Brute Ratel Config Decoding update

By Jason Reaves

Published: 2022-10-25 · Archived: 2026-04-05 19:23:28 UTC

By: Jason Reaves

There have been a few reports on how to decrypt Brute Ratels[1] configuration data along with a few decryptors created[2,3]. However, the developer added in the release notes that they changed it to be a dynamic key instead of the hardcoded key everyone refers to. The hardcoded key is still used and exists for decrypting some of the strings on board.

Press enter or click to view image in full size

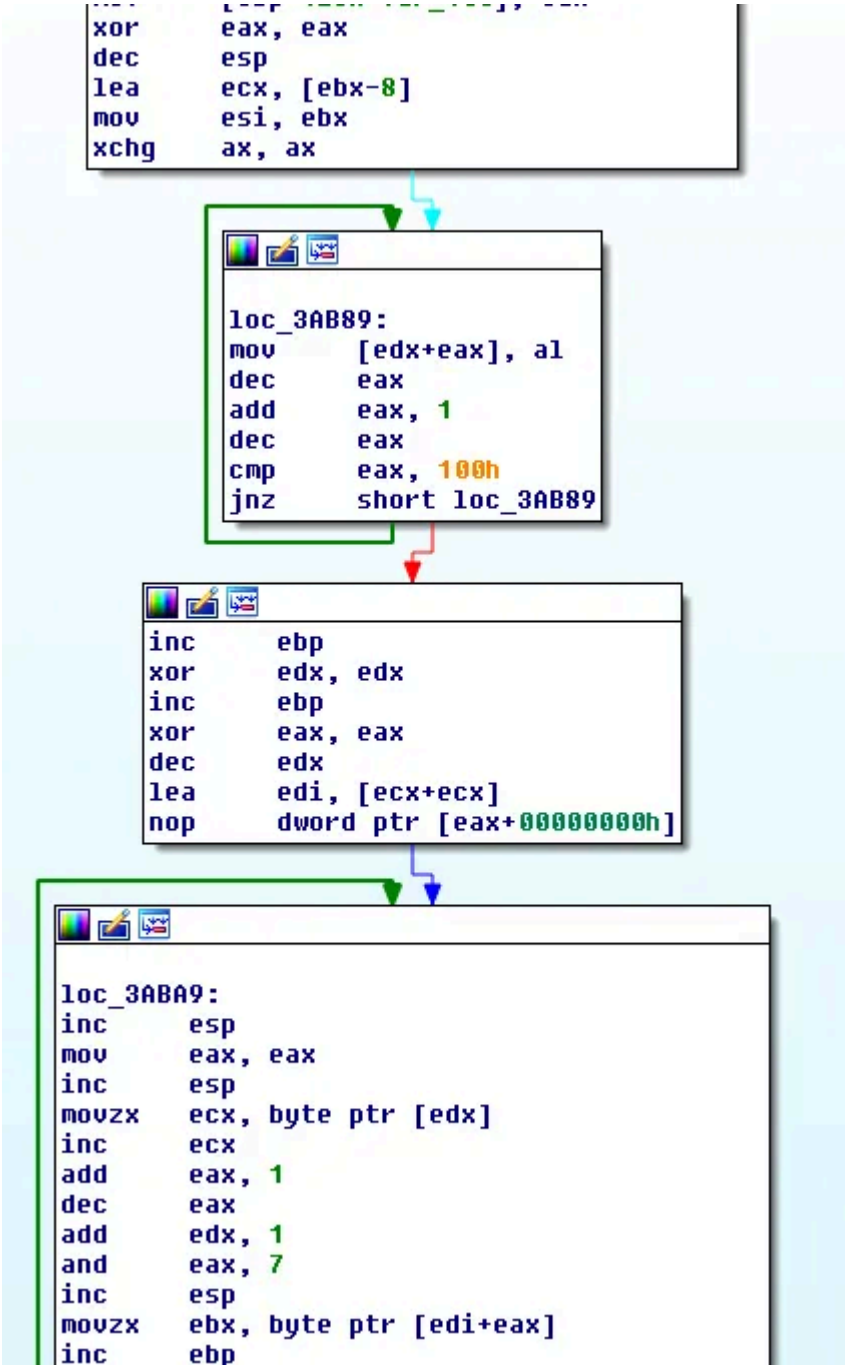
Ref: https://bruteratel.com/release_notes/releases.txt

We start with a sample from a TrendMicro report on BlackBasta actors leveraging QBot to deliver Brute Ratel and CobaltStrike:

```
62cb24967c6ce18d35d2a23ebed4217889d796cf7799d9075c1aa7752b8d3967
```

The shellcode-based loader is stored onboard and is loaded into memory. The shellcode stager uses a few Anti Debugging checks such as checking the NtGlobalFlag.

