# Unpacking Smokeloader and Reconstructing PE Programatically using LIEF

m.alvar.es/2020/06/unpacking-smokeloader-and.html

This article holds notes on my experience unpacking a *Smokeloader* 2020 sample. The unpacked payload is further used for composing a valid *PE* file. The outcome is a *PE32* executable containing clean code ready for reversing.

First things first, here is the sample used in this research:

| | |
|---|---|
| Size | *308.17* KB (*315568* bytes) |
| Type | PE32 executable for MS Windows (GUI) Intel 80386 32-bit |
| First seen | 2020-03-06 21:45:11 |
| md5 | c067e0a2d7fc6092bb77abc7f7156b60 |
| sha1 | 52f68073caec0fd424c7cbaaed5f5221d7103d20 |
| sha256 | 25959cfe4619126ab554d3111b875218f1dbfadd79eed1ed0f6a8c1900fa36e0 |

You can find it in VirusTotal [1].

This sample does regular already documented *Smokeloader* checks before unpacking the main payload, such as:

- checks if the process is running in the context of a debugger using "*kernel32.isDebuggerPresent*" function [2];
- makes a copy of *ntdll.dll*, loads it and uses it instead. This technique helps to evade some sandboxes and has been described already in this article here [3];
- looks for specific patterns in registry keys to check if the sample is running under a virtualised environment.

It also performs a small profiling of the hosting machine in order to decide which payload to inject. *Smokeloader* has specific code for both main architectures x86 and x64. In this article, we gonna unpack the x86 payload of the above mentioned sample.

*Smokeloader* has been using various techniques to inject its final payload into the user file management process "*explorer.exe*". The sample analysed uses Rtl*CreateUserThread* approach in order to copy the final payload to the targeted process. This injection method is better described in this Endgame/Elastic article [4].

So our game plan is:

1. pause execution before the unpacked payload is executed by "*explorer.exe*";
2. transplant this code to a dummy PE shell;
3. fix PE header values and section boundaries;
4. patching *Smokeloader* code preamble;
5. test unpacked *Smokeloader* PE;
6. how to do all this programatically using LIEF [5].

**1.0 - FETCHING MAIN PAYLOAD**

*Smokeloader* 1st stage decompresses its payload using *ntdll.RtlDecompressBuffer* [6] after few anti-analysis checks described above. It does not call this function from the initially loaded *ntdll.dll* but from a copy of it loaded afterwards. So breakpoints should be set after the binary loads the copy of *ntdll.dll*. Figure 01 presents a screenshot of this specific code *IDA*.

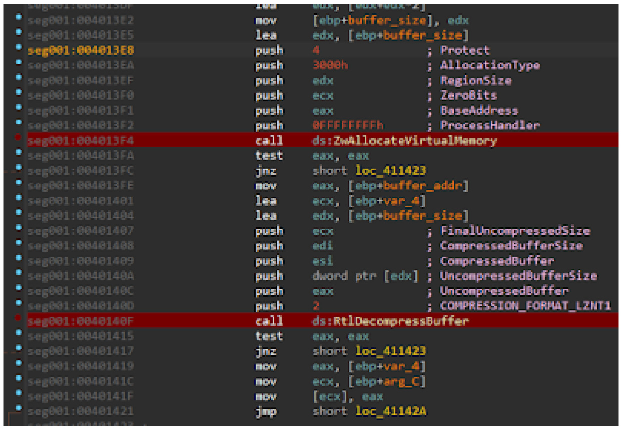


Figure 01: *Smokeloader* first stage decompression code

This code allocates a buffer with *0x2D000* bytes using *ZwAllocateVirtualMemory* which stores the main decompressed payload [7]. This code is still transformed before being injected into "*explorer.exe*". The following steps are performed during injection:

- fetches *explorer.exe* PID by calling *GetShellWindow* and *GetWindowThreadProcessId*;
- sections and maps are created in the current and remote processes using *ZwCreateSection* and *ZwMapViewOfSection*;
- main payload is copied to local section and reflected in the remote section;
- data section is created in the remote process for holding parameters and dynamically created Import Table;
- A new thread is created in the remote process by invoking *RtlCreateUserThread*.

