks only (performance)
  - So it can provide new “allocated” chunks

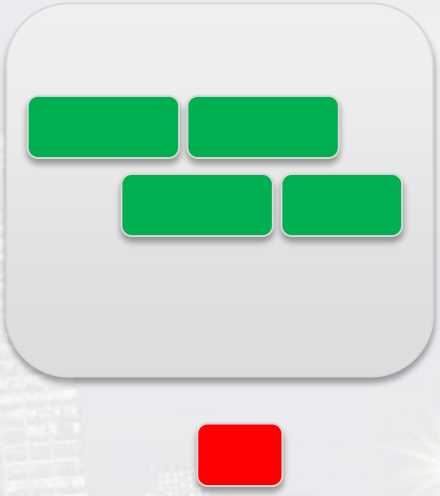


dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use
  - And tracks free chunks only (performance)
  - So it can provide new “allocated” chunks



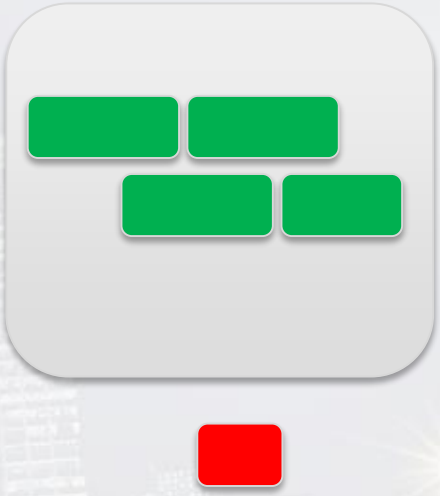
dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use
  - And tracks free chunks only (performance)
  - So it can provide new “allocated” chunks

- Chunk = memory returned by heap allocator

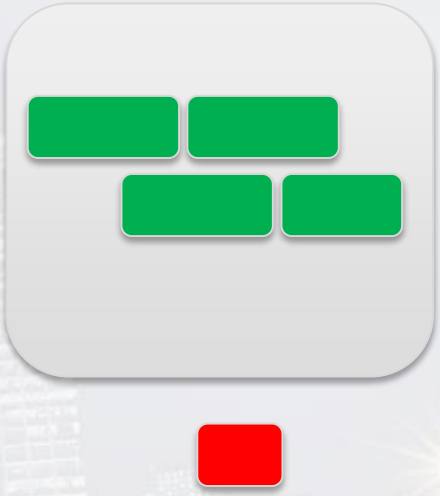


dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use
  - And tracks free chunks only (performance)
  - So it can provide new “allocated” chunks



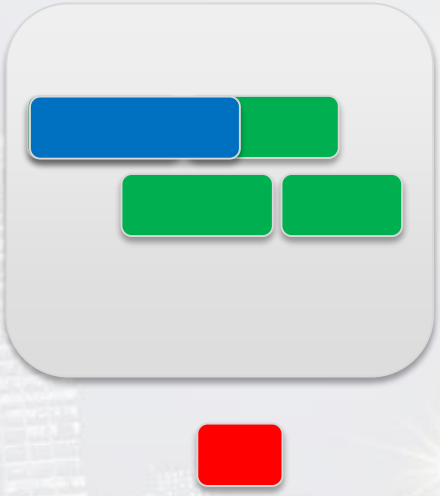
- Chunk = memory returned by heap allocator
- Metadata inlined in the chunk

dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use
  - And tracks free chunks only (performance)
  - So it can provide new “allocated” chunks



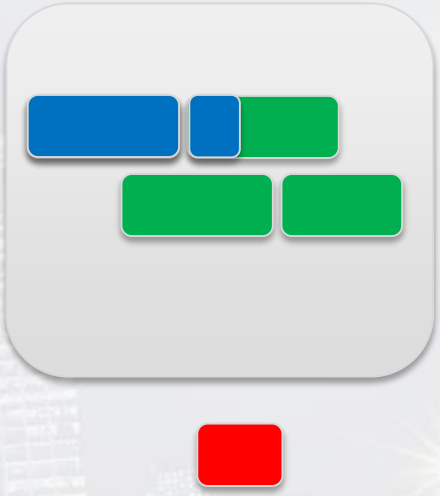
- Chunk = memory returned by heap allocator
- Metadata inlined in the chunk

dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)

- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use
  - And tracks free chunks only (performance)
  - So it can provide new “allocated” chunks

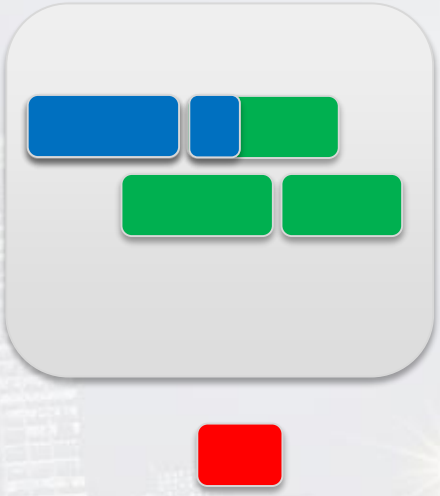


- Chunk = memory returned by heap allocator
- Metadata inlined in the chunk

dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)



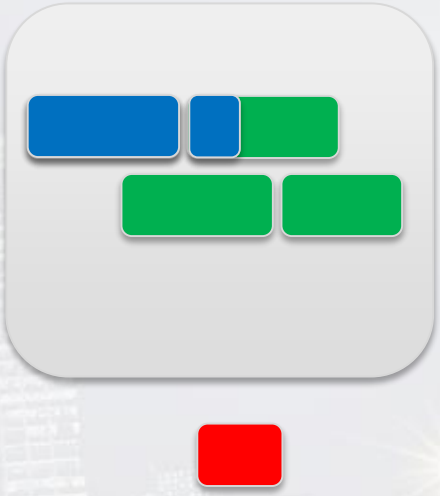
- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use
  - And tracks free chunks only (performance)
  - So it can provide new “allocated” chunks

- Chunk = memory returned by heap allocator
- Metadata inlined in the chunk
- Bin = tracks free chunks for a given size

dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

Well-known heap allocators

- dlmalloc = Doug Lea allocator (since 1993)
- ptmalloc = Per-Thread allocator (since 1999)
- GNU libc = widely used everywhere as part of Linux (glibc.so)



- A heap manager assists us in allocating/freeing chunks
  - Provides the APIs: `malloc`/`free`/etc.
  - Generally has some kind of region it can use
  - And tracks free chunks only (performance)
  - So it can provide new “allocated” chunks

- Chunk = memory returned by heap allocator
- Metadata inlined in the chunk
- Bin = tracks free chunks for a given size
- All open-source

dlmalloc	ptmalloc	glibc
dlmalloc 2.5.x	N/A	N/A
dlmalloc 2.6.x	ptmalloc	N/A
dlmalloc 2.7.x	ptmalloc2	forked
dlmalloc 2.8.x	ptmalloc3	N/A

ASA heap allocator?

- Let's focus on 32-bit for now (e.g. asa924-k8.bin)
- "lina" uses a version of dlmalloc compiled directly into its ELF binary (rather than external glibc)
- "lina" does link to glibc but won't end up using its allocation functions

```
.text:09BE4DF0   mov     dword ptr [esp+8], 0B0Ah
.text:09BE4DF8   mov     dword ptr [esp+4], offset aMalloc_c ; "malloc.c"
.text:09BE4E00   mov     dword ptr [esp], offset aNext_pinuseP_0 ; "(next_pinuse(p))"
.text:09BE4E07   call    __lina_assert
```

```
dlmalloc$ grep "(next == m->top || cinuse(next))" *
malloc-2.8.0.c:      assert(next == m->top || cinuse(next));
malloc-2.8.1.c:      assert(next == m->top || cinuse(next));
malloc-2.8.2.c:      assert(next == m->top || cinuse(next));
malloc-2.8.3.c:      assert(next == m->top || cinuse(next));
```

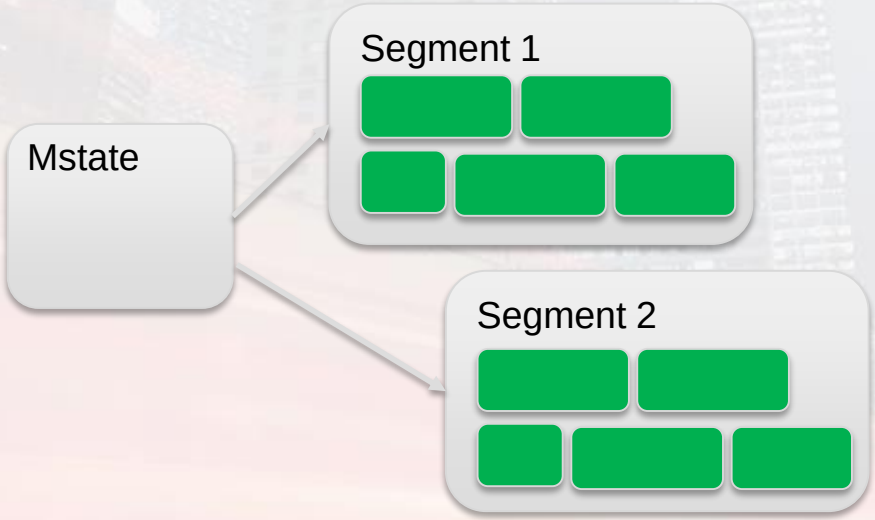
```
dlmalloc$ grep "segment_holds(sp, (char\*)sp)" *
malloc-2.8.3.c:        assert(segment_holds(sp, (char*)sp));
malloc-2.8.4.c:        assert(segment_holds(sp, (char*)sp));
malloc-2.8.5.c:        assert(segment_holds(sp, (char*)sp));
malloc-2.8.6.c:        assert(segment_holds(sp, (char*)sp));
```

➔ dlmalloc 2.8.3 used on ASA

➔ dlmalloc embedded in "lina" does NOT have safe-unlinking (enabled for dlmalloc >= 2.8.5)

dlmalloc2.8

- “Mstates” and “Mspaces”
- “Memory Segment” = tracks a region of memory
- “Mstate" contains different memory segments potentially allocated at different places



