

A Look into SUNBURST's DGA

By asuna amawaka

Published: 2020-12-22 · Archived: 2026-04-06 00:57:21 UTC



6 min read

Dec 20, 2020

Many fellow researchers have written very good analysis on SUNBURST malware, so I shall not do a walkthrough on reverse engineering it. But I've been intrigued by the domain generation algorithm. The RedDrip Team wrote a nice decoder [1]. Folks at NETRESEC has an improved version [2]. However, there are still some generated DGA strings that cannot be decoded. I want to know what these are! *Edit: Just moments before I click on Publish for this post, I saw that Kaspersky's analysts wrote their code in C [3] with nice writeup [4] that also handled decoding of all two types of DGA string. But hey, I got Python :)

Before I go on to explain what I did to decode these DGA strings, I would like to summarize the related functions and variable names so you won't be lost.

Victims of SUNBURST are uniquely identified with a GUID that is created within *OrionImprovementBusinessLayer.GetOrCreateUserID()*. This GUID is a 8-byte value made up of the victim machine's MAC address, MachineGUID value read from HKLM\Software\Microsoft\Cryptography\MachineGuid and the victim machine's domain name. These three information are concatenated and MD5-hashed. The MD5 value is then "cut" into 2 and XORed (where 1st byte is XORed with the 9th byte; 8th byte is XORed with 16th byte), hence we end up with a 8-byte irreversible unique identifier.

OrionImprovementBusinessLayer.Update() is the function that calls the respective functions to generate the DGA strings and responsible for handling the stuff that happens after a DNS response is received.

OrionImprovementBusinessLayer.CryptoHelper.GetStatus() returns the concatenation of ".appsync-api.<one of four below>.avsvmcloud.com".

- eu-west-1
- us-west-2
- us-east-1
- us-east-2

Four functions within *OrionImprovementBusinessLayer.CryptoHelper* are the ones that are called upon to generate the DGA strings:

GetNextStringEx(), *GetNextString()*, *GetPreviousString()* and *GetCurrentString()*.

Within them, the functions *CreateSecureString()*, *CreateString()*, *DecryptShort()*, *Base64Decode()* and *Base64Encode()* are responsible for the encoding process.

(At this point, I would like to comment on the names of the functions and variables — geez most of them are not in any way descriptive of the actual meaning of the task they perform or the value they hold. If anything, they are there to mislead the analyst.)

CreateSecureString() does nothing to create secure strings. What it really does is XOR-encode some data with a random byte as XOR key, prepend this byte to the data and then return a Base32-like encoded string of the data. The Base32-like encoding is performed by *Base64Encode()*. Look at that nasty confusing name!

The other related function, *Base64Decode()* is sort of a substitution cipher where by default the characters' are shifted by 4 to the right according to a custom alphabet. If a special character "0", ".", "-", or "_" is encountered, then a random shift value is selected. Either *Base64Decode()* and *Base64Encode()* is called within *DecryptShort()*.

CreateString() creates a 1 byte value that encodes the index of the DGA string. In the event that multiple DGA strings are required to fully represent the victims' domain, this index numbering (I call it the "chunk index", and it ranges from 0 to 35) will help the receiving end to piece back the domain. This is because the DGA string is capped at 32 bytes (excluding the fixed ".appsync-api...avsvmcloud.com"), so encoded victim's domain that is longer than 16 bytes would have to be expressed across multiple DGA strings (the first 16 bytes in each of such DGA string is taken up by victim's GUID and the chunk index). Having a chunk index of 35 means that this DGA string is the last piece.

To explain this concept of "index numbering", let's look at the output of my decoder. After decoding the victim GUID and the index, I was able to link up two DGA strings (involving abit of manual searching) that makes up one victim's domain. Chunk Index of "0" means it is the first piece, and Chunk Index of "35" means it is the last. If there had been another piece in the middle, it would be index "1".

```
3993 jga7cjdpausatposyovi0t1fj2o10kovi.appsynchronic.us-east-2.avsvmcloud.com
3994 Victim GUID    = 6370938ad5638b2
3995 Chunk Index   = 0
3996 Victim Domain = int.lukoil-int
3997 -----

5713 s1bvlrpo8tbeelo6rsvuio2vu10if7.appsynchronic.us-east-2.avsvmcloud.com
5714 Victim GUID    = 6370938ad5638b2
5715 Chunk Index   = 35
5716 Victim Domain = ernational.uz
5717 -----
```

