Adjusting the Anchor

Published: 2021-07-15 · Archived: 2026-04-05 16:58:10 UTC

Overview

AnchorDNS is a backdoor used by the TrickBot actors to target selected high value victims. It has been seen delivered by both TrickBot and Bazar[1] malware campaigns[2]. AnchorDNS is particularly difficult to track given that it is deployed only post-infection and that too only after a period of reconnaissance, once the malware operators have established that the target is of special interest.

Following analysis of AnchorDNS samples published in recent reporting[2][3], we have observed that the C2 communications protocol of AnchorDNS has changed. We also see the use of another Anchor component called AnchorAdjuster. The newer variants contain a modification to the structure of the messages sent to the C2, and have added additional encryption routines when creating the DNS queries. Data received from the C2 is now encoded, thereby making the traffic less obvious.

In this post we analyze the role that AnchorAdjuster plays and outline the changes made to the communication protocol by the recent AnchorDNS samples.

AnchorAdjuster

AnchorAdjuster is a tool that is used to modify an AnchorDNS sample with an updated config and the victim's UUID. The tool is executed by an external command and has been seen being run by CobaltStrike[2].

A valid series of arguments need to be passed to the AnchorAdjuster for it to execute succesfully. If arguments are not passed, the tool outputs a message onto the console detailing the arguments required:

```
 using:
anchorAdjuster* --source=<source file> --target=<target file> --domain=<domain name>
    --period=<recurrence interval, minutes, default value 15>-guid
```

Below is a description of the arguments:

Argument	Description	Requirements
`--source`	AnchorDNS sample with a blank config	Required
`--target`	Name to save the modified AnchorDNS sample	Required
`--domain`	Domain C2s to save as config	Required
`--period`	Interval between each cycle of DNS queries; default is 15 minutes	Optional

Argument	Description	Requirements
`--lasthope`	Number of communication attempts; default is 100	Optional
`-guid`	Flag for initializing the Victim's `UUID` in the sample	Required

The AnchorAdjuster tool works as follows:

Firstly, if it finds a 16 byte string of `AAAAAAAAAAAAAAAA` in the AnchorDNS bot, it rewrites it with a `UUID` that it generates by calling `CoCreateGuid`. This creates a `UUID` unique to the victim machine. The string `AAAAAAAAAAAAAAAA` acts as a placeholder for the `UUID` and is typically stored in the `.rand` section.

Secondly, if it finds a 66 byte string of all `B`s, it overwrites this string with XOR encoded C2s. The C2s are the values that were passed to the AnchorAdjuster's `--domain` argument. The XOR key used is a hardcoded hex value `0x23`.

Finally, using the name passed to the `--target` argument, the tool creates a new AnchorDNS bot with these modifications.

Below is an example standard output log from the tool after successful execution:

```
 source file size 347648
guid: 743E900F5861EF468E120559E9D23EF8, shift 0x00053C00(343040)
domain: shift 0x00053A04(342532)
OK
```

This technique reuses an AnchorDNS sample to be able to communicate to new C2s that it provides, without having to re-compile an entirely new AnchorDNS binary. This also helps the threat actors to hide any new C2s created, especially if the AnchorDNS sample were to be discovered by a threat researcher.

AnchorDNS

AnchorDNS communicates to its C2 servers using DNS Tunnelling. Using the DNS protocol for command & control benefits AnchorDNS because such requests are often allowed to pass through firewalls. Using this method, AnchorDNS is able to exfiltrate data to its C2s in the form of DNS queries. The data is encoded and made to appear as subdomains. In addition, the C2 can communicate back to the bot by sending information in the form of DNS A records whereby the data is reconstructed by the bot based on AnchorDNS's specific format.

Review on how AnchorDNS works

To get a better grasp on what new changes have been implemented to this DNS communication, this section will do a quick high-level review on how AnchorDNS works.

1. Upon initial execution, AnchorDNS gains persistence on the machine by creating a scheduled task that is set to run every 15 minutes.

    - The frequency of the scheduled task can be modified again by the bot if the C2 sends a command with instructions to do so.
2. Each run cycle involves a series of commands transmitted as DNS queries between the bot and the C2.
    - Initial beacon message.
    - Request from the bot for command to be executed.
    - Request from the bot for a payload (if the command requires one).
    - Send report on the command's execution.