The encoded onboard DLL is still stored RC4 encrypted as mentioned in the MDSec blog[3] the key is the last 8 bytes:



RC4

Manually decoding:

```
>>> data[-8:]
'*%@{.de|'
>>> rc4 = ARC4.new(data[-8:])
>>> t = rc4.decrypt(data)
>>> t[:1000]
'zn<dq{f%\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
...snip...
```

As we previously mentioned, the RC4 key for the config is no longer the hardcoded value in the DLL. Instead, it is now the last 8 bytes from the decoded DLL blob:

```
>>> a = base64.b64decode('FE2frlPu/3cYTkUYWP9aoUwTUKZ778EWaz5b2nzDTz2OAR2qI5Jvqozn6a2BTADp7kUTrsTI6s
>>> rc4 = ARC4.new('\x24\x7b\x29\x75\x5e\x2f\x2e\x70')
>>> rc4.decrypt(a)
'0|5|5|||||eyJjaGFubmVsIjoi|In0=|0|1|symantecuptimehost.com|8080|Mozilla/5.0 (Windows NT 10.0; Win64
```

So, if we wanted to automate, we need to account for two methods I've seen being used for loading the config and DLL data by the shellcode layer.

Get Jason Reaves's stories in your inbox

Join Medium for free to get updates from this writer.

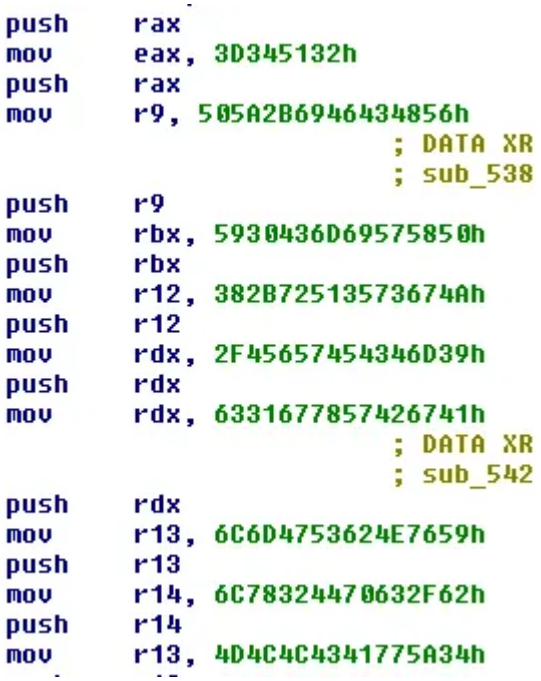
Remember me for faster sign in

The call over method which calls over the relevant data causing it's address to be pushed onto the stack:

Press enter or click to view image in full size

Call over method

Also the stack load method where chunks of the data are pushed onto the stack causing it to be rebuilt:



Stack load method

For the call over method, we just look for the instructions leading to the call and then pull out the data. I'll be using a naive method, but I would recommend switching the code to using YARA as your decoder will last much longer.

```
cfg_off = blob.find('\x5a\xe8\x00\x00\x00\x00\x59\x48\x01\xd1\x48\x83\xc1\x0a\xff\xd1')
cfg_len = struct.unpack_from('<I', blob[cfg_off-4:])[0]
cfg_off += 16
cfg = blob[cfg_off:cfg_off+cfg_len]
```

For finding the data in this scenario, we use a similar approach by just finding the call instruction sequence and pulling out the length while we are there:

```
if cfg != '':#Few ways to find the end
    #way1
    off1 = blob.find('\x41\x59\xe8\x00\x00\x00\x00\x41\x58')
    l = struct.unpack_from('<I', blob[off1-4:])[0]
    bb = blob[off1+19:]
    bb = bb[:l]
```

Decoding the config, then just involves first decrypting the DLL and recovering the key:

```
    rc4 = ARC4.new(bb[-8:])
    decoded = rc4.decrypt(bb[:-8])    rc4 = ARC4.new(decoded[-8:])
    decoded_cfg = rc4.decrypt(base64.b64decode(cfg))    print(decoded_cfg)
```