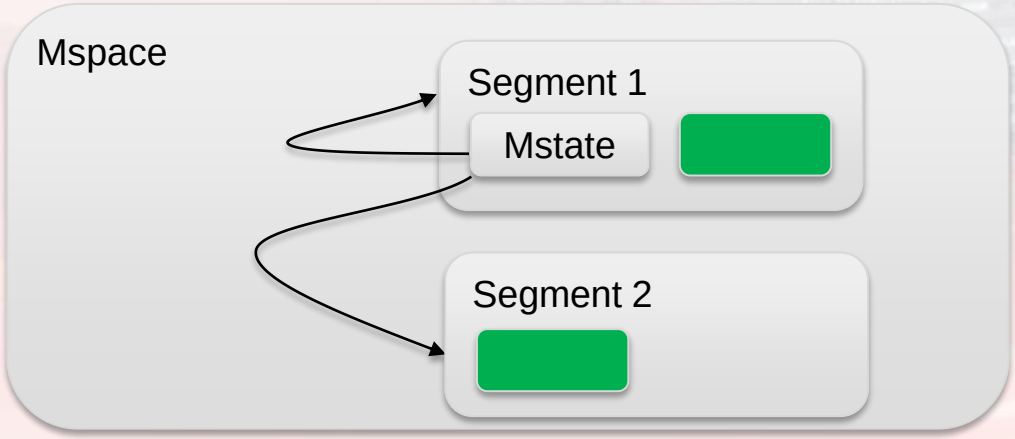
ASA mstate

- Mstate address can be easily determined as passed to `mspace_malloc()`

```
(gdb) dlmstate 0xa8400008
struct dl_mstate @ 0xa8400008 {
smallmap    = 0b00011110110000100000111111111100
treemap     = 0b00000000000000001011011010000111
dvsize      = 0xed268
topsize     = 0x2eac55c0
least_addr  = 0xa8400000
dv          = 0xad03d778
top         = 0xad13aa10
trim_check  = 0x200000
magic       = 0x2900d4d8
smallbin[00] (sz 0x0)   = 0xa840002c, 0xa840002c [EMPTY]
smallbin[01] (sz 0x8)   = 0xa8400034, 0xa8400034 [EMPTY]
smallbin[02] (sz 0x10)  = 0xacff8068, 0xa88647f0
…
treebin[00] (sz 0x180)      = 0xacff0e98
treebin[01] (sz 0x200)      = 0xacfefc00
…
seg = struct malloc_segment @ 0xa84001d4 {
base        = 0xa8400000
size        = 0x33800000
next        = 0x0
sflags      = 0x8
```

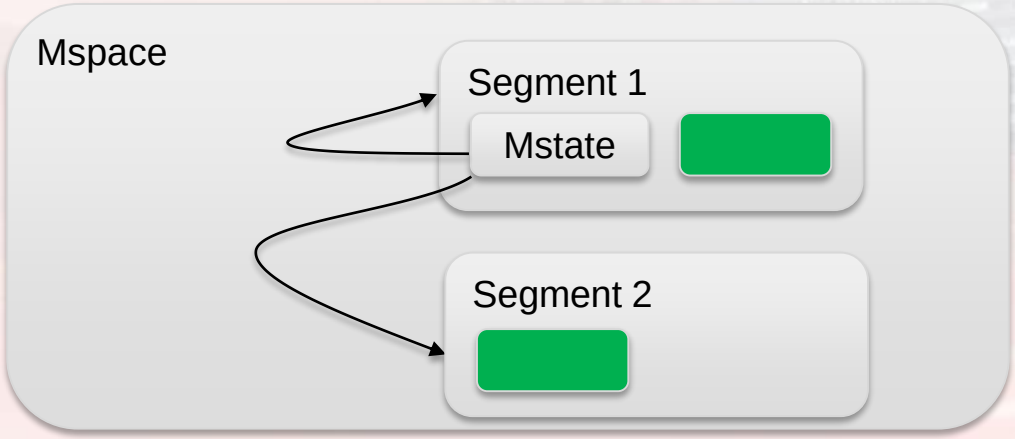
dlmalloc2.8

- “mspace” introduced – facilitates the management of multiple discrete heaps
- By defining `MSPACES`, `mspace_malloc()`/`mspace_free()`/etc. functions are enabled
- Create a dedicated mspace by using `create_mspace()`
  - Allocator maps a memory segment and stores an “mstate” into its first chunk
- “mspace” passed as first arg, e.g. `void* mspace_malloc(mspace msp, size_t bytes);`



dlmalloc2.8

- “mspace” introduced – facilitates the management of multiple discrete heaps
- By defining `MSPACES`, `mspace_malloc()`/`mspace_free()`/etc. functions are enabled
- Create a dedicated mspace by using `create_mspace()`
  - Allocator maps a memory segment and stores an “mstate” into its first chunk
- “mspace” passed as first arg, e.g. `void* mspace_malloc(mspace msp, size_t bytes);`
- Can still use `malloc()`/`free()`/etc. independently



dlmalloc2.8

- “mspace” introduced – facilitates the management of multiple discrete heaps
- By defining `MSPACES`, `mspace_malloc()`/`mspace_free()`/etc. functions are enabled
- Create a dedicated mspace by using `create_mspace()`
  - Allocator maps a memory segment and stores an “mstate” into its first chunk
- “mspace” passed as first arg, e.g. `void* mspace_malloc(mspace msp, size_t bytes);`
- Can still use `malloc()`/`free()`/etc. independently

dlmalloc2.8

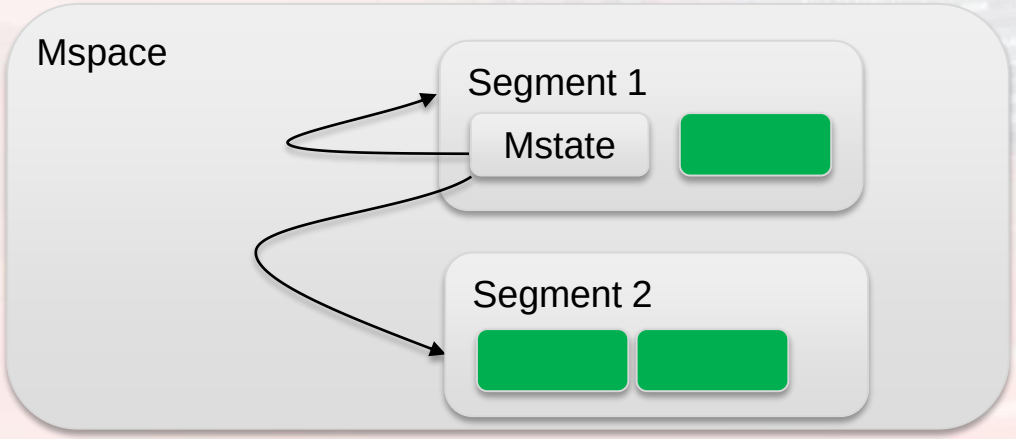
- “mspace” introduced – facilitates the management of multiple discrete heaps
- By defining `MSPACES`, `mspace_malloc()`/`mspace_free()`/etc. functions are enabled
- Create a dedicated mspace by using `create_mspace()`
  - Allocator maps a memory segment and stores an “mstate” into its first chunk
- “mspace” passed as first arg, e.g. `void* mspace_malloc(mspace msp, size_t bytes);`
- Can still use `malloc()`/`free()`/etc. independently

dlmalloc2.8

- “mspace” introduced – facilitates the management of multiple discrete heaps
- By defining `MSPACES`, `mspace_malloc()`/`mspace_free()`/etc. functions are enabled
- Create a dedicated mspace by using `create_mspace()`
  - Allocator maps a memory segment and stores an “mstate” into its first chunk
- “mspace” passed as first arg, e.g. `void* mspace_malloc(mspace msp, size_t bytes);`
- Can still use `malloc()`/`free()`/etc. independently

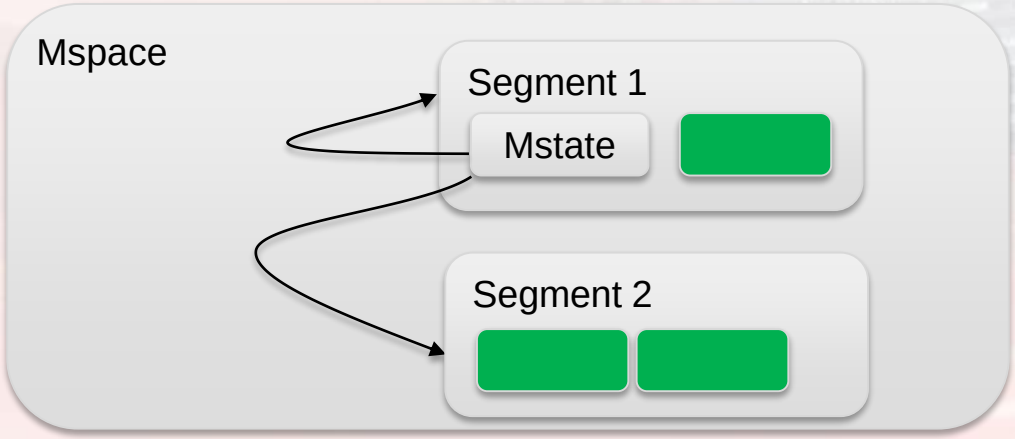
dlmalloc2.8

- “mspace” introduced – facilitates the management of multiple discrete heaps
- By defining `MSPACES`, `mspace_malloc()`/`mspace_free()`/etc. functions are enabled
- Create a dedicated mspace by using `create_mspace()`
  - Allocator maps a memory segment and stores an “mstate” into its first chunk
- “mspace” passed as first arg, e.g. `void* mspace_malloc(mspace msp, size_t bytes);`
- Can still use `malloc()`/`free()`/etc. independently



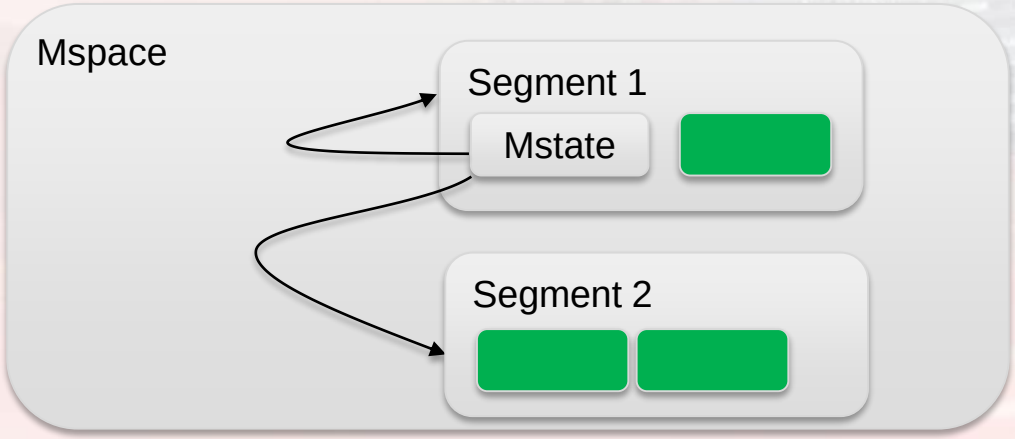
dlmalloc2.8

- “mspace” introduced – facilitates the management of multiple discrete heaps
- By defining `MSPACES`, `mspace_malloc()`/`mspace_free()`/etc. functions are enabled
- Create a dedicated mspace by using `create_mspace()`
  - Allocator maps a memory segment and stores an “mstate” into its first chunk
- “mspace” passed as first arg, e.g. `void* mspace_malloc(mspace msp, size_t bytes);`
- Can still use `malloc()`/`free()`/etc. independently



dlmalloc2.8

- “mspace” introduced – facilitates the management of multiple discrete heaps
- By defining `MSPACES`, `mspace_malloc()`/`mspace_free()`/etc. functions are enabled
- Create a dedicated mspace by using `create_mspace()`
  - Allocator maps a memory segment and stores an “mstate” into its first chunk
- “mspace” passed as first arg, e.g. `void* mspace_malloc(mspace msp, size_t bytes);`
- Can still use `malloc()`/`free()`/etc. independently
- Except if you define `ONLY_MSPACES`



malloc() ASA implementation?

