Dropping the Anchor

netscout.com/blog/asert/dropping-anchor

- malware
- Banking Trojans
- Bot

Trickbot Anchor Analysis

by Suweera De Souza on October 26th, 2020

Executive Summary

Trickbot has long been one of the key banking malware families in the wild. Despite recent disruption events, the operators continue to drive forward with the malware and have recently begun porting portions of its code to the Linux operating system. As this technical deep dives shows, the communication between the command-and-

control (C2) server and the bot are extremely complex. Additionally, we have analyzed the C2 communication process of the Linux2 version of Trickbots' Anchor module.

Key Findings

- Trickbot operators leverage a complex communication schema to control infected machines.
- Recent efforts show the operators moving portions of their code to Linux, thus increasing the portability and range of possible victims.
- The Anchor module can implement techniques such as process hollowing and process doppelgänging to evade analysis.
- Both Windows- and Linux -based bots have the ability to install additional modules in a victim's system.

Communication Setup

Trickbot's Anchor framework is a backdoor module discovered in 2018. Unlike Trickbot's typically broad-based campaigns, Anchor is deployed exclusively on selected targets. Anchor's communication with the C2 currently uses DNS tunneling, which we will break down later.

This communication involves a few different steps. The purpose of this blog is to make sense of this communication and understand how the C2 operates the bot.

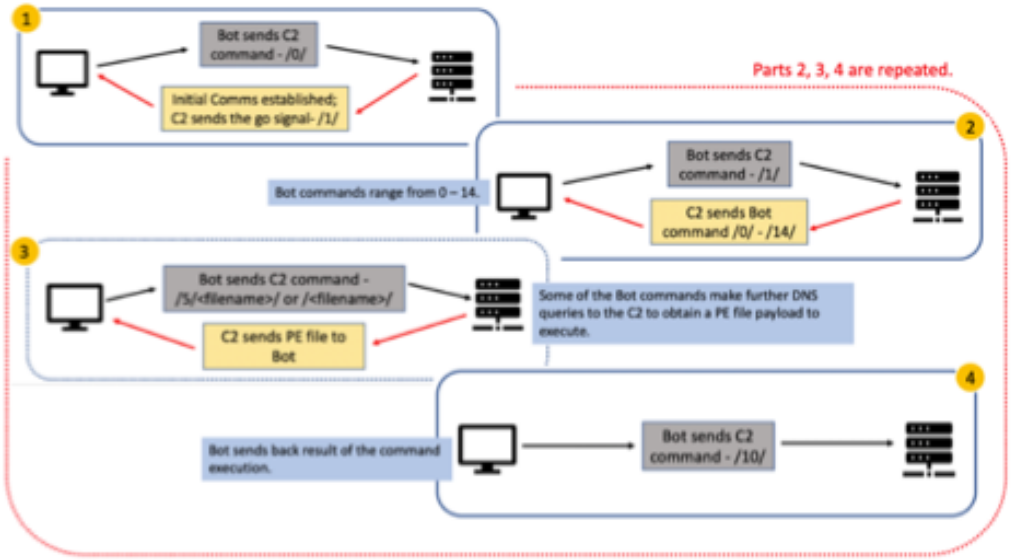

Figure 1: Parts of communication between Bot and C2

From a high-level view, **Figure 1** shows the flow of communication between the bot and the C2. Throughout the communication process; there are commands meant for the bot (which will be termed as *bot_commands*) as well as the C2 (termed as *c2_commands*). A description of the commands seen in **Figure 1** is provided in **Table 1** below.

1. Part 1 of the communication is the initial setup between the bot and the C2. The bot sends the *c2_command 0* to the C2, which contains information about the client, including the bot ID (part 1 of **Figure 1**). Once the initial communication is established, the C2 responds with a message that contains the signal */1/*.
2. In part 2 of the communication, the bot sends back the same signal (which is also the *c2_command 1*), and the C2 responds with a *bot_command* (part 2 of **Figure 1**).
3. The bot may make further request for the C2 to send an executable file, depending on the initial bot command received (part 3 of **Figure 1**).
4. Finally, the bot sends back the result of the execution to the C2 (part 4 of **Figure 1**).

