Gozi V3 Technical Update

fidelissecurity.com/threatgeek/threat-intelligence/gozi-v3-technical-update/

May 17, 2018

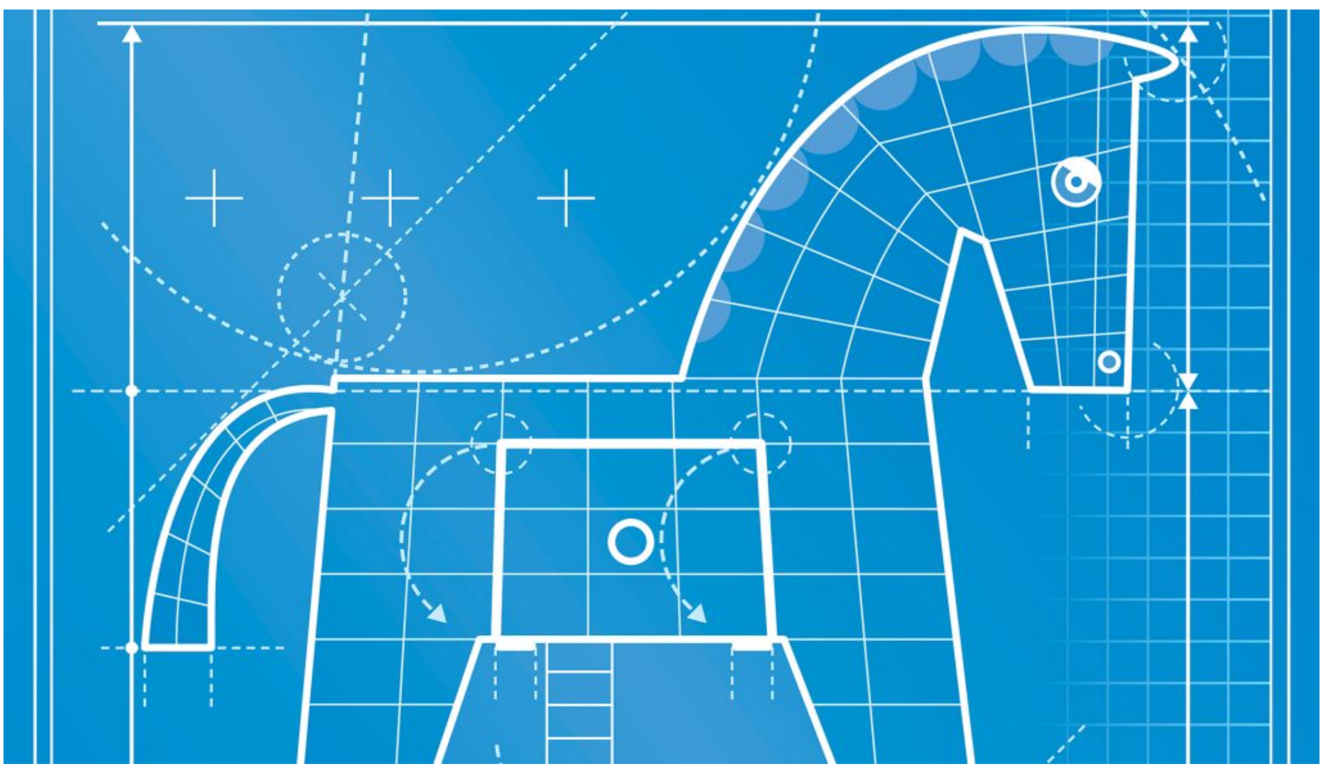
Author

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.

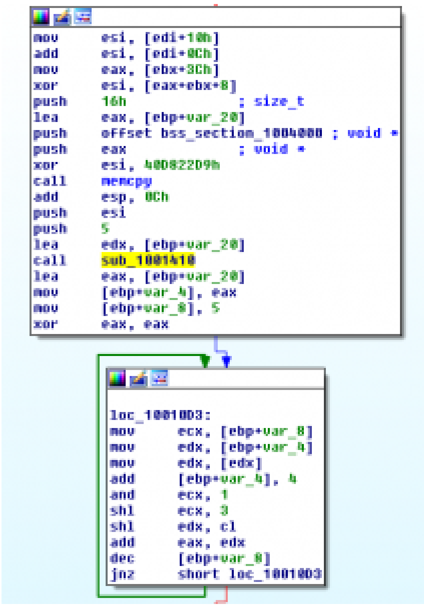


Figure 1 BSS decode

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.

