Cisco ASA Episode 1: A Fragment to rule them all - Exploiting the IKEv1 heap overflow

WarCon – June 2017

Agenda

- Previous work
- ASA internals
- Brainstorming
- Heap feng shui
- From mirror write to RCE
- Conclusion

Previous work

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 and ASA 9.2.4 only

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

Pwnie for Best Server-Side Bug

Awarded to the researchers who discovered or exploited the most technically sophisticated and interesting server-side bug. This includes any software that is accessible remotely without using user interaction.

- Cisco ASA IKEv1/IKEv2 Fragmentation Heap Buffer Overflow (CVE-2016-1287)
  Credit: David Barksdale, Jordan Gruskovnjak, and Alex Wheeler

  Cisco's ASA (Ancient Security Architecture) firewalls had a vulnerability in their IKE fragment re-assembly that permitted remote unauthenticated heap memory corruption. Thanks to a lack of non-executable memory and ASLR protections, these Exodus researchers were able to turn this vulnerability into an epic win just as if they were exploiting a late 90's Linux box. It just turns out that this late 90's Linux box happens to be your firewall/NIDS/VPN/IRC Bouncer. Yay.

Execute My Packet

Open Problems

Reliability:

- Didn't try to achieve gov grade exploit (Pareto's law is a good metric for exploit dev). Just look at the timeline to see it'll take forever
- Concurrent connections will mess with the heap

Targeting:

- Shellcode is not version independant (hardcoded values)
- Need to have a binary version of firmware to add a new target

Non-Factors:

- ASLR / DEP mitigation
- Up to date dlmalloc implementation (safe unlinking)
- 64-bit binaries will probably need different exploit technique (bigger heap metadata size)

Other questions

- How to improve the reliability of the current exploit?
- What about IKEv1?
- What mitigations in newer firmware and how to bypass them?
- How an attacker can leak the ASA version?
- What heap manager do they really use?

Today’s objective

- Previously ported XI IKEv2 exploit to all ASA versions → used internally by pentesters
- Clients are disabling IKEv2 and moving back to IKEv1, WTF!?

- Let’s build an exploit for IKEv1
- This presentation demonstrates the involved methodology
  - Ideology: Solving one problem at a time
  - Is it really exploitable? PoC||GTFO
  - Finding the quickest way to achieve RCE

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

- Presentation focuses on 32-bit
  - E.g. hardcoded sizes
  - Most concepts apply to 64-bit too

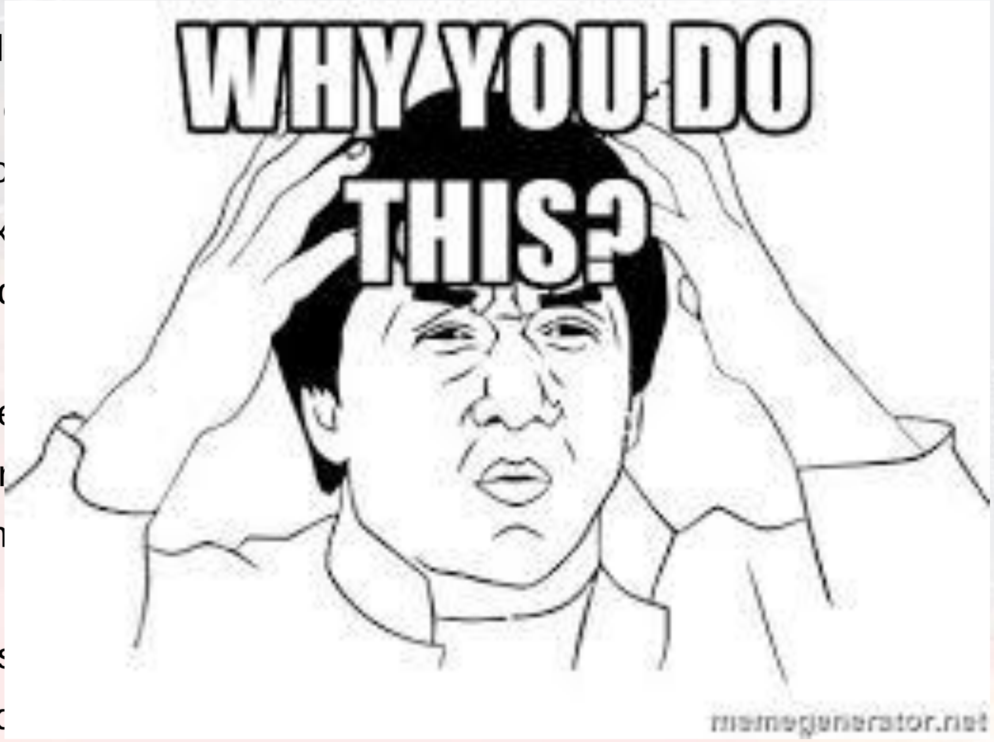
Today's objective

- Previously ported XI IKEv2 exploit to all ASA versions → used internally by pentesters
- Clients are disabling IKEv2 and moving back to IKEv1, WTF!?

- Let's build an expl
- This presentation
  - Ideology: Sc
  - Is it really ex
  - Finding the

- Exploit Developme
  - Cedric Halbr
  - Aaron Adam

- Presentation focus
  - E.g. hardcoc
  - Most concepts apply to 64-bit too



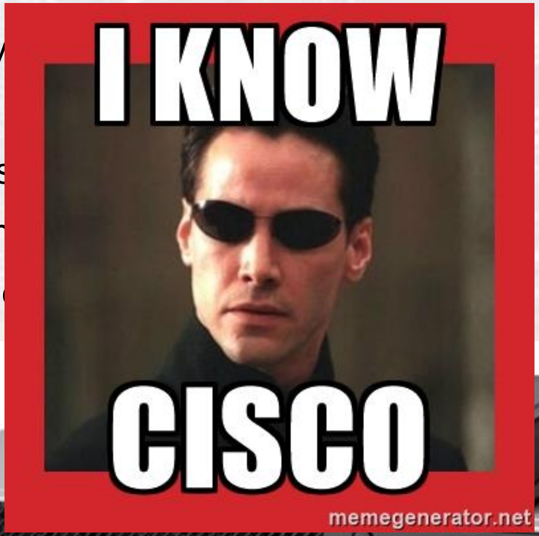
ASA internals

ASA

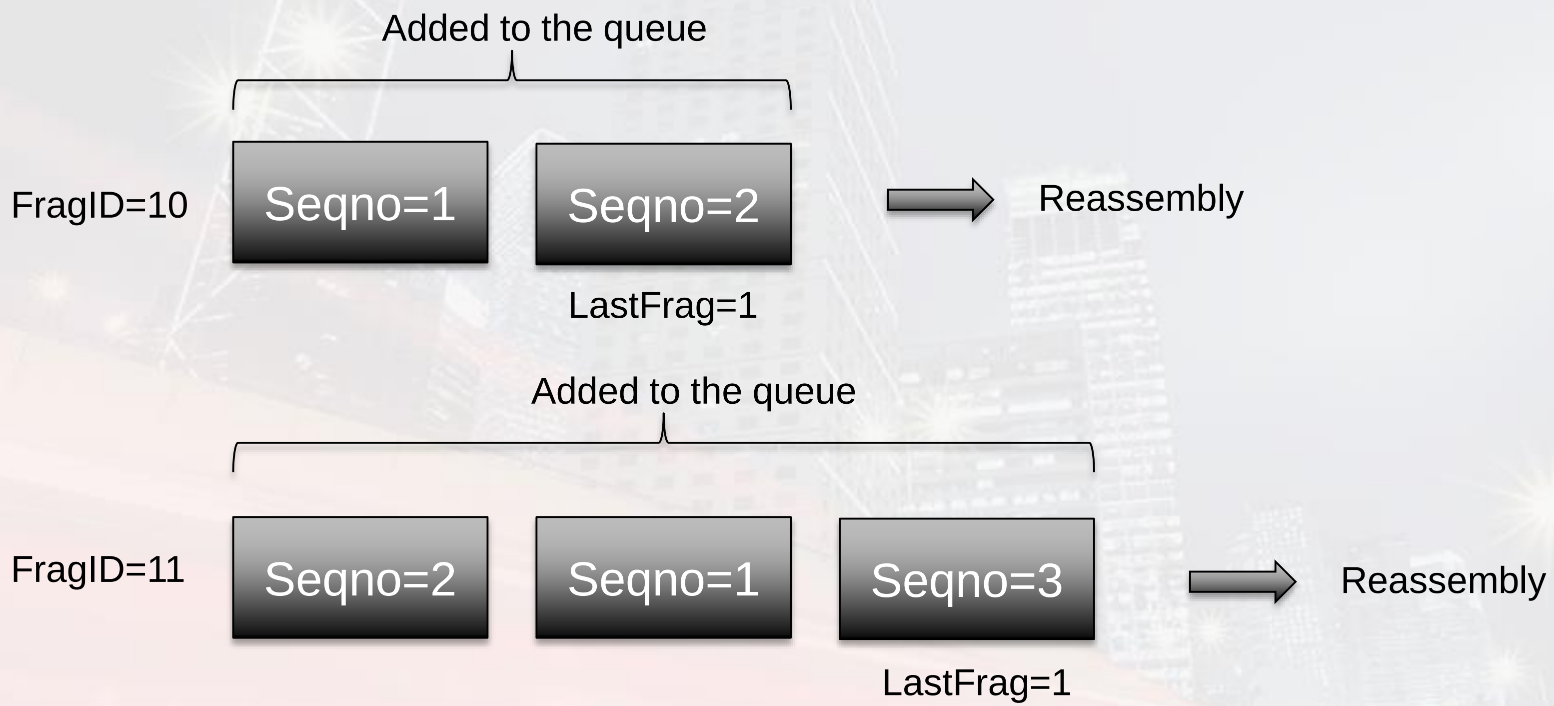
- ASA stands for “Adaptive Security Appliance”
- Different hardware but same software underneath
- x86 or x86-64 (SMP, ASAv)
- Features: firewall, VPN gateway, router
- ASA = Linux + “/asa” folder
    - Different than IOS which is a proprietary OS
- “/asa/bin/lina” contains everything (ELF is 40MB)
    - E.g.: no network at Linux level as handled by “lina”

ASA

- ASA stands for “Adaptive Security Appliance”
- Different hardware but same software underneath
- x86 or x86-64 (SMP, ASAv)
- Features: firewall, VPN gateway
- ASA = Linux + “/asa” folder
    - Different than IOS which is
- “/asa/bin/lina” contains everythin
    - E.g.: no network at Linux l



