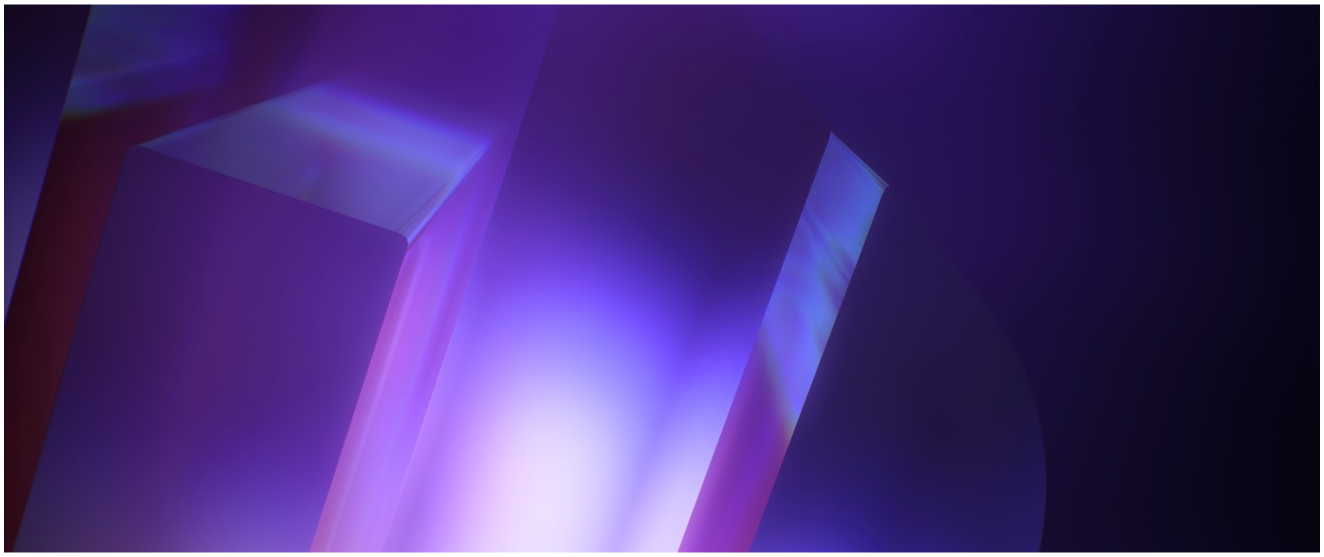
Sophisticated New Packer Identified in CryptXXX Ransomware Sample

sentinelone.com/blog/sophisticated-new-packer-identified-in-cryptxxx-ransomware-sample/

May 5, 2016



Summary of Execution

This sample is being delivered using the Angler Exploit Kit within a drive-by download attack. The exploit is the latest Adobe Flash exploit.

The packing on the sample is very sophisticated; far more sophisticated than the final payload. The packer looks to be written in C and utilizes many techniques used by shellcode, but is repurposed to evade static analysis. The packer also has many unnecessary API calls whose purpose could only be to confuse dynamic and behavioral detection. Multiple layers of packing are used to further conceal itself.

The ransomware payload is written in Delphi and only has a few dirty tricks to mask it's malicious intent. In addition to the ransom, the payload hunts down bitcoin wallet files on the disk to steal their contents.

Why would an adversary spend more resources in the packing of ransomware instead of the ransomware itself? I maintain that they are trying to increase the time between the initial release of the malware and the time of detection by the antivirus software. This longer time period will net them more infections, and in turn, increase revenue on an individual campaign. It makes economical sense to spend more resources on gumming up the antivirus analysis and detection process, than building a better ransomware.

Packed Sample

Filename: api-ms-win-system-hid-l1-1-0.dll
File size: 252,416 bytes
MD5: 8D044D1FC07526FA0B1ADADD1FBDAA28
SHA-1: 41A70AD7D7D43C33C73485C827ADAEF30C86597A
SHA-256: 175e01b113bbd7637adb88e4f1d3bc526dc429b221465a7a1fbd5bf1ed22662f

Unpacked Sample

File size: 394,240 bytes
MD5: BAF5EC28F7E25FB3C54153C509940712
SHA-1: C47A6626CAB9BE3904AA70CF24910A26E46F16BF
SHA-256:
A386FE46D245BC3C53DD9154266557E19219DB0086DCBE1BC6FD7B9B0EC9B70B

Packer Layer 0 (Psychological)

From an initial inspection, the sample looks like an innocuous DLL from Microsoft. The file has a good sounding scary-to-delete-me name. The version strings in the resource section are formatted exactly to Microsoft style. Most malware I've seen that tries to masquerade as a Microsoft binary forgets certain features like the copyright symbol.

All the DLL export symbol names are in the format lua_*. It looks like a legitimate Lua library, but the disconnect begins here. When have you ever heard of a Microsoft product written in Lua?

All of these exports are just stub functions. Just standard procedure prolog and epilogs with no implementation in-between.

The sample is also hard to get started unless you have full context of how it is supposed to be run.

The sample is initially run using rundll32.exe

```
C:WindowsSystem32rundll32.exe api-ms-win-system-hid-l1-1-0.dll,Working
```

When first loaded, a call to the dll entry point is made, then a call to Working(). The packed DLL does not have an export named Working(), but the unpacked payload does. Working() will check the the command line string to ensure that it was launched with rundll32.exe. If it isn't, It will create a new process using rundll32.exe. Once Working() is sure that the process was created with rundll32.exe, it will then start a new process using rundll32.exe, but this time it will run the AccessToken() export. AccessToken() is the beginning of the ransomware code.