So, at this point, you could ask me: what is the relevance of describing all these call names to the final goal of this article? the answer is: so you can reproduce exactly what I'm describing in here. :D

Next step is setting up a break point in *RtlCreateUserThread* (from the copy of *ntdll.dll*) and dump the final payload. It is also necessary to take note of few important addresses: (*i*) entry point of the thread created in the remote processes and (*ii*) base addresses for injected code in virtual process.

Figure 02 shows a screenshot of *IDApro* showing the call to *RtlCreateUserThread* (where we should pause the execution).


Figure 02: Call to RtlCreateUserThread after injecting code into remote process

By stoping the execution on this call we can collect all data we need to move on to the next step:

| | |
|---|---|
| **Base address code** | *0x02060000* |
| **Base address data** | *0x00B60000* |

| | |
|---|---|
| **Data payload** | *a01751fb6eb3f19d9b010818bbecc23c* [8] |
| **Code payload** | *2547231b4ae82ea9e395fb0c8a308982* [9] |
| **Code entry point** | *0x02061734* |

Code payload is the final unpacked *Smokeloader* code adjusted to run on Virtual Address with base equal to *0x02060000*. The created thread receives the base address of the data segment (*0x00B60000*) as parameter ("*StartParameter"* parameter of "*RtlCreateUserThread*" call).

## 2.0 - TRANSPLANTING PAYLOAD TO PE

*Smokeloader* loads all resources necessary to its execution dynamically. This article here [10] describes how *Smokeloader* builds up its import table and how to prepare patch an IDB to overcome this technique before starting reversing. So this main payload does not need any specific setup of *imports*.

In this section we will use 010 Hex Editor [11] to transplant a PE header from a random executable. 010 Hex Editor has a PE format template [12]. Although any other valid PE32 binary could be used in this experiment, we used a PE header extracted from an executable listed in this Sotirov's blog post [13][14].

*Smokeloader* code payload has *0x1000* null bytes at offset zero, so we copied the first *0x1000* bytes containing the PE header from *tinype.exe* to this region.

Coincidently, *.text* section will be already pointing to the beginning of the our payload at offset *0x1000*. Probably the malware author just wiped out the PE header before creating the payload and left the null bytes there. Next step is to paste all *20480* bytes (*0x5000*) of our data payload in the offset *0x4400*.

Figure 03 shows the new layout of our binary containing the PE header in the beginning followed by *0x3400* bytes of code payload (at offset *0x1000*) and finally *0x5000* bytes of data from our data payload (at offset *0x4400*).

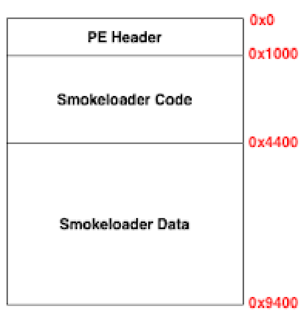


Figure 03: Initial layout of new handcrafted PE binary

## 3.0 - FIXING PE HEADERS

It is time to adjust our implanted PE header manually using *010 Hex Editor*. At this point, all fields in this header are still set up according "*tinype.exe*". From now on, we gonna use this schematic as reference to PE header internal structures [15].

The first adjustment is to change the number of sections to *2* for holding code and data. This field is located in the "*COFFHeader.NumberOfSections*". Now our binary will list only *2* sections named "*.text*" and "*.rdata*" we can rename this second one to "*.data*" by changing "*SectionHeaders[1].Name*".

Next step is make sure that both sections have correct permissions. "*SectionHeaders[0].Characteristics*" ("*.text*") should have *CODE*, *EXECUTE* and *READ* flags active and "*SectionHeaders[1].Characteristics*" ("*.data*") should have the

*INITIALIZED_DATA*, *READ* and *WRITE* flags active. Still on SectionHeaders, we can setup the bounds and virtual addresses. "*SectionHeaders[0].SizeOfRawData*" should be set to *0x3400* (*13312* Bytes), "*SectionHeaders[0].PointerToRawData*" should be set to "*0x1000*" and finally "*SectionHeaders[0].VirtualAddress*" should be set to *0x1000*. For "*SectionHeaders[1]"* ("*.data*") we gonna set "*SizeOfRawData"* to *0x5000*, "*PointerToRawData"* to *0x4400* and "*VirtualAddress"* to *0x5000*. These changes means that these sections will be mapped in memory in base_address (defined in the *OptionalHeader*) shifted by each section Virtual Addresses offset. There is an Union inside these section headers called "*PhysicalAddress*" and "*VirtualSize*", these fields should hold the same value as "*SizeofRawData*".