Table 1: Parts of communication between bot and C2

C2 Commands	Purpose	Bot Commands (Windows)	Purpose	Bot Commands (Linux)	Purpose
0	Initial C2 Comms setup/register bot	0	execute instruction via cmd.exe	0	Execute instruction via cmd.exe in the Windows shares
1	Ask C2 for *bot_command*	1 or 2	Execute EXE in *%TEMP%*	1 or 2	Execute file in Windows share
5	Obtain PE file	3 or 4	Execute DLL in *%TEMP%*	3 or 4	Execute DLL with export *control_RunDLL* in Windows shares
<filename>	Obtain PE file	5 or 6	Execute PE file using process hollowing	10, 11, or 12	Execute Linux file
10	Result of the execution of the *bot_command*	7 or 8	Execute PE using process doppelganging	100	Check bot GUID
		9	Execute instruction via pipe object to cmd.exe		
		10	Execute instruction via pip object to powershell.exe		
		11 or 12	Inject PE into multiple process		
		13	Change the bot's scheduled task		
		14	Uninstall the bot		

Creation of DNS queries

Every part of communication (**Figure 1)** made to the C2 follows a sequence of 3 different DNS queries (**Figure 2**). NTT previously published an article that explores how the DNS queries were created by the bot to the Anchor C2 server. In this blog, we researched further into the protocol to better understand the role of each part of the DNS query.

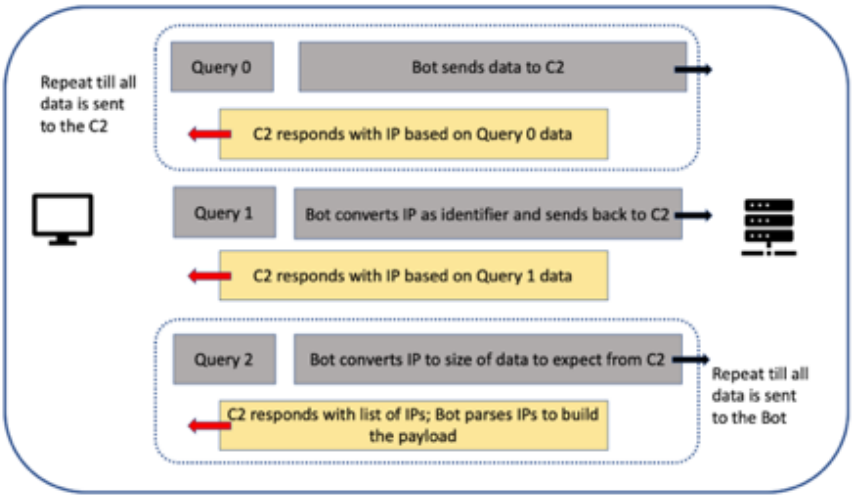

Figure 2: High-level overview of the DNS queries

Figure 2 gives a high-level overview of what the DNS queries look like. Each of these queries have their own format on the type of data that is sent to the malware C2, as explained below:

Query 0

Bot DNS Query

```
0<UUID bytes><current_part><total_parts>/anchor_dns/<Bot_GUID>/<c2_command>/<content>/
```

- 0 – Indicating type 0 query
- UUID – 16 bytes in length generated by the bot
- current_part – The current part of the data being sent (this is further explained below)
- total_parts – Total number of parts the data is divided into
- anchor_dns – The type of Anchor bot that is communicating to the C2.
- Bot_GUID – The GUID generated is different for Windows and Linux platforms
- c2_command – The command meant for the C2
- content – The content to send based on the type of command (**Table 2**)

The Anchor module generates a GUID which is different for each platform:

- Windows - <hostname_windowsVersion>.<32 bytes client id>
- Linux – <system_linnuxVersion>.<32 bytes client id>

Each command sent to the C2 is followed by its own set of content (**Table 2**):

Table 2: Content for c2_command

c2_command	Content
0	/<Windows OS Info>/1001/<bot_IP><64 bytes random bytes><32 bytes random alphanumeric characters>/
1	/<32 bytes random alphanumeric characters>/
5	/<filename>/
<filename>	N/A
10	/<bot_command>/<bot_command_ID>/<result of bot execution>/

Since the maximum length of a DNS name is 255 bytes, the data sent for the first query gets sent in parts. This also explain the fields *current_path* and *total_parts* seen in the type 0 query. Below is the pseudocode on how many parts the data gets divided into:

```
def get_total_parts(c2, data):
    divider = ((0xfa - len(c2)) >> 1) - 0x18
    size = len(data)
    return (size / divider) + 1
```

The data sent to the C2 is crafted as subdomains after it is xor'ed with the key. The key continues to remain 0xb9.