- On 32-bit (e.g. `asa924-k8.bin`)
- `*malloc()` → `resMgrCalloc()` → `mem_mh_calloc()` → `mspace_malloc(mspace msp, size_t bytes)`
- `MSPACES = 1` and `ONLY_MSPACES = 1`
- Statically compiled into `lina`

Understanding mempools

Mempool

- How do they handle the different mstates/mspaces?
  - Using mempools

- Mempool = region of mapped memory dedicated for allocations by some Cisco ASA components
  - Two main mempools on ASA

```
(gdb) python import libmempool as lmp
(gdb) python m = lmp.mempool_list(addr=0x0a9a50b0)
(gdb) python print(m)
struct mempool_list @ 0xa9a50b0 {
 offset       = 0x68
 head         = 0xa8400c80
 unk          = 0x0
struct mempool @ 0xa8400c18 {
 dlmstate      = 0xa5c00008
 pool_name     = MEMPOOL_DMA
 field_58      = 0x0
 mempool_id    = 0x6
 next          = 0xa8400be8
struct mempool @ 0xa8400b80 {
 dlmstate      = 0xa8400008
 pool_name     = MEMPOOL_GLOBAL_SHARED
 field_58      = 0x0
 mempool_id    = 0x0
 next          = 0xa9a50b4
```

DMA-related allocations

General purpose allocations (> 99%)

Allocator function wrappers

- On 32-bit (e.g. asa924-k8.bin)
- `*malloc()` → `resMgrCalloc()` → `mem_mh_calloc()` → `mspace_malloc(mspace msp, size_t bytes)`
- `resMgrCalloc` does not do much except dispatching
- `mem_mh_calloc` injects bookkeeping structures
  - Allocates more space

```
.text:09BEB3DC  mov     [esp], esi      ; mspace = 0xa8800008
.text:09BEB3DF  lea     ebx, [eax+edx] ; length += some overhead (0 mostly)
.text:09BEB3E2  lea     eax, [ebx+24h] ; length += sizeof(mp_header)
.text:09BEB3E5  mov     [esp+4], eax
.text:09BEB3E9  call    mspace_malloc  ; call dlmalloc2.8.3 code
```

  - Fills the mempool header

```
.text:09BEB3FE  mov     [eax+4], ebx                        ; length
.text:09BEB401  mov     [eax+8], 0
.text:09BEB408  mov     [eax], 0A11C0123h                   ; magic header
.text:09BEB40E  mov     dword ptr [eax+ebx+20h], 0A11CCDEFh   ; magic footer
```

  - Don't forget the caller ☺

```
; some non-relevant instructions skipped below
.text:09BEB496  mov     ecx, ebp            ; get a pointer to the stack
...
.text:09BEB4CE  mov     [edx+0x18], eax      ; save caller address
```

“mh” a.k.a. mempool header?

```
struct mp_header {
    unsigned            mh_magic;       // 0xa11c0123
    unsigned            mh_len;         // length of the data, not including mp_header
    unsigned            mh_refcount;    // also used for free magic when freed
    unsigned            mh_unused;
    struct mp_header *  mh_fd_link;     // points to next mp_header of same bin size
    struct mp_header *  mh_bk_link;     // points to previous mp_header of same bin size
    void *              alloc_pc;       // function allocating chunk
    void *              free_pc;        // function freeing chunk
};
```

```
(gdb) dlchunk 0xa8a21740 -v
struct malloc_chunk @ 0xa8a21740 {
prev_foot    = 0x8140d4d0
size         = 0x38 (CINUSE|PINUSE)
struct mp_header @ 0xa8a21748 {
mh_magic       = 0xa11c0123
mh_len         = 0xc
mh_refcount    = 0x0
mh_unused      = 0x0
mh_fd_link     = 0xacfe2130
mh_bk_link     = 0xa84002c4
alloc_pc       = 0x868200c (IKE_AddRcvFrag+0x13c)
free_pc        = 0x806ab8c (sub_806AB30+0x5c)
```

```
(gdb) dlchunk 0xacfe2130-0x8 -v
struct malloc_chunk @ 0xacfe2128 {
prev_foot    = 0x8140d4d0
size         = 0x38 (CINUSE|PINUSE)
struct mp_header @ 0xacfe2130 {
mh_magic       = 0xa11c0123
mh_len         = 0xc
mh_refcount    = 0x0
mh_unused      = 0x0
mh_fd_link     = 0xacfe4bc0
mh_bk_link     = 0xa8a21748
alloc_pc       = 0x868200c (IKE_AddRcvFrag+0x13c)
free_pc        = 0x806ab8c (sub_806AB30+0x5c)
```

```
(gdb) dlchunk 0xa8a21740
0xa8a21740 M sz:0x00038 fl:CP alloc_pc:0x0868200c,IKE_AddRcvFrag+0x13c
(gdb) dlchunk 0xacfe2130-0x8
0xacfe2128 M sz:0x00038 fl:CP alloc_pc:0x0868200c,IKE_AddRcvFrag+0x13c
```

“mh” a.k.a. mempool header?

```
struct mp_header {
    unsigned            mh_magic;       // 0xa11c0123
    unsigned            mh_len;         // length of the data, not including mp_header
    unsigned            mh_refcount;    // also used for free magic when freed
    unsigned            mh_unused;
    struct mp_header *  mh_fd_link;     // points to next mp_header of same bin size
    struct mp_header *  mh_bk_link;     // points to previous mp_header of same bin size
    void *              alloc_pc;       // function allocating chunk
    void *              free_pc;        // function freeing chunk
};
```

```
(gdb) dlchunk 0xa8a21740 -v
struct malloc_chunk @ 0xa8a21740 {
prev_foot    = 0x8140d4d0
size         = 0x38 (CINUSE|PINUSE)
struct mp_header @ 0xa8a21748 {
mh_magic       = 0xa11c0123
mh_len         = 0xc
mh_refcount    = 0x0
mh_unused      = 0x0
mh_fd_link     = 0xacfe2130
mh_bk_link     = 0xa84002c4
alloc_pc       = 0x868200c (IKE_AddRcvFrag+0x13c)
free_pc        = 0x806ab8c (sub_806AB30+0x5c)
```

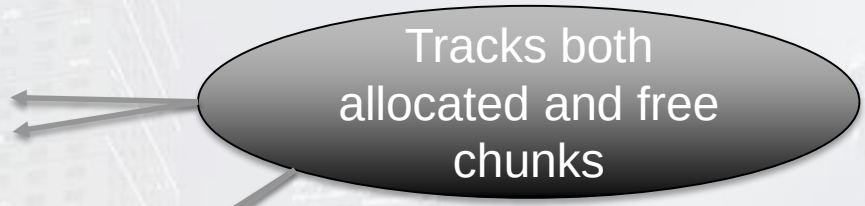
```
(gdb) dlchunk 0xacfe2130-0x8 -v
struct malloc_chunk @ 0xacfe2128 {
prev_foot    = 0x8140d4d0
size         = 0x38 (CINUSE|PINUSE)
struct mp_header @ 0xacfe2130 {
mh_magic       = 0xa11c0123
mh_len         = 0xc
mh_refcount    = 0x0
mh_unused      = 0x0
mh_fd_link     = 0xacfe4bc0
mh_bk_link     = 0xa8a21748
alloc_pc       = 0x868200c (IKE_AddRcvFrag+0x13c)
free_pc        = 0x806ab8c (sub_806AB30+0x5c)
```

```
(gdb) dlchunk 0xa8a21740
0xa8a21740 M sz:0x00038 fl:CP alloc_pc:0x0868200c,IKE_AddRcvFrag+0x13c
(gdb) dlchunk 0xacfe2130-0x8
0xacfe2128 M sz:0x00038 fl:CP alloc_pc:0x0868200c,IKE_AddRcvFrag+0x13c
```

Mempool-related thoughts

- "show mem detail" is able to track how many chunks are currently in use

```
ciscoasa# show mem detail
...
MEMPOOL_DMA POOL STATS:

Non-mmapped bytes allocated =     41041920
Number of free chunks       =           66
Number of mmapped regions   =            0
Mmapped bytes allocated     =            0
...
MEMPOOL_GLOBAL_SHARED POOL STATS:

Non-mmapped bytes allocated =    859832320
Number of free chunks       =          251
Number of mmapped regions   =            0
Mmapped bytes allocated     =            0
...
```



- When a chunk is freed, there is no safe unlinking on the `mp_header` doubly linked list
  - Used by Exodus Intel's IKEv2 exploit to achieve two mirror writes

- A heap-based memory revelation bug can give a lot of really useful information
  - Heap addresses (`mh_fd_link`/`mh_bk_link`)
  - A `.text` address (`alloc_pc`/`free_pc`)

Footers' ASA's modifications

dlmalloc 2.8's footers

```
struct malloc_chunk {
  size_t               prev_foot;  /* Size of previous chunk (if free).  */
  size_t               head;       /* Size and inuse bits. */
  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;
};
```

- Usually, heap managers reuse first field as a spillover area when chunk is allocated
  - Saves space and simplifies alignment

- Defining "FOOTERS" in dlmalloc2.8
  - When creating an inuse chunk, dlmalloc stores a "prev_foot" value into the adjacent chunk
  - Not called "prev_size" anymore
  - Purpose: like a global cookie

If FOOTERS is defined nonzero, then each allocated chunk carries an additional check word to verify that it was malloced from its space. **These check words are the same within each execution of a program using malloc, but differ across executions, so externally crafted fake chunks cannot be freed.** This improves security by rejecting frees/reallocs that could corrupt heap memory, in addition to the checks preventing writes to statics that are always on. This may further improve security at the expense of time and space overhead. (Note that FOOTERS may also be worth using with MSPACES.)