Press enter or click to view image in full size

```

3504 // Token: 0x000090CC RID: 2548 RVA: 0x0004745C File Offset: 0x0004565C
3505 public string GetNextString(bool flag)
3506 {
3507     byte[] array = new byte[(OrionImprovementBusinessLayer.svcList.Length * 2 + 7) / 8];
3508     Array.Clear(array, 0, array.Length);
3509     for (int i = 0; i < OrionImprovementBusinessLayer.svcList.Length; i++)
3510     {
3511         int num = Convert.ToInt32(OrionImprovementBusinessLayer.svcList[i].stopped) | Convert.ToInt32(OrionImprovementBusinessLayer.svcList[i].running) << 1;
3512         byte[] array2 = array;
3513         int num2 = array.Length - 1 - i / 4;
3514         array2[num2] |= Convert.ToByte(num << i % 4 * 2);
3515     }
3516     return OrionImprovementBusinessLayer.CryptoHelper.CreateSecureString(this.UpdateBuffer(2, array, flag), false) + this.GetStatus();
3517 }
3518
3519 // Token: 0x000090CD RID: 2549 RVA: 0x000474FB File Offset: 0x000456FB
3520 public string GetNextStringEx(bool flag)
3521 {
3522     return OrionImprovementBusinessLayer.CryptoHelper.CreateSecureString(this.UpdateBuffer(1, null, flag), false) + this.GetStatus();
3523 }
3524
3525 // Token: 0x000090CE RID: 2510 RVA: 0x00047518 File Offset: 0x00045718
3526 public string GetPreviousString(out bool last)
3527 {
3528     string text = OrionImprovementBusinessLayer.CryptoHelper.CreateSecureString(this.guid, true);
3529     int num = 32 - text.Length - 1;
3530     string result = "";
3531     last = false;
3532     if (this.offset >= this.dnStr.Length || this.mCount > 36)
3533     {
3534         return result;
3535     }
3536     int num2 = Math.Min(num, this.dnStr.Length - this.offset);
3537     this.dnStrLower = this.dnStr.Substring(this.offset, num2);
3538     this.offset += num2;
3539     if (OrionImprovementBusinessLayer.ZipHelper.Unzip("0403AAA=").Contains(this.dnStrLower[this.dnStrLower.Length - 1]))
3540     {
3541         if (num2 == num)
3542         {
3543             this.offset--;
3544             this.dnStrLower = this.dnStrLower.Remove(this.dnStrLower.Length - 1);
3545         }
3546         this.dnStrLower += "0";
3547     }
3548     if (this.offset >= this.dnStr.Length || this.mCount > 36)
3549     {
3550         this.mCount = -1;
3551     }
3552     result = text + OrionImprovementBusinessLayer.CryptoHelper.CreateString(this.mCount, text[0]) + this.dnStrLower + this.GetStatus();
3553     if (this.mCount >= 0)
3554     {
3555         this.mCount++;
3556     }
3557     last = (this.mCount < 0);
3558     return result;
3559 }
3560
3561 // Token: 0x000090CF RID: 2511 RVA: 0x00047600 File Offset: 0x00045800
3562 public string GetCurrentString()
3563 {
3564     string text = OrionImprovementBusinessLayer.CryptoHelper.CreateSecureString(this.guid, true);
3565     return text + OrionImprovementBusinessLayer.CryptoHelper.CreateString((this.mCount > 0) ? (this.mCount - 1) : this.mCount, text[0]) + this.dnStrLower + this.GetStatus();
3566 }

```

What made decoding the victims’ domain possible is because the malware generated the DGA string via GetPreviousString() and GetCurrentString(). The victims’ domain is encoded and included in the DGA string through the variables dnStr and dnStrLower. The decoders out there would try to reverse the DGA string to decode dnStrLower to retrieve the victim’s domain.

Get asuna amawaka’s stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

This is how the DGA string would look like. I call this “Type 1” DGA string.

Press enter or click to view image in full size



Notice that in GetNextString() and GetNextStringEx(), dnStr and dnStrLower are not used. Instead, another function UpdateBuffer() is called.

Press enter or click to view image in full size

```

3474
3475 // Token: 0x060009CA RID: 2586 RVA: 0x00047338 File Offset: 0x00045538
3476 private static int GetStringHash(bool flag)
3477 {
3478     return ((int)((DateTime.UtcNow - new DateTime(2010, 1, 1, 0, 0, 0, DateTimeKind.Utc)).TotalMinutes / 30.0) & 524287) << 1 | (flag ? 1 : 0);
3479 }
3480
3481 // Token: 0x060009CB RID: 2587 RVA: 0x00047384 File Offset: 0x00045584
3482 private byte[] UpdateBuffer(int sz, byte[] data, bool flag)
3483 {
3484     byte[] array = new byte[this.guid.Length + ((data != null) ? data.Length : 0) + 3];
3485     Array.Clear(array, 0, array.Length);
3486     Array.Copy(this.guid, array, this.guid.Length);
3487     int stringHash = OrionImprovementBusinessLayer.CryptoHelper.GetStringHash(flag);
3488     array[this.guid.Length] = (byte)((stringHash & 983040) >> 16 | (sz & 15) << 4);
3489     array[this.guid.Length + 1] = (byte)((stringHash & 65280) >> 8);
3490     array[this.guid.Length + 2] = (byte)(stringHash & 255);
3491     if (data != null)
3492     {
3493         Array.Copy(data, 0, array, array.Length - data.Length, data.Length);
3494     }
3495     for (int i = 0; i < this.guid.Length; i++)
3496     {
3497         byte[] array2 = array;
3498         int num = i;
3499         array2[num] ^= array[this.guid.Length + 2 - i % 2];
3500     }
3501     return array;
3502 }

```

As such, a different kind of DGA string is generated. I call this “Type 2” DGA string.

