Old dog, with new tricks - ISFB v3 loader

Maciej Kotowicz

Introduction

Some time ago I released an extensive paper on ISFB[1], a modular malware that is notoriously used by cyber-criminals all over the world to steal money. ISFB is an offspring of the legendary Gozi. This infamous malware family is with us already for over 10 years, having it ups and downs but still being actively developed and forked by various criminal groups - especially this year it seems like a go to tool for some of the most notorious gangs.

Loader

When it comes to binary file formats, the Portable Executable (PE) format is not very complicated one. This means, writing a custom loader is quite simple and many malware families are exploiting this. However, not many of them employ their own proprietary data structures resembling a file format to store a decoupled binary. While a PE file stores a lot of information, only some of it is needed to actually successfully execute the binary. One can split a PE file into a few basic blocks

- Header represented by `IMAGE_NT_HEADERS`
- Sections represented by an array of `IMAGE_SECTION_HEADER`s
- Extra Data represented by an array of `IMAGE_DATA_DIRECTORY`s

Those are the most important pieces of an ordinary PE file, which contain information on which bytes should be loaded where and how memory in a newly spawned process should be protected. Other relevant information that is contained in those headers are for example related to resolving external symbols (e.g. to the Windows API). Of course those parts can be further decoupled and mixed but for this report we will focus on how the authors/maintainers of ISFB approached this challenge.

Rebuilding an Image

Let's tackle the main header first. After some reversing we can describe the main header as the following C-structure:

```
struct __declspec(align(4)) LDR::header
{
  DWORD exports[3];
  DWORD NtHeadersOffset;
  DWORD TotalSize;
  DWORD HeadersSize;
  WORD gap18;
  WORD field_1A;
  DWORD ImportsSize;
  DWORD ImportsOffset;
  DWORD field_24;
  DWORD field_28;
  DWORD ExportOffset;
  LDR::data_entry DataDirectory[3];
  DWORD field_54;
  DWORD RelocsSize;
  DWORD RelocsOffset;
  WORD field_60;
  unsigned __int16 NumberOfSections;
  DWORD EntryPoint
};
```

As we can see it is not a very complex structure and fortunately most of data of an original PE file is preserved inside the binary as a blobs that are copied into their proper locations.

After this header follows the table containing a modified `IMAGE_SECTION_HEADER` that looks like this:

```
struct LDR::section
{
 DWORD VirtualAddress;
 DWORD VirtualSize;
 DWORD OffsetData;
 DWORD SizeOfRawData;
 DWORD Characteristics;
}
```

This structure is totally redundant because the original section headers are already included in data pointed to by `NtHeadersOffset`. On the other hand, they can be very useful to us because we can use this information to re-map sections from a given BLOB into proper offsets in the file like this:

```
    for i in range(hdr.NumberOfSections):
        s = m.read_struct(LDR_section)
        st = pe.get_section_by_rva(s.VirtualAddress)
        out.seek(st.PointerToRawData,os.SEEK_SET)
        out.write(m.read_at(s.DataOffset,st.SizeOfRawData))
```

After this step, we are almost done with rebuilding a whole legitimate PE file. What is left is to map all remaining data structures into their proper location, for example the Import Address Table (IAT):

```
    ## imports
    imp_rva = pe.OPTIONAL_HEADER.DATA_DIRECTORY[IMP_IDX].VirtualAddress
    imp_off = pe.get_offset_from_rva(imp_rva)
    out.seek(imp_off,os.SEEK_SET)
    out.write(m.read_at(hdr.ImportsOffset,hdr.ImportsSize))
```

Lastly, don't forget about those data directories from the PE header. Those are very important and in next section we will explain why.

Static Configuration

For as long as we remember ISFB had their static configuration stored somewhere in the last section with a pointer to it hidden just after the section table. This has changed! THe authors of version 3 decided that it is about time to make some significant changes and they moved the pointer to the static configuration into the data directory. To be more precise, the abuse the field designated for `ENTRY_SECURITY`.

Why is this important? Because this data directory is one of the `LDR::directory` fields you can see in the `LDR::header`! This discovery makes our life much easier since we don't have to rebuild a whole binary every time. In order to extract the static configuration data, we only need to get the second data directory and sections headers.[1] Using Python, this can be achieved like this:

```
def make_memory(data):
    m = ISFBv3M(data)
    hdr = m.read_struct(LDR_header)
    doff  = hdr.DataDirectory[1].Offset
    dsize = hdr.DataDirectory[1].Size
    sections = []
    for i in range(hdr.NumberOfSections):
        s = m.read_struct(LDR_section)
        sections.append(s)
    m.load_rsrc(m.read(doff,dsize))
    m.load_sections(sections)
    return m
```

[1] we need to map rvas from config into file offsets

There are a couple of more cosmetic changes done in v3 regarding the static configuration:

- no more `FJ` or similar tell-tale constants in the binary, the real magic-tags are obfuscated
- a simplified and obfuscated joined resource header
- simplified flags around this joined resource - now the only flag is either 0 or 1 depending on whether the item is compressed or not

Lets take a quick look at how the new joined resource header is constructed, since this is the only relevant change.

In previous versions the joined resource header looked like this

```
struct CONFIG::Item::Hdr
{
  WORD  magic; // FJ, J1, JJ etc
    WORD  skip;
  DWORD addr;
    DWORD size;
    DWORD crc32_name;
    DWORD flags;
}
```

Those fields could be re-arranged depending on version. In contrast, now it looks like this:

```
struct CONFIG::Item::Hdr
{
    DWORD flags; // also xor key
    DWORD crc32_name;
    DWORD size;
  DWORD addr;
}
```

The flags also act as a XOR decryption key for the remaining fields, meaning that one cannot simply write rules looking for known `crc32_name` values in order to detect unpacked samples. Armed with this knowledge it's quite simple to extract all the relevant information. Fortunately, no changes where made to the method by which parameters[2] are parsed so we can happily re-use our code for previous versions.

Closing Remarks

It is extremely hard to track variants of this family. One big reason is that most researchers and anti-x companies are using a wild mix or sometimes similar

[2] item with `crc32_name` equals to 0x8fb1dde1

names (mostly "Gozi") for different malware samples, collapsing a quite complex family structure into a supposedly singular super cluster. While this is acceptable for detection purposes since their capabilities are not tremendously different, it can get quite messy if you want to track or research a particular actor group and/or development strain. At the time of writing, I'm pretty sure there are 3 independent groups maintaining and developing the main bot code:

- `Dreambot` - the most prominent developers that are behind most changes
- `IAP` - probably the original authors - they have some working relationship with `Dreambot` guys because changes are going back and forth between them
- `TheLoader` - looks like new kids on the block. They can be seen as a first stage downloader in various spam and malvertasing campaings which later are pushing a worker module or separate additional malware families such as Rammnit or Dridex.

That being said, without direct access to the criminal underground it remains pure speculation, since some groups may come and go. Also, the level of cooperation beetween potential independent groups is unknown. Take it with a grain of salt, but I put a lot of hope into the future of this malware, as this is one of not so many families that are still creatively and actively developed and bring us as analysts interesting work.

References

[1] M. Kotowicz, "ISFB: Still live and kicking," *The Journal on Cybercrime & Digital Investigations*, vol. 2, no. 1, 2017 [Online]. Available: https://journal.cecyf.fr/ojs/index.php/cybin/article/view/15