Threat Spotlight: MenuPass/QuasarRAT Backdoor

threatvector.cylance.com/en_us/home/threat-spotlight-menupass-quasarrat-backdoor.html

The BlackBerry Cylance Threat Research Team

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:


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":


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:


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:


Figure 4: Use of CppHostCLR technique


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:


Figure 6: Example AES and XOR decryption keys


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:


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”:


Figure 12. Backdoor assembly in memory (version 2.0.0.0)


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:


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:


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:


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:

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	Variant	Size	File Names
e24f56ed330e37b0d52d362eeb66c148d09c25721b1259900c1da5e16f70230a	**1**	153600	prints.dll
9bbc5b8ad7fb4ce7044a2ced4433bf83b4ccc624a74f8bafb1c5932c76511308	**1**	153600	EntApp.dll
fe65e5c089f8a09c8a526ae5582aef6530e1139d4a995eb471349de16e76ec71	**1**	153600	LSMsvc.dll
cf08dec0b2d1e3badde626dbbc042bc507733e2454ae9a0a7aa256e04af0788d	**1**	155136	useracc.dll
239e9bc49de3e8087dc5e8b0ce7494dabce974de220b0b04583dec5cd4af35e5	**2**	166912	Sezlnsrsvc.dll
cf981bda89f5319a4a30d78e2a767c54dc8075dd2a499ddf79b25f12ec6edd64	**2**	166912	wlytkansvc.dll
41081e93880cc7eaacd24d5846ae15016eb599d745809e805deedb0b2f7d0859	**2**	166912	Wbyfziosrvc.dll
1ddb533be5fa167c9a6fce5d1777690f26f015fcf4bd82efebd0c5c0b1e135f2	**2**	167728	tk.dll
26866d6dcb229bf6142ddfdbf59bc8709343f18b372f3270d01849253f1caafb	**3**	268872	Mpnrrdim.dll
7f7fc0db3ea3545f114ed41853e4dc3764addfa352c28b1f6643d3fdaf7076c5	**3**	268872	Witwaservc.dll
c8c707575bb87c17ec17c4517c99229a993f80a76261191b2b89d3cb88e24aea	**3**	268872	Icyowsvcext.dll
6037b5ce5e7eda68972c7d6dfe723968bea7b40ac05b0f8c779a1f1d542b4ae4	**3**	268872	Upqmnnphost.dll
cc02561e5632a2c8b509761ee7a23a75e3899441f9c77d778d1a770f0f82a9b7	**5**	297984	Pnniorpauto.dll, SvchostSvc.dll
c8f2cc7c4fdf8a748cb45f6cfb21dd97655b49dd1e13dd8cc59a5eab69cc7017	**5**	297984	UsyaerDataAccessRes.dll
0eff243e1253e7b360402b75d7cb5bd2d3b608405daece432954379a56e27bff	**6**	403948	11-PrivateBatch.dll
31f0ff80534007c054dcdbaf25f2449ee7856aceac2962f4d8463f89f61bb3b0	**6**	399280	Wostqrkfolderssvc.dll
e8f00263b47b8564da9bc2029a18a0fec745377372f6f65a21aa2762fc626d4c	**6**	400947	11-PrivateBatch.dll
56f727b3ced15e9952014fc449b496bfcf3714d46899b4bb289d285b08170138	**6**	358867	daoris.dll
721caf6de3086cbab5a3a468b21b039545022c39dc5de1d0f438c701ecc8e9df	**6**	349810	updgwnphost.dll
f8a7e8a52de57866c6c01e9137a283c35cd934f2f92c5ace489b0b31e62eebe7	**6**	377236	USHBEERDATAACCESSRES.DLL, 10-FileCopy.dll
f1c5a9ad5235958236b1a56a5aa26b06d0129476220c30baf0e1c57038e8cddb	**N/A[1]**	79360	ZpNxNaQ.dll, SvchostSvc.dll
0aa3d394712452bba79d7a524a54aa871856b4d340daae5bf833547da0f1d844	**N/A4**	73728	SvchostSvc.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[\\]^_`ABCDEFGHIJKLMNOPQRSTUVWXYZ{|}~" 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):

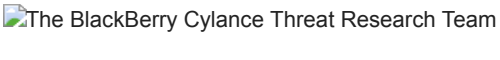
```
import "pe"
import "math"

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:

[1] https://github.com/quasar/QuasarRAT
[2] https://code.msdn.microsoft.com/windowsdesktop/CppHostCLR-e6581ee0
[3] https://github.com/quasar/QuasarRAT
[4] Service installer (a.k.a DILLJUICE)
[5] https://threatvector.cylance.com/en_us/home/cylance-vs-future-threats-the-predictive-advantage.html



About The BlackBerry Cylance Threat Research Team

The BlackBerry Cylance Threat Research team examines malware and suspected malware to better identify its abilities, function and attack vectors. Threat Research is on the frontline of information security and often deeply examines malicious software, which puts us in a unique position to discuss never-seen-before threats.