Cisco Fragmentation basics



- 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

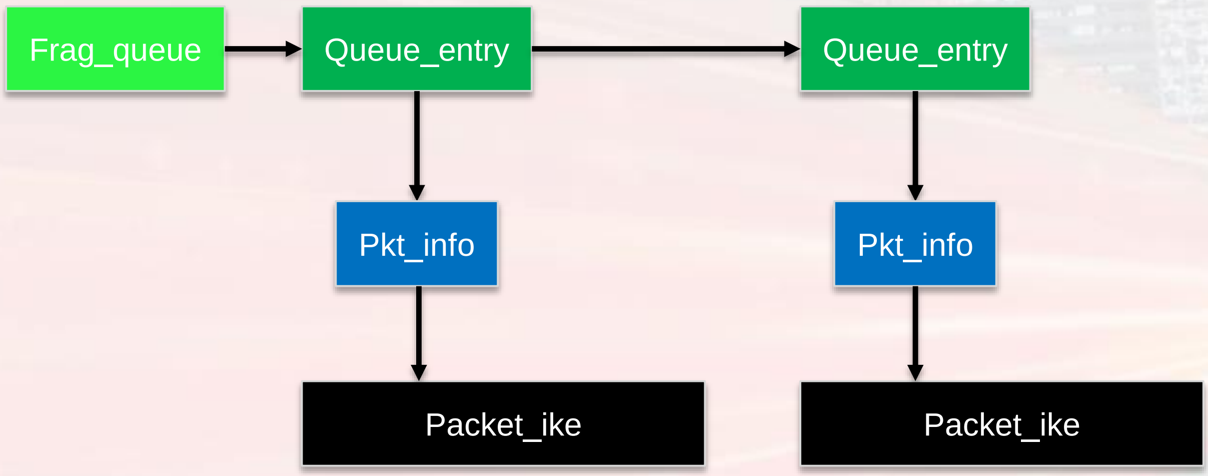
Reversing – Packet allocation

- IKEv1 packet handled by the IKE receiver thread
  - Allocate a buffer to hold the IKE packet before sending it over IPC to the right thread
    - Pkt_info: `malloc(0x24)`
    - Packet_ike: `malloc(msg->len)`

- After some validation, `ikev1_parse_packet()` is called
  - Check if the embedded payload is a Cisco Fragment
    - Call two functions
      - `IKE_AddRcvFrag()`
      - `IKE_GetAssembledPkt()`

Reversing – Fragment processing

- `IKE_AddRcvFrag()`
  - If queue does not exist
    - Frag_queue: `malloc(0x14)`
  - If LastFrag=1, save the Seqno to LastFrag_Seqno
  - Update the total length. Underflow happens here
    - `assembled_len += (fragment_payload->payload_length - 8)`
  - Add fragment entry to the queue list
    - Queue_entry: `malloc(0xC)` tracking Packet_ike



Reversing – Fragment processing

- `IKE_GetAssembledPkt()`
  - Exit if number of fragments is different than LastFrag_Seqno
  - Reass_pkt: `malloc(assembled_len + 20)`
    - Extra 20 is to hold `assembled_len` before actual data
  - Loop on all fragments
    - Search for Seqno=1, then Seqno=2, etc.
    - When the Seqno > LastFrag_Seqno, successfully exit the loop
    - If one Seqno is not found, exit the loop (failure)
    - Otherwise `memcpy()` the fragment into the reassembled packet

Reversing – Incomplete check

- `IKE_GetAssembledPkt()`
  - There is actually a check before `memcpy()` fragment to make sure we don't copy OOB

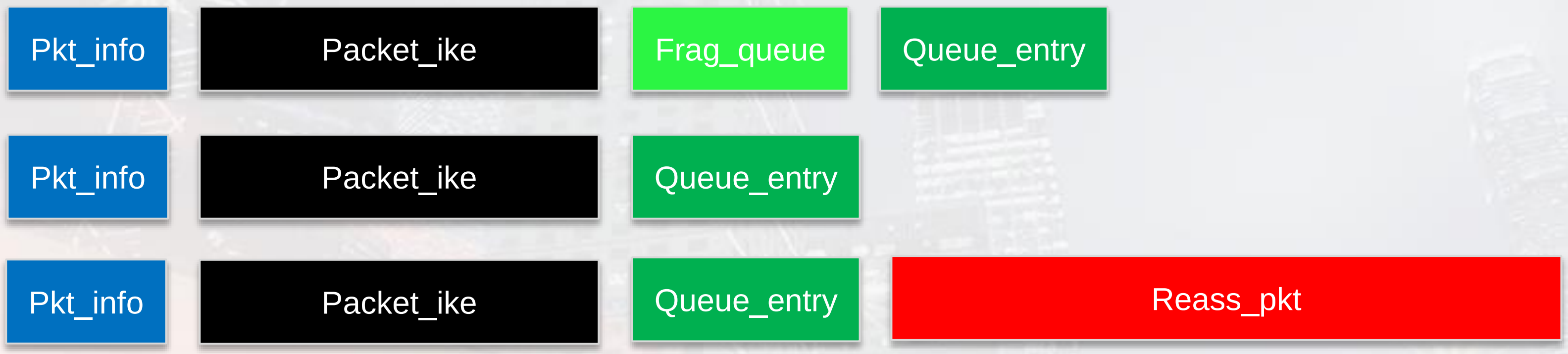
```
// allocate reassembled packet. Note the extra 20 for the size of struct reass_pkt
int alloc_size = assembled_len + 20;
struct reass_pkt* reass_pkt = malloc(alloc_size);
int curr_reass_len = 0;

while (TRUE) {
    ...
    // update the reassembled packet length
    int curr_frag_len = payload_length - 8;
    curr_reass_len += curr_frag_len;
    ...

    // Incomplete check. Does not take into account sizeof(struct reass_pkt)
    if (alloc_size < curr_reass_len) {
        es_PostEvent("Error assembling fragments! Fragment data longer than packet.");
        goto free_buffer;
    }
```

Dynamic analysis

- Logging allocation when fragments are received
  - By setting a breakpoint on `malloc`/`free` in the IKEv1 thread



- Small fixed allocations: Pkt_info: 0x50 | Frag_queue: 0x48 | Queue_entry: 0x38
- Packet_ike: variable length
- Reass_pkt: variable length
- An additional 0x48 allocation but free right away
- Sizes apply to ASA 32-bit

- Do not allocate temporary buffers (used by XI IKEv2 exploit)
- Do not allocate buffers for fragments only – keep a reference to the complete IKEv1 packet

➔ Relatively few allocations (layout completely different than IKEv2 – but actually simpler ☺)

Brainstorming

Exploit strategy

- Constraint: max 20-byte between
  - The reassembly length
  - The length provided to `memcpy()`

- Fragments added to the queue:
  - Seqno=0 and Seqno=3 have a length of 1 (resulting in -7-7)
  - Seqno=4 has a length of 2 (resulting in -6)
  - Seqno=1 is the only fragment with a valid length (e.g. ~0x200 bytes)
- In total -7-7-6 = -20 is added to the reassembly length

- Reassembly
  - Loop begins at 1 and exits as soon as a Seqno is not found
    - Seqno=1 is copied
    - All other fragments are skipped because Seqno=2 cannot be found

➔ Initial overflow very similar to XI approach for IKEv2

Heap metadata

- `malloc(int len)` → `resMgrMalloc()` → `mem_mh_malloc()` → `mspace_malloc()`
  - `resMgrMalloc()`: resource manager dispatches to the right underlying function
  - `mem_mh_malloc()`
    - “mh" likely stands for mempool header (Cisco specific) / mempool abbreviated “mp”
    - Allocates `len+0x24` (0x20 for `mp_header` / 0x4 for `mp_footer`)
  - `mspace_malloc()` actually allocates memory (dlmalloc.c)
    - Allocates `len+0x24+0x8`
  - After `mspace_malloc()` returns, `mem_mh_malloc()` fills the `mp_header`/`mp_footer`

```
struct malloc_chunk @ 0xacb96a08 {
prev_foot = 0x8180d4d0
size      = 0x1d0 (CINUSE|PINUSE)
struct mp_header @ 0xacb96a10 {
mh_magic     = 0xa11c0123
mh_len       = 0x1a4
mh_refcount  = 0x0
mh_unused    = 0x0
mh_fd_link   = 0xacb85b30
mh_bk_link   = 0xa8800604
allocator_pc = 0x86816b3 (IKE_GetAssembledPkt+0x53)
free_pc      = 0x868161d (IKE_FreeAllFrags+0xfd)
0x1a8 bytes of chunk data:
0xacb96a30: 0x394d3943  0x59305239  0x747490ad  0x00163dff
0xacb96a40: 0x08021084  0x01000000  0xd4010000  0xb8010000
...
0xacb96bd0: 0x00000000  0xa11ccdef
```

```
struct malloc_chunk @ 0xacb96bd8 {
prev_foot = 0x8180d4d0
head      = 0x30 (PINUSE)
fd        = 0xac825ab8
bk        = 0xa880005c
struct mp_header @ 0xacb96be8 {
mh_refcount  = 0xf3ee0123
mh_unused    = 0x0
mh_fd_link   = 0x0
mh_bk_link   = 0x0
allocator_pc = 0x0
free_pc      = 0x0
0x8 bytes of chunk data:
0xacb96c00: 0x00000000  0xf3eecdef
```

mspace & mstate

- dlmalloc

```
/* mspace_malloc behaves as malloc, but operates within the given space. */
void* mspace_malloc(mspace msp, size_t bytes);
```

- By reversing, we determined the mspace contains the dlmalloc mstate followed by a Cisco-specific mempool structure

DLMALLOC MSTATE				
Bin	**Bin size**	**fd**	**bk**	**Note**
Smallbin[00]	0x0	0xa880002c	0xa880002c	
…				
Smallbin[31]	0xf8	0xad010e70	0xa8c647f0	Free chunks
Treebin[00]	0x180	0xa9906708	-	Free chunks
…				
Treebin[31]	0xffffffff	0x0	-	
MEMPOOL MSPACE				
Bin	**Bin size**	**cnt**	**mh_fd_link**	**Note**
Mp_smallbin[00]	0x0	0x0000	0x0	
…				
Mp_smallbin[31]	0xf8	0x0049	0xa98b1780	Allocated chunks
Mp_treebin[00]	0x100	0x01ac	0xacb85b30	Allocated chunks
…				
Mp_treebin[31]	0xffffffff	0x1	0xaba41748	Allocated chunks

Tracks free chunks

Tracks allocated chunks

Checkheaps

- Mechanism introduced in Cisco IOS
- Detailed by Michael Lynn in 2005

- Checks periodically if the chunks metadata are corrupted
  - Scans memory linearly (from lower to higher addresses)
  - Encounters both allocated and freed chunks

- Implementation
  - dlmalloc compiled with DEBUG set
  - A few time-consuming checks removed

- Free chunk `fd`/`bk` pointers checked
  - Even though safe unlinking not present, since it is dlmalloc debug code
- Alloc chunk `mh_fd_link`/`mh_bk_link` pointers not checked
  - They have not modified the dlmalloc DEBUG code!

