

# Gozi V3 Technical Update

[fidelissecurity.com/threatgeek/threat-intelligence/gozi-v3-technical-update/](https://fidelissecurity.com/threatgeek/threat-intelligence/gozi-v3-technical-update/)

May 17, 2018

## Author

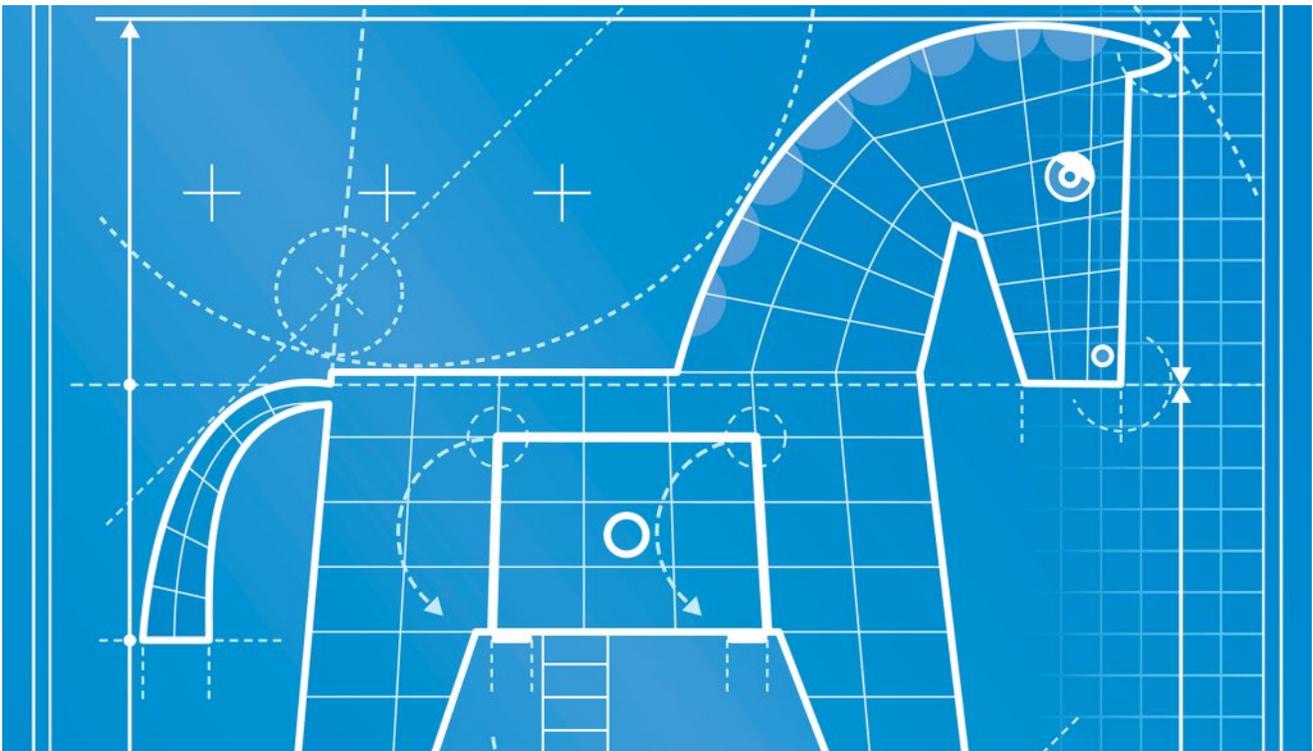


Threat Research Team

The Fidelis Threat Research team is comprised of expert security researchers whose sole focus is generating accurate and actionable intelligence to better secure customers. Together, they represent over... [Read More](#)

## Comments

May 17, 2018



In 2017 Gozi was updated[1] to include protections of the onboard configuration known as INI PARAMS[3]. That update was likely in response to an excellent article written by @maciekkotowicz[2], or possibly because infection rates had dropped due to increased

coverage through various IOC extraction programs[4,7,8]. This post aims to fill any technical gaps related to the changes in this new evolution as compared to previous versions to show the similarities and differences between this new version and the previous one.

Previous major versions of Gozi include Dreambot[9] or the addition of P2P[9] mechanisms and IAP[2], which is an evolution of ISFB where serpent encryption was added and the panel was changed. These distinctions are important because while older ISFB code versions were leaked, these other code bases are not so widely spread.