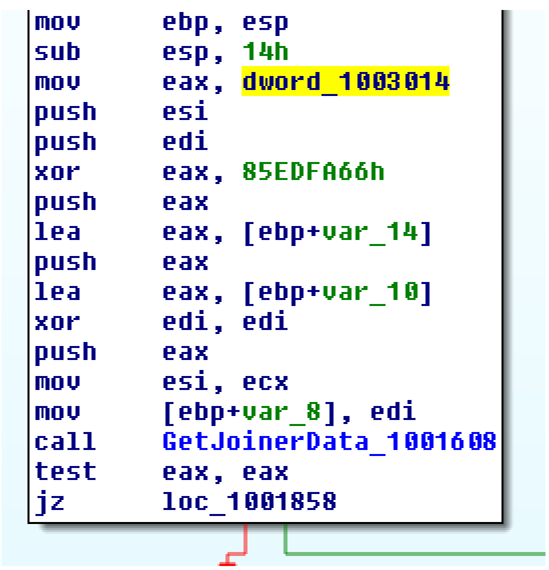


Figure 2 GetJoinerData

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.

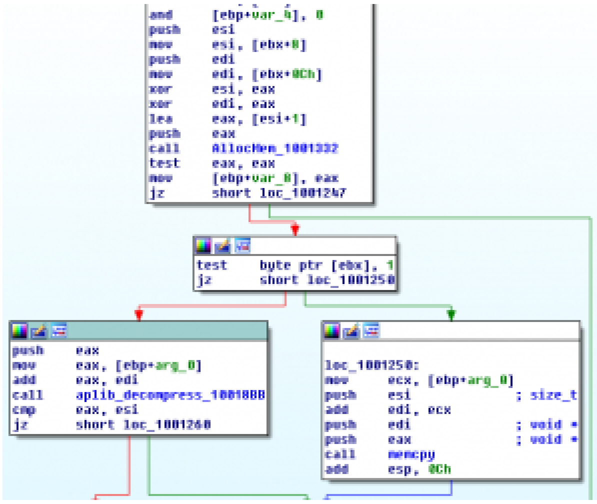


Figure 3 Xor Table and Compression Check

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:

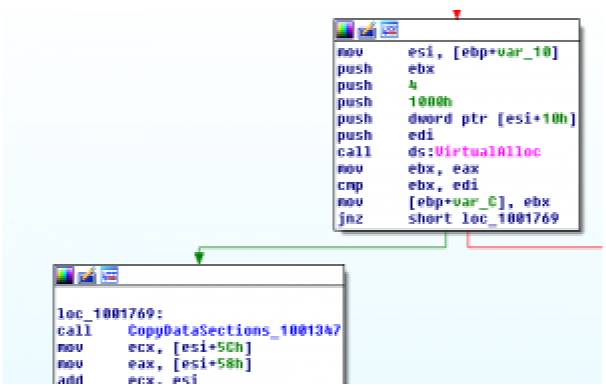


Figure 4 Reconstruct DLL Overview

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).

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.

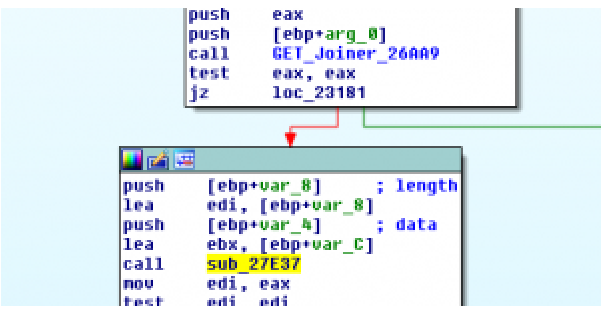


Figure 6 Get joiner section and decode

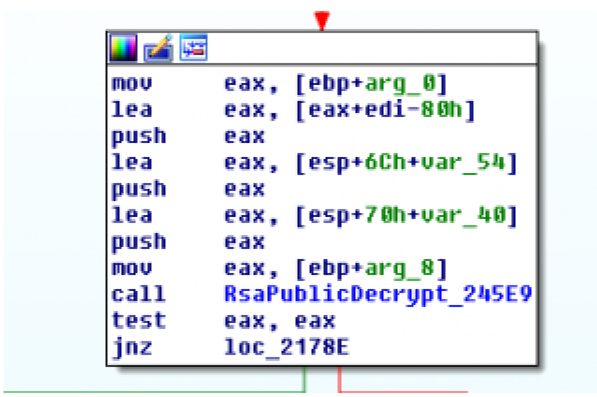


Figure 7 RSAPublicDecrypt from ISFB

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

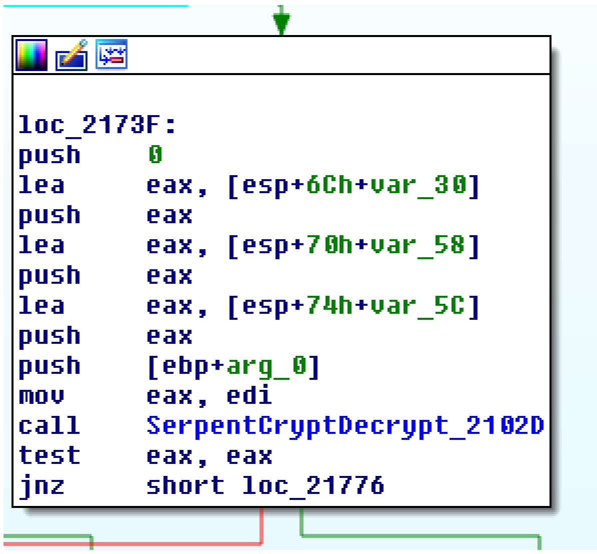


Figure 8 Rest of data Serpent decrypted

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'.