DEBUG dlmalloc

```
/*
DEBUG                    default: NOT defined
  The DEBUG setting is mainly intended for people trying to modify
  this code or diagnose problems when porting to new platforms.
  [...]
  The checking is fairly extensive, and will slow down
  execution noticeably.
  [...]
*/

#if DEBUG
...
/* Check properties of inuse chunks */
static void do_check_inuse_chunk(mstate m, mchunkptr p) {
  do_check_any_chunk(m, p);
  assert(cinuse(p));
  assert(next_pinuse(p));
  /* If not pinuse and not mmapped, previous chunk has OK offset */
  assert(is_mmapped(p) || pinuse(p) || next_chunk(prev_chunk(p)) == p);
  if (is_mmapped(p))
    do_check_mmapped_chunk(m, p);
}
```

- All the asserts were very useful to match the exact version of dlmalloc
  - Retrieve source code of checkheaps: achieved!

Checkheaps implementation

- Checkheaps thread calls `validate_buffers()` (default interval: 60 sec)
  - Takes a few ms

```
int ch_is_validating = 0;

void validate_buffers(int check_depth)
{
      if (ch_is_validating != 0)
            return;
      ch_is_validating = 1;

      // loop on all mspaces
      while (...)
      {
            //...
            // custom version of dlmalloc function
            // note this is inlined...
            custom_traverse_and_check(cur_dlmstate, check_depth);
      }

finished:
    ch_is_validating = 0;
    return;
}
```

- We can bypass checkheaps by setting `ch_is_validating` to a value != 0
  - `validate_buffers()` will exit each time it is called

Initial hypothesis

- We assume the device has been started recently
  - So the heap is not too fragmented
  - Bad hypothesis for real world but will help building a reliable exploit
- We assume Checkheaps disabled
  - We can win the race against Checkheaps (as it only runs for a few msec every 60 sec)
  - We know we can “easily” disable it (changing one global variable)
- Strategy
  - Target either dlmalloc free lists or mempool alloc lists to get a mirror write
  - Mirror write: unlinking an element from a doubly-linked list will actually trigger two write operations
    - One operation is the useful one, the other is a side effect
    - Constraint: both need to be writable addresses

Triggering a useful overflow

- Allocated chunk: up to half `mp_header->mh_len`
- Free chunk: up to half `malloc_chunk->bk`
- Note: both overflow 18 bytes (instead of 20 due to some alignment in `reass_pkt` struct)

previous mp magic footer: 0xa11ccdef

```
struct malloc_chunk @ 0xacb96a08 {
prev_foot = 0x8180d4d0
size      = 0x1d0 (CINUSE|PINUSE)
struct mp_header @ 0xacb96a10 {
mh_magic     = 0xa11c0123
mh_len       = 0x1a4
mh_refcount  = 0x0
mh_unused    = 0x0
mh_fd_link   = 0xacb85b30
mh_bk_link   = 0xa8800604
allocator_pc = 0x86816b3 (IKE_GetAssembledPkt+0x53)
free_pc      = 0x868161d (IKE_FreeAllFrags+0xfd)
0x1a8 bytes of chunk data:
0xacb96a30: 0x394d3943  0x59305239  0x747490ad  0x00163dff
0xacb96a40: 0x08021084  0x01000000  0xd4010000  0xb8010000
...
0xacb96bd0: 0x00000000  0xa11ccdef
```

Possible overflow

previous mp magic footer: 0xa11ccdef

```
struct malloc_chunk @ 0xacb96bd8 {
prev_foot = 0x8180d4d0
head      = 0x30 (PINUSE)
fd        = 0xac825ab8
bk        = 0xa880005c
struct mp_header @ 0xacb96be8 {
mh_refcount  = 0xf3ee0123
mh_unused    = 0x0
mh_fd_link   = 0x0
mh_bk_link   = 0x0
allocator_pc = 0x0
free_pc      = 0x0
0x8 bytes of chunk data:
0xacb96c00: 0x00000000  0xf3eecdef
```

Triggering a useful overflow

- Allocated chunk: up to half `mp_header->mh_len`
- Free chunk: up to half `malloc_chunk->bk`
- Note: both overflow 18 bytes (instead of 20 due to some alignment in `reass_pkt` struct)

previous mp magic footer: 0xa11ccdef

```
struct malloc_chunk @ 0xacb96a08 {
prev_foot = 0x8180d4d0
size      = 0x1d0 (CINUSE|PINUSE)
struct mp_header @ 0xacb96a10 {
mh_magic     = 0xa11c0123
mh_len       = 0x1a4
mh_refcount  = 0x0
mh_unused    = 0x0
mh_fd_link   = 0xacb85b30
mh_bk_link   = 0xa8800604
allocator_pc = 0x86816b3 (IKE_GetAssembledPkt+0x53)
free_pc      = 0x868161d (IKE_FreeAllFrags+0xfd)
0x1a8 bytes of chunk data:
0xacb96a30: 0x394d3943  0x59305239  0x747490ad  0x00163dff
0xacb96a40: 0x08021084  0x01000000  0xd4010000  0xb8010000
...
0xacb96bd0: 0x00000000  0xa11ccdef
```

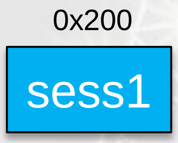
Chosen overflow

previous mp magic footer: 0xa11ccdef

```
struct malloc_chunk @ 0xacb96bd8 {
prev_foot = 0x8180d4d0
head      = 0x30 (PINUSE)
fd        = 0xac825ab8
bk        = 0xa880005c
struct mp_header @ 0xacb96be8 {
mh_refcount  = 0xf3ee0123
mh_unused    = 0x0
mh_fd_link   = 0x0
mh_bk_link   = 0x0
allocator_pc = 0x0
free_pc      = 0x0
0x8 bytes of chunk data:
0xacb96c00: 0x00000000  0xf3eecdef
```

Heap feng shui

Heap feng shui 1

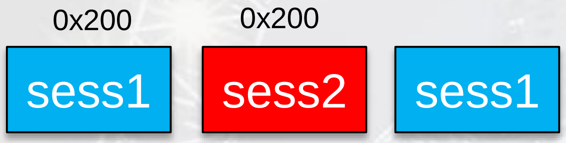


- 2 IKEv1 sessions of 0x200 fragments
  - Send one fragment in sess1, sess2, sess1, sess2, etc.
  - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
  - 0x30 size was changed into 0x90
  - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R’)
  - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
  - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
  - They are all fragments
  - IKEv2 XI exploit used Option Lists but they don’t exist in IKEv1

Heap feng shui 1

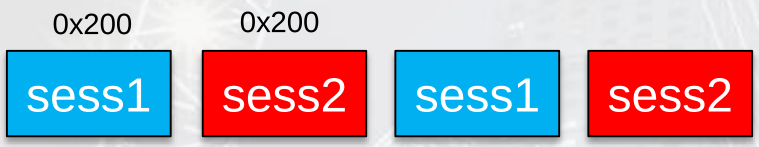
- 2 IKEv1 sessions of 0x200 fragments
  - Send one fragment in sess1, sess2, sess1, sess2, etc.
  - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
  - 0x30 size was changed into 0x90
  - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R')
  - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
  - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
  - They are all fragments
  - IKEv2 XI exploit used Option Lists but they don't exist in IKEv1

Heap feng shui 1



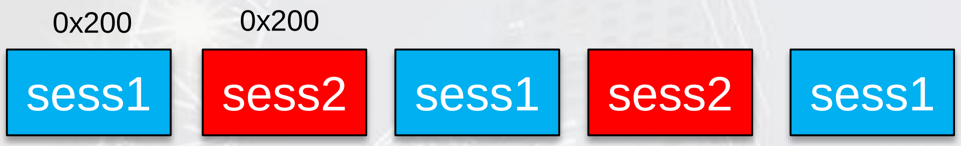
- 2 IKEv1 sessions of 0x200 fragments
    - Send one fragment in sess1, sess2, sess1, sess2, etc.
    - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
    - 0x30 size was changed into 0x90
    - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R')
    - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
    - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
    - They are all fragments
    - IKEv2 XI exploit used Option Lists but they don't exist in IKEv1

Heap feng shui 1



- 2 IKEv1 sessions of 0x200 fragments
  - Send one fragment in sess1, sess2, sess1, sess2, etc.
  - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
  - 0x30 size was changed into 0x90
  - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R’)
  - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
  - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
  - They are all fragments
  - IKEv2 XI exploit used Option Lists but they don’t exist in IKEv1

Heap feng shui 1

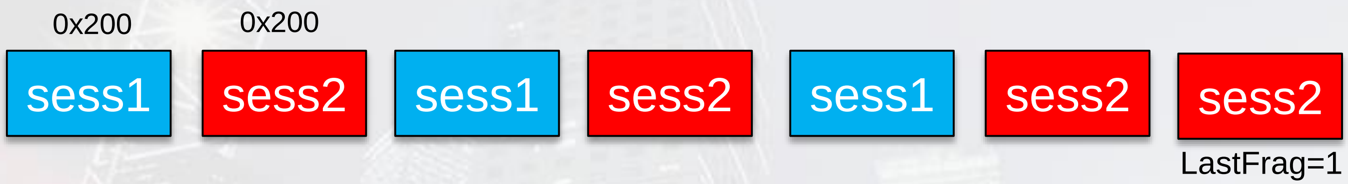


- 2 IKEv1 sessions of 0x200 fragments
  - Send one fragment in sess1, sess2, sess1, sess2, etc.
  - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
  - 0x30 size was changed into 0x90
  - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R')
  - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
  - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
  - They are all fragments
  - IKEv2 XI exploit used Option Lists but they don't exist in IKEv1

