A quirk in the SUNBURST DGA algorithm

blog.cloudflare.com/a-quirk-in-the-sunburst-dga-algorithm/

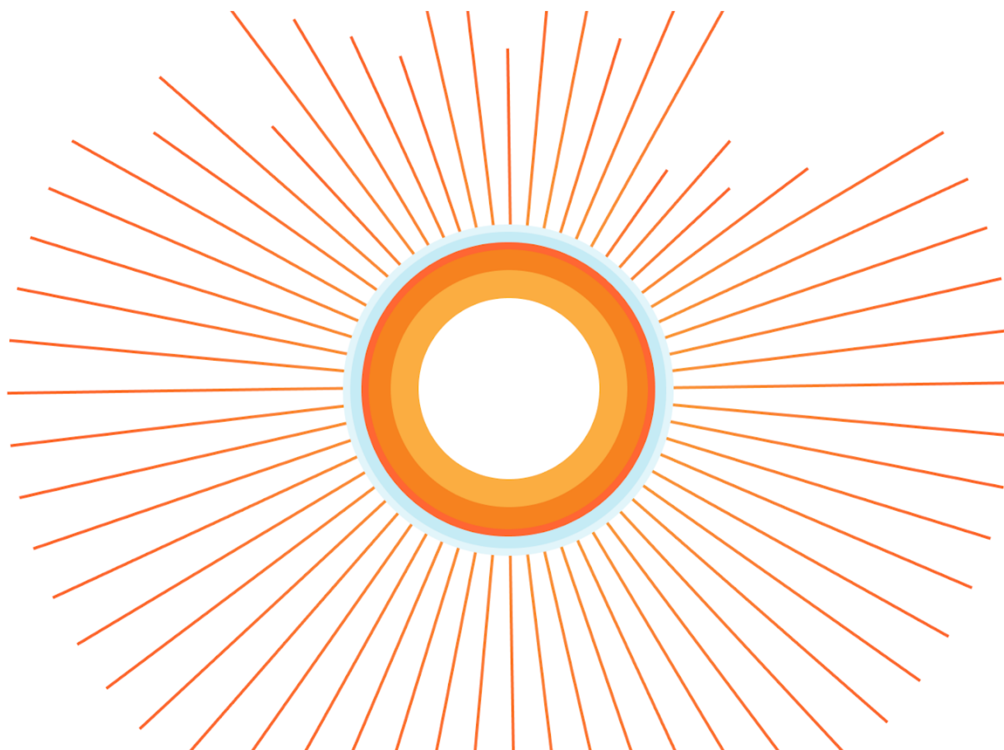
Nick Blazier December 18, 2020

Loading...

- Nick Blazier

-

  Jesse Kipp



On Wednesday, December 16, the RedDrip Team from QiAnXin Technology released their discoveries (tweet, github) regarding the random subdomains associated with the SUNBURST malware which was present in the SolarWinds Orion compromise. In studying queries performed by the malware, Cloudflare has uncovered additional details about how the Domain Generation Algorithm (DGA) encodes data and exfiltrates the compromised hostname to the command and control servers.

Background

The RedDrip team discovered that the DNS queries are created by combining the previously reverse-engineered unique guid (based on hashing of hostname and MAC address) with a payload that is a custom base 32 encoding of the hostname. The article they published includes screenshots of decompiled or reimplemented C# functions that are included in the compromised DLL. This background primer summarizes their work so far (which is published in Chinese).

RedDrip discovered that the DGA subdomain portion of the query is split into three parts:

```
<encoded_guid> + <byte> + <encoded_hostname>
```

An example malicious domain is:

```
7cbtailjomqle1pjvr2d32i2voe60ce2.appsync-api.us-east-1.avsvmcloud.com
```

Where the domain is split into the three parts as

Encoded guid (15 chars)	byte	Encoded hostname

Encoded guid (15 chars)	byte	Encoded hostname
7cbtailjomqle1p	j	vr2d32i2voe60ce2

The work from the RedDrip Team focused on the encoded hostname portion of the string, we have made additional insights related to the encoded hostname and encoded guid portions.

At a high level the encoded hostnames take one of two encoding schemes. If all of the characters in the hostname are contained in the set of domain name-safe characters `"0123456789abcdefghijklmnopqrstuvwxyz-_."` then the `OrionImprovementBusinessLayer.CryptoHelper.Base64Decode` algorithm, explained in the article, is used. If there are characters outside of that set in the hostname, then the `OrionImprovementBusinessLayer.CryptoHelper.Base64Encode` is used instead and '00' is prepended to the encoding. This allows us to simply check if the first two characters of the encoded hostname are '00' and know how the hostname is encoded.

These function names within the compromised DLL are meant to resemble the names of legitimate functions, but in fact perform the message encoding for the malware. The DLL function Base64Decode is meant to resemble the legitimate function name base64decode, but its purpose is actually to perform the encoding of the query (which is a variant of base32 encoding).

The RedDrip Team has posted Python code for encoding and decoding the queries, including identifying random characters inserted into the queries at regular character intervals.

One potential issue we encountered with their implementation is the inclusion of a check clause looking for a '0' character in the encoded hostname (line 138 of the decoding script). This line causes the decoding algorithm to ignore any encoded hostnames that do not contain a '0'. We believe this was included because '0' is the encoded value of a '0', '.', '-' or '_'. Since fully qualified hostnames are comprised of multiple parts separated by '.'s, e.g. 'example.com', it makes sense to be expecting a '.' in the unencoded hostname and therefore only consider encoded hostnames containing a '0'. However, this causes the decoder to ignore many of the recorded DGA domains.