Preparing the messages for the C2

The name of the bot at the start of the message has changed from `anchor_dns` to `stickseed`. This new name is very different from that of the name used in the past variants[4]. One possible explanation is that `tick` in `stickseed` represents the Windows API `GetTickCount` and `seed` for a pseudorandom number generator, the two functions that we see being frequently used in the new variant.

The `GUID` created by the Bot is recorded by the C2 to keep track of the different infected machines. The format of the `GUID` is as follows:
`<Computer_Name>_W<major version><minor version><version build number>.<16 bytes UUID>`

The 16 byte `UUID` is hardcoded in the `.rand` section of the AnchorDNS PE file. If there are no 16 bytes in the `.rand` section or if there is a string `AAAAAAAAAAAAAAAA` in that section, the bot skips making any DNS queries.

Example `GUID`:
`ADMINWIN10_W629200.1BDD88D8278746A68CE4BCF8DCF27B7E`

Below is a summary of the messages and the command sent to the C2:

C2 Command	Description	Info sent by New Variant	Info sent by Previous Variant
0	Register Bot	`/stickseed/<GUID>/0/<Windows OS Type>/1001/<Bot IP>/<32 random hex bytes>/<32 random alphanumeric characters>/`	`/anchor_dns/<GUID>/0/<Windows OS Type>/1001/<Bot IP>/<32 random hex bytes>/<32 random alphanumeric characters>/`
1	Request Bot command	`/stickseed/<GUID>/1/<32 random alphanumeric characters>/`	`/anchor_dns/<GUID>/1/<32 random alphanumeric characters>/`
5	Request File	`/stickseed/<GUID>/5/<filename>`	`/anchor_dns/<GUID>/5/<filename>`
10	Send result of Bot command execution	`/stickseed/<GUID>/10/<Bot Command>/<Bot Command ID>/<Result of Command execution>/`	`/anchor_dns/<GUID>/10/<Bot Command>/<Bot Command ID>/<Result of Command execution>/`

The DNS Queries

Each message above, made by the AnchorDNS bot, to send to the C2 involves a sequence of 3 types of DNS queries[5]. This order is still maintained in the new variants. The table below shows a summary of the sequence of DNS queries made:

Query Order	Info Sent	Info Received
0	Send info including command	Receive IP record from C2
1	Convert IP to identifier and send to C2	Receive IP record from C2
2	Convert IP to size; send identifier and size to C2	Receive data in the form of multiple IP records

Crafting the Queries

The new variants make changes to the way in which the queries are crafted.

Old Variant:

To better understand the changes made, this section will briefly review how the queries were crafted in the previous variants. Each query would contain information about the query type and a 16 byte UUID. The query type would inform the C2 on what type of message it is receiving and the UUID helps it keep track of the queries. If the crafted query is type 0, the message gets divided into parts. This is to ensure that the length of the query remains under 255 characters. Finally, the queries are XOR'ed with the key `0xb9`. This is the only encoding we see in the previous variants.

The table below summarizes the queries crafted in the old variants[6]:

Query Order	Query Type	Old Variant Format	Encoding
0	0	`0<UUID><(BYTE)Current Part><(BYTE)Total Parts><Divided Message>`	xor with `0xb9`
1	1	`1<UUID><(DWORD)Identifier>`	xor with `0xb9`
2	2	`2<UUID><(DWORD)Identifier><(DWORD)Size>`	xor with `0xb9`

New Variant:

In the new variant, before a query is crafted, the message in each DNS query type is XOR'ed with the key `United States of America (USA)`. After encoding the message, a 16 byte UUID is generated for each query type (like the previous variant, the UUID is for the C2 to keep track of the query) and is further encoded with a custom Base32 algorithm using the custom dictionary `dghbcijklmnfqrwxyz23stuopaev4569`.

The bot then calculates if the message needs to be divided into parts for all 3 DNS query types (in the previous variant we see this for only the query type 0).