Heap feng shui 1

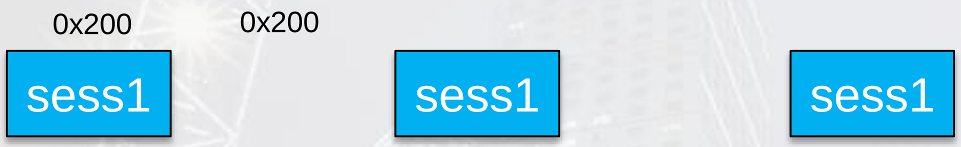
- 2 IKEv1 sessions of 0x200 fragments
  - Send one fragment in sess1, sess2, sess1, sess2, etc.
  - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
  - 0x30 size was changed into 0x90
  - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R')
  - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
  - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
  - They are all fragments
  - IKEv2 XI exploit used Option Lists but they don't exist in IKEv1

Heap feng shui 1



- 2 IKEv1 sessions of 0x200 fragments
  - Send one fragment in sess1, sess2, sess1, sess2, etc.
  - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
  - 0x30 size was changed into 0x90
  - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R')
  - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
  - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
  - They are all fragments
  - IKEv2 XI exploit used Option Lists but they don't exist in IKEv1

Heap feng shui 1



- 2 IKEv1 sessions of 0x200 fragments
  - Send one fragment in sess1, sess2, sess1, sess2, etc.
  - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
  - 0x30 size was changed into 0x90
  - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R')
  - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
  - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
  - They are all fragments
  - IKEv2 XI exploit used Option Lists but they don't exist in IKEv1

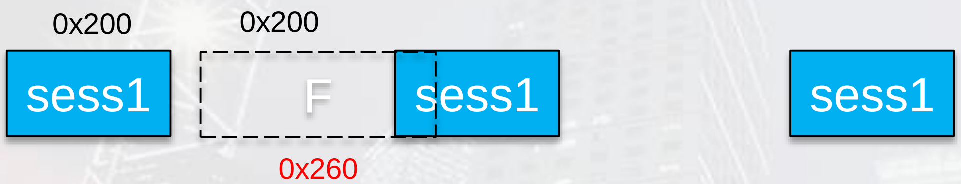
Heap feng shui 1

- 2 IKEv1 sessions of 0x200 fragments
  - Send one fragment in sess1, sess2, sess1, sess2, etc.
  - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
  - 0x30 size was changed into 0x90
  - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R')
  - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
  - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
  - They are all fragments
  - IKEv2 XI exploit used Option Lists but they don't exist in IKEv1

Heap feng shui 1

- 2 IKEv1 sessions of 0x200 fragments
  - Send one fragment in sess1, sess2, sess1, sess2, etc.
  - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
  - 0x30 size was changed into 0x90
  - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R')
  - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
  - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
  - They are all fragments
  - IKEv2 XI exploit used Option Lists but they don't exist in IKEv1

Heap feng shui 1

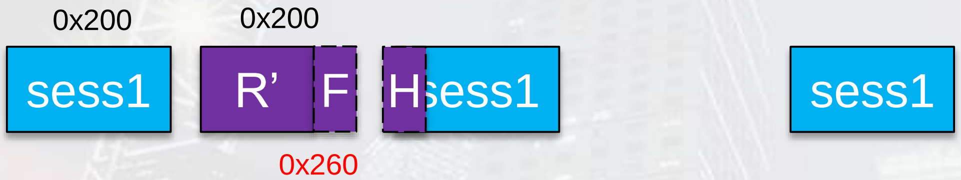


- 2 IKEv1 sessions of 0x200 fragments
  - Send one fragment in sess1, sess2, sess1, sess2, etc.
  - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
  - 0x30 size was changed into 0x90
  - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R')
  - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
  - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
  - They are all fragments
  - IKEv2 XI exploit used Option Lists but they don't exist in IKEv1

Heap feng shui 1

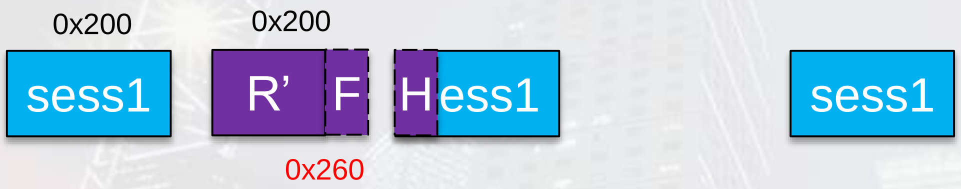
- 2 IKEv1 sessions of 0x200 fragments
  - Send one fragment in sess1, sess2, sess1, sess2, etc.
  - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
  - 0x30 size was changed into 0x90
  - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R')
  - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
  - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
  - They are all fragments
  - IKEv2 XI exploit used Option Lists but they don't exist in IKEv1

Heap feng shui 1



- 2 IKEv1 sessions of 0x200 fragments
    - Send one fragment in sess1, sess2, sess1, sess2, etc.
    - Trigger reassembly of sess2 to free sess2 fragments
- Trigger reassembly in a 0x1d0 chunk (R) to overflow metadata of a 0x30 free chunk
- When 0x1d0 is free (invalid reassembly), it is coalesced with the adjacent chunk
    - 0x30 size was changed into 0x90
    - 0x1d0 + 0x90 = 0x260 free chunk added to the bin list
- Fill the 0x260 encompassing chunk (R')
    - Corrupt the following 0x200 chunk `mp_fd_link`/`mp_bk_link` (H)
    - Craft a fake 0x30 free chunk with corrupted `fd`/`bk` (F)
- Notes
    - They are all fragments
    - IKEv2 XI exploit used Option Lists but they don't exist in IKEv1

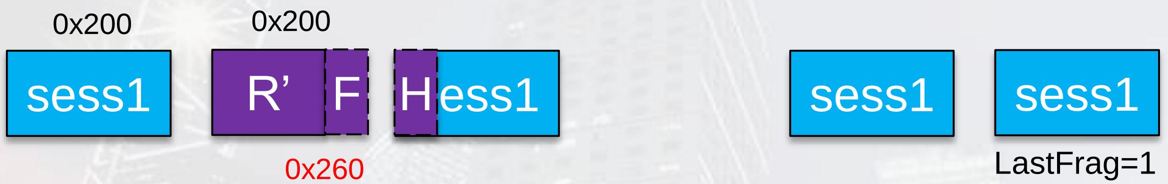
Heap feng shui 1 – WTF!?



- Trigger reassembly of sess1 to free sess1 fragments
- We are expecting to crash when the alloc list pointers are accessed
  - `mh_mem_free()` first unlinks the chunk from the mempool alloc list
  - Then it calls `mspace_free()` responsible to coalesce with adjacent free chunks
- Instead it crashes when the freelist pointers are accessed
- Problem is even though our corrupted sess1 is freed
  - Fragment queue is LIFO so they are free in reverse order
  - Also they are in the same alloc list bin in `mp_mspace`
  - So our corrupted sess1 chunk gets its pointers overwritten/restored before they are used!

➔ We need a different heap feng shui where we can arbitrary free one chunk

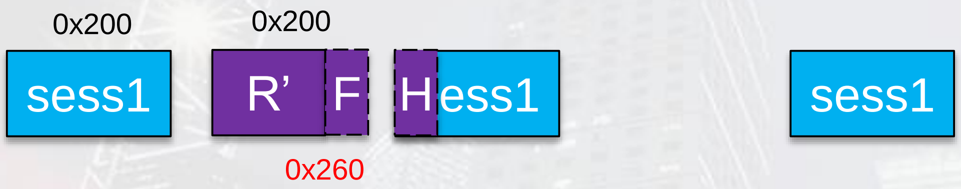
Heap feng shui 1 – WTF!?



- Trigger reassembly of sess1 to free sess1 fragments
- We are expecting to crash when the alloc list pointers are accessed
  - `mh_mem_free()` first unlinks the chunk from the mempool alloc list
  - Then it calls `mspace_free()` responsible to coalesce with adjacent free chunks
- Instead it crashes when the freelist pointers are accessed
- Problem is even though our corrupted sess1 is freed
  - Fragment queue is LIFO so they are free in reverse order
  - Also they are in the same alloc list bin in `mp_mspace`
  - So our corrupted sess1 chunk gets its pointers overwritten/restored before they are used!

➔ We need a different heap feng shui where we can arbitrary free one chunk

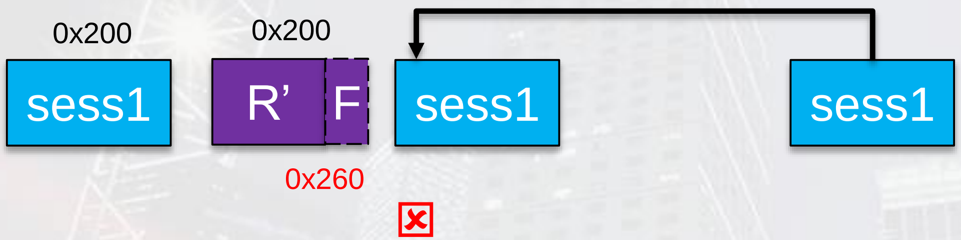
Heap feng shui 1 – WTF!?



- Trigger reassembly of sess1 to free sess1 fragments
- We are expecting to crash when the alloc list pointers are accessed
  - `mh_mem_free()` first unlinks the chunk from the mempool alloc list
  - Then it calls `mspace_free()` responsible to coalesce with adjacent free chunks
- Instead it crashes when the freelist pointers are accessed
- Problem is even though our corrupted sess1 is freed
  - Fragment queue is LIFO so they are free in reverse order
  - Also they are in the same alloc list bin in `mp_mspace`
  - So our corrupted sess1 chunk gets its pointers overwritten/restored before they are used!

➔ We need a different heap feng shui where we can arbitrary free one chunk

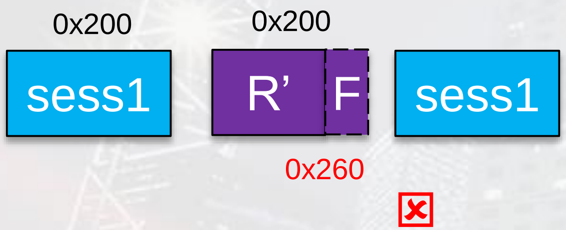
Heap feng shui 1 – WTF!?



- Trigger reassembly of sess1 to free sess1 fragments
- We are expecting to crash when the alloc list pointers are accessed
  - `mh_mem_free()` first unlinks the chunk from the mempool alloc list
  - Then it calls `mspace_free()` responsible to coalesce with adjacent free chunks
- Instead it crashes when the freelist pointers are accessed
- Problem is even though our corrupted sess1 is freed
  - Fragment queue is LIFO so they are free in reverse order
  - Also they are in the same alloc list bin in `mp_mspace`
  - So our corrupted sess1 chunk gets its pointers overwritten/restored before they are used!