Figure 04 shows a diagram of a Section header. Each section in the binary has an instance of this header associated to it.

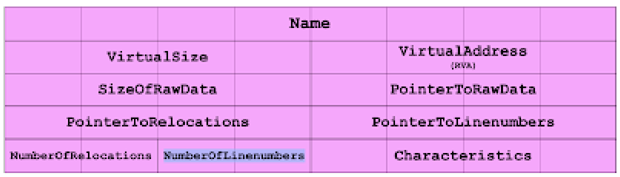


Figure 04: PE Section Header

Now we need to adjust few fields in the Optional Header. In this header we will need to change the following fields:

| | | |
|---|---|---|
| ImageBase | Virtual Address where binary will be mapped | 0x02060000 |
| SizeOfCode | size of .text section | 0x3400 bytes |
| SizeOfInitializedData | size of .text and .data sections together | 0x8400 bytes |
| AddressOfEntryPoint | offset of the entry point code | 0x1734 |
| BaseOfCode | .text section Virtual Address | 0x1000 |
| BaseOfData | .data section Virtual Address | 0x5000 |
| SectionAlignment | Virtual Addresses have to be multiple of this value | 0x1000 |
| FileAlignment | file offsets have to be multiple of this value | 0x200 |
| SizeOfImage | total size of binary headers + sections | 0x9400 |
| Checksum | PE file checksum - use PE Explorer [16] or Hiew [17] to calculate this value | -------- |

"*ImageBase*" has to match the base of the code section we dumped from "*explorer.exe*" (*0x02060000*). As we will not export or import anything all "*Data Directories*" inside the Optional Header can be *zeroed* as well.

Summarising the whole process:

1. Transplanting PE header from a dummy PE;
2. Fix sections sizes, boundaries, permissions and Virtual Addresses in *SectionHeaders*;
3. Setup section contents;
4. Setup Optional Header fields;
5. Setup PE checksum;

Here is the version of our binary after following up all steps described above [18]. This binary is a valid executable and we can load it in any debugger or disassembly but we still need to change one last thing before call it a valid unpacked *Smokeloader* sample.

**4.0 - PATCHING BINARY**

Figure 05 shows our reconstructed PE loaded in *IDApro* paused on the correct Entry Point.


Figure 05 - Reconstructed PE paused on Entry Point

We can notice that the entry point function receives an argument (*0x02061737*) and loads it into *ECX* and then calls another function located in *0x02061743* which is just below the current function. This argument is the address of the data segment. This data segment will be used for various tasks during *Smokeloader* execution including holding the dynamically created import table.

If we execute this file without a valid value in *ECX* it will break when the main payload tries to write into the data segment (invalid address in *ECX*). Figure 06 shows what happens when we try to execute our binary the way it is right now.

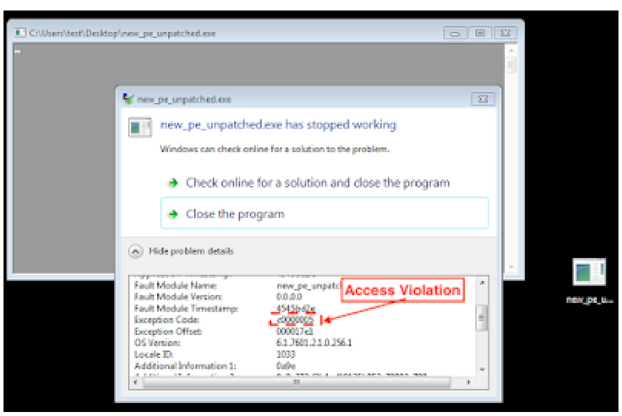


Figure 06 - Access Violation exception when executing unpatched reconstructed binary

The plan now is to patch this binary to load the correct address of the data segment into *ECX* before calling "*sub_2061743*". Since both functions are consecutive and the function on top does not do much - we gonna replace all *15* bytes of this function (*0x02061743* - *0x02061734*). Figure 07 shows the new patched code.

