GATEWATCHER

MALWARE ANALYSIS

NSIS Packer

The NSIS (Nullsoft Scriptable Install System) installer is more and more used by malicious actors as a first unpacking stage. As it is open source (https://nsis.sourceforge.io/Download), it is a convenient base for malware author to use and eventually modify. The Gatewatcher purple team analysed this packing method, we'll describe it taking the following sample as an example:

SHA256:3807df31a0db07c4ea7b4e06c0f8c2db76e3c84071319d1e1b356c62f3872512 obtained on 05/11/2022.

This sample general unpacking process is as follow:

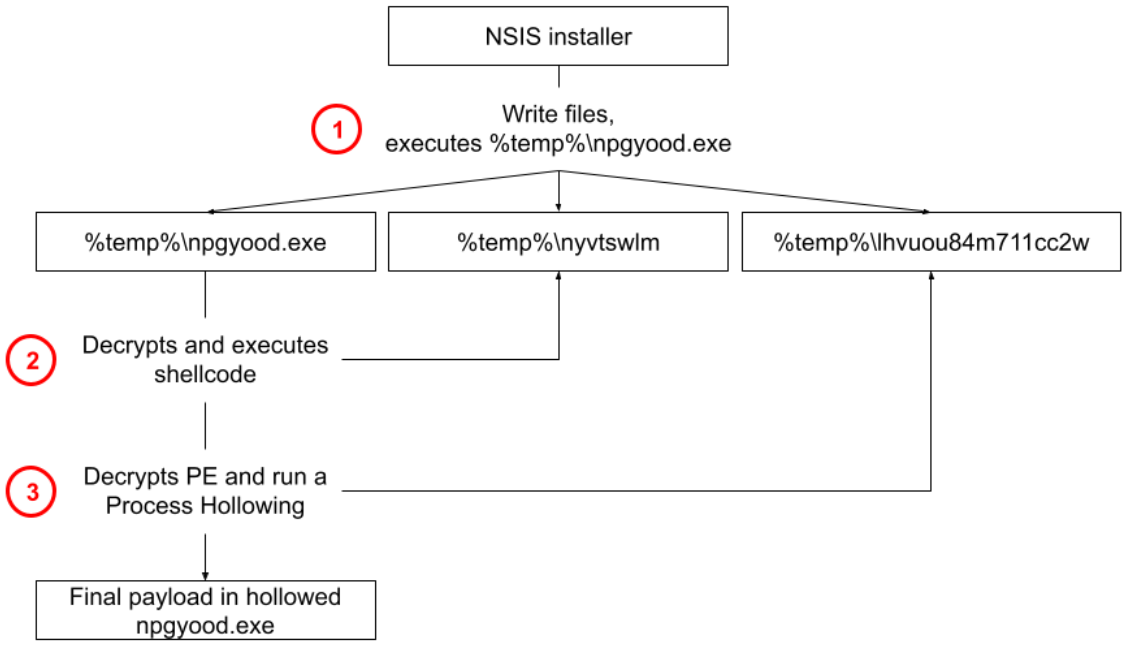

Figure 1: unpacking process

1. The NSIS installer is used to drop 3 files in the %temp% folder: one PE executable, and 2 other unidentified files. It then runs the PE before exiting.
2. This new process simply opens one of the other dropped files, decrypts it, and jumps on its content (it's a shellcode).
3. The shellcode opens the third file, decrypt it, and runs a process hollowing on a copy of its own process with the decrypted payload (a PE file).

Each of this step will be analyzed in more details in this article.

1 NSIS installer

1.1 Files using BZIP2 compression

The sample we analyzed uses the BZIP2 algorithm to compress its data. The NSIS file follows this structure:

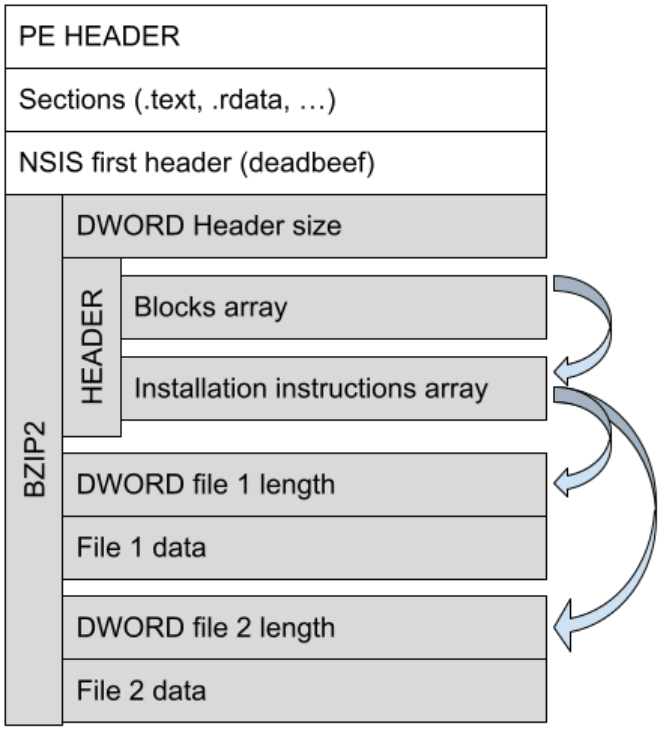

Figure 2: structure of the NSIS file, when using BZIP2

The data related to the installation are placed out of the PE sections, at the end of the file. They are not present in the process memory.