➔ We need a different heap feng shui where we can arbitrary free one chunk

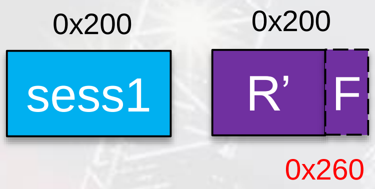
Heap feng shui 1 – WTF!?



- Trigger reassembly of sess1 to free sess1 fragments
- We are expecting to crash when the alloc list pointers are accessed
  - `mh_mem_free()` first unlinks the chunk from the mempool alloc list
  - Then it calls `mspace_free()` responsible to coalesce with adjacent free chunks
- Instead it crashes when the freelist pointers are accessed
- Problem is even though our corrupted sess1 is freed
  - Fragment queue is LIFO so they are free in reverse order
  - Also they are in the same alloc list bin in `mp_mspace`
  - So our corrupted sess1 chunk gets its pointers overwritten/restored before they are used!

➔ We need a different heap feng shui where we can arbitrary free one chunk

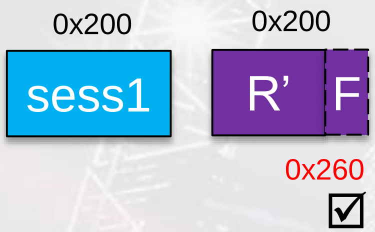
Heap feng shui 1 – WTF!?



- Trigger reassembly of sess1 to free sess1 fragments
- We are expecting to crash when the alloc list pointers are accessed
  - `mh_mem_free()` first unlinks the chunk from the mempool alloc list
  - Then it calls `mspace_free()` responsible to coalesce with adjacent free chunks
- Instead it crashes when the freelist pointers are accessed
- Problem is even though our corrupted sess1 is freed
  - Fragment queue is LIFO so they are free in reverse order
  - Also they are in the same alloc list bin in `mp_mspace`
  - So our corrupted sess1 chunk gets its pointers overwritten/restored before they are used!

➔ We need a different heap feng shui where we can arbitrary free one chunk

Heap feng shui 1 – WTF!?



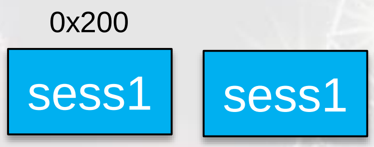
- Trigger reassembly of sess1 to free sess1 fragments
- We are expecting to crash when the alloc list pointers are accessed
  - `mh_mem_free()` first unlinks the chunk from the mempool alloc list
  - Then it calls `mspace_free()` responsible to coalesce with adjacent free chunks
- Instead it crashes when the freelist pointers are accessed
- Problem is even though our corrupted sess1 is freed
  - Fragment queue is LIFO so they are free in reverse order
  - Also they are in the same alloc list bin in `mp_mspace`
  - So our corrupted sess1 chunk gets its pointers overwritten/restored before they are used!

➔ We need a different heap feng shui where we can arbitrary free one chunk

Heap feng shui 2

- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes
- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

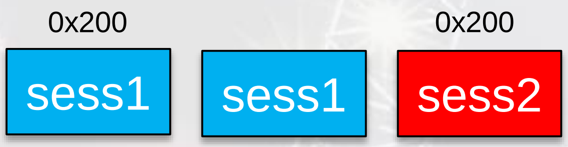
Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes

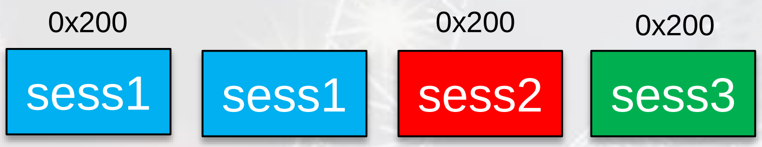
- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes
- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

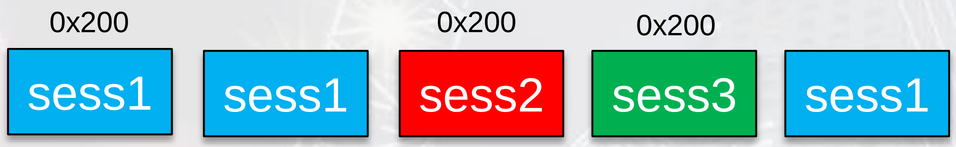
Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
    - Fill holes with sess1
    - Send one fragment only in sess2, sess3
    - Send additional fragments in sess1 (avoid coalescing)
    - Trigger reassembly of sess2 to free sess2 fragment
    - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
        - Success: mempool alloc list mirror write triggered!
        - It gives us 2 mirror writes

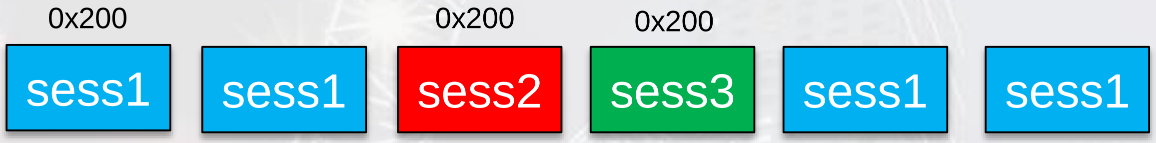
- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes
- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

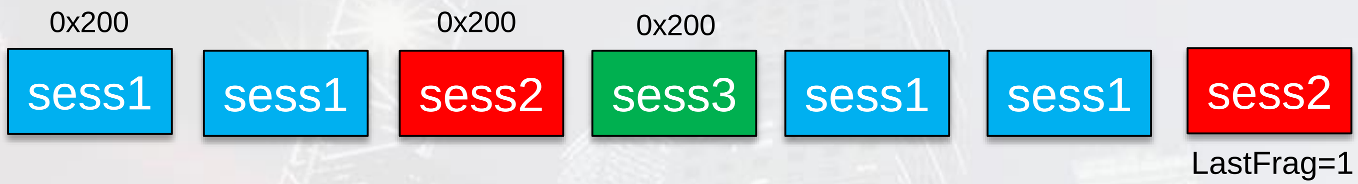
Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes

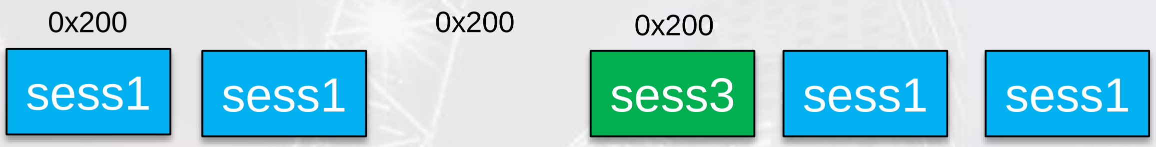
- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes
- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

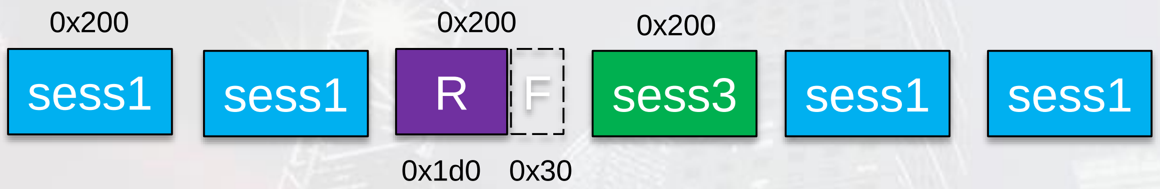
Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes

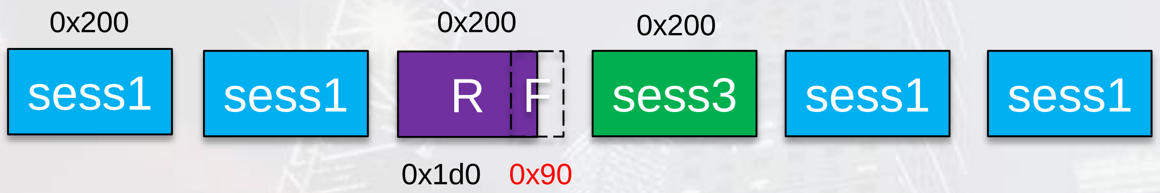
- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

Heap feng shui 2



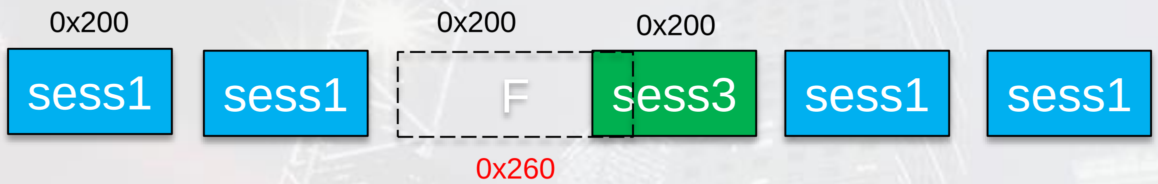
- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
    - Fill holes with sess1
    - Send one fragment only in sess2, sess3
    - Send additional fragments in sess1 (avoid coalescing)
    - Trigger reassembly of sess2 to free sess2 fragment
    - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
        - Success: mempool alloc list mirror write triggered!
        - It gives us 2 mirror writes
- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes
- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

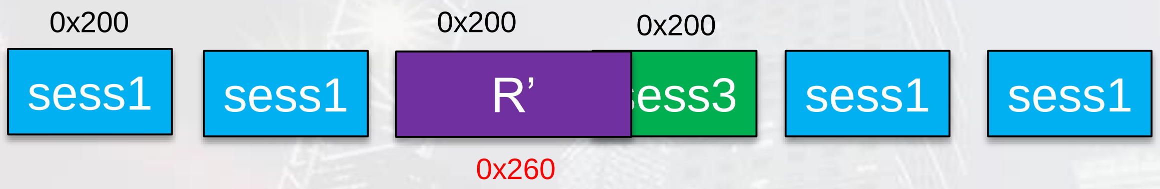
Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes

- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

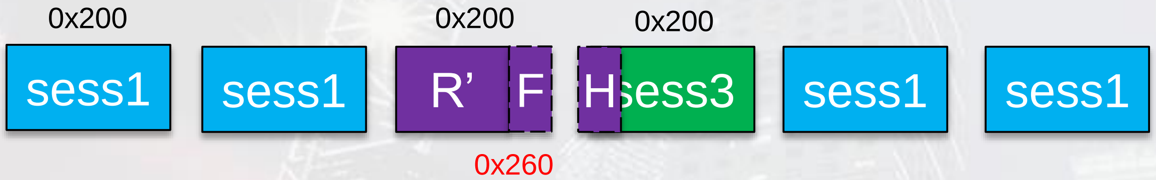
Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes

- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

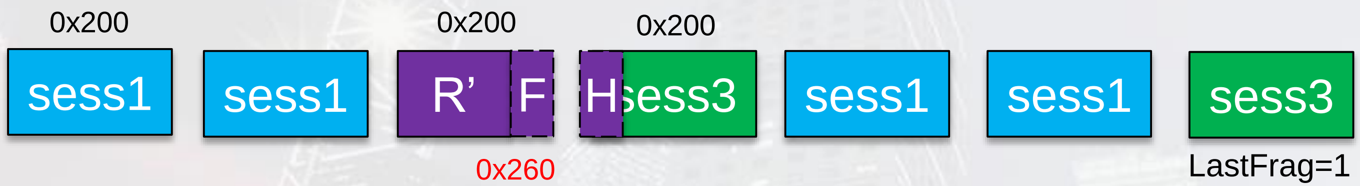
Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes

- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

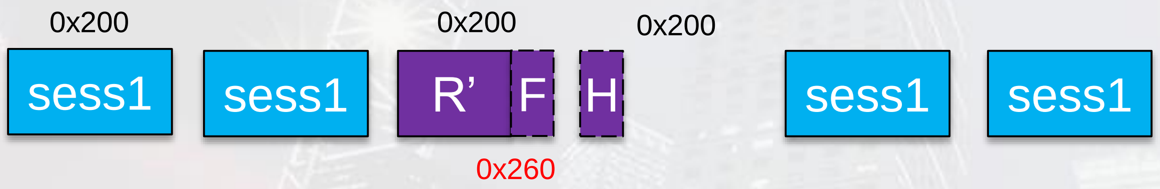
Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes

- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

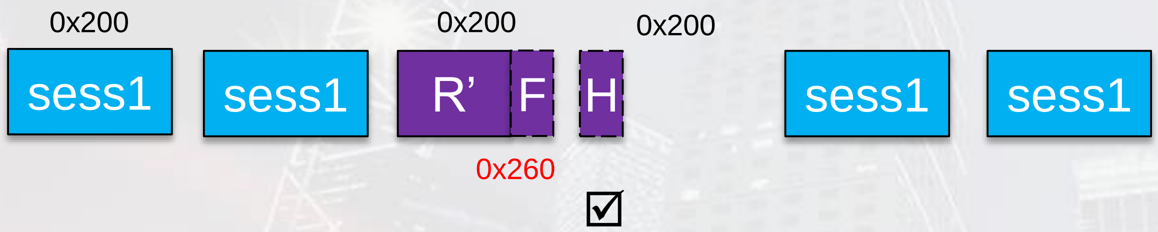
Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes

- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

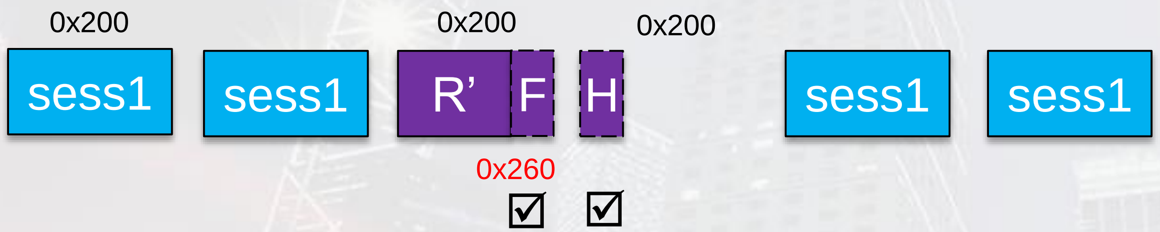
Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes

- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

Heap feng shui 2



- Create one hole before one fragment from a session with only this fragment (sess3)
- 3 IKEv1 sessions of 0x200 fragments
  - Fill holes with sess1
  - Send one fragment only in sess2, sess3
  - Send additional fragments in sess1 (avoid coalescing)
  - Trigger reassembly of sess2 to free sess2 fragment
  - After memory corruption, trigger reassembly of sess3 to free sess3 fragment
    - Success: mempool alloc list mirror write triggered!
    - It gives us 2 mirror writes
- Generalize this method using as many sessions as required (with one fragment each) if we need more mirror writes

From mirror write to RCE

Follow the white rabbit

- On IKEv2, XI targeted `list_add()` called to add a fragment to the queue
  - A global pointer is stored in memory
  - Used when a fragment is received and we control its content so contains our shellcode
  - Not possible on IKEv1 as it does not use the same list format
- I looked for a function pointer to overwrite in IDA…
  - IKEv1-related functions
- Best candidate I found is `IKEMM_BuildMainModeMsg2()`
  - EDX is a pointer to a pointer to our IKE packet. Our shellcode is at @packet_ike+0x6a
  - Can be triggered by sending an SA INIT (first IKE packet)

```
(gdb) i r edx
edx            0xacaa8334       -1398111436
(gdb) x /wx 0xacaa8334
0xacaa8334:     0xadc17670
(gdb) x /150bx 0xadc17670
0xadc17670:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00    //packet_ike
...
0xadc176d0:     0x00    0x04    0x00    0x00    0x70    0x80    0x00    0x00
0xadc176d8:     0x0f    0xb0    0x90    0x90    0x90    0x90    0x90    0x90    //shellcode
0xadc176e0:     0x90    0x90    0x90    0x90    0xcc    0xcc    0xcc    0xcc
0xadc176e8:     0xcc    0xcc    0xcc    0xcc    0xcc    0xcc    0xcc    0xcc
0xadc176f0:     0xcc    0xcc    0xcc    0xcc    0xcc    0xcc    0xcc    0xcc
0xadc176f8:     0xcc    0xcc    0xcc    0xcc    0xcc    0xcc    0xcc    0xcc
0xadc17700:     0xcc    0xcc    0xcc    0xcc    0xcc    0xcc
```

Calling IKEMM_BuildMainModeMsg2

- `FSM_SMDriver()`
  - Get global `IKEmmStateTable`
  - `IKEMM_BuildMainModeMsg2_ptr= IKEmmStateTable[sizeof(void*)*0x32c]`
  - `IKEMM_BuildMainModeMsg2 = *IKEMM_BuildMainModeMsg2_ptr`
  - Call `IKEMM_BuildMainModeMsg2`

- Memory layout

```
.data:0A46B680 IKEmmStateTable dd offset off_9E7F000
.data:0A46B684                 dd offset off_9E7F020
...
.data:0A46C330                 dd offset IKEMM_BuildMainModeMsg2_ptr


.rodata:09E7F240 IKEMM_BuildMainModeMsg2_ptr dd offset IKEMM_BuildMainModeMsg2
```

- Easiest is to overwrite `IKEMM_BuildMainModeMsg2_ptr` in `IKEmmStateTable`

Execute my *real* packet ©

- XI actually executed an IKE Fragment payload, we execute our whole IKE packet ☺
  - 2 mirror writes to overwrite function pointer
  - As many mirror writes as required for the trampoline

- Part of memory RWX: we choose 0xc2000000-0xc2ffffff

```
a6000000-a8724000 rwxs 00000000 00:0e 1740       /dev/udma0
a8800000-ab400000 rwxs 00000000 00:0b 0          /SYSV00000002 (deleted)
ab800000-abc00000 rwxs 03000000 00:0b 0          /SYSV00000002 (deleted)
ac400000-dbc00000 rwxs 03c00000 00:0b 0          /SYSV00000002 (deleted)
```

- Trampoline

```
// edx is a pointer to our packet
8b 12          mov    edx,DWORD PTR [edx]     // access our packet
83 c2 6a       add    edx,0x6a                // point to our shellcode within packet
ff e2          jmp    edx                     // jump to it
c2             .byte 0xc2
```

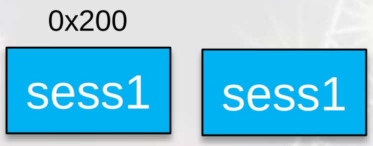
- 4 mirror writes

```
*0x0a46c330 = 0xc2831200 (IKEMM_BuildMainModeMsg2_ptr)
*0xc2831200 = 0xc2831204 (fake IKEMM_BuildMainModeMsg2)
*0xc2831204 = 0xc283128b (trampoline)
*0xc2831208 = 0xc2e2ff6a (trampoline 2)
```

Summary

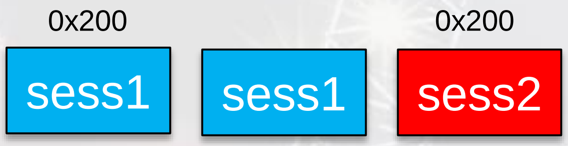
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



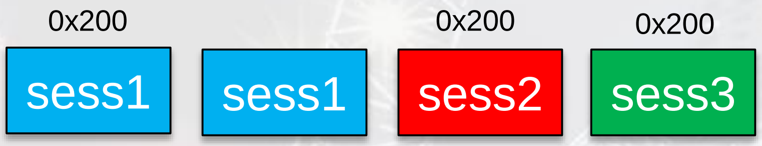
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



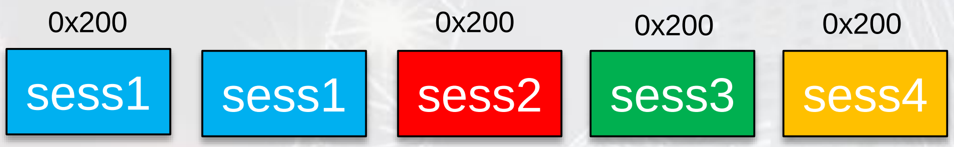
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



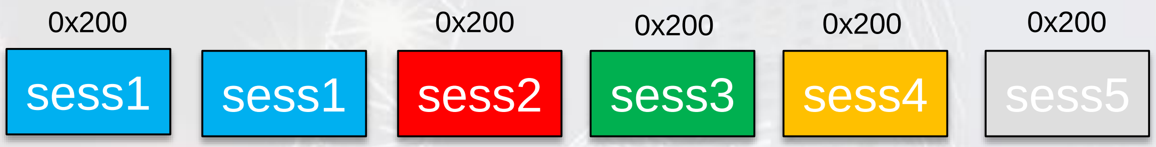
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