### **Key findings of this report:**

1. Bot DLL changes in how it's protected and stored in the loader
2. Onboard configuration changes in how the Bott DLL is protected and stored
3. Changes in joiner elements stored in the binary
4. Bot DLL now can come chopped up with a missing DOS header

Historically Gozi can be broken down into two major components; the loader portion and the DLL. Some actors have reused the DLL part since it was leaked with the ISFB leak in order to add a banking trojan module for added functionality(GOZNYM)[10].

Some of this usage as a module has caused quite a bit of confusion with naming, which in my mind just makes me think we should name distinct parts of malware and not just the entire package. More in-depth naming doesn't seem to happen until something is added, removed, or spun off, and then researchers are left to perform historical analysis and time consuming mapping of genealogies[2] and even then, sometimes get it wrong. For the purpose of this paper, however, we'll be using naming based on recovered panel code and major version changes since the ISFB leak along with historical analysis already conducted[2].

### **Gozi Loader**

---

As per the previous versions the loader still decodes it's bss section where it keeps all the strings that it will use.

```

mov     esi, [edi+10h]
add     esi, [edi+0Ch]
mov     eax, [ebx+3Ch]
xor     esi, [eax+ebx+8]
push   10h           ; size_t
lea     eax, [ebp+var_20]
push   offset bss_section_1004000 ; void *
push   eax           ; void *
xor     esi, 40082209h
call   memcpy
add     esp, 0Ch
push   esi
push   5
lea     edx, [ebp+var_20]
call   sub_1001418
lea     eax, [ebp+var_20]
mov     [ebp+var_4], eax
mov     [ebp+var_8], 5
xor     eax, eax

```

```

loc_1001003:
mov     ecx, [ebp+var_8]
mov     edx, [ebp+var_4]
mov     edx, [edx]
add     [ebp+var_4], 4
and     ecx, 1
shl     ecx, 3
shl     edx, cl
add     eax, edx
dec     [ebp+var_8]
jnz     short loc_1001003

```

Figure 1 BSS decode

Most of the important data is still stored using the same Joiner code from the ISFB code on github[3], however instead of having all the data with an ADDON\_MAGIC stored in code caves, the data is instead stored as a table with a single 2 byte ADDON\_MAGIC value serving as a way to locate it.

```

mov     ebp, esp
sub     esp, 14h
mov     eax, dword_1003014
push   esi
push   edi
xor     eax, 85EDFA66h
push   eax
lea     eax, [ebp+var_14]
push   eax
lea     eax, [ebp+var_10]
xor     edi, edi
push   eax
mov     esi, ecx
mov     [ebp+var_8], edi
call   GetJoinerData_1001608
test   eax, eax
jz     loc_1001858

```

Figure 2 GetJoinerData

The addon descriptor table has changed slightly and the relevant flags are part of the XOR value in the table. The only relevant flag currently used is relating to whether or not the data is compressed. In the event the data is compressed, it is decompressed using APLIB – if the data is not compressed, it is copied over.



Figure 3 Xor Table and Compression Check

The loader is now based on an IAP variant and now comes with an onboard mangled DLL, the DLL is reconstructed using tables of offsets tacked on top as you can see below:

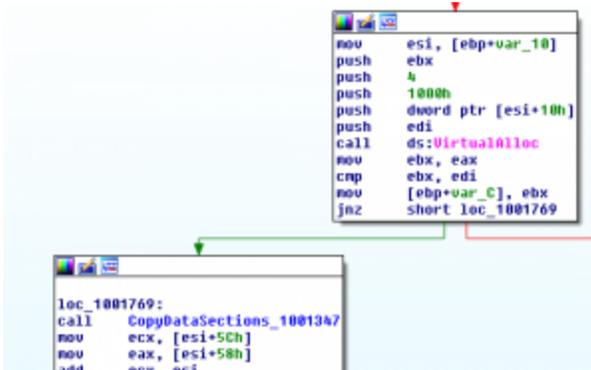


Figure 4 Reconstruct DLL Overview

After being reconstructed and having its imports fixed, you are left with a memory mapped DLL at the magic bytes PE but with the PE already stripped out. Fixing the code for static analysis involves either reconstructing the missing data – basically everything before the NT headers, or letting the malware load everything into memory and then dumping it. A walkthrough of the reconstruction process can be seen later in this write-up.

Most of the functions for this version are resolved manually, you can let the malware resolve its own dependencies and then use a script to auto rename the functions in the malware, or use any of a number of scripts available to rebuild the IAT from a dump[5].