To retrieve this content, the installer starts by opening its own file, and searches a header structure (like an egg hunting):

- A DWORD flag (with a value inferior to 16),
- a DWORD: 0xdeadbeef,
- the ASCII string NullSoftInst,
- 2 DWORD: the length of the header, and the length of the following data.

This header is followed by a custom version of BZIP2 compressed data (some block size has been changed compared to the official implementation, but it is overall the same algorithm).

The decompression is progressive (not all data are decompressed at once), and the decompressed data are saved in a random file in %temp% (used like a byte array or variable length).

All the file data are compressed in a single stream, and each field size (header, file length, …) is stored as a DWORD inside the compressed stream (which is a main difference with LZMA, as we'll see later).

The compressed data starts by a header, with an array of blocks. Each block points toward data (strings for examples), and one of them points to an array of installation instructions, each of them described by a first DWORD representing the type of instruction, and 6 other DWORD used as parameter, whose signification varies depending on the instruction (making the total size of an instruction entry 0x1c).

There are around 60 instructions, depending on the compilation options. A result can be stored by some instruction to be used by the next one, like some form of a lite stack-based programming language.

In our sample, the instruction number to write a file is 0x14, the offset of the file data structure inside the decompressed BZIP2 being placed in the second parameter (out of the 6). The file data structure is a simple DWORD representing the size of the file, followed by the file itself.

The LZMA compression algorithm can be used instead of BZIP2, with the exact same structure. We also came across samples using LZMA with a different structure.

1.2 Files using LZMA compression by blocks

Some NSIS samples uses LZMA compression algorithm, with a different structure as shown below:

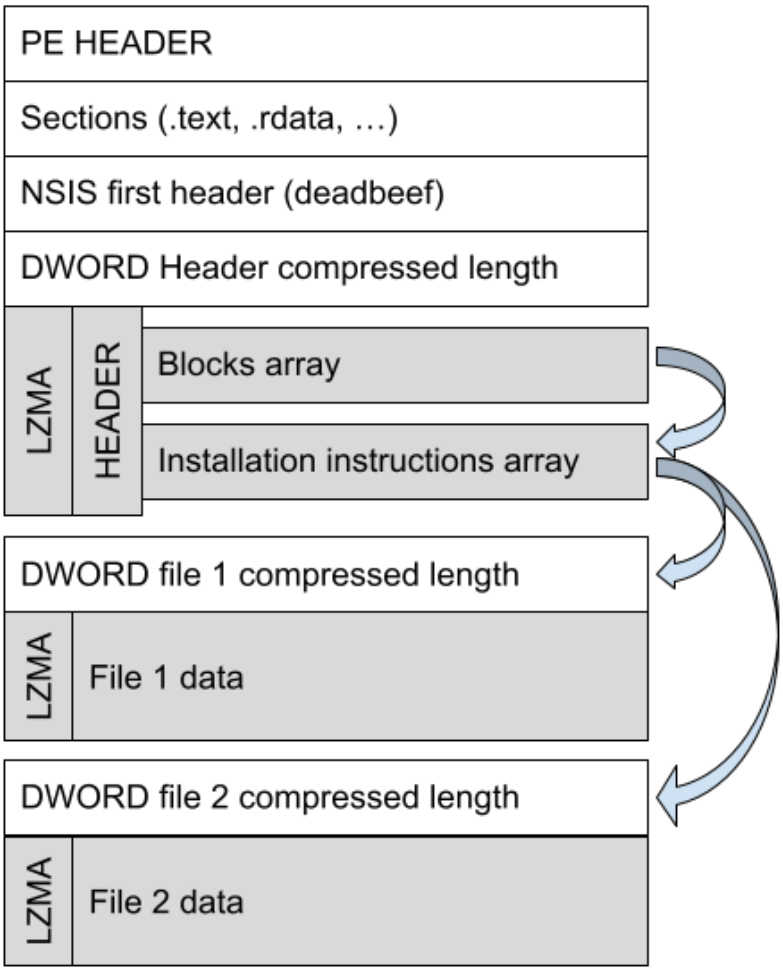

Figure 3 : File structure when using LZMA

The general structure when using LZMA is the same, but each field starts with a clear text DWORD, indicating the size of the raw data to use for LZMA decompression (this DWORD is a flag, and must be xored with 0x7fffffff to get the actual size). Each field is compressed independently.

2 Dropped files

The analysed sample drops 3 files in %temp%, one of them being a PE executable being launched at the end of the installation. The rest of the unpacking process happens only in memory, there are no other content written in the filesystem.

