Gh0stBins, Chinese RAT: Malware Analysis, Protocol Description, RDP Stream Recovery

By Electron, Jane and kinoshi

Published: 2023-06-22 · Archived: 2026-04-05 23:05:22 UTC

Electron

I'm a malware analyst. I love CTF, reversing, and pwn. Off-screen, I enjoy the simplicity of biking, walking, and hiking.

Jane

I'm ANY.RUN ambassador and a real network traffic numismatist. I also love penguins and tortoises. My motto is to do good and throw it into the sea.

kinoshi

I'm a dedicated programmer and malware analyst. I derive immense joy from the art of coding and have a deep passion for both low-level and system-level programming. I thoroughly enjoy delving into the intricacies of software and exploring how it operates at a fundamental level. My expertise extends to solving crackme challenges and participating in online CTF competitions, where I tackle complex tasks to enhance my skills.

It’s not every day that you come across a DLL so new that even VirusTotal draws a blank. But it’s even rarer when this sample turns out to be a sophisticated RAT from China.

But this is exactly what happened in our recent case. **We discovered what may be a previously unseen version of the Gh0stBins RAT** — an obscure malware family originating from the Middle Kingdom and sparsely studied in the field. Naturally, we had to analyze it.

The Chinese malware scene has recently undergone something of an industrial revolution, making modern Chinese malware a serious threat. In this article, **we’ll dive deep into this new Gh0stBins variant — and show you how to detect it with Suricata and YARA rules as well as recover leaked data using a Python script**.

Let’s get started.

How we Discovered this Gh0stBins Sample

At ANY.RUN, our team is always monitoring network activity of public samples, constantly on the lookout for signs of suspicious actions. We classify them into three main categories: backdoors, stealers, and loaders.

Today’s case started when we detected loader-type activity. This detection was achieved through a two-fold approach. First, using a unique rule specifically designed for xored files of PE EXE or DLL format. Second, by analyzing certain statistical features of the encrypted file — notably the autocorrelation function, a concept that will be discussed more comprehensively in the section on network rules.

As we continued our analysis, we discovered a significant similarity in the structure of packets from the system-installed backdoor to the structure of Gh0stRat packets. You’ll find these similar packets highlighted with the same color in the attached screenshots, and we’ll be discussing these similarities in greater detail in the following sections.

Gh0stRAT: https://app.any.run/tasks/f50156b5-c387-40a1-8eca-8f913babca06/

Gh0stBins: https://app.any.run/tasks/3b14ef62-5d21-48bb-a5e4-5b3fed402fb7/

Our sample's packets are suspiciously similar to Gh0stRat's

Stage 1: Loader Analysis

The initial loader consists of two files:

- the legitimate application 'net-service.exe' (part of VMware Workstation), which has a *valid digital signature* from "VMware, Inc"
- the malicious DLL 'shfolder.dll'

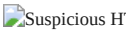

Process tree of the loader

At the time of writing this article, the malicious DLL has only received *3 detections* on VirusTotal.


3 detections on VirusTotal

Firstly, the main process with PID 3508 restarts itself from the same location. Secondly, it creates its own copy in the same directory with the name "vmnet.exe" and starts itself again.

Additionally, we discovered that two processes made HTTP requests to http://49[.]235.129.40/update/. This indicates that the loader may be attempting to download or update a payload:


Suspicious HTTP requests

The malicious library is loaded into all three processes using Search Order Hijacking technique (T1574.001) which was documented in the old CVE-2019-5526.

It is interesting that "shfolder.dll" has an artifact – a PDB path with Chinese characters translated as "over start":

```
E:\MyProjects\过启动\FakeDll\Release\shfolder.pdb
```

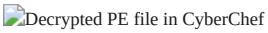
So far, a malicious code starts its execution at the initialization routine where static objects or libraries need to be initialized before the program execution:


Static objects and libraries are initialized before the program executes

The initialization routine of the loader unpacks two payloads that are encrypted with a XOR key '12345678AABBCCDD':

- shellcode is used to load an executable PE file;
- the malicious executable (not found on a VirusTotal).

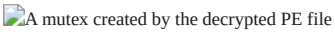
The following image shows the decrypted PE file with the help of CyberChef:


Decrypted PE file in CyberChef

The shellcode will be written to the main module's Entry Point using 'WriteProcessMemory' function, ensuring that when we reach that point, it will be executed, and the decrypted PE file will be mapped to memory:

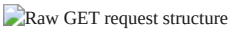

Decrypted PE file

The decrypted PE file creates a mutex, which is likely associated with the date of a sample compiled '2023.01.18.18.45':


A mutex created by the decrypted PE file

It is worth noting that the date is written in the Chinese date format, using the pattern **"yyyy年mm月dd日."** This observation could potentially indicate that the attacker has Chinese origins or is associated with China in some way.