To successfully launch the ransomeware, you must pass the “Working” or “AccessToken” parameter in the rundll32 command line. Without it, it will not run. Why would the author be troubled to limit the infection?
I postulate that the author did this on purpose. I can see the scenario playing out in my head: A forensics analyst has found this DLL, and determines that it is related to the compromise. He sends the sample
to a malware analyst who tries running the DLL, but can’t get the ransomeware to invoke.
The malware analyst will incorrectly respond to the forensic analyst, saying that there must be more to the infection. This will slow down the initial response process, allowing more infections. Before the malware analyst can get the ransomeware to run, he will have to unpack the sample to get the payload, and see the two exports of the payload.

The packer is riddled with API calls that have no effect whatsoever. This makes it a nightmare for manual dynamic analysis. It was hard to tell what code is important, and what is just a red herring. An analyst who sees a call to GetLogicalDrives() might think, “I must be really close to the ransomware code!” but only to be disappointed to find that the ransomware isn’t even unpacked. Here is a call to MoveFile(), that might look like it’s using obfuscated strings statically, but dynamically, you can see the function return 0 indicating failure.

There are imports to winmm.dll (multi-media) that are never used. This might be an attempt
to make the DLL look more legitimate (since the filename has HID in the name and most MIDI devices also have a HID component), but I believe the author’s reasoning to include all these unused imports is to hide one crucial import: VirtualProtect().

This one call to VirtualProtect() opens the gate to the rest of the packer.

Packer Layer 1

The first software layer of protection used by the packer is an encrypted region of code. The code is decrypted in place, but because the “.text” section is marked as read-execute in the PE header, a call to VirtualProtect() must be made to allow write access.

Once the region is decrypted, it is run. Unfortunately, OllyDBG has determined this region to be data, not code. So in the debugger, Olly will not display the disassembly. I eventually discovered that you can remove Olly's analysis for a section using ctrl+backspace:

At this point, I made a dump of the DLL to analyze the uncovered region.

Packer Layer 2

The windows loader was not able to patch the relocations for this encrypted region, because it was still encrypted during the loading process. As a result, many restrictions are on the author for this region of code.

All the code must be written similar to shellcode. It has to run wherever it is loaded in memory. This means no access to global variables, the import table, or string literals.

The first thing that is done, is a large structure is allocated on the stack. This structure contains the 'global variables' for this section of code. because It doesn't know where the ".data" segment is to access normal global variables.

You can see in my reverse engineered version of this structure, there are pointers to API calls. This is done because the encrypted portion of the packer was not linked during compile time. It also hides these entries from the import section.

Now, the base address of the DLL is recovered using a technique similar to an egghunter. a call $+5; pop eax is used to find the value of EIP, and memory is search for 'MZ' at the start of a each 4k page. This value is stored for later use.

Because this section of code isn't linked into the main binary at compile time, its imports have to manually be resolved. This is done with a technique commonly used in windows shellcode. A pointer to a linked list of loaded modules is stored inside the Process Environment Block (PEB). By traversing this linked list, the required modules containing the exported functions needed can be found. Once the modules are found, the individual exports can be found by following the PE structure. Also, because this section of code can't use any string literals, instead of using a string compare, all strings needed are hashed and the hashes are compared.

In the following image, the hash value of the string "kernel32.dll" is 0x6a4abc5b.

The most important symbols to resolve are LoadLibrary() and GetProcAddress(). Both are in kernel32. Using these two functions, all other symbols can be resolved. But because of the limitation of no string literals, some strings have to be built on the stack first.

Now that all the prep-work for this stage is done, it must now unpack the payload and position it in memory.

First the packed dll is read off of the disk and into memory. The region containing the compressed payload is found.

The payload dll is decompressed using “RtlDecompressBuffer()”. The PE header is parsed and the image is reconstructed in memory. This image must have its imports manually linked in and relocations patch. But the address range the payload image has been reconstruced in is not it’s final resting place. The packed dll is occupying that region.

To remove the packed image from memory to make room for the final position of the unpacked payload DLL, a bit of code must be placed outside of these two regions. This portion of the code is identified by a ‘MOV EAX, 0x11223344’ instruction. A VirtualAlloc() is performed, and the code is copied to the newly allocated region. Execution continues at this region.

This last bit of code will unmap the packed dll from memory using a call to UnmapViewOfFile(). Next, it reallocates this exact same region to copy the unpacked payload image. The unpacked payload is freed from it’s temporary position and execution continues at the entry point of the payload dll.

From here, the ransomware runs.

Ransomware

The ransomware communicates to one of two hard coded command servers.

The IP addresses of the command servers are 146.185.250.152 and 217.23.6.40. There are no domain names stored in the sample, only the IP addresses. Whois records for these ip ranges appear to be issued to Ukraine and Netherlands.

Domain names that are associated with these ip addresses in the past are:

146.185.250.152
f0rget–a00lls.com
weeell-drive.com
647hnhjnnc-nje00l.com
feeeel–t00ll.com
daertnw90-kola.com
217.23.6.40
p3.regularclass.com

regularclass.com
ns2.streamingmp4.net
ns1.streamingmp4.net
ns2.streamingmp4.net
ns1.streamingmp4.net
ns2.streamingmp4.net
ns1.streamingmp4.net
cdn78.livefile.org
venusbjerget.dk
semi