## Gozi DLL

The DLL is similar to previous versions. It has an onboard public key, a wordlist that it will use to generate pseudo random strings and INI parameters. Also it comes with onboard algorithms used by previous versions, APLib(ISFB), Serpent CBC(IAP) and custom RSA encrypt/decrypt(ISFB).

```

push    eax
push    [ebp+arg_0]
call    GET_Joiner_26AA9
test    eax, eax
jnz     short loc_2A25F

loc_2A25F:
mov     eax, ds:CS_COOKIE_2F508
xor     eax, 9A8FF5C3h
push    eax
push    ebx
lea     eax, [ebp+var_8]
push    eax
lea     eax, [ebp+var_4]
push    eax
push    [ebp+arg_0]
call    GET_Joiner_26AA9 ; word list
test    eax, eax
jz      short loc_2A22B

mov     eax, [ebp+var_4]
mov     ecx, [ebp+var_8]
mov     [ecx+eax-1], dl
call    ParseStringIntoWordTokens_22B88
mov     esi, eax

```

Figure 5 Parse onboard word list

The INI parameters are now protected a little more as compared to previous versions, the bot takes the last 128 bytes of data and then uses the ISFB routine RSAPublicDecrypt[6] to decrypt this block of data and parse out the encrypted data it wants to use.

```

push    eax
push    [ebp+arg_0]
call    GET_Joiner_26AA9
test    eax, eax
jz      loc_23181

push    [ebp+var_8] ; length
lea     edi, [ebp+var_8]
push    [ebp+var_4] ; data
lea     ebx, [ebp+var_C]
call    sub_27E37
mov     edi, eax
test    edi, edi

```

Figure 6 Get joiner section and decode

```

mov     eax, [ebp+arg_0]
lea     eax, [eax+edi-80h]
push    eax
lea     eax, [esp+6Ch+var_54]
push    eax
lea     eax, [esp+70h+var_40]
push    eax
mov     eax, [ebp+arg_8]
call    RsaPublicDecrypt_245E9
test    eax, eax
jnz     loc_2178E

```

Figure 7 RSAPublicDecrypt from ISFB

In this case, the data that is parsed out ends up being the Serpent key to decrypt the data itself.

```

loc_2173F:
push    0
lea     eax, [esp+6Ch+var_30]
push    eax
lea     eax, [esp+70h+var_58]
push    eax
lea     eax, [esp+74h+var_5C]
push    eax
push    [ebp+arg_0]
mov     eax, edi
call    SerpentCryptDecrypt_2102D
test    eax, eax
jnz     short loc_21776

```

Figure 8 Rest of data Serpent decrypted

To do this in python we encrypt the data with the RSA public key which decrypts out the data we need. After skipping 16 bytes the bot takes the next 16 bytes and uses this as a Serpent key which is then used to decrypt the INI parameters in CBC mode with a NULLed IV – similar to how it would previously encode its URI string (python example code can be seen in the appendix [A 1]). In order to utilize the RSA public key however we need to do a bit of conversion work and decompress it if the flag is set [A 2].

## Reconstructing the mangled DLL

When reconstructing the DLL, we find that it gets APLib decompressed with another magic two bytes on top 'PX'.

```

Offset (h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 50 58 00 00 E1 6B 3A 64 25 F1 00 00 65 0B 00 00  PX..skidv..e...
00000010 00 20 01 00 C0 01 00 00 CC D3 00 00 B4 00 00 00  ..k...i0...
00000020 CC 00 00 00 50 E1 00 00 41 00 00 00 F0 03 00 00  I..P..A...e...
00000030 00 00 00 00 70 02 00 00 80 01 00 00 00 18 01 00  .b.p...e.....
00000040 8C 01 00 00 19 09 00 00 00 00 00 00 00 00 00 00  x...t.....
00000050 00 00 00 00 00 10 01 00 A8 05 00 00 31 04 00 00  .....l.....
00000060 4C 01 05 00 18 16 00 00 10 00 00 00 C0 00 00 00  L...s.....k...
00000070 25 00 00 00 00 B4 00 00 20 00 00 60 00 00 00 00  b.....'..b...
00000080 00 20 00 00 25 C1 00 00 12 00 00 40 00 00 40 00  .s.k.....@..@
00000090 00 F0 00 00 10 00 00 25 D3 00 00 00 06 00 00 00  .e.....b0....
000000A0 40 00 00 C0 00 00 01 00 00 10 00 25 D9 00 00 00  B..k.....@0..
000000B0 00 08 00 00 40 00 00 C0 00 10 01 00 00 10 00 00  ....B..k.....
000000C0 25 E1 00 00 10 00 00 40 00 00 40 54 D6 00 00 00  %a.....e..8T0..
000000D0 00 00 00 00 00 00 00 18 D7 00 00 14 D1 00 00 00  .....*...00...
000000E0 E8 D5 00 00 00 00 00 00 00 30 D7 00 00 00 00 00  #0.....0+...
000000F0 68 D1 00 00 BC D4 00 00 00 00 00 00 00 00 00 00  hR..40.....
00000100 0A D9 00 00 3C D0 00 00 08 D6 00 00 00 00 00 00  .u..<b...o...
00000110 00 00 00 00 34 D9 00 00 88 D1 00 00 80 D4 00 00 00  ...40..R..e0..
00000120 00 00 00 00 00 00 00 88 D9 00 00 00 00 00 00 00 00  .....u..b...
00000130 20 D6 00 00 00 00 00 00 00 00 00 00 00 F8 DC 00 00  .o.....e0...
00000140 AD D1 00 00 DC D5 00 00 00 00 00 00 00 00 00 00 00  R..u0.....
00000150 98 E0 00 00 5C D1 00 00 E8 D6 00 00 00 00 00 00 00  ~a..N..e0....
00000160 00 00 00 00 BC E0 00 00 68 D2 00 00 00 00 00 00 00 00  ...Na..h0....
00000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....