The primary objective of the decrypted PE file, which is relatively small in size (around 7KB), is to download and execute a payload from a remote server. To achieve this task, it utilizes WinAPI functions such as ‘connect’, ‘WriteFile’, and ‘ReadFile’ to create a GET request. The structure of the GET request can be observed in the accompanying picture:


Raw GET request structure

At the time of writing the article, the remote server was still active. However, instead of returning the expected payload, it displayed a directory listing. Consequently, when the loader attempted to download the payload, it encountered an unexpected response, leading to a crash. The loader was originally designed to download a PE executable, and the directory listing caused an error in its execution.


The remote server displayed a directory at the time of writing, which led to loader crashing

In case when the payload was successfully downloaded, it needed to be decrypted using t**he XOR key ‘12345678AABBCCDD’.**

> **To proceed with our analysis, we manually downloaded the payload and decrypted it.**

You can examine the operational payload at this link.

Now, let us move on to the next stage.

Stage 2: RAT Analysis

The downloaded payload is a DLL with one exported function ‘shellcode_entry’:


The downloaded payload is a DLL

The DLL is a modular Remote Access Trojan (RAT) written in C++, and it is not currently present on VirusTotal (VT). The downloaded DLL is also a kernel module that serves as a connector for all the other components of the RAT.

The main execution flow of the RAT can be described roughly as follows:

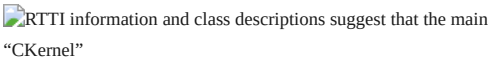

main execution flow of the RAT

The RAT is an IOCP and asynchronous client, which has a complex multithreaded structure, primarily based on the events. However, the detailed description of this structure is beyond the scope of this article. Instead, we will focus on discussing the exchange protocol in detail and highlight a few aspects of the RAT below.

It is interesting that the RAT contains forgotten debug logs, which can prove helpful for debugging purposes:


Forgotten debug logs we found in the RAT

Furthermore, the RAT includes RTTI (Run-Time Type Information) information and class descriptions. This tells us that the main class of the current module is likely named ‘CKernel’:


RTTI information and class descriptions suggest that the main class of the current module is named “CKernel”

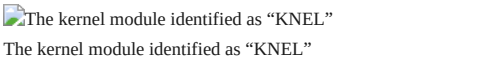
Let us now turn to discuss the exchange protocol.

Stage 3. Traffic Analysis

We’re going to analyze traffic based on this task. To perform a thorough analysis of the traffic, we recommend either downloading the PCAP (Packet Capture) file or following the network stream in the static discovery window available on ANY.RUN.

Initial Request: Module Registration

After establishing the connection, it is observed that the first outgoing packet always consists of 4 bytes, which describes the module connecting to the Command and Control (C2) server. In this particular case, the kernel module is identified by its short alias “KNEL”:


The kernel module identified as “KNEL”

Our hunting team has also discovered an RDP module, identified by the alias ‘RDTP’. Furthermore, through the process of reverse engineering the code, we can deduce the existence of additional modules. We can speculate about their intended purposes based on their names:

Name	Alias	Module description
kernel	KNEL	The heart of the RAT, a connector for all other modules
chat	unknown	Enables communication and interaction with the RAT operator or other users.
filemgr	unknown	Manages files and directories on the compromised system
rd	RDTP	Remote Desktop: Allows remote access and control of the target system's desktop.
camera	unknown	Controls and accesses the target system's camera for capturing images or videos.
microphone	unknown	Controls and accesses the target system's microphone for recording audio.
filedownloader	unknown	Downloads files from the internet onto the compromised system
kblog	unknown	Logs and records keystrokes on the target system
socksproxy	unknown	Sets up a SOCKS proxy server on the compromised system, allowing network traffic to be routed through it
cmd	unknown	Executes commands on the target system, providing remote control and administration capabilities

Initial Response: Registration Confirmed

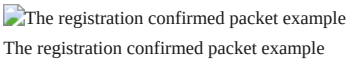
The server responds to the received ‘module registration’ packet with the following ‘registration confirmed’ packet:

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>a</th><th>b</th><th>c</th><th>d</th><th>e</th><th>f</th></tr>
<tr><td rowspan="2">0</td><td colspan="16">header</td></tr>
<tr><td colspan="4">magic bytes</td><td colspan="4">packet size</td><td colspan="4">decompressed size</td><td colspan="4">packet type</td></tr>
<tr><td>1</td><td colspan="2">p_type</td><td colspan="14">payload</td></tr>
</table>