As we explain below, we believe that long domains are split across multiple queries where the second half is much shorter and unlikely to include a '.'. For example 'www2.example.c' takes up one message, meaning that in order to transmit the entire domain 'www2.example.c' a second message containing just 'om' would also need to be sent. This second message does not contain a '.' so its encoded form does not contain a '0' and it is ignored in the RedDrip team's implementation.

The quirk: hostnames are split across multiple queries

A list of observed queries performed by the malware was published publicly on GitHub. Applying the decoding script to this set of queries, we see some queries appear to be truncated, such as `grupobazar.loca`, but also some decoded hostnames are curiously short or incomplete, such as “com”, “.com”, or a single letter, such as “m”, or “l”.

When the hostname does not fit into the available payload section of the encoded query, it is split up across multiple queries. Queries are matched up by matching the GUID section after applying a byte-by-byte exclusive-or (xor).

Analysis of first 15 characters

Noticing that long domains are split across multiple requests led us to believe that the first 16 characters encoded information to associate multipart messages. This would allow the receiver on the other end to correctly re-assemble the messages and get the entire domain. The RedDrip team identified the first 15 bytes as a GUID, we focused on those bytes and will refer to them subsequently as the header.

We found the following queries that we believed to be matches without knowing yet the correct pairings between message 1 and message 2 (payload has been altered):

Part 1 - Both decode to “www2.example.c”

```
r1q6arhpujcf6jb6qqqb0trmuhd1r0ee.appsync-api.us-west-2.avsvmcloud.com
r8stkst71ebqgj66qqqb0trmuhd1r0ee.appsync-api.us-west-2.avsvmcloud.com
```

Part 2 - Both decode to “om”

```
0oni12r13ficnkqb2h.appsync-api.us-west-2.avsvmcloud.com
ulfmcf44qd58t9e82h.appsync-api.us-west-2.avsvmcloud.com
```

This gives us a final combined payload of **www2.example.com**

This example gave us two sets of messages where we were confident the second part was associated with the first part, and allowed us to find the following relationship where message1 is the header of the first message and message2 is the header of the second:

```
Base32Decode(message1) XOR KEY = Base32Decode(message2)
```

The KEY is a single character. That character is xor’d with each byte of the Base32Decoded first header to produce the Base32Decoded second header. We do not currently know how to infer what character is used as the key, but we can still match messages together without that information. Since A XOR B = C where we know A and C but not B, we can instead use A XOR C = B. This means that in order to pair messages together we simply need to look for messages where XOR’ing them together results in a repeating character (the key).

```
Base32Decode(message1) XOR Base32Decode(message2) = KEY
```

Looking at the examples above this becomes

	Message 1	Message 2
Header	r1q6arhpujcf6jb	0oni12r13ficnkq
Base32Decode (binary)	101101000100110110111111011 010010000000011001010111111 011110001010011101000000101	110110010010000011010010000 001000110110110100111100100 001000111111100000000100

We’ve truncated the results slightly, but below shows the two binary representations and the third line shows the result of the XOR.

1011010001001101101111110110100100000000110010101111110111100010100111 01
1101100100100000110100100000010001101101101001111001000010001111111 10000
0110110101101101011011010110110101101101011011010110110101101101

We can see the XOR result is the repeating sequence ‘01101101’meaning the original key was 0x6D or ‘m’.

We provide the following python code as an implementation for matching paired messages (Note: the decoding functions are those provided by the RedDrip team):

```
# string1 is the first 15 characters of the first message
# string2 is the first 15 characters of the second message
def is_match(string1, string2):
    encoded1 = Base32Decode(string1)
    encoded2 = Base32Decode(string2)
    xor_result = [chr(ord(a) ^ ord(b)) for a,b in zip(encoded1, encoded2)]
    match_char = xor_result[0]
    for character in xor_result[0:9]:
        if character != match_char:
            return False, None
    return True, "0x{:02X}".format(ord(match_char))
```

The following are additional headers which based on the payload content Cloudflare is confident are pairs (the payload has been redacted because it contains hostname information that is not yet publicly available):

Example 1:

vrffaikp47gnsd4a

aob0ceh5l8cr6mco

xorkey: 0x4E

Example 2:

vrffaikp47gnsd4a

vrffaikp47gnsd4a

aob0ceh5l8cr6mco

xorkey: 0x54

Example 3:

vvu7884g0o86pr4a

6gpt7s654cfn4h6h

xorkey: 0x2B

We hypothesize that the xorkey can be derived from the header bytes and/or padding byte of the two messages, though we have not yet determined the relationship.

Update (12/18/2020):

Erik Hjelmvik posted a blog explaining where the xor key is located. Based on his code, we provide a python implementation for converting the header (first 16 bytes) into the decoded GUID as a string. Messages can then be paired by matching GUID's to reconstruct the full hostname.

```
def decrypt_secure_string(header):
    decoded = Base32Decode(header[0:16])
    xor_key = ord(decoded[0])
    decrypted = ["{0:02x}".format(ord(b) ^ xor_key) for b in decoded]
    return ''.join(decrypted[1:9])
```

Updated example:

	Message 1	Message 2
Header	r1q6arhpujcf6jb	0oni12r13ficnkq
Base32Decode Header (hex)	b44dbf6900cafde29d05	d920d2046da7908ff004
Base32Decode first byte (xor key)	0xb4	0xd9
XOR result (hex)	00f90bddb47e495629	00f90bddb47e495629

Cloudflare Zero Trust Cloudflare Gateway Deep Dive