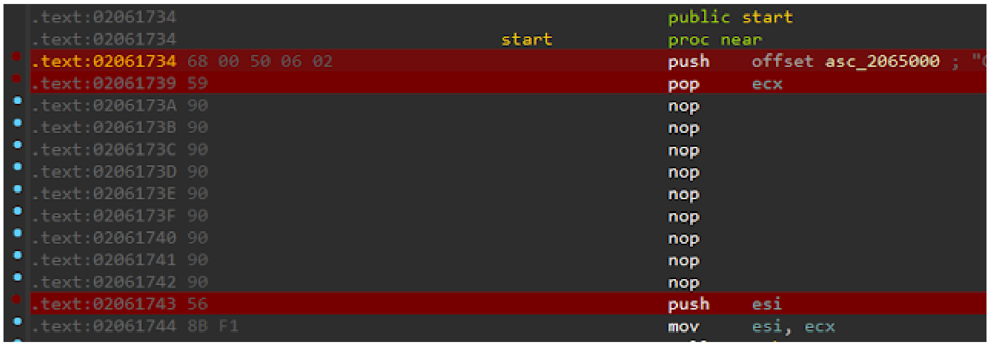


Figure 07 - Code after patching

In this new code the entry point remains the same. We can see that we loaded *ECX* with the address of the data segment by using the push and pop instructions and then we filled the rest of the remaining bytes with *NOP* (*0x90*). We can see the beginning of the second function at the same address as before (*0x02061743*). Of course there are many ways to achieve this same result but this was the simplest approach we could think of.

The final step is to update the PE checksum field inside the *Optional Heade*r again and we will have a fully unpacked *Smokeloader* sample. Here are the last version of our reconstructed binary [19]:

| | |
|---|---|
| File name | new_pe_patched.bin |
| File type | PE32 executable for MS Windows (console) Intel 80386 32-bit |
| Size | 37888 bytes |
| md5 | f401109ae24aaf47dce75266ffc049f8 |
| sha1 | 49e7ed68b9569e0e987da71b3c678974d8ed7c81 |
| sha256 | cd42f017913034d527d90a84feebcde015e714baa03714c83f80608555e52386 |

## 5.0 - TESTING RECONSTRUCTED PE

For testing our branding new reconstructed PE we ran it into Cuckoo sandbox [20] to analyse its behaviour [21]. As we can see in figure 08 and 09, the binary was executed properly and we got it checking in and contacting its controllers.