Setting dlmalloc 2.8's footer

- Footer "`prev_foot`" is calculated when chunk is allocated
  - `M` = mstate / `p` = pointer to inuse chunk / `s` = size of chunk

```
/* Set foot of inuse chunk to be xor of mstate and seed */
#define mark_inuse_foot(M,p,s)
  (((mchunkptr)((char*)(p) + (s)))->prev_foot = ((size_t)(M) ^ mparams.magic))
```

- Security relies on the inability to predict both `M` and `mparams.magic`
- `mparams.magic` is initialized by `init_params()` and uses either
  - `/dev/urandom` to read `sizeof(size_t)` bytes
  - `time(0)`

Checking dlmalloc 2.8's footer

- Additional checks on `prev_foot` at runtime when `FOOTER = 1`

```
void mspace_free(mspace msp, void* mem) {
  if (mem != 0) {
    mchunkptr p  = mem2chunk(mem);
    mstate fm = get_mstate_for(p);
    if (!ok_magic(fm)) {
      USAGE_ERROR_ACTION(fm, p);
      return;
    }
```

- Fetches what it believes is an mstate address from a chunk

```
#define get_mstate_for(p)
  ((mstate)(((mchunkptr)((char*)(p) + (chunksize(p))))->prev_foot ^ mparams.magic))
```

- Validates that the magic member of the mstate matches the `mparams.magic` global

```
#define ok_magic(M)      ((M)->magic == mparams.magic)
```

➔ Interesting security mechanism when `MSPACES` and `FOOTERS` constants are defined

ASA's footer implementation

- `USE_DEV_RANDOM = 0`
- `time(0)` is used instead of copying cookie from `/dev/urandom`

```
.text:09BE4AE0 loc_9BE4AE0:                    ; CODE XREF: create_mspace_with_base+E
.text:09BE4AE0   mov     ds:mparams_mmap_threshold, 40000h
.text:09BE4AEA   mov     ds:mparams_trim_threshold, 200000h
.text:09BE4AF4   mov     ds:mparam_default_mflags, 3
.text:09BE4AFE   mov     dword ptr [esp], 0
.text:09BE4B05   call    __wrap_time
.text:09BE4B0A   mov     edx, ds:mparams_magic
.text:09BE4B10   test    edx, edx
.text:09BE4B12   jnz     short loc_9BE4B2E
.text:09BE4B14   xor     eax, 55555555h
.text:09BE4B19   or      eax, 8
.text:09BE4B1C   and     eax, 0FFFFFFF8h
.text:09BE4B1F   mov     ds:mparams_magic, eax
```

- Called early enough during boot that `__wrap_time(0)` returns a static value: `0x7c558180`

```
.text:09BC4E73   mov     eax, [ebp+arg_0]
.text:09BC4E76   pop     ebp
.text:09BC4E77   mov     eax, [eax]
.text:09BC4E79   add     eax, 7C558180h
.text:09BC4E7E   retn
```

- Corresponds to NTP timestamp max 'Wed Feb 6 23:28:16 2036' (new epoch)

ASA's footer implementation

- After following operations, initialize `mparams.magic = 0x2900d4d8`

```
s = (size_t)(time(0) ^ (size_t)0x55555555U);
s |= (size_t)8U;    /* ensure nonzero */
s &= ~(size_t)7U;   /* improve chances of fault for bad values */
```

- Confirmed by dlmstate command and never changes across builds/reboots

```
(gdb) dlmstate 0xa8400008
struct dl_mstate @ 0xa8400008 {
...
magic       = 0x2900d4d8
...
```

- `prev_foot = M ^ mparams.magic = 0xa8400008 ^ 0x2900d4d8 = 0x8140d4d0`

```
(gdb) dlchunk 0xacfe5458 -v
struct malloc_chunk @ 0xacfe5458 {
prev_foot   = 0x8140d4d0
head        = 0x10 (PINUSE)
fd          = 0xacff2ca0
bk          = 0xacff3e58
```

- How predictable is the mstate address?
  - On ASA 5505 (no-ASLR), we only observe 2 different addresses for the main mstate
- Variance still non-ideal if we need to overwrite `prev_foot` when overwriting chunks
- Turns out another customisation Cisco made makes it all irrelevant anyway ☺

ASA's footer implementation

- dlmalloc2.8.3 implementation

```
void mspace_free(mspace msp, void* mem) {
  if (mem != 0) {
    mchunkptr p  = mem2chunk(mem);
#if FOOTERS
    mstate fm = get_mstate_for(p);
#else /* FOOTERS */
    mstate fm = (mstate)msp;
#endif /* FOOTERS */
    if (!ok_magic(fm)) {
      USAGE_ERROR_ACTION(fm, p);
      return;
    }
```

- Approximate implementation on ASA 9.2.4

```
void mspace_free(mspace msp, void* mem) {
  if (mem != 0) {
    mchunkptr p  = mem2chunk(mem);
    if (msp != NULL)  {
      mstate fm = (mstate)msp;
    } else {
      mstate fm = get_mstate_for(p);
    }
    // this will always be good if msp != NULL
    if (!ok_magic(fm)) {
      USAGE_ERROR_ACTION(fm, p);
      return;
    }
```

- If mspace passed is non NULL, assume it is the correct mspace
- Does not rely on using the pointer derived from a footer

➔ We don't even have to correctly guess the footer at all!

- Confirm this in practice by testing Exodus Intel's CVE-2016-1287 exploit
  - They use a static `prev_foot` that when decoded gives an mspace at `0xc8000008`
  - We only observe two different mspace (`0xa8400008` or `0xa8800008`)
  - Their exploit should fail on our device if it was using an unmodified dlmalloc2.8.3 but it works!

And?


nccgroup
3 tools to rule them all

Create holes

- Count how many chunks live in a specific bin

```
(gdb) dlmstate 0xa8400008 -c --depth 100

smallbin[00] (sz 0x0)   = 0xa840002c, 0xa840002c [EMPTY]
smallbin[01] (sz 0x8)   = 0xa8400034, 0xa8400034 [EMPTY]
smallbin[02] (sz 0x10)  = 0xac620b88, 0xa88647f0 [77]
smallbin[03] (sz 0x18)  = 0xacfee0b8, 0xa9689a28 [68]
smallbin[04] (sz 0x20)  = 0xac96b920, 0xa87206f8 [40]
...
smallbin[24] (sz 0xc0)  = 0xac789038, 0xac789038 [EMPTY]
...
```

- Send lots of fragments of same size in two different sessions and free one of them only
  - 0x70-byte IKEv1 fragments give 0xc0-byte allocations

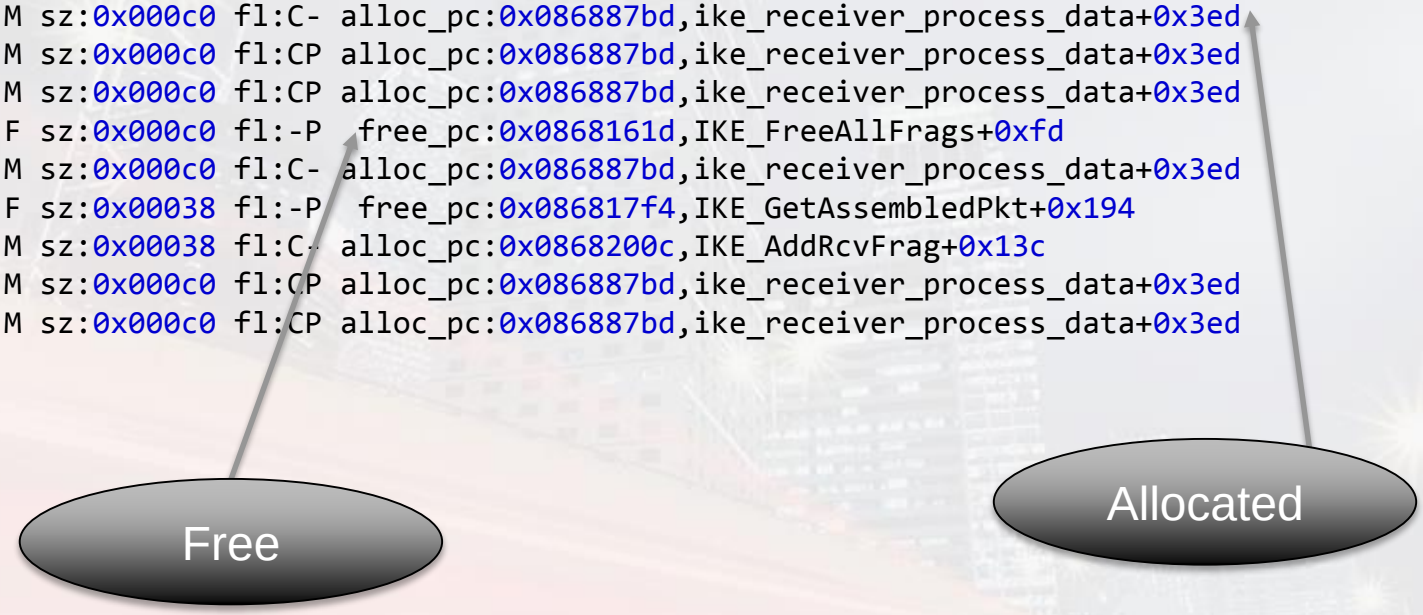
```
(gdb) dlmstate 0xa8400008 -c --depth 100

smallbin[00] (sz 0x0)   = 0xa840002c, 0xa840002c [EMPTY]
smallbin[01] (sz 0x8)   = 0xa8400034, 0xa8400034 [EMPTY]
smallbin[02] (sz 0x10)  = 0xad03b680, 0xa88647f0 [100+]
smallbin[03] (sz 0x18)  = 0xacfee0b8, 0xa9689a28 [68]
smallbin[04] (sz 0x20)  = 0xac79a038, 0xa87206f8 [41]
...
smallbin[24] (sz 0xc0)  = 0xad01a0b8, 0xac799da8 [10]
...
```

Comes from our feng shui

```
  (gdb) dlchunk 0xad01a0b8
0xad01a0b8 F sz:0x000c0 fl:-P  free_pc:0x0868161d,IKE_FreeAllFrags+0xfd
```

Analyzing memory layout

```
(gdb) dlchunk 0xad01a3f8 -c 10
0xad01a3f8 F sz:0x000c0 fl:-P  free_pc:0x0868161d,IKE_FreeAllFrags+0xfd
0xad01a4b8 M sz:0x000c0 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed
0xad01a578 M sz:0x000c0 fl:CP alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed
0xad01a638 M sz:0x000c0 fl:CP alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed
0xad01a6f8 F sz:0x000c0 fl:-P  free_pc:0x0868161d,IKE_FreeAllFrags+0xfd
0xad01a7b8 M sz:0x000c0 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed
0xad01a878 F sz:0x00038 fl:-P  free_pc:0x086817f4,IKE_GetAssembledPkt+0x194
0xad01a8b0 M sz:0x00038 fl:C- alloc_pc:0x0868200c,IKE_AddRcvFrag+0x13c
0xad01a8e8 M sz:0x000c0 fl:CP alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed
0xad01a9a8 M sz:0x000c0 fl:CP alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed
```