The packet has the following fields:

- **magic bytes:** always contains the value “BINS” for all subsequent communications.
- **packet size:** the size of the packet excluding the header.
- **decompressed size:** is used only when the payload is compressed, and it represents the size of the decompressed data.
- **packet type**: type of the packet, which can have 2 values: **0x0** denotes a data packet and **0xABCDEF** indicates that the packet is a “heartbeat”.
- **p_type**: can have 2 types of values: **0x9C78**: payload is compressed with ‘zlib’ using fixed Huffman coding and **any** represents a command to process.
- **payload**: compressed or raw data.

Below, you can see an example of the “registration confirmed” packet:


The registration confirmed packet example

The decompressed command from the payload in the above picture can be viewed in CyberChef:

Decompressed command from the picture above

So that the server asks the client to send information about the host.

Client Identity

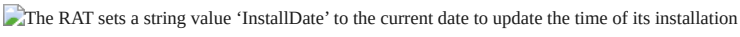
In response to the command received from the server, the client starts collecting information about the victim.

They do it in the following order:

1. Get IP address using WinAPI “getsockname”
2. Get computer name
3. Get user name
4. Get the Windows version using the WinAPI function “GetNativeSystemInfo” to obtain bitness and information from the registry key:

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Product

5. Create a registry key “HKEY_CURRENT_USER\SOFTWARE\HHClient”
   if it didn’t exist before. It also updates the date of the RAT installation by setting a string value ‘InstallDate’ to the current date:


The RAT sets a string value ‘InstallDate’ to the current date to update the time of its installation

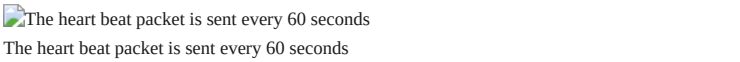
6. Get information about the processor from ‘HKLM\HARDWARE\DESCRIPTION\System\CentralProcessor\0’ and using GetSystemInfo API
7. Get information about drives via GetLogicalDrives and GetDiskFreeSpaceExW
8. Get memory size using GlobalMemoryStatusEx API
9. Check if the C2 is available by sending a echo-request (PING) packet to the attacker server
10. Check if a victim has a camera by enumerating available devices
11. Check if an attacker’s comment of the victim exists in the key “HHClient”

After collecting all the information, the RAT prepends it with a 2-byte prefix ‘0xEE01’, indicating that it is a client identity response, compresses it with “zlib” and sends it to the C2:


Exfiltrating data to C2

HeartBeat

Every 60 seconds the RAT sends the heartbeat packet (packet type is equal to 0xABCDEF) to the server to ensure the connection is still active. The server has to respond with the same packet type and zero payload len immediately:


The heart beat packet is sent every 60 seconds

Modules Downloading and Executing

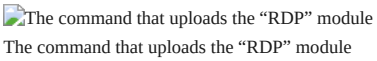
When the attacker decides to execute a command on the victim host, they send a packet similar to the ‘registration confirmed’ packet, but with a different command ID. The command ID is always 2 bytes in length. Depending on the packet type, the command ID can either be compressed or located in the position of the ‘zlib’ header.

Below is a list of all the available command IDs:

#	Cmd ID req	Cmd ID resp	Description
1	0x4552	0xEE01	Send victim info
2	0xDD01	0xEA05	Prepare for loading ‘cmd’ module
3	0xDD02	0xEA05	Prepare for loading “chat’ module
4	0xDD03	0xEA05	Prepare for loading “file manager” module
5	0xDD04	0xEA05	Prepare for loading “RDP” module
6	0xDD05	0xEA05	Prepare for loading “camera” module

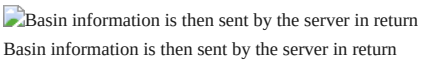
#	Cmd ID req	Cmd ID resp	Description
7	0xDD06	0xEA05	Prepare for loading “microphone” module
8	0xDD07	0xEA05	Prepare for loading “file uploader” module
9	0xDD08	-	Exit
10	0xDD09	0xEA05	Prepare for loading ‘keyboard log’ module
11	0xDD0A	0xEA08	Create a LNK file in the startup menu with name of “VMware NAT Service”
12	0xDD0B	0xEA08	Add itself to autorun via “HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run” with name “VMware NAT Service”
13	0xDD0C	0xEA05	Prepare for loading ‘socks proxy’ module
14	0xDD0D	-	Is not developed, has a comment “OnUtilsOpenWebPage”
15	0xEA04	-	Restart itself
16	0xEA07	0xFA00	Prepare memory for the payload
17	0xEE02	-	Reboot system
18	0xEE03	-	Force system shutdown
19	0xEE04	0xEE05	Save comment about the victim host to the registry
20	0xFA01		A part of the payload is received