```

Figure 9 Mangled DLL

Taking another look at the copy screenshots above we can see that the 5<sup>th</sup> dword in will be the total memory section to be allocated:

```

mov     esi, [ebp+var_10]
push   ebx
push   4
push   1000h
push   dword ptr [esi+10h]
push   edi
call   ds:VirtualAlloc
mov     ebx, eax
cmp    ebx, edi
mov    [ebp+var_C], ebx
jnz    short loc_1001769

loc_1001769:
call   CopyDataSections_1001347
mov    ecx, [esi+5Ch]
mov    eax, [esi+58h]
add    ecx, esi

```

Figure 10 DLL Reconstruction memory allocation

From there execution is handed off to a routine that will be responsible for parsing the headers of the mangled DLL data to properly map it into memory. This routine uses the word value at offset 0x62 to perform a loop involving a call to copy data into our newly allocated section:

```

push   ecx
and    [ebp+var_8], 0
cmp    word ptr [esi+62h], 0
push   edi
jbe    short loc_10013A3

lea    edi, [esi+6Ch]

loc_1001358:
mov    eax, [edi+8]
push   eax
mov    [ebp+var_4], eax
mov    eax, [edi+4]
add    eax, esi
push   eax
mov    eax, [edi-4]
add    eax, ebx
push   eax
call   memcpy
mov    eax, [edi]
add    esp, 0Ch
cmp    eax, [ebp+var_4]
jbe    short loc_1001394

```

Figure 11 DLL Reconstruct – Section Copy Loop

The word value at offset 0x62 then is our number of sections we will be mapping into memory, from there the following code is given a pointer to offset 0x6c – where it begins copying data based on what it reads at offset -4, 0, +4 and +8. So, a list of structures starts at 0x68 offset and the length of list is at offset 0x62. Since the data is immediately passed to memcpy it makes it easier to parse the meaning of the values:

```

struct section {
    int to_offset;
    int final_length;
    int from_offset;
    int length;
}