Free chunk

```
(gdb) dlchunk 0xad01a0b8
0xad01a0b8 F sz:0x000c0 fl:-P  free_pc:0x0868161d,IKE_FreeAllFrags+0xfd

(gdb) dlchunk 0xad01a0b8 -v -x
struct malloc_chunk @ 0xad01a0b8 {
prev_foot    = 0x8140d4d0
head         = 0xc0 (PINUSE)
fd           = 0xad01a3f8
bk           = 0xa84000ec
struct mp_header @ 0xad01a0c8 {
mh_refcount    = 0xf3ee0123
mh_unused      = 0x0
mh_fd_link     = 0xad01a000
mh_bk_link     = 0xad01a180
alloc_pc       = 0x86887bd (ike_receiver_process_data+0x3ed)
free_pc        = 0x868161d (IKE_FreeAllFrags+0xfd)
0x98 bytes of chunk data:
0xad01a0e0:      0x50595143 0x35464131 0x114b6970 0x768ba780
0xad01a0f0:      0x08021084 0x01000000 0x94000000 0x78000000
0xad01a100:      0x00012000 0x42424242 0x42424242 0x42424242
0xad01a110:      0x42424242 0x42424242 0x42424242 0x42424242
0xad01a120:      0x42424242 0x42424242 0x42424242 0x42424242
0xad01a130:      0x42424242 0x42424242 0x42424242 0x42424242
0xad01a140:      0x42424242 0x42424242 0x42424242 0x42424242
0xad01a150:      0x42424242 0x42424242 0x42424242 0x42424242
0xad01a160:      0x42424242 0x42424242 0x42424242 0x42424242
0xad01a170:      0x42424242 0xf3eecdef
```

Allocated chunk

```
(gdb) dlchunk 0xad01a4b8
0xad01a4b8 M sz:0x000c0 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed

(gdb) dlchunk 0xad01a4b8 –v -x
struct malloc_chunk @ 0xad01a4b8 {
prev_foot    = 0xc0
size         = 0xc0 (CINUSE)
struct mp_header @ 0xad01a4c0 {
mh_magic       = 0xa11c0123
mh_len         = 0x94
mh_refcount    = 0x0
mh_unused      = 0x0
mh_fd_link     = 0xad01a300
mh_bk_link     = 0xad01a580
alloc_pc       = 0x86887bd (ike_receiver_process_data+0x3ed)
free_pc        = 0xc6eee941
0x98 bytes of chunk data:
0xad01a4e0:      0x38575156 0x38523747 0x4697cf22 0x9a6489b9
0xad01a4f0:      0x08021084 0x01000000 0x94000000 0x78000000
0xad01a500:      0x00021100 0x41414141 0x41414141 0x41414141
0xad01a510:      0x41414141 0x41414141 0x41414141 0x41414141
0xad01a520:      0x41414141 0x41414141 0x41414141 0x41414141
0xad01a530:      0x41414141 0x41414141 0x41414141 0x41414141
0xad01a540:      0x41414141 0x41414141 0x41414141 0x41414141
0xad01a550:      0x41414141 0x41414141 0x41414141 0x41414141
0xad01a560:      0x41414141 0x41414141 0x41414141 0x41414141
0xad01a570:      0x41414141 0xa11ccdef
```

Analyzing the designated victim

- Designated victim?

```
(gdb) dlmstate 0xa8400008
struct dl_mstate @ 0xa8400008 {
smallmap    = 0b00000100000110100000011011111100
treemap     = 0b00000000000000000000000000000001
dvsize      = 0x80
topsize     = 0x2ebc17c8
least_addr  = 0xa8400000
dv          = 0xad03b8e8
top         = 0xad03e808
```

- Chunk that we can easily replace

```
(gdb) dlchunk 0xad03b8e8 -v
struct malloc_chunk @ 0xad03b8e8 {
prev_foot   = 0x8140d4d0
head        = 0x80 (PINUSE)
fd          = 0x5ee3fedc
bk          = 0xc
struct mp_header @ 0xad03b8f8 {
mh_refcount   = 0xf3ee0123
mh_unused     = 0x0
mh_fd_link    = 0xad03bbd8
mh_bk_link    = 0xa84002c4
alloc_pc      = 0x868200c (IKE_AddRcvFrag+0x13c)
free_pc       = 0x86817f4 (IKE_GetAssembledPkt+0x194)
```

Analyzing mempool bins

```
(gdb) mpmstate 0xa84001e4
struct mp_mstate @ 0xa84001e4 {
mp_smallbin[00] - sz: 0x00000000 cnt: 0x0000, mh_fd_link: 0x0
mp_smallbin[01] - sz: 0x00000008 cnt: 0x0000, mh_fd_link: 0x0
mp_smallbin[02] - sz: 0x00000010 cnt: 0x0000, mh_fd_link: 0x0
mp_smallbin[03] - sz: 0x00000018 cnt: 0x0000, mh_fd_link: 0x0
mp_smallbin[04] - sz: 0x00000020 cnt: 0x0000, mh_fd_link: 0x0
mp_smallbin[05] - sz: 0x00000028 cnt: 0x0000, mh_fd_link: 0x0
mp_smallbin[06] - sz: 0x00000030 cnt: 0x0211, mh_fd_link: 0xacfeb290
mp_smallbin[07] - sz: 0x00000038 cnt: 0x0e44, mh_fd_link: 0xad0183c0
mp_smallbin[08] - sz: 0x00000040 cnt: 0x1c99, mh_fd_link: 0xad019bd0
...
mp_smallbin[24] - sz: 0x000000c0 cnt: 0x014c, mh_fd_link: 0xad01a0c0
...
mp_treebin[31]  - sz: 0xffffffff cnt: 0x0001, mh_fd_link: 0xab641758 [UNSORTED]

(gdb) mpheader 0xad01a0c0
struct mp_header @ 0xad01a0c0 {
mh_magic      = 0xa11c0123
mh_len        = 0x94
mh_refcount   = 0x0
mh_unused     = 0x0
mh_fd_link    = 0xad03b830
mh_bk_link    = 0xad01a0c0
alloc_pc      = 0x86887bd (ike_receiver_process_data+0x3ed)
free_pc       = 0x6967736d

(gdb) dlchunk 0xad01a0c0-8
0xad01a0c0 M sz:0x000c0 fl:CP alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed
```

Finding a chunk

```
(gdb) mpbinwalk -s 0x41414141 -c 5 0xc0
[libmempool] mp_header @ 0xa84004e4 - mh_len: 0x00000000, alloc_pc: 0x00000000 [BIN HEAD]
[libmempool] mp_header @ 0xad03b830 - mh_len: 0x00000094, alloc_pc: 0x086887bd
[libmempool] mp_header @ 0xad03b590 - mh_len: 0x00000094, alloc_pc: 0x086887bd
[libmempool] mp_header @ 0xad03b2f0 - mh_len: 0x00000094, alloc_pc: 0x086887bd
[libmempool] mp_header @ 0xad03b050 - mh_len: 0x00000094, alloc_pc: 0x086887bd

(gdb) dlchunk 0xad03b830-8 -v -x
struct malloc_chunk @ 0xad03b828 {
prev_foot    = 0x8140d4d0
size         = 0xc0 (CINUSE|PINUSE)
struct mp_header @ 0xad03b830 {
mh_magic       = 0xa11c0123
mh_len         = 0x94
mh_refcount    = 0x0
mh_unused      = 0x0
mh_fd_link     = 0xad03b590
mh_bk_link     = 0xad03b9c0
alloc_pc       = 0x86887bd (ike_receiver_process_data+0x3ed)
free_pc        = 0x66666972
0x98 bytes of chunk data:
0xad03b850:      0x38575156 0x38523747 0x4697cf22 0x9a6489b9
0xad03b860:      0x08021084 0x01000000 0x94000000 0x78000000
0xad03b870:      0x00631100 0x41414141 0x41414141 0x41414141
...
0xad03b8d0:      0x41414141 0x41414141 0x41414141 0x41414141
0xad03b8e0:      0x41414141 0xa11ccdef
```

Filling holes

- Chunks are adjacent after filling all the holes

```
malloc: 0xad03c970, realsz 0x0190, reqsz 0x0164 - struct packet_ike*
malloc: 0xa98af380, realsz 0x0150, reqsz 0x011d - struct packet_ike*
malloc: 0xadbb2958, realsz 0x01f0, reqsz 0x01c2 - struct packet_ike*
malloc: 0xadbbf9d0, realsz 0x0190, reqsz 0x0164 - struct packet_ike*
malloc: 0xadbbfb30, realsz 0x0190, reqsz 0x0164 - struct packet_ike*
malloc: 0xadbc5bc8, realsz 0x0190, reqsz 0x0164 - struct packet_ike*
malloc: 0xadbbfb30, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xad896018, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xad896218, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xac8e40e0, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xac8e42e0, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
...
malloc: 0xadbd0118, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xadbd0318, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xadbd0518, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xadbd0718, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xadbd0918, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xadbd0b18, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xadbd0d18, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xadbd0f18, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
malloc: 0xadbd1118, realsz 0x0200, reqsz 0x01d4 - struct packet_ike*
```

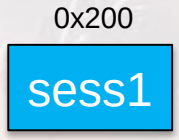
Simple reassembly (32-bit dlmalloc)