In the analyzed task, the attacker sends a command 0xDD04 to upload the “RDP” module. In response, the client sends a confirmation of readiness to accept the payload with the bytes ‘rd’ at the end — the type of module to be loaded:

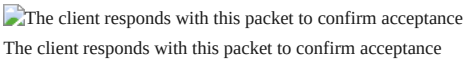

The command that uploads the “RDP” module

The server, in turn, sends basic information about the expected payload:

- command 0xEA07
- total size
- *resulting hash value* obtained by simply summing up all the bytes included in the payload after the final assembly
- ‘rd’ confirmation


Basin information is then sent by the server in return

The client allocates memory for the payload and confirms its acceptance by sending the following packet:

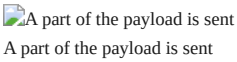

The client responds with this packet to confirm acceptance

This packet includes:

- confirmation command 0xFA00
- expecting payload size
- expecting payload hash
- the number of the received part
- the maximum size of the expecting part
- “rd” confirmation

It is also worth noting that the above packet will be sent to the server as confirmation of receiving every part with the only difference in the number of the received part.

Starting from this moment, the server will send the result payload part by part with a size that was agreed upon with the client. Each subsequent packet will have a structure similar to the following:


A part of the payload is sent

This data packet includes:

- command 0xFA01
- expecting payload size
- the size of the current part
- payload
- 4 bytes hash at the end of each packet calculated only for the current payload's part; the hashing algorithm used will be the same as described earlier

When the transaction is complete, the server may send a 0xDD08 command to exit from the kernel module, as was the case in our task.

At this moment, the downloaded RDP module is mapped to the memory and executed, which can be observed through the newly created connection:


A new connection indicates that the downloaded RDP is executed

To simplify the task of constructing the resulting payload, we have written a Python script that is already **available in our GitHub repository**. You can download the PCAP file and save the raw TCP stream 0 to a separate file. Then, you can apply our script, which will rebuild the payload from the captured traffic dump. As a result, you will obtain a new DLL containing the malicious RDP module.

Or, you could download a constructed payload with the simple DLL loader for your own analysis.

Stage 4. RDP module: basic description and protocol

The RDP module, the same as the 'kernel' module, is a DLL compiled against static CRT and OpenCL libraries. It includes an exported function called "ModuleEntry". This function takes the host and port as input arguments:


The exported function named "ModuleEntry"

The structure of the RDP module is similar to the "kernel" module, as it is also based on asynchronous events. It has its own commands and includes forgotten logging functions, which can be observed if we execute the module from the console:


A logging function was likely forgotten

It is worth mentioning that the RDP module also possesses a debug filename artifact, displaying the same developer's directory as the "kernel" module:


Debug filename artifact of the RDP module

The traffic structure of the RDP module is like the kernel's, except for the initial registration packet, which contains the keyword "RDTP":


The traffic structure of the RDP module

One interesting aspect to note is that the RDP module will not function properly if it is started by an external loader, as it lacks the call to the "WSAStartup" routine. This absence of initialization will result in a failure, leading to the module's exit. This could be a clever trick to protect the module from dynamic analysis, as well as a programmer mistake.

We won't spend our time analyzing the internal workings of the RDP. Instead, let's move on to a more interesting task: recovering a video stream.

Stage 5. RDP Module – Recovering a Video Stream and Leaked Data

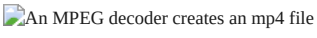
During our analysis, we wonder if it is possible to restore the video stream received by the attacker to gain insights into the leaked data. The answer is **yes** — we can do it.

To begin with, we discovered that the RDP protocol contains a NALU header with information about the upcoming video stream. In particular, we observed that the stream is encoded using the H.264 codec:

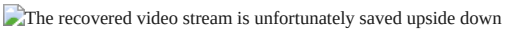

The video stream is encoded with the H.264 codec

Secondly, we have developed a Python script, **available in our GitHub** repository, which is capable of extracting the encapsulated video stream from the RAT traffic. The script concatenates the extracted data and saves it as a separate file.

Finally, we used a MPEG decoder to create an mp4 file:


MPEG decoder creates an mp4 file

As a result, we have the full video stream captured by the attacker, but upside down! Just compare the screen to the analyzed task:


We did it! But the recovered video stream is unfortunately saved upside down

Thus we may conclude that the stream is not encrypted at all which, for example, might help you to write a Suricata signature.