Below is a python function that calculate the number of parts a message would get divided into and the size of each part:

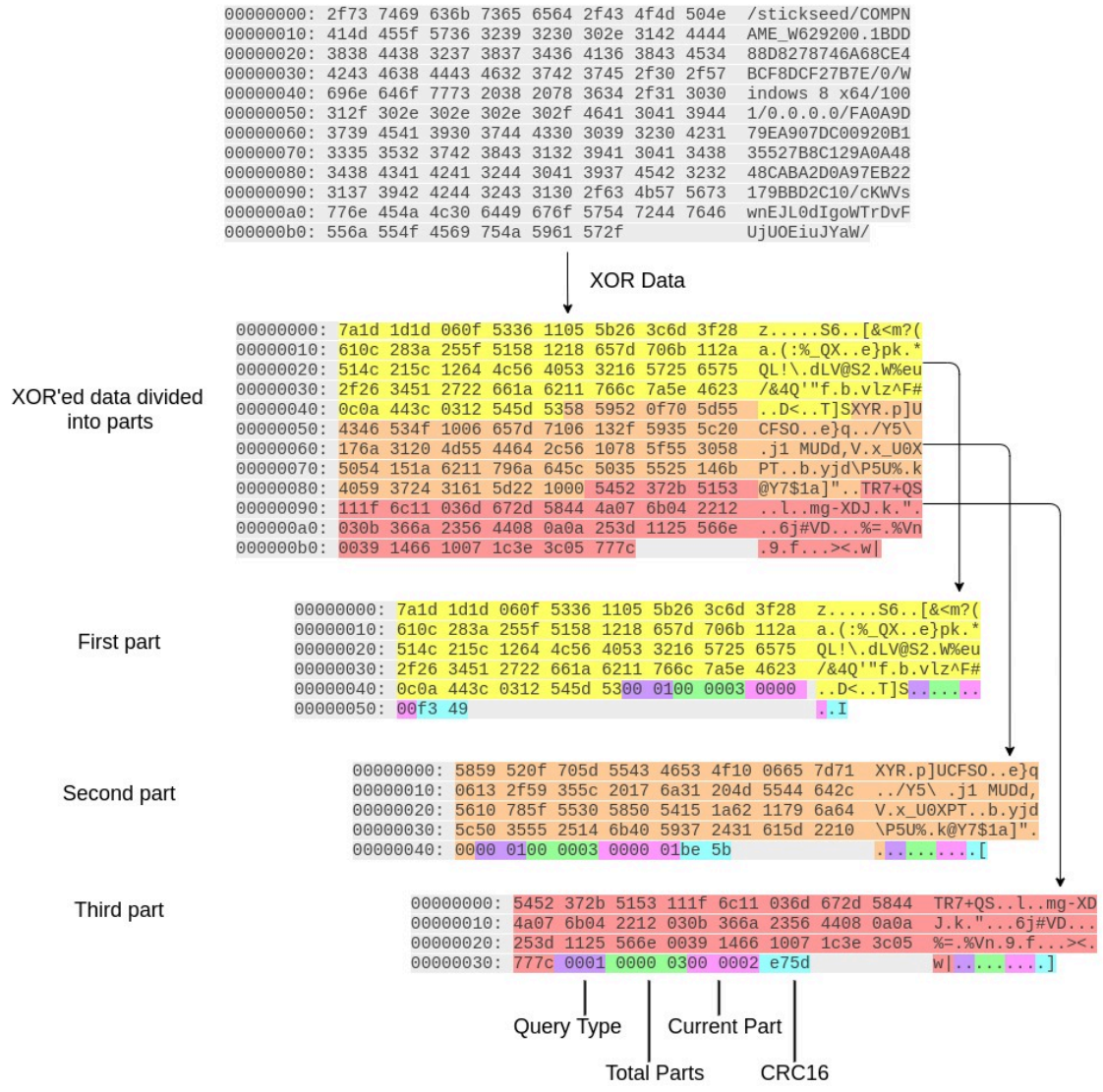
```
 import random

def get_parts(msg_len: int, c2_len: int) -> list():
    blocks = list()
    foo = 5 * (0xba - 0x1a - c2_len - 8)
    fee = ((foo & 7) + foo) >> 3
    faa = fee * 0.85
    if faa > (fee - 5):
        faa = (fee - 5) * 0.85
    i, count = 0, 0
    while i < msg_len:
        block_sz = msg_len - i
        if (msg_len - i) > fee:
            rand = random.randint(0,0x7fff)
            fii = fee - 5
            if count:
                fii = fee
            block_sz = int(((rand * (fii - faa)) / 32767.0) + faa)
        i += block_sz
        count += 1
        blocks.append(block_sz)
    return blocks
```