- Sending 3 fragments and asking for a reassembly

```
FUNC?     ADDRESS?    CHUNK SIZE? FLAGS?                    WHO ALLOCATED THIS?                    WHAT FOR?
// fragment 1
malloc -> 0xacb889d8 M sz:0x00050 fl:CP alloc_pc:0x0868875e,ike_receiver_process_data+0x38e // pkt info
malloc -> 0xacb88dd8 M sz:0x00090 fl:CP alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed // packet ike
malloc -> 0xad424e10 M sz:0x00048 fl:CP alloc_pc:0x08688eb7,enqueue_ike_ext_action+0x17
calloc -> 0xacb89190 M sz:0x00048 fl:CP alloc_pc:0x086821dc,IKE_AddRcvFrag+0x30c            // frag queue
malloc -> 0xacb98c00 M sz:0x00038 fl:CP alloc_pc:0x0868200c,IKE_AddRcvFrag+0x13c            // queue entry
free   -> 0xad424e10 M sz:0x00048 fl:CP alloc_pc:0x08688eb7,enqueue_ike_ext_action+0x17

// fragment 2
malloc -> 0xacb98c38 M sz:0x00050 fl:CP alloc_pc:0x0868875e,ike_receiver_process_data+0x38e // pkt info
malloc -> 0xacb99408 M sz:0x00090 fl:CP alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed // packet ike
malloc -> 0xad424e10 M sz:0x00048 fl:CP alloc_pc:0x08688eb7,enqueue_ike_ext_action+0x17
malloc -> 0xacb99498 M sz:0x00038 fl:CP alloc_pc:0x0868200c,IKE_AddRcvFrag+0x13c            // queue entry
free   -> 0xad424e10 M sz:0x00048 fl:CP alloc_pc:0x08688eb7,enqueue_ike_ext_action+0x17

// fragment 3
malloc -> 0xacb89e50 M sz:0x00050 fl:CP alloc_pc:0x0868875e,ike_receiver_process_data+0x38e // pkt info
malloc -> 0xacb89ea0 M sz:0x00090 fl:CP alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed // packet ike
malloc -> 0xad424e10 M sz:0x00048 fl:CP alloc_pc:0x08688eb7,enqueue_ike_ext_action+0x17
malloc -> 0xacb88870 M sz:0x00038 fl:CP alloc_pc:0x0868200c,IKE_AddRcvFrag+0x13c            // queue entry
malloc -> 0xacb985d8 M sz:0x00100 fl:CP alloc_pc:0x086816b3,IKE_GetAssembledPkt+0x53        // reassembled packet
```

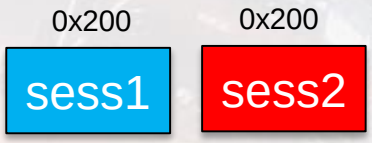
Reassembly in a hole (32-bit dlmalloc)

- Scenario



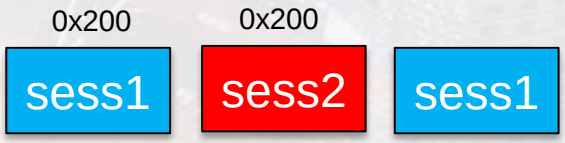
Reassembly in a hole (32-bit dlmalloc)

- Scenario



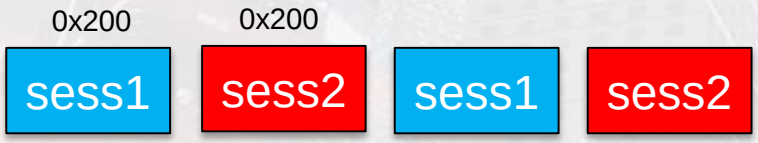
Reassembly in a hole (32-bit dlmalloc)

- Scenario



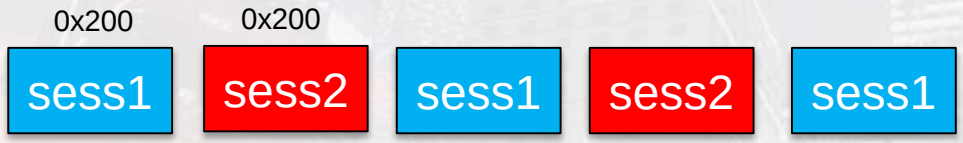
Reassembly in a hole (32-bit dlmalloc)

- Scenario



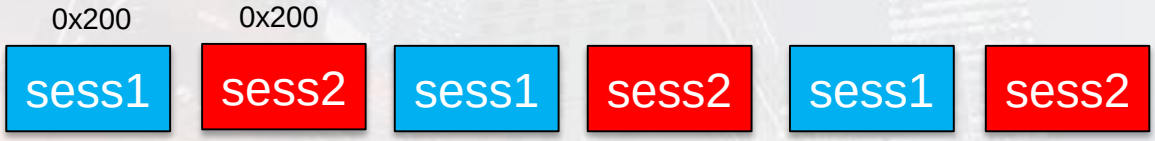
Reassembly in a hole (32-bit dlmalloc)

- Scenario



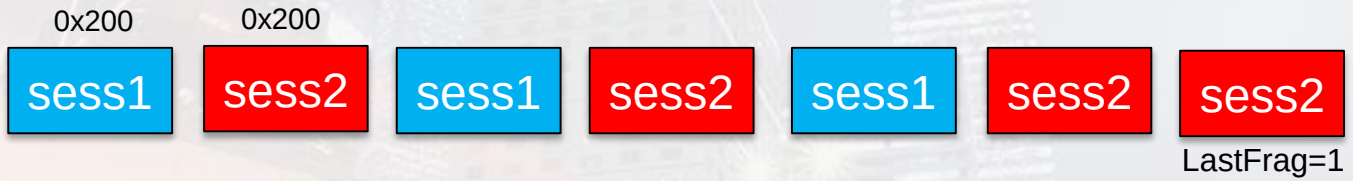
Reassembly in a hole (32-bit dlmalloc)

- Scenario



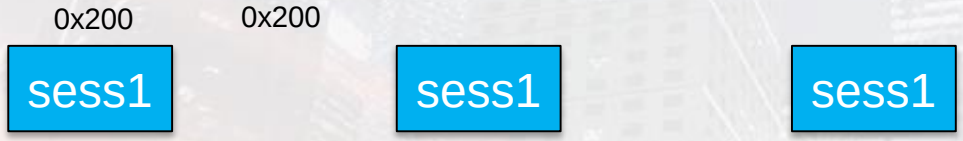
Reassembly in a hole (32-bit dlmalloc)

- Scenario



Reassembly in a hole (32-bit dlmalloc)

- Scenario



- Creating holes

```
0xa9901a18 M sz:0x00200 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed   //sess1
0xa9901c18 F sz:0x00200 fl:-P  free_pc:0x0868161d,IKE_FreeAllFrags+0xfd             //sess2 - hole
0xa9901e18 M sz:0x00200 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed   //sess1
0xa9902018 F sz:0x00200 fl:-P  free_pc:0x0868161d,IKE_FreeAllFrags+0xfd             //sess2 - hole
0xa9902218 M sz:0x00200 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed   //sess1
```

Reassembly in a hole (32-bit dlmalloc)

- Scenario

- Creating holes

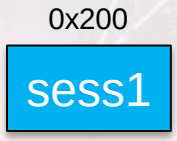
```
0xa9901a18 M sz:0x00200 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed   //sess1
0xa9901c18 F sz:0x00200 fl:-P  free_pc:0x0868161d,IKE_FreeAllFrags+0xfd             //sess2 - hole
0xa9901e18 M sz:0x00200 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed   //sess1
0xa9902018 F sz:0x00200 fl:-P  free_pc:0x0868161d,IKE_FreeAllFrags+0xfd             //sess2 - hole
0xa9902218 M sz:0x00200 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed   //sess1
```

- Triggering reassembly into a hole

```
0xa9901a18 M sz:0x00200 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed
0xa9901c18 M sz:0x001d0 fl:CP alloc_pc:0x086816b3,IKE_GetAssembledPkt+0x53   //reassembled packet
0xa9901de8 F sz:0x00030 fl:-P  free_pc:0x42424242,-                          //remaining 0x30 hole
0xa9901e18 M sz:0x00200 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed
0xa9902018 F sz:0x00200 fl:-P  free_pc:0x0868161d,IKE_FreeAllFrags+0xfd             //hole
0xa9902218 M sz:0x00200 fl:C- alloc_pc:0x086887bd,ike_receiver_process_data+0x3ed
```