Stage 6. Fake RAT Server

In order to simplify the process of the protocol analysis and only for educational purposes we wrote a simple fake server for the RAT, which can only accept the client, send a registration packet, and a heartbeat. This script is available on **our GitHub page**.


We've spun up a fake RAT server strictly for educational purposes

Stage 7. Suricata Signature

We've developed 4 Suricata rules for detecting Gh0stBINS in network traffic. You can find them in **our GitHub repository**.

As an example, let's look at the key points of the Gh0stBins rule (sid: 8000054).

Suricata keyword	Description
flow: established, to_client;	Defines the direction of data packet transmission — from the remote PC to the client
dsize: 24;	The size of the payload of the transmitted packet is 24 bytes
content: "BINS";depth: 4;	Magic constant — beginning of the data packet
content: "|789c 0300 0000 0001|"; distance: 12; within: 8;	Payload of the Gh0stBins protocol, which is an empty zlib archive

Stage 8. YARA Rules

We've developed multiple YARA rules for detecting Gh0stBINS in memory and files. You can familiarize yourself with them in detail in our GitHub repository.

These YARA rules are designed to detect:

1. Malicious DLL, used for CVE-2019-5526
2. Core and RDP modules
3. Decryptor and loader shellcode

Conclusion

We hope that you've learned something new from today's analysis. Gh0stBins is indeed an unusual sample. Despite its challenges, analyzing it was highly rewarding and may provide insights into the strategies used by adversaries from China.

Don't forget, that we've written a Python script that can construct the payload from captured traffic dump for further analysis. We encourage you to download and try it. The script is available on our GitHub.

Interested in more malware deep dives? Read how we deobfuscated GuLoader, or how we examined the encryption and decryption of PrivateLoader.

Appendix 1: IOCs

Analyzed files:

Name	payload_decrypted.bin net-service.exe 7f426b327c878f799c74bb4b8a532cb3.exe shfolder.dll
MD5	4FEB48DDEB3F2BD55B2AF31BD77EAB2E D9B422F37FCAF61BD80E12CC03E84816 7F426B327C878F799C74BB4B8A532CB3 d

Name	**payload_decrypted.bin net-service.exe 7f426b327c878f799c74bb4b8a532cb3.exe shfolder.dll**
SHA1	20B5B6C2F24C2FDB9778BDFF5BC5976997C7E2AD 1D9D212620F342AE0D5440A067F4DE3AE12877F9 0315CC83C6D781DB16
SHA256	16F3191FF882670F1288E1836CF4683C7A74863AD0BFFE153FE4A668995A714B 4395003E0D81C685BE47C80DFF9DACCC2F0A3

Connections (IP)

- “118[.]107.7.166”
- “193[.]134.208.217”
- “49[.]235.129.40”

HTTP Request

- http://118[.]107[.]7[.]166/foxx/64.bin
- http://49[.]235.129.40/update/

Appendix 2: MITRE MATRIX

Tactics	**Techniques**	**Description**
TA0007: Software discovery	T1082: System Information Discovery	Collects system data
TA0011: Command and Control	T1071.001: Application Layer Protocol	Sending collected data to the control server
	T1105 Ingress Tool Transfer	Requests binary from the Internet
	T1572 – Protocol Tunneling	GhostBins protocol uses RDP
TA0005: Defense Evasion	T1027 – Obfuscated Files or Information	Attempt to make an executable or file difficult to discover or analyze by encrypting XOR
	T1140 – Deobfuscate/Decode Files or Information	Decrypts unpack file with XOR key
TA0005: Defense Evasion	T1574.001 – Hijack Execution Flow: DLL Search Order Hijacking	CVE-2019-5526

Electron

Leading malware analyst

I'm a malware analyst. I love CTF, reversing, and pwn. Off-screen, I enjoy the simplicity of biking, walking, and hiking.

Jane

Leading network traffic analysis expert at ANY.RUN

I'm ANY.RUN ambassador and a real network traffic numismatist. I also love penguins and tortoises. My motto is to do good and throw it into the sea.

kinoshi

Malware analyst at ANY.RUN

I'm a dedicated programmer and malware analyst. I derive immense joy from the art of coding and have a deep passion for both low-level and system-level programming. I thoroughly enjoy delving into the intricacies of software and exploring how it operates at a fundamental level. My expertise extends to solving crackme challenges and participating in online CTF competitions, where I tackle complex tasks to enhance my skills.