In the new variant, the DNS query types are labeled differently (but still follow the same order as the previous):

Query Order	Query Type	Message
0	`0x0001`	`/stickseed/<GUID>/<C2 Command>/<Info if any>/`
1	`0xfffe`	`<Identifier DWORD>`
2	`0xffff`	`<Identifier DWORD><Size in DWORD of data received>`

For each divided message part, additional information is appended. The image below gives an example of a message for DNS query type 0x0001 and how each divided part is crafted:



Message being built for the queries

The resulting data is encoded with a custom Base32 algorithm and the encoded Base32 UUID is appended at the end. So for example, the message parts above would result in the following types of DNS queries being made:

```
efkezwpdxpsq3lsdv2mp3u5kl.mppdslkiaohiqmhplaekp.rrzynhijic42cljjandescbf4nim
    .anoopcsmswhzpqeyphgvzre3oqsz.ygndzp3glhsnojidcddddjddddddabb.ygacsziqpmpqcvdkb2zhu2gjzg
        .domain.com

pnuctkdw5ntjcbrnxhcqy2txz3gjzo.cftgod2flrzglesnzlcbfqildx9ntdbqgns
    .nisgziha3eljwgntmtnhnqrdnuwb2cjgfoch.ldddlddddqddddhdpby.ygacsziqpmpqcvdkb2zhu2gjzg
        .domain.com

s2sw3tcn3nc6guihblvwuudfc22wytzdhz.cjyipjnvlqihggnyhn26chizt4jdcksya
    .dzbyb6gxnyvgdgdddygdddydjlqd.ygacsziqpmpqcvdkb2zhu2gjzg.domain.com
```

Query Responses

The query responses for each DNS query type have been slightly modified. Before the start of making the 3 types of DNS queries, the bot tries to resolve the C2 domain to an IP address. This IP address is used as a check by the bot to confirm if the C2 has received the message. Below is a table on what each response means.

C2 IP Record Response	Description
`255.255.255.255`	Retry, cannot reach
<C2_IP>	Message received by C2, send next message part of the query type
`239.255.255.255`	Sleep and retry
Single IP	For query type `0xfffe`, the IP is the identifier
Multiple IPs	For query type `0xffff`, the IPs form as a structure for the Bot to parse to data

As with the previous version, the DNS query type `0xffff` responds with multiple IP records. These records form a particular structure (that has been reverse engineered before[56]), whereby the final message is constructed. The change seen is that the resulting data built from the IP records is `xor` encoded. The key to decode the message is `Miguel de Cervantes Saavedra`.

Conclusion

Despite their simplicity, the changes seen in AnchorDNS are still effective in evading detection. The use of AnchorAdjuster allows the threat actors to modify the AnchorDNS backdoor in-place, providing a stealthy way to add fresh C2s that have been created for new targets. The actors behind AnchorDNS continue to actively develop their toolset, increasing flexibility and raising the barrier for detection.

IOCs

SHA256	Description
cbff159d0b178734248209ae70565d09dddf397ea4e897bf99206ddd74673e6f	AnchorDNS 64-bit DLL
a8a8c66b155fcf9bfdf34ba0aca98991440c3d34b8a597c3fdebc8da251c9634	AnchorDNS 64-bit DLL
9fdbd76141ec43b6867f091a2dca503edb2a85e4b98a4500611f5fe484109513	AnchorDNS 64-bit DLL
ba801f1c2e2c5f5cd961e887cb0776f2d5cee8d17164f29b138a8952dd162165	AnchorDNS 64-bit DLL
0d6a10df6eeb1dbb88b4d625873ed13daa367e165374a72daa16170af3ee31a0	AnchorDNS 64-bit DLL
f93b838dc89e7d3d47b1225c5d4a7b706062fd8a0f380b173c099d0570814348	AnchorAdjuster 64-bit EXE
3ab8a1ee10bd1b720e1c8a8795e78cdc09fec73a6bb91526c0ccd2dc2cfbc28d	AnchorAdjuster 64-bit EXE

SHA256	Description
c1ae70683da042792a504847b426a55cdcbca80dca12517f581a4e089a1f8932	AnchorAdjuster 64-bit EXE

C2s

```
 farfaris[.]com
kalarada[.]com
xyskencevli[.]com
sluaknhbsoe[.]com
jetbiokleas[.]com
nyhgloksa[.]com
```

References