The PE executable simply opens, and decrypts (a simple XOR-like loop) one of the other 2 dropped files before jumping on its content:

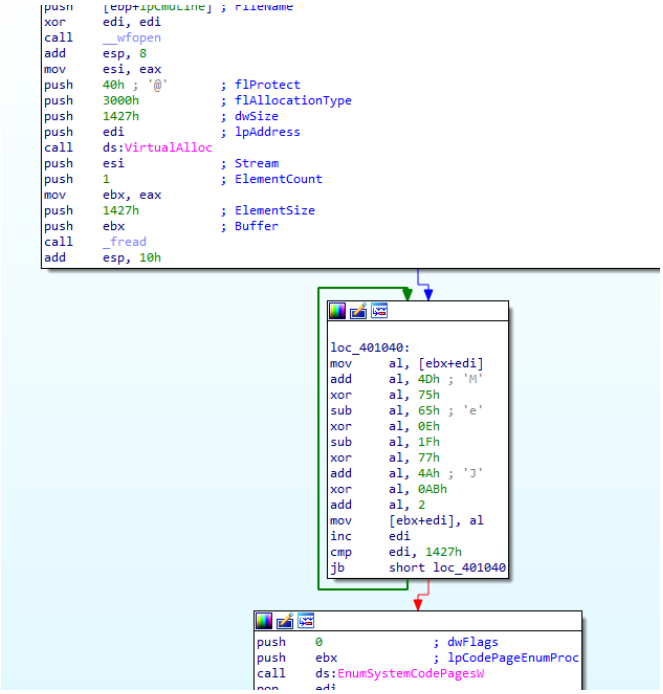

Figure 4: second unpacking step

This shellcode obtains kernel32.dll address from the PEB, and uses a custom import by hash to obtain several other functions:

```
call    RE_get_kernel32_dll
mov     [ebp+kernel32_dll], eax
mov     edx, 7FC01DAEh
mov     ecx, [ebp+kernel32_dll]
call    RE_custom_GetProcAddress ; GetTempPathW
mov     [ebp+GetTempPathW], eax
mov     edx, 0FF7F721Ah
mov     ecx, [ebp+kernel32_dll]
call    RE_custom_GetProcAddress ; GetModuleFileNameW
mov     [ebp+GetModuleFileNameW], eax
```

Figure 5: call of the custom import by hash

It then proceeds to open the third and last dropped file, decrypt it with another (very long) XOR-like loop:

Figure 6 : partial view of the decryption basic blocks, based on successive ADD, OR, SUB, XOR, NOT and NEG operations.

Finally, the shellcode starts its process executable again, and does a process hollowing on its copy, loading the decrypted PE in memory.

The final payload of this sample was detected as a common version of Loki.

3 Hunting

We found many other samples using NSIS as a first packing step, the LZMA compression by blocks method being the most used. The dropped files are on many cases very similar to the ones we presented above. The exact decryption routine varies:

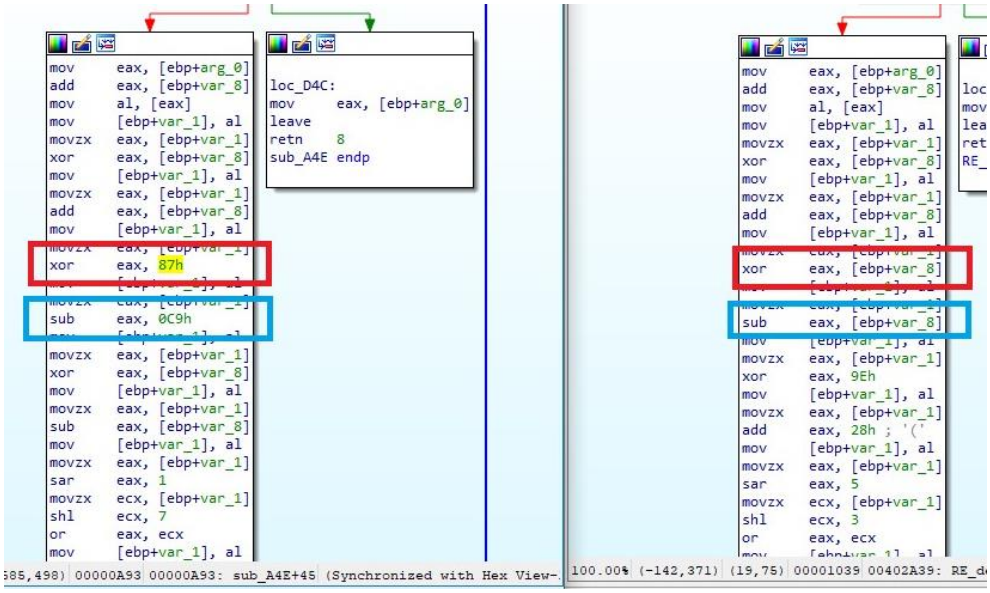

Figure 7 : 2 decryption routine present in the shellcode for step3.

In the example above, the decryption follows the same flow, but the operations (XOR, ADD, etc …) and the value used (either the current position, or a constant) are different : it is likely randomly generated.

Jérémy Beaume


GATEWATCHER