```

Figure 12 DLL Reconstruct – section structure

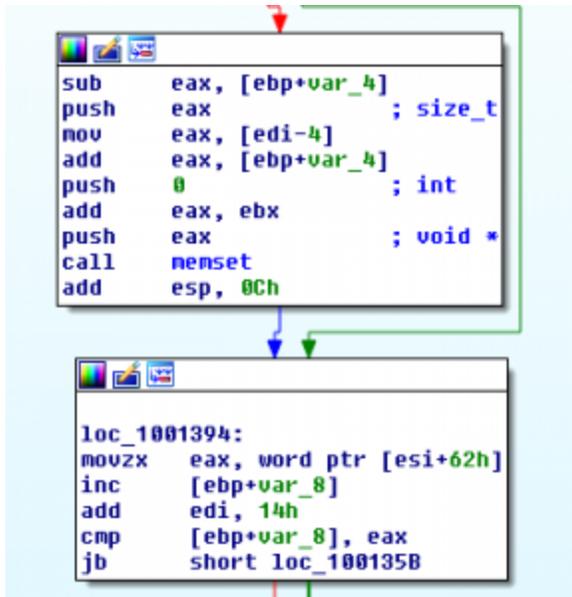


Figure 13 DLL Reconstruct – Next section

To get to the structure 0x14 is added to the pointer meaning that each structure in the list takes up 20 bytes. Reconstruction can be seen a little easier through python pseudocode:

```
(dc,dc,dc,dc,sz) = struct.unpack_from('<IIIII', data)
ret_out = 'x00'*sz
num_secs = struct.unpack_from('<H', data[0x62:])[0]
temp = data[0x6c-4:]
for i in range(num_secs):
    (to_off,final_l,from_off,l) = struct.unpack_from('<IIII', temp)
    ret_out =
ret_out[:to_off]+data[from_off:from_off+l]+ret_out[to_off+l:]
    temp = temp[0x14:]
return ret_out
```

Figure 14 DLL Reconstruct – Python code example

After being reconstructed you are left with a DLL that has been mapped into memory at the start of the IMAGE\_NT\_HEADERS but with the “PE” wiped out.

Oddly enough the dreambot version for v3 does not come with a mangled DLL but instead is APLib compressed -> structified -> serpent CBC encrypted. The serpent key is hidden at the end, similar to the INI parameters, as previously explained by using RSAPublicDecrypt.

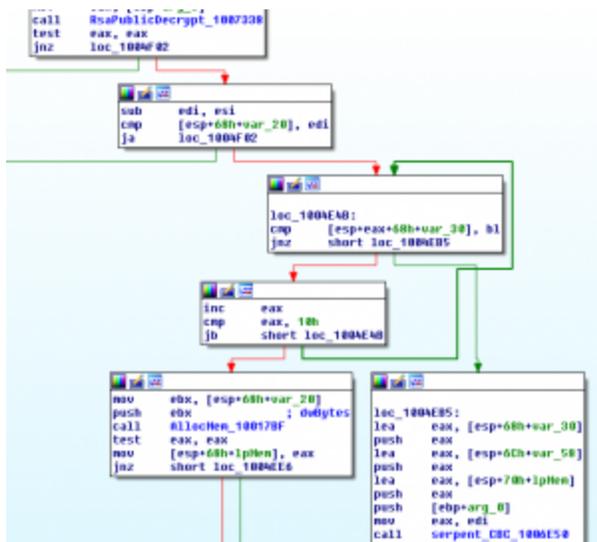


Figure 15 RSAPublicDecrypt followed by serpent decrypt

Within the DLL whether decompressed or reconstructed we can find the INI parameters that are most interesting to people as it's where the C2 information is stored. From the previous version just add an RSAPublicDecrypt, parse out the serpent key and then use serpent CBC to decrypt the data.

## Conclusion

There have been a number of smaller and less talked about versions pop up aside from this one, so what makes this one special? It's very common for malware authors to reuse proven code libraries and code bases to either enhance their own malware or to create a variant of an older version. So what makes this v3? The answer is that it's code and obfuscation that appears to be expanding upon the last major version outlined within the community. Whether or not that is the case or if that code base was packaged up and sold off remains to be seen.

A number of other versions of this malware family have popped up over the years where people have performed slight modifications, for example changing the ADDON\_MAGIC in the ISFB code base. These sorts of one off versions have also popped up in the versions after IAP – which, as you might recall from the introduction, has a code base that is not as readily available as ISFB. So whatever version this one wants to be is fine but at the end of the day it's a new variant of Gozi and hopefully this paper has helped explain how it fits into the family.

## IOCs:

V3:

1d8a0f9c987bf0332fbb3d41b002c0d379c38564ceeae402c0a0681ecb93be1  
 92e0f1754394b5a19595c7c5ce03c0d29be1f0e28b5e9c9c61bde2918572f31a  
 2d2e4985cc102109505c1a69d24ead1664adfe3ba382fc330ba73771d64cd924

## One offs:

63813e71ffad159f8d8a1e54fc1bc256a7592406ffd7fb4e11a538cfd7ae7932 – “J1” magic val  
134463122c569995795bc0857f70f1dcaa572a599bb4fed6c22692df6c94e869 – “J1” magic  
val  
48e9227077ba672530c0c55867b8380b9155f026f65cc74bf4cfe5a7b1f539f7 – “JJ” magic val  
with different order of section length/offset + custom loader DLL parsing with missing MZ and  
PE and an abnormal INI params parsing.

## **References:**

Appendix A 1 Python RSADecrypt and SerpentDecrypt

Appendix A 2 Python convert RSA public key