The example below shows what this data would look like with content sent to c2_command 0 and how many parts it gets divided into:

- 0<GUID>\x00\x03/anchor_dns/WIN-COMP_W617601.HGDJ3748EURIHDGV192873645672DFGW/0/Windows 7/1001/0.0.0.0/
- 0<GUID>\x01\x03EAA477CDE0E29EF989E433E633F545A09FD3178993712114490620 2B0EFD32CB/Tb1i5Xc
- 0<GUID>\x02\x03Zih0P1wW70rhjGp7G75WsFu69/

C2 Response

After every part of the query gets sent, the C2 responds with an IP. The bot uses this IP to obtain the identifier value that will be used in the next query sequence.

```
def get_identifier(IP):
    return inet_aton(IP) >> 6
```

Query 1:

Bot DNS Query

Both platforms have the same query for type 1. And similarly, the data is created as a subdomain after being xor'ed with 0xb9.

```
1<UUID><dw_Identifier>
```

dw_Identifier – The same value that was sent by the C2 to the bot for query type 0

C2 Response

The C2 responds with an IP. This IP is also passed through the same function routine as *get_identifier* in the pseudocode above, and the resulting value is the size of the data to be expected from the final query type.

Query 2:

Bot DNS Query

Both platforms have the same query for type 2.

```
2<UUID><dw_Identifier><dw_DataReceivedSize>
```

- dw_Identifier – The same value that was sent by the C2 to the bot for query type 0
- dw_DataReceivedSize – The size of the data that has been received thus far.

The bot keeps sending a query type 2 request until the total size of the data received from the C2 matches that of the value sent by C2 in response to query type 1.

C2 Response

With each type 2 DNS query made by the bot, the C2 responds with a list of IP records. This list of IPs (**Figure 3**) is a structure on how the data is constructed and is exactly as that mentioned by NTT.

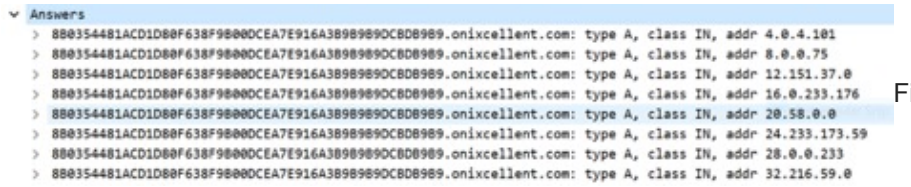

Figure 3: Records of IPs sent by the C2

The first dotted decimal number of the IP gives the order in which the list of IPs is parsed by the bot.

- IPs of this form 4.?.?.? show how much data has been sent by the C2, with the last 3 dotted decimal numbers of the IP indicating the size.
- IPs of this form 8.?.?.? show the size of data in the current record list, with the last 3 dotted decimal numbers of the IP indicating the value.
- The additional IPs in Figure 3 are all data in which the last 3 dotted decimal numbers of the IP are concatenated together.

In **Figure 4** below, we see an example PE file payload of IP records sent by the C2.

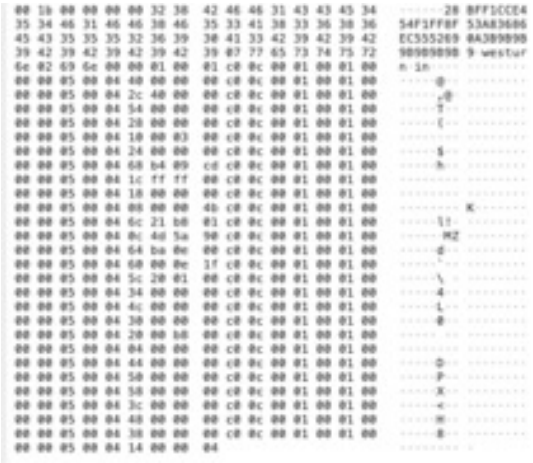
Figure 4: Example PE file payload

Windows Anchor

The final data received from the C2 has the following structure:

/<bot_command>/anchor_dns/<Bot_GUID>/<32_bytes_random_alphanumeric>/<bot_command_ID>/\r\n<base64_encoded_d

base64_encoded_data: Information used by the resulting bot_command subroutine.

Bot_Command 0

base64_encoded_data from C2 – decodes to a series of arguments

Executes a series of arguments through cmd.exe

Bot_Command 1 & 2

base64_encoded_data from C2 – decodes to filename and/or file arguments

- The bot makes DNS queries to C2 for a PE file.
  - bot_command 1 sends c2_command 5
  - bot_command 2 sends c2_command <filename>
- The EXE is created in the *%TEMP%* directory with a prefix *tcp* and executed.

Bot_Command 3 & 4

base64_encoded_data from C2 – decodes to filename and/or file arguments

- The bot makes DNS queries to C2 for a PE file.
  - bot_command 3 sends c2_command 5
  - bot_command 4 sends c2_command <filename>
- The DLL is created in the *%TEMP%* directory with a prefix *tcp* and executed (**Figures 5** and **6**).