Figure 9 Mangled DLL

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

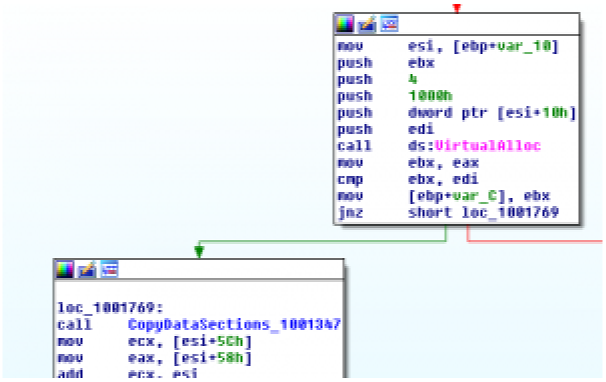


Figure 10 DLL Reconstruction memory allocation

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:

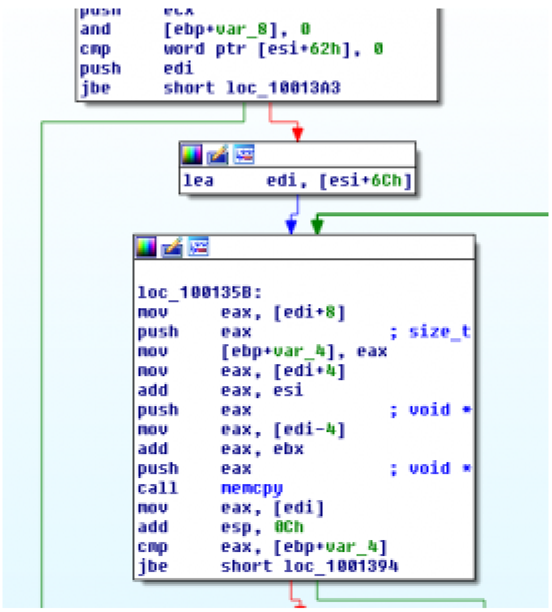


Figure 11 DLL Reconstruct – Section Copy Loop

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

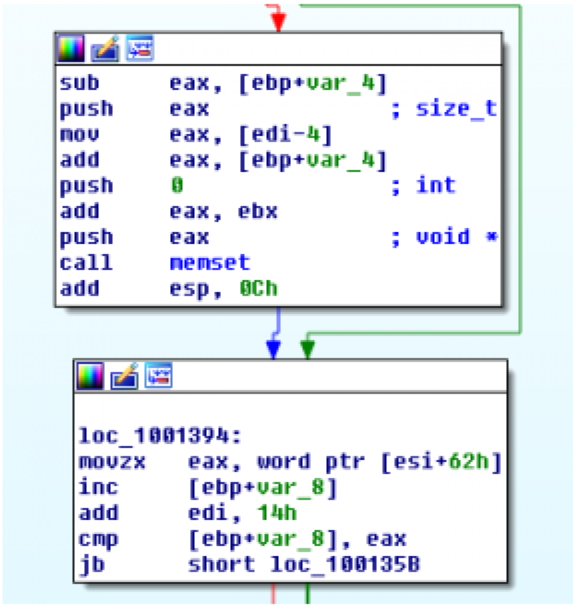


Figure 13 DLL Reconstruct – Next section

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.

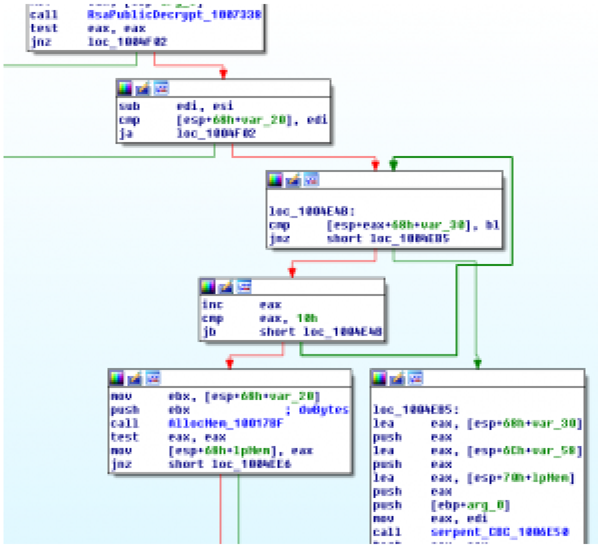


Figure 15 RSAPublicDecrypt followed by serpent decrypt

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:
1d8a0f9c987bf0332fbb3d41b002c0d379c38564ceeaee402c0a0681ecb93be1
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