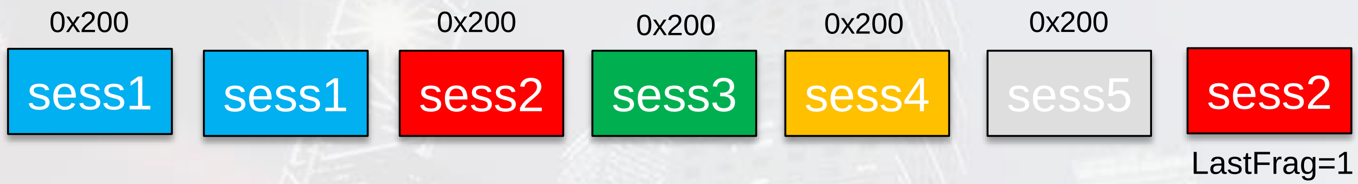
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



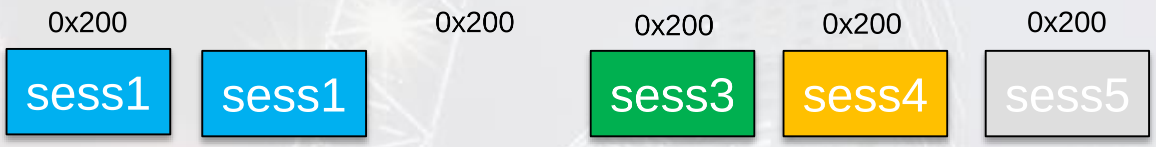
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



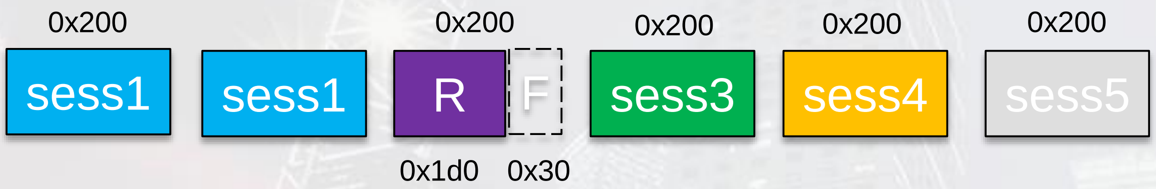
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary

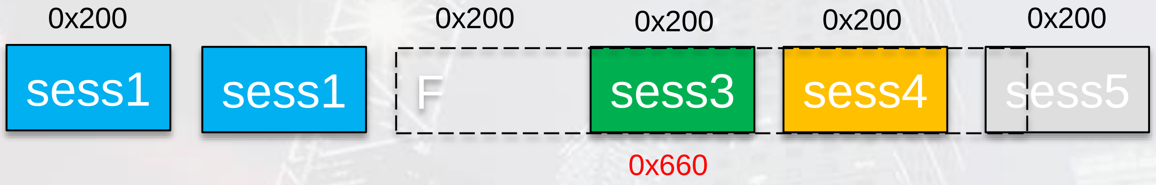


- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary

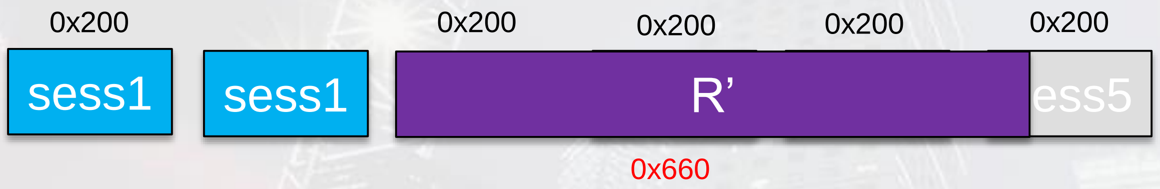
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



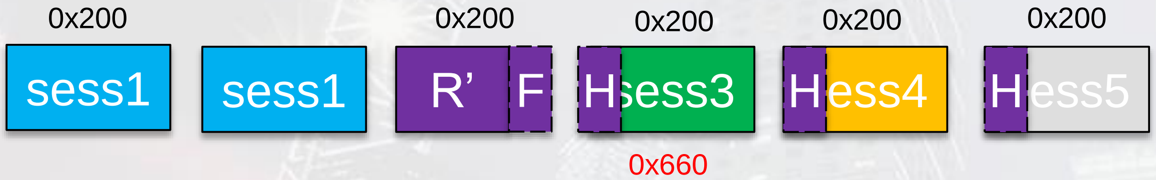
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



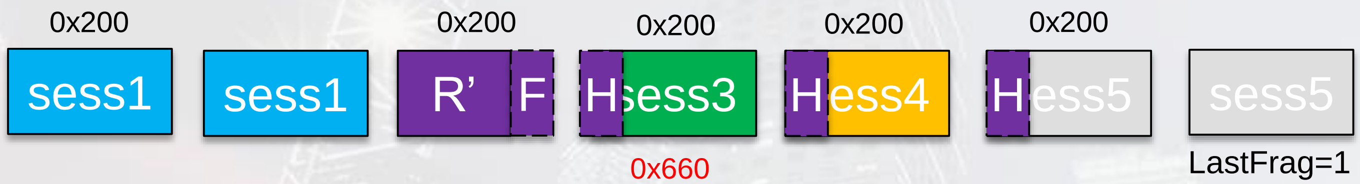
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



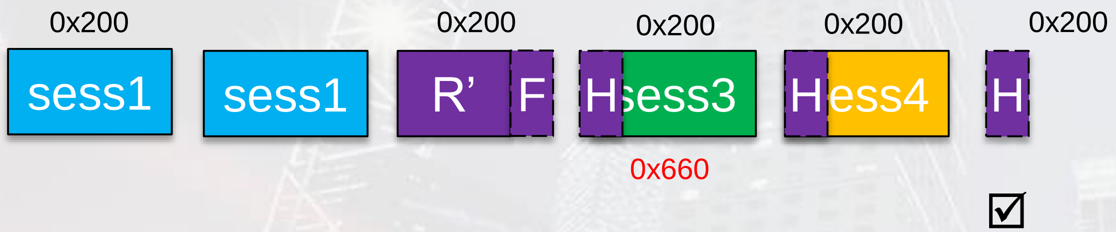
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



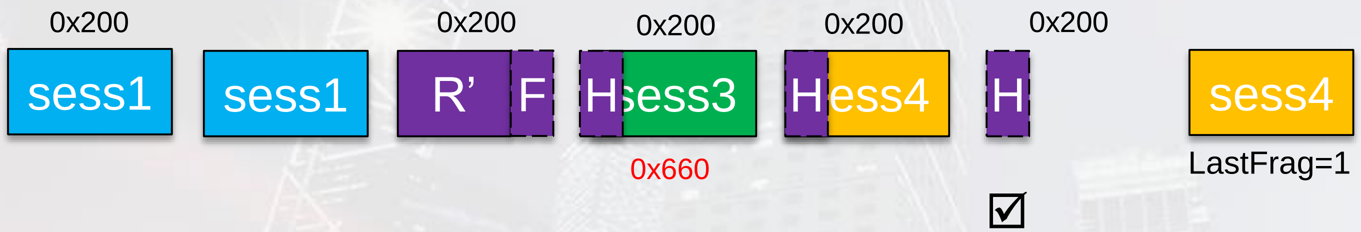
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



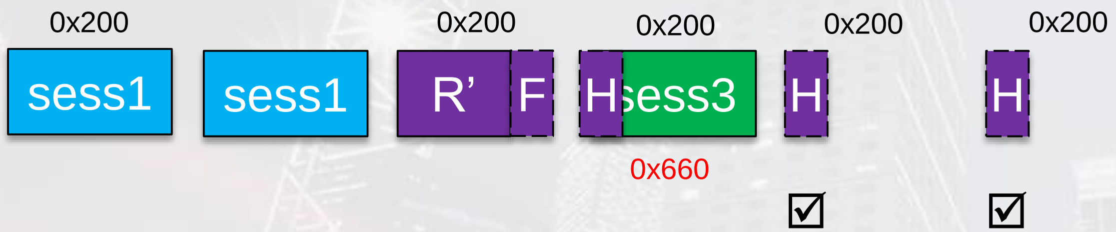
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



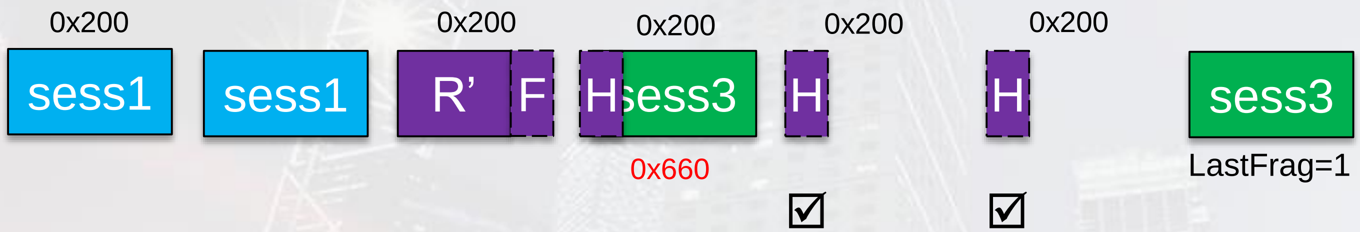
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



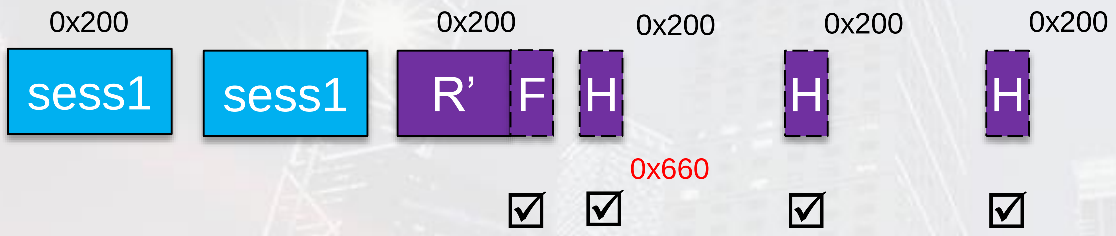
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



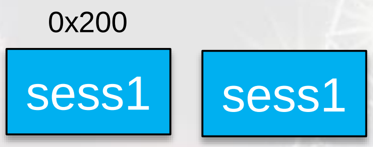
- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Summary