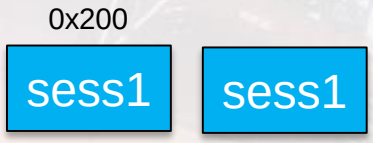
Brainstorming

- Exploit scenario



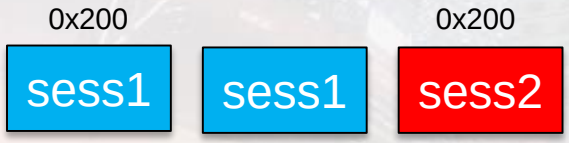
Brainstorming

- Exploit scenario



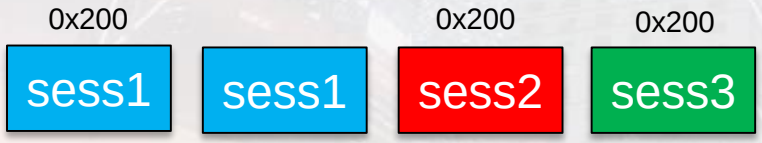
Brainstorming

- Exploit scenario



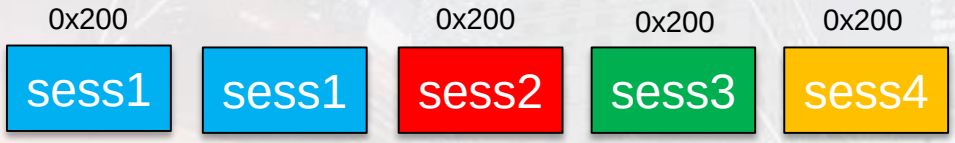
Brainstorming

- Exploit scenario



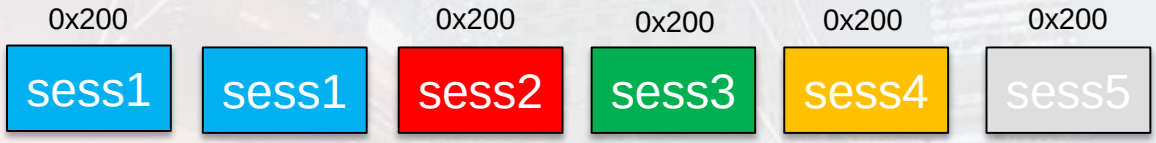
Brainstorming

- Exploit scenario



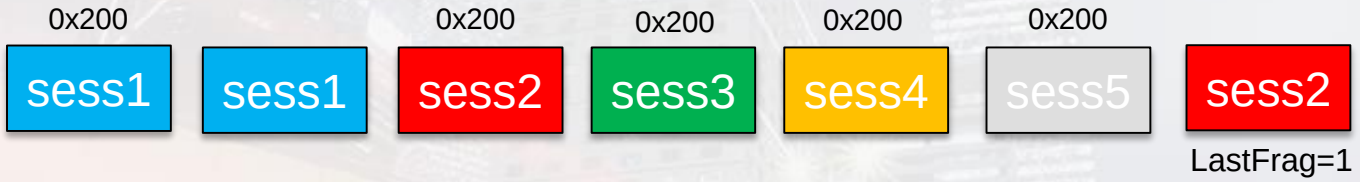
Brainstorming

- Exploit scenario



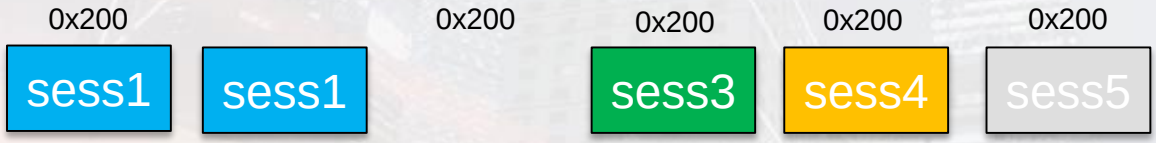
Brainstorming

- Exploit scenario



Brainstorming

- Exploit scenario



Brainstorming

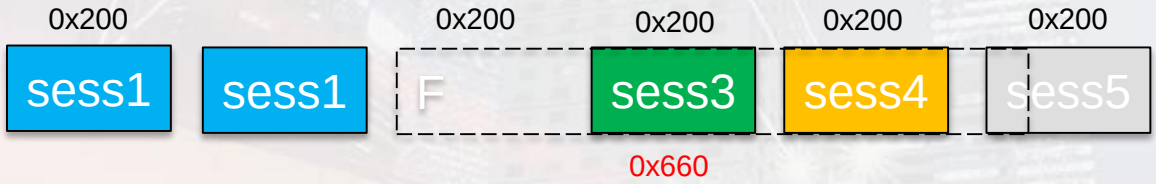
- Exploit scenario

Brainstorming

- Exploit scenario

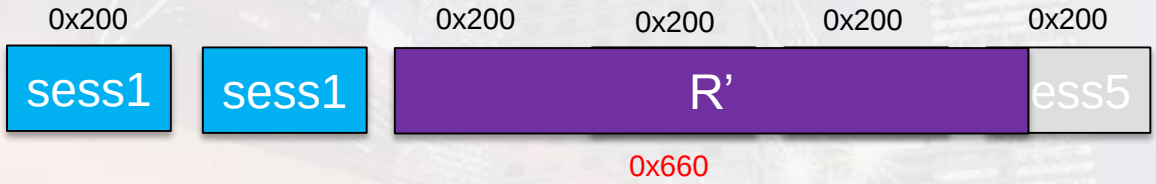
Brainstorming

- Exploit scenario



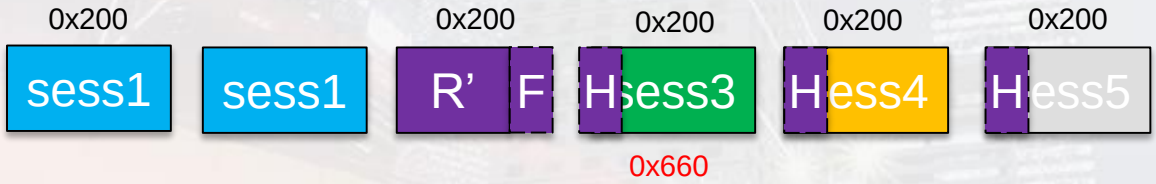
Brainstorming

- Exploit scenario



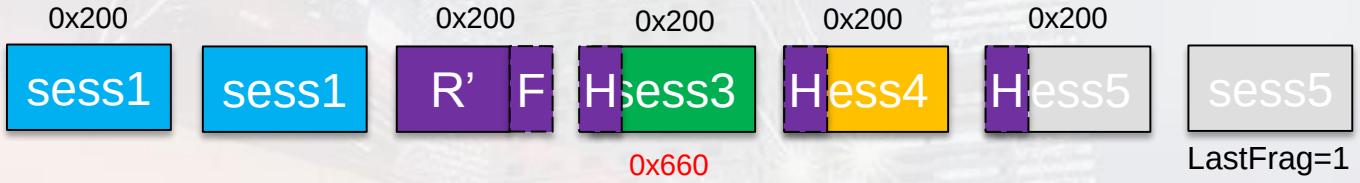
Brainstorming

- Exploit scenario



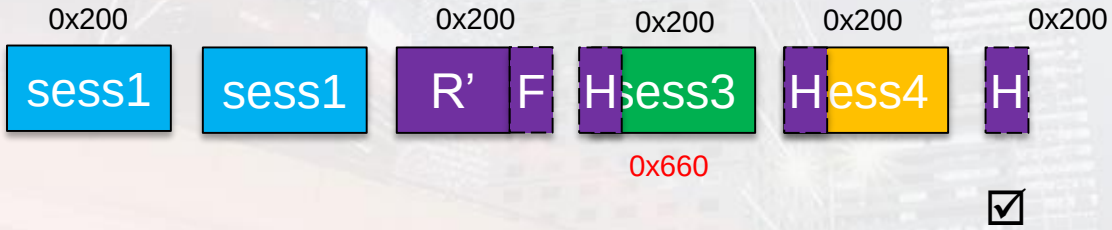
Brainstorming

- Exploit scenario



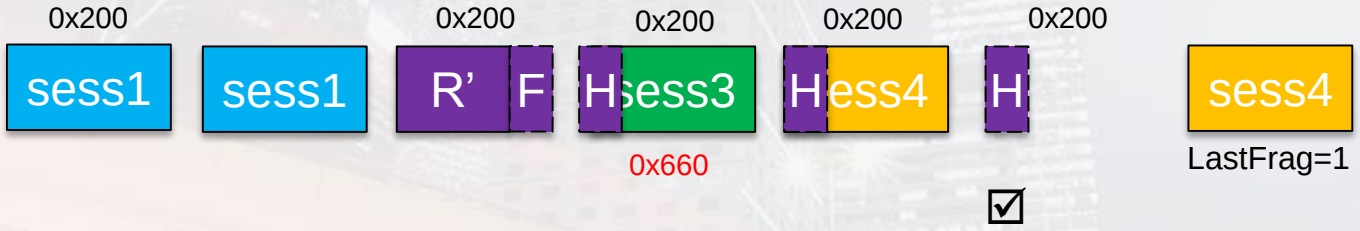
Brainstorming

- Exploit scenario



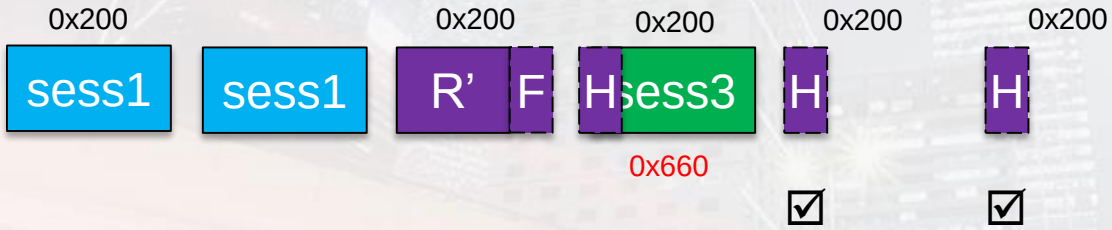
Brainstorming

- Exploit scenario



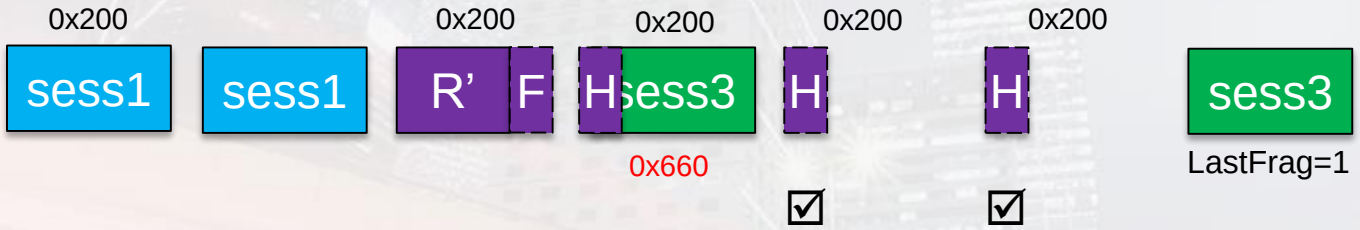
Brainstorming

- Exploit scenario



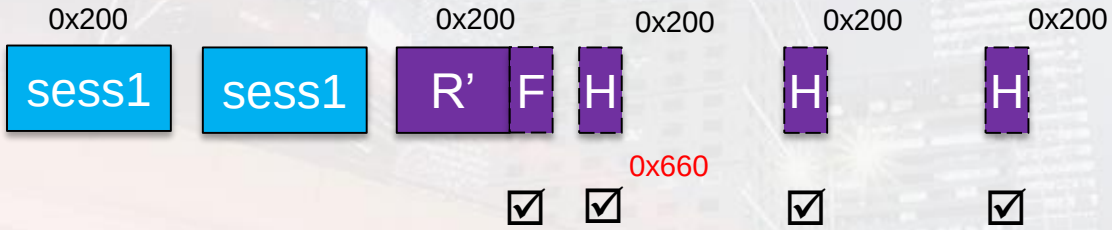
Brainstorming

- Exploit scenario



Brainstorming

- Exploit scenario



Reassembly before n frags (64-bit dlmalloc)

- Creating a hole before chunks from different IKE sessions

```
(gdb) dlchunk 0x7fff9a002bf0 -x -c 6
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x180 bytes of chunk data:
0x7fff9a002c30:     0xf3ee0123    0x39414958    0x87f71899    0x00010dcf
[...]
0x7fff9a002da0:     0x00000000    0x00000000    0x00000000    0xa11ccdef
--
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
[!] Empty chunk?
--
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
0x1c0 bytes of chunk data:
0x7fff9a002e30:     0x51354745    0x304c4739    0xc766cacb    0x5e5f8351
[...]
0x7fff9a002fe0:     0x43434343    0x43434343    0x43434343    0xa11ccdef
--
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
0x1c0 bytes of chunk data:
0x7fff9a003030:     0x594a4d52    0x4e374352    0xf6fd0875    0x4618d8ff
[...]
0x7fff9a0031e0:     0x44444444    0x44444444    0x44444444    0xa11ccdef
--
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
0x1c0 bytes of chunk data:
0x7fff9a003230:     0x595a5533    0x58324634    0x693dce53    0x48b693ac
[...]
0x7fff9a0033e0:     0x45454545    0x45454545    0x45454545    0xa11ccdef
--
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
0x1c0 bytes of chunk data:
0x7fff9a003430:     0x58374f44    0x4a36544e    0x19cca95d    0x37842435
[...]
0x7fff9a0035e0:     0x46464646    0x46464646    0x46464646    0xa11ccdef
```

Debugging overflow (64-bit dlmalloc)

- Before overflow

```
malloc: 0x7fff9a002bf0, realsz 0x01c0, reqsz 0x017c - reassembled packet
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 43434343
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 44444444
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 45454545
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 46464646
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

Debugging overflow (64-bit dlmalloc)

- Before overflow

```
malloc: 0x7fff9a002bf0, realsz 0x01c0, reqsz 0x017c - reassembled packet
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 43434343
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 44444444
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 45454545
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 46464646
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After overflowing the following chunk's size field

```
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x006b0 fl:-P  free_pc:0x00000202,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

Debugging overflow (64-bit dlmalloc)

- Before overflow