Press enter or click to view image in full size

20 or 23 bytes encoded data made up of
(8 bytes XOR-encoded GUID + 3 bytes timestamp + optional 2 bytes info on # of security tools' processes) .appsync-api.us-west-2.avsmcloud.com

Within UpdateBuffer(), a 3-bytes time value is calculated through GetStringHash(). The last two bytes of this time value is going to be used as the XOR key to encode the 8-bytes victim GUID. UpdateBuffer() returns a 11-bytes value made up of 8-bytes encoded GUID and 3-bytes time/XOR key. If data is provided to the function (data that describes the number of security tools' processes present somehow), then UpdateBuffer() returns a 13-bytes value, with the additional 2 bytes appended behind the time/XOR key.

The value is then encoded through CreateSecureString(), which applies the Base32-like encoding. The ending DGA value is a 20 or 23 bytes string. Interesting, I can use this as a condition to identify this form of DGA string.

Alright, stop talking. Let's decode!

I made 2 assumptions to try to differentiate between the two types DGA strings.

- If the decoded chunk index is 0, the length of the DGA string cannot be less than 32 bytes (which should not be happening, because if data is “overflowing” into another DGA string, then the first should be filled up to the max length). Else, it could be a Type 2 DGA string.
- If chunk index is successfully decoded to 35, then it is a Type 1 DGA string.(what are the chances of getting this exact value using the 16th byte and the 1st byte? I think low enough for this assumption to work.)

Along with the expected length for Type 2 DGA strings, I'm able to come up with the following if-else checks:

```
# the idea: if chunk_index == 0, it means that the domain name is truncated
# and this is the first "portion" of the truncated data.
# then the length of the DGA string must be 32 bytes (anything less won't have caused truncation)
# if chunk_index == 0 but total length of DGA string != 32,
# means can try doing the other type of decoding (the one with timestamp XOR)

if ((chunk_index == 0 and len(data) != 32) or (chunk_index != 35) and (len(data) == 20 or len(data) == 23)):
    guid = getVictimGUID_fromDGA_type2(data)
    print line.rstrip()
    print "Victim GUID = {}".format(guid)
else:
    guid = getVictimGUID_fromDGA_type1(encoded_guid)
    if encoded_domain[0] == "0" and encoded_domain[1] == "0":
        encoded_domain = encoded_domain[2:]
        domain = reverse_Base64Encode(encoded_domain)
    else:
        domain = reverse_Base64Decode(encoded_domain)
    print line.rstrip()
    print "Victim GUID = {}".format(guid)
    print "Chunk Index = {}".format(chunk_index)
    print "Victim Domain = {}".format(domain)
```

With the decoded Victim GUIDs from Type 1 and Type 2 DGA strings, we can identify the related DNS queries, and how many different machines are infected within the same domain.

Let's see some examples. I worked with data from here:

<https://github.com/bambenek/research/blob/main/sunburst/uniq-hostnames.txt>

```
128 0l7bgt0lhmk4pmrcuhs0ee2sd0eovir1.appsync-api.us-east-1.avsvmcloud.com
129 Victim GUID = ee52cf7f78fb90b7
130 Chunk Index = 0
131 Victim Domain = amr.corp.intel
132 -----
133 0lgjmdgj8qk63ldcuhs0ce2sd0govir1.appsync-api.us-east-1.avsvmcloud.com
134 Victim GUID = aae8eb8eaf6fbe6a1
135 Chunk Index = 0
136 Victim Domain = amr.corp.intel

1173 5f19k0dd4qlv4ibh5onrloipe2hh0e12.appsync-api.us-west-2.avsvmcloud.com
1174 Victim GUID = 59219a2ddb5b8320
1175 Chunk Index = 0
1176 Victim Domain = fidelitycomm.lo

3892 j5vautirmsvcpbjhgjie72mg.appsync-api.us-west-2.avsvmcloud.com
3893 Victim GUID = 59219a2ddb5b8320
```

If anyone has a list of all such DNS queries from within their network, I guess it would also be helpful to be able to decode Type 2 DGA strings to identify all the different affected machines.

Here's the link to the script on Github:

<https://github.com/asuna-amawaka/SUNBURST-Analysis>

If anyone is keen to discuss (or point out where I can improve in the code), DM me on Twitter!

References:

[1] https://github.com/RedDrip7/SunBurst_DGA_Decode

[2] <https://www.netresec.com/?page=Blog&month=2020-12&post=Reassembling-Victim-Domain-Fragments-from-SUNBURST-DNS>

[3] https://github.com/2igosha/sunburst_dga

[4] <https://securelist.com/sunburst-connecting-the-dots-in-the-dns-requests/99862/>

Source: <https://medium.com/insomniacs/a-look-into-sunbursts-dga-ba4029193947>