For the stack-based loading, I will be using the Unicorn[5] emulator which I've used for decoding data out of previous malware samples. First, we need the config data:

```
else:
    #need to pull from stack
    offset = data.find(needle)    blob = data[offset:]    STACK=0x90000
    code_base = 0x10000000
    mu = Uc(UC_ARCH_X86,UC_MODE_64)    test = re.findall(r'''4883e4f04831c050.+4889e168''',binascii.
    temp = [test[0][:-2]]
    mu.mem_map(code_base, 0x100000)    mu.mem_map(STACK, 4096*10)
    for i in range(len(temp)):
        #print(temp[i])
        try:
            blob = binascii.unhexlify(temp[i])
        except:
            blob = binascii.unhexlify(temp[i][1:])
        mu.mem_write(code_base, '\x00'*0x100000)
        mu.mem_write(STACK, '\x00'*(4096*10))        mu.mem_write(code_base,blob)
        mu.reg_write(UC_X86_REG_ESP,STACK+4096)
        mu.reg_write(UC_X86_REG_EBP,STACK+4096)
        try:
            mu.emu_start(code_base, code_base+len(blob), timeout=10000)
        except:
            pass
        a = mu.mem_read(STACK,4096*10)
        b = a.rstrip('\x00')
        b = b.lstrip('\x00')
        cfg = str(b)
```

For the data, we just need to account for a larger stack size:

```
    mu = Uc(UC_ARCH_X86,UC_MODE_64)#045e95f1a5bcc1ce2eeb905ab1c5f440a42364a170008309faef1cfdba296644
    test = re.findall(r'''00005a4[89].+4989e068''',binascii.hexlify(blob))
    if len(test) > 0:
        temp = [test[0][6:-2]]
        mu.mem_map(code_base, 0x100000)        mu.mem_map(STACK, 4096*200)
        for i in range(len(temp)):
            try:
                blob = binascii.unhexlify(temp[i])
            except:
                blob = binascii.unhexlify(temp[i][1:])
            mu.mem_write(code_base, '\x00'*0x100000)
            mu.mem_write(STACK, '\x00'*(4096*200))            mu.mem_write(code_base,blob)
            mu.reg_write(UC_X86_REG_ESP,STACK+(4096*100))
            mu.reg_write(UC_X86_REG_EBP,STACK+(4096))
            mu.emu_start(code_base, code_base+len(blob), timeout=100000)
```

```
            a = mu.mem_read(STACK,4096*200)        b = a.rstrip('\x00')
        b = b.lstrip('\x00')
        b = str(b)
```

Decoding the config is then the same process of first decrypting the DLL:

```
        rc4 = ARC4.new(b[-8:])
        t = rc4.decrypt(b[:-8])        rc4 = ARC4.new(t[-8:])        decoded_cfg = rc4.decrypt(base6
        print(decoded_cfg)
```

While enumerating samples off VirusTotal, we also discovered what looks more like a stager version:

```
d79f991d424af636cd6ce69f33347ae6fa15c6b4079ae46e9f9f6cfa25b09bb0
```

This version just loads a bytecode blob onto the stack:

```
push    r12
mov     rax, 23B66D2870547D1Bh
push    rax
mov     r13, 0EAE37AB0F8C09711h
push    r13
mov     r8, 57390FFD7AFA6E5Eh
push    r8
mov     rsi, 2F5FAADDB5A8323Bh
push    rsi
mov     rdi, 0A5B20234C20C06EFh
push    rdi
mov     r11, 8AC521558108B90Ah
push    r11
mov     r13, 75059F5DFA393EDEh
push    r13
mov     rcx, rsp
push    0FCh
pop     rdx
push    rbp
sub     rsp, 10h
call    sub_140004E1E
add     rsp, 10h
pop     rsp
pop     r15
pop     r14
pop     r13
```

Stager like version

The decoding of the bytecode config is once again just the last 8 bytes as an RC4 key:

```
|{"channel":"|"}|1|login.offices365.de|443|Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537
```

IOCs

```
symantecuptimehost.com
login.offices365.de
```

References

1: https://bruteratel.com/

2: https://github.com/Immersive-Labs-Sec/BruteRatel-DetectionTools/blob/main/ConfigDecoder.py

3: https://www.mdsec.co.uk/2022/08/part-3-how-i-met-your-beacon-brute-ratel/

4: https://www.trendmicro.com/en_us/research/22/j/black-basta-infiltrates-networks-via-qakbot-brute-ratel-and-coba.html

5: https://www.unicorn-engine.org/