```
malloc: 0x7fff9a002bf0, realsz 0x01c0, reqsz 0x017c - reassembled packet
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 43434343
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 44444444
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 45454545
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 46464646
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After overflowing the following chunk's size field

```
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x006b0 fl:-P  free_pc:0x00000202,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

Debugging overflow (64-bit dlmalloc)

- Before overflow

```
malloc: 0x7fff9a002bf0, realsz 0x01c0, reqsz 0x017c - reassembled packet
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 43434343
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 44444444
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 45454545
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 46464646
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After overflowing the following chunk's size field

```
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x006b0 fl:-P  free_pc:0x00000202,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

Debugging overflow (64-bit dlmalloc)

- Before overflow

```
malloc: 0x7fff9a002bf0, realsz 0x01c0, reqsz 0x017c - reassembled packet
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 43434343
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 44444444
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 45454545
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 46464646
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After overflowing the following chunk's size field

```
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x006b0 fl:-P  free_pc:0x00000202,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

Debugging overflow (64-bit dlmalloc)

- Before overflow

```
malloc: 0x7fff9a002bf0, realsz 0x01c0, reqsz 0x017c - reassembled packet
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 43434343
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 44444444
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 45454545
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 46464646
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After overflowing the following chunk's size field

```
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x006b0 fl:-P  free_pc:0x00000202,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After corrupted chunk is coalesced

```
0x7fff9a002bf0 F sz:0x00870 fl:-P  free_pc:0x10dcf87f71899,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

Debugging overflow (64-bit dlmalloc)

- Before overflow

```
malloc: 0x7fff9a002bf0, realsz 0x01c0, reqsz 0x017c - reassembled packet
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 43434343
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 44444444
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 45454545
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 46464646
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After overflowing the following chunk's size field

```
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x006b0 fl:-P  free_pc:0x00000202,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After corrupted chunk is coalesced

```
0x7fff9a002bf0 F sz:0x00870 fl:-P  free_pc:0x10dcf87f71899,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

Debugging overflow (64-bit dlmalloc)

- Before overflow

```
malloc: 0x7fff9a002bf0, realsz 0x01c0, reqsz 0x017c - reassembled packet
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 43434343
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 44444444
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 45454545
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 46464646
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After overflowing the following chunk's size field

```
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x006b0 fl:-P  free_pc:0x00000202,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After corrupted chunk is coalesced

```
0x7fff9a002bf0 F sz:0x00870 fl:-P  free_pc:0x10dcf87f71899,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

Debugging overflow (64-bit dlmalloc)

- Before overflow

```
malloc: 0x7fff9a002bf0, realsz 0x01c0, reqsz 0x017c - reassembled packet
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 43434343
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 44444444
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 45454545
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 46464646
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After overflowing the following chunk's size field

```
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x006b0 fl:-P  free_pc:0x00000202,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After corrupted chunk is coalesced

```
0x7fff9a002bf0 F sz:0x00870 fl:-P  free_pc:0x10dcf87f71899,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After sending a packet to replace chunk

```
0x7fff9a002bf0 M sz:0x00870 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
0x7fff9a003460 M sz:0x00190 fl:CP alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

Debugging overflow (64-bit dlmalloc)

- Before overflow

```
malloc: 0x7fff9a002bf0, realsz 0x01c0, reqsz 0x017c - reassembled packet
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 43434343
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 44444444
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 45454545
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 46464646
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After overflowing the following chunk's size field

```
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x006b0 fl:-P  free_pc:0x00000202,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After corrupted chunk is coalesced

```
0x7fff9a002bf0 F sz:0x00870 fl:-P  free_pc:0x10dcf87f71899,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After sending a packet to replace chunk

```
0x7fff9a002bf0 M sz:0x00870 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
0x7fff9a003460 M sz:0x00190 fl:CP alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

Debugging overflow (64-bit dlmalloc)

- Before overflow

```
malloc: 0x7fff9a002bf0, realsz 0x01c0, reqsz 0x017c - reassembled packet
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x00040 fl:-P  free_pc:0x00000040,-
0x7fff9a002df0 M sz:0x00200 fl:C- alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 43434343
0x7fff9a002ff0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 44444444
0x7fff9a0031f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 45454545
0x7fff9a0033f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360  // 46464646
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After overflowing the following chunk's size field

```
0x7fff9a002bf0 M sz:0x001c0 fl:CP alloc_pc:0x00a06235,IKE_GetAssembledPkt+0x75
0x7fff9a002db0 F sz:0x006b0 fl:-P  free_pc:0x00000202,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After corrupted chunk is coalesced

```
0x7fff9a002bf0 F sz:0x00870 fl:-P  free_pc:0x10dcf87f71899,-
0x7fff9a003460 M sz:0x00190 fl:C- alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

- After sending a packet to replace chunk

```
0x7fff9a002bf0 M sz:0x00870 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
0x7fff9a003460 M sz:0x00190 fl:CP alloc_pc:0x4646464646464646,-
0x7fff9a0035f0 M sz:0x00200 fl:CP alloc_pc:0x00a0cc50,ike_receiver_process_data+0x360
```

Conclusion

64-bit vs 32-bit

- Old 64-bit = same as 32-bit

- On newer 64-bit (e.g. asav962-7.qcow2)
- `*malloc()` → `resMgrCalloc()` → `mem_mh_calloc()` → `mem_mh_generic_malloc()` → `___libc_malloc(size_t)`
- Call external `malloc()` from glibc.so

- Default heap allocator on Cisco ASA
  - Old versions (32-bit + 64-bit): dlmalloc2.8.3
  - Recent versions (64-bit only): Glibc

- No safe unlinking on mempool header even on newer 64-bit devices
  - Even though glibc/ptmalloc2 having safe unlinking

- Need a library to parse ptmalloc2 chunks (glibc)

3 libs to rule them all

- Very similar to “libtalloc” by Aaron adams or “shadow” (jemalloc) by argp / huku

- libmempool: gdb plugin to analyse Cisco-specific metadata
  - Metadata embedded into both dlmalloc2.8 and ptmalloc2 chunks
  - Tested on both 32-bit and 64-bit
  - Uses ret-sync to retrieve symbols from IDA Pro
- libdlmalloc
  - Tested on both 32-bit and 64-bit
  - Supports callback to integrate with libmempool
- libptmalloc
  - Tested on 64-bit
  - Fork of libheap developed by cloudburst
  - Main reasons for developing our own
    - Consistency with other commands in libtalloc, libdlmalloc, libmempool
    - Support callback to integrate with libmempool

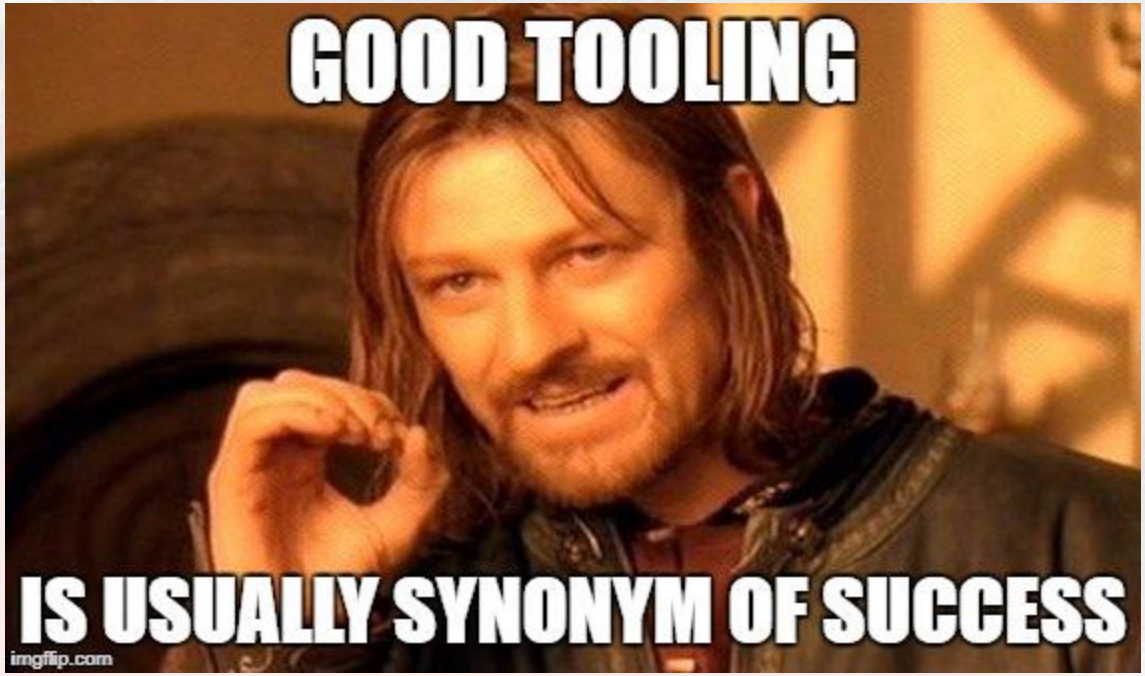
- See references at the end of the slides deck

Cisco ASA summary

Version	Heap	Mempool header	Heap safe unlinking	Mempool safe unlinking	Footer
<= 9.3.1	dlmalloc2.8.3	Yes	No	No	Broken
>= 9.3.2.200	ptmalloc2/glibc	Yes	Yes	No	TBC

- Cisco added security mitigations over time
  - Heap safe-unlinking
  - ASLR
  - DEP

Conclusion



Future work

- Abstract out the gdb-related portions so the tools can be easily integrated into other debuggers
  - As done in the more recent versions of shadow/libheap
- Test on more targets
  - dlmalloc/ptmalloc
  - Other Cisco devices (like non ASAs)
  - Other embedded devices
- Implement other commands
  - Easy bin walking/searching
  - Linear in-use searching in a given segment

Questions?

- If you have any questions, contact me:
  - cedric.halbronn@nccgroup.trust
  - @saidelike

References

- https://github.com/nccgroup/libdlmalloc
- https://github.com/nccgroup/libptmalloc
- https://github.com/nccgroup/libmempool

- http://gee.cs.oswego.edu/pub/misc/malloc-2.8.3.h
- http://gee.cs.oswego.edu/pub/misc/malloc-2.8.3.c
- https://github.com/cloudburst/libheap
- https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20160210-asa-ike
- https://cansecwest.com/slides/2016/CSW2016_Wheeler-Barksdale-Gruskovnjak_ExecuteMyPacket.pdf
- https://blog.exodusintel.com/2016/02/10/firewall-hacking/
- https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2017/june/a-warcon-2017-presentation-cisco-asa-exploiting-the-ikev1-heap-overflow-cve-2016-1287/