Figure 5: 64bit Anchor PE run being run

Figure 6: 64bit Anchor PE executing a DLL

Bot_Command 5 & 6

base64_encoded_data from C2 – decodes to filename and/or file arguments

- The bot makes DNS queries to C2 for a PE file.
  - bot_command 5 sends c2_command 5
  - bot_command 6 sends c2_command <filename>
- The PE file is injected into a process via process hollowing.

Bot_Command 7 & 8

base64_encoded_data from C2 – decodes to filename and/or file arguments

- The bot makes DNS queries to C2 for a PE file.
- bot_command 5 sends c2_command 5
- bot_command 6 sends c2_command <filename>
- The PE file is injected into a process via process doppelgänging.

Bot_Command 9

base64_encoded_data from C2 – decodes to a series of arguments

Executes a series of arguments through a pipe object to cmd.exe

Bot_Command 10

base64_encoded_data from C2 – decodes to a series of arguments

Executes a series of arguments through a pipe object to powershell.exe

Bot_Command 11 & 12

base64_encoded_data from C2 – decodes to filename

- The bot makes DNS queries to C2 for a PE file.
  - bot_command 11 sends c2_command 5
  - bot_command 12 sends c2_command <filename>
- This PE file is injected into 3 different running processes that get created.
- These processes are explorer.exe, mstsc.exe, and notepad.exe

Bot_Command 13

base64_encoded_data from C2 – decodes to a scheduled task string

The bot changes the scheduled task of the bot

Bot_Command 14

Uninstall the Anchor

Linux Anchor

The Linux Anchor module was first discovered by Stage 2 Security. The analysis done by Stage 2 Security is extensive, but we wanted to take a closer look at the C2 communications in the same way we did with the Windows version above.

The *bot_commands* from 0-4 contains smb2 information (which includes domain, user, and password) to be used by the Linux module when attempting to connect to any Windows shares. The module has an embedded PE file that is used to execute commands or files on the Windows shares.

Bot_Command 0

base64_encoded_data from C2 - decodes to a series of arguments

Executes a series of arguments through cmd.exe on the Windows shares.

Bot_Command 1 & 2

base64_encoded_data from C2 - decodes to filename and/or file arguments

- The bot makes DNS queries to C2 for a PE file.
  - bot_command 1 sends c2_command 5
  - bot_command 2 sends c2_command <filename>
- The file gets executed.

Bot_Command 3 & 4

base64_encoded_data from C2 - decodes to filename and/or file arguments

- The bot makes DNS queries to C2 for a PE file.
  - bot_command 3 sends c2_command 5
  - bot_command 4 sends c2_command <filename>
- The DLL files export function Control_RunDLL gets executed.

Bot_Command 10 & 11 & 12

base64_encoded_data from C2

- For bot_command 10, the encoded data is a Linux file that is executed by the bot.
- Bot_Commands 11 and 12 make DNS queries to C2 for a Linux file.
    - bot_command 11 sends c2_command 5
    - bot_command 12 sends c2_command <filename>
- The bot sets the file's permissions to 777 and executes it.

Bot_Command 100

base64_encoded_data from C2 - decodes to a GUID

- The bot checks whether the GUID sent by the C2 matches that of the bot's GUID.
- If the GUIDs do not match, the bot terminates C2 communication.

Conclusion

The complexity of Anchor's C2 communication and the payloads that the bot can execute reflect not only a portion of the Trickbot actors' considerable capabilities, but also their ability to constantly innovate, as evidenced by their move to Linux. It is important to note that Trickbot operators aren't the only adversaries to realize the value of targeting other operation systems. Earlier this year, we analyzed a DDoS bot called Lucifer designed to run on both Windows and Linux platforms. As we see more adversaries building cross-compile malware families, it seems clear that security professionals must re-evaluate security practices for Linux systems to ensure they are well-prepared to defend against these increasing threats.

Indicators of Compromise:

Anchor C2s:

- westurn[.]in
- onixcellent[.]com
- wonto[.]pro
- ericrause[.]com

Anchor PE 64bit

- SHA256 - c427a2ce4158cdf1f320a1033de204097c781475889b284f6815b6d6f4819ff8
- SHA256 - 4e5fa5dcd972170bd06c459f9ee4c3a9683427d0487104a92fc0aaffd64363b2

Anchor ELF 64bit

SHA256 - 4655b4b44f6962e4f9641a52c24373390766c50b62fcc222e40511c0f1ed91d2

Anchor PE 32bit Helper file for Linux

SHA256 - 7686a3c039b04e285ae2e83647890ea5e886e1a6631890bbf60b9e5a6ca43d0a

Posted In

Malware

Subscribe

Sign up now to receive the latest notifications and updates from NETSCOUT's ASERT.