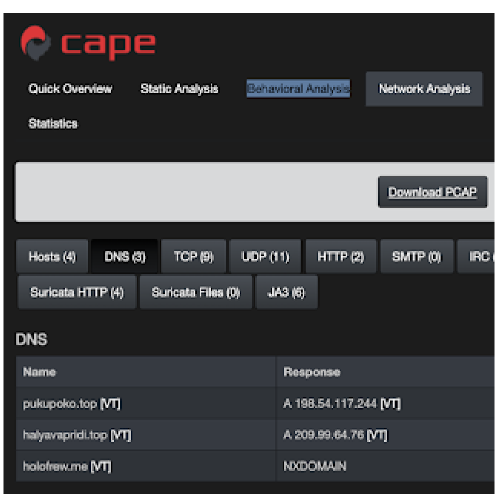

Figure 08 - Reconstructed sample connecting back to Controllers

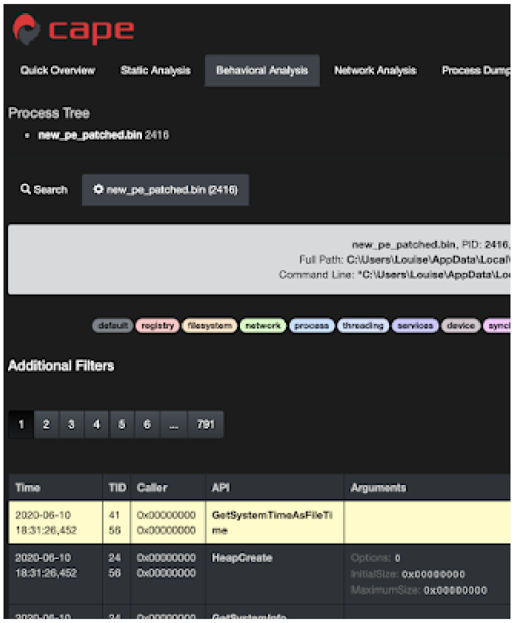


Figure 09 - List of API calls intercepted by Cuckoo

As we can see we got the sample connecting back to three controller *URLs* and many pages of intercepted *API* calls in the behavioural analysis. This is an indication that our unpacked and reconstructed *Smokeloader* sample is functional.

## 6.0 - DOING IT ALL PROGRAMATICALLY

So far we described how to unpack *Smokeloader* main payload and how to manually reconstruct a valid PE file out of this. Now we will automate what we did manually by transplanting a PE header from a dummy binary ("*tinype.exe*"). The Python library used in this experiment is *LIEF* [5]. We extended this example called "Create a PE from scratch" they have in their official documentation [22].

The following code does exactly the same as the manual approach but using *LIEF*.

The final binary generated by *LIEF* is:

| | |
|---|---|
| File name | unpacked_smokeloader.exe |
| File type | PE32 executable for MS Windows (console) Intel 80386 32-bit |
| Size | *34816* bytes |
| md5 | a0aebc61bc89208be0585eca4d1ed00c |
| sha1 | ea2f3c914dec6bb36832abc313b3fce826cdecb0 |
| sha256 | 0247de510507792fcbf425fab9dbbc2f067c25dc7e4e80a958d1ebfb0505f6e6 |

We uploaded it for testing to Virustotal [23] and CAPE sandbox [24] and is a valid unpacked *Smokeloader* PE32 executable.

**REFERENCES:**


[1] https://www.virustotal.com/gui/file/25959cfe4619126ab554d3111b875218f1dbfadd79eed1ed0f6a8c1900fa36e0/details
[2] https://docs.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-isdebuggerpresent
[3] https://malwareandstuff.com/examining-smokeloaders-anti-hooking-technique/
[4] https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process
[5] https://github.com/lief-project/LIEF
[6] https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-rtldecompressbuffer
[7] https://www.virustotal.com/gui/file/756dd799c8195b98f295baa210ac1807f7d1d86de2736f559c76ce1c7816d0ee/detection
[8] https://www.virustotal.com/gui/file/787c762f7384dc8f427f76fb313e3bc248b09516a917426a533ed295bf1b593b/detection
[9] https://www.virustotal.com/gui/file/3cae0d6a80a716b1c823e1f4011b3d07c4b3475a894751006874729d2145bbcf/detection
[10] http://security.neurolabs.club/2019/10/dynamic-imports-and-working-around.html
[11] https://www.sweetscape.com/010editor/
[12] https://www.sweetscape.com/010editor/repository/templates/file_info.php?file=EXE.bt
[13] http://www.phreedom.org/research/tinype/
[14]
https://www.virustotal.com/gui/file/718b03e878080c34f4a51c7243bb60024c7734f23d5715d6b97a5a4d54d7a630/detection
[15] https://upload.wikimedia.org/wikipedia/commons/1/1b/Portable_Executable_32_bit_Structure_in_SVG_fixed.svg
[16] http://www.heaventools.com/
[17] http://www.hiew.ru/
[18] https://www.virustotal.com/gui/file/3e474e495e11716f6eab40ee3c353602da2683dbefd900f6d90dfcedff2fa93d
[19] https://www.virustotal.com/gui/file/cd42f017913034d527d90a84feebcde015e714baa03714c83f80608555e52386/detection
[20] https://www.capesandbox.com/
[21] https://www.capesandbox.com/analysis/7521/#behavior
[22] https://lief.quarkslab.com/doc/latest/tutorials/02_pe_from_scratch.html
[23] https://www.virustotal.com/gui/file/0247de510507792fcbf425fab9dbbc2f067c25dc7e4e80a958d1ebfb0505f6e6
[24] https://www.capesandbox.com/analysis/7643/