- When packet is reassembled, we have: an allocated 0x1d0 before a free 0x30
- After initial memory corruption, we have an allocated 0x1d0 before a corrupted free 0x490
- Reassembled packet is free, we have a free 0x660
- Reallocate the 0x660 chunk to corrupt alloc lists for the 3 adjacent 0x200 chunks and craft a fake 0x30 free chunk
- Free one fragment at a time to trigger the different mirror writes

Fixing the heap



- After triggering the 4 mirror writes, there are two possible views
  - From R’, we see an allocated 0x660 chunk
  - But at offset R’+0x1d0 there is a free 0x630 free chunk
- Easiest is to patch R’ size (`malloc_chunk`/`mp_header`) to be a 0x1d0 chunk
  - Retrieve the dlmalloc mstate address from a global pointer (mempool_array)
  - Access the mempool mspace
  - Look for the right alloc list bin (sz: 0x800) and fix the corrupted chunk

- We clear all sessions from Cisco shell to free all our packets and check our ASA is still alive ☺

```
asa(config)# clear crypto ikev1 sa
```

- Now we can deliver a Cisco CLI to the attacker ☺

Fixing the heap

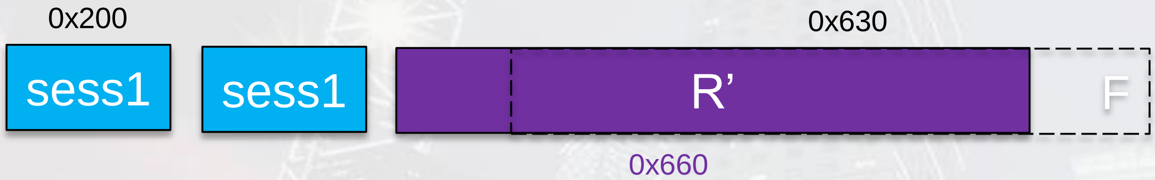
- After triggering the 4 mirror writes, there are two possible views
  - From R’, we see an allocated 0x660 chunk
  - But at offset R’+0x1d0 there is a free 0x630 free chunk
- Easiest is to patch R’ size (`malloc_chunk`/`mp_header`) to be a 0x1d0 chunk
  - Retrieve the dlmalloc mstate address from a global pointer (mempool_array)
  - Access the mempool mspace
  - Look for the right alloc list bin (sz: 0x800) and fix the corrupted chunk

- We clear all sessions from Cisco shell to free all our packets and check our ASA is still alive ☺

```
asa(config)# clear crypto ikev1 sa
```

- Now we can deliver a Cisco CLI to the attacker ☺

Fixing the heap



- After triggering the 4 mirror writes, there are two possible views
  - From R’, we see an allocated 0x660 chunk
  - But at offset R’+0x1d0 there is a free 0x630 free chunk
- Easiest is to patch R’ size (`malloc_chunk`/`mp_header`) to be a 0x1d0 chunk
  - Retrieve the dlmalloc mstate address from a global pointer (mempool_array)
  - Access the mempool mspace
  - Look for the right alloc list bin (sz: 0x800) and fix the corrupted chunk
- We clear all sessions from Cisco shell to free all our packets and check our ASA is still alive ☺

```
asa(config)# clear crypto ikev1 sa
```

- Now we can deliver a Cisco CLI to the attacker ☺

Fixing the heap

- After triggering the 4 mirror writes, there are two possible views
  - From R', we see an allocated 0x660 chunk
  - But at offset R'+0x1d0 there is a free 0x630 free chunk
- Easiest is to patch R' size (`malloc_chunk`/`mp_header`) to be a 0x1d0 chunk
  - Retrieve the dlmalloc mstate address from a global pointer (mempool_array)
  - Access the mempool mspace
  - Look for the right alloc list bin (sz: 0x800) and fix the corrupted chunk
- We clear all sessions from Cisco shell to free all our packets and check our ASA is still alive ☺

```
asa(config)# clear crypto ikev1 sa
```

- Now we can deliver a Cisco CLI to the attacker ☺

Restore execution

- Restore overwritten pointer to function pointer

```
IKEmmStateTable[index] = IKEMM_BuildMainModeMsg2_ptr
```

- Jump to original function: `IKEMM_BuildMainModeMsg2()`
- After doing that, we realized it crashed after `IKEMM_BuildMainModeMsg2()` returns
  - Because `IKEMM_BuildMainModeMsg2_ptr` was also saved at `ebp-0x24`
  - And it was reused
- So we fix it as well before calling original function

```
*(ebp-0x24) = IKEMM_BuildMainModeMsg2_ptr
```

Checkheaps bypass?

- We already bypass Checkheaps
  - Even though there is some misalignment issue, we are fine as long as our last 0x200 chunk is allocated (as it contains a fake header to keep things aligned)
  - Because Checkheaps checks chunks linearly
  - But then there is a race between checkheaps and our shellcode that will fix that
  - There is still a risk Checkheaps detects us if it is already running and is analysing the chunk we are actively corrupting

Demo

Mitigations

```
$ ./info.py  -l
Using dbname targets.json
 -------------------------------------------------------------------------------------------------------
| ID  | Version    |Arch|ASLR| NX |PIE|Can|RELRO|Sym|Strip|    Linux | Glibc | Firmware                  |
 -------------------------------------------------------------------------------------------------------
| 000 |      8.0.2 | 32 |  N |  N | N | N |   N | N |  N  | 2.6.17.8 |     ? |              asa802-k8.bin |
| 001 |      8.0.3 | 32 |  N |  N | N | N |   N | N |  N  | 2.6.17.8 |     ? |              asa803-k8.bin |
...
| 018 |      8.2.3 | 32 |  N |  N | N | N |   N | N |  N  | 2.6.29.6 | 2.3.2 |              asa823-k8.bin |
| 019 |      8.2.3 | 32 |  N |  N | N | N |   N | N |  N  | 2.6.29.6 | 2.3.2 |          asa823-smp-k8.bin |
...
| 048 |      8.4.1 | 32 |  N |  N | N | N |   N | N |  N  | 2.6.29.6 |   2.9 |              asa841-k8.bin |
| 049 |      8.4.1 | 64 |  N |  N | N | N |   N | N |  N  | 2.6.29.6 |   2.9 |          asa841-smp-k8.bin |
...
| 105 |      9.1.6 | 32 |  N |  N | N | N |   N | N |  N  | 2.6.29.6 |   2.9 |              asa916-k8.bin |
| 106 |      9.1.6 | 64 |  N |  N | N | N |   N | N |  N  | 2.6.29.6 |   2.9 |          asa916-smp-k8.bin |
...
| 123 |      9.2.4 | 32 |  N |  N | N | N |   N | N |  N  | 2.6.29.6 |   2.9 |              asa924-k8.bin |
| 124 |      9.2.4 | 64 |  N |  N | N | N |   N | N |  N  | 2.6.29.6 |   2.9 |          asa924-smp-k8.bin |
...
| 135 |  9.3.2.200 | 64 |  N |  N | N | N |   N | N |  N  |  3.10.19 |  2.18 |      asa932-200-smp-k8.bin |
| 136 |  9.3.2.200 | 64 |  N |  N | N | N |   N | N |  N  |  3.10.19 |  2.18 |asav932-200-from-qcow2.bin |
...
| 155 |      9.4.3 | 64 |  N |  Y | N | N |   N | N |  N  |  3.10.55 |  2.18 |          asa943-smp-k8.bin |
| 157 |      9.4.4 | 64 |  N |  Y | N | N |   N | N |  N  |  3.10.55 |  2.18 |          asa944-smp-k8.bin |
...
| 158 |      9.5.1 | 64 |  Y |  N | Y | N |   N | N |  N  |  3.10.62 |  2.18 |          asa951-smp-k8.bin |
| 159 |      9.5.2 | 64 |  Y |  N | Y | N |   N | Y |  N  |  3.10.62 |  2.18 |          asa952-smp-k8.bin |
...
| 170 |      9.7.1 | 64 |  Y |  Y | Y | N |   N | Y |  N  |  3.10.62 |  2.18 |          asa971-smp-k8.bin |
| 171 |      9.7.1 | 64 |  Y |  Y | Y | N |   N | Y |  N  |  3.10.62 |  2.18 |    asav971-from-qcow2.bin |
 -------------------------------------------------------------------------------------------------------
```

Conclusion

Conclusion

- First IKEv1 exploit
  - Targets 32-bit and 64-bit (dlmalloc)

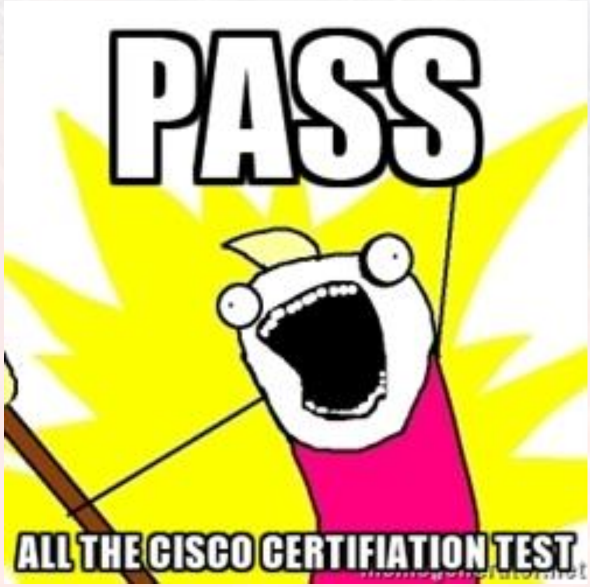
- Versions vulnerable to the IKE heap overflow

Version	Heap	Heap safe unlinking	Mempool safe unlinking	ASLR	NX
< 9.0.4.38	dlmalloc	No	No	No	No
< 9.1.6.11	dlmalloc	No	No	No	No
< 9.2.4.5	dlmalloc	No	No	No	No
< 9.3.3.7	ptmalloc	Yes	No	No	No
< 9.4.2.4	ptmalloc	Yes	No	No	No
< 9.5.2.2	ptmalloc	Yes	No	Yes	No

- Next steps
  - 64-bit ptmalloc
  - ASLR / Safe-unlinking for free lists (not for mempool alloc list!)

Questions?

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



References

- David Barksdale, Jordan Gruskovnjak, Alex Wheeler (Exodus Intel) – Execute my packet
- Alec Stuart-Muirk - Breaking bricks and Plumbing pipes – Cisco ASA: A super Mario adventure
- Michael Lynn – The Holy Grail: Cisco IOS Shellcode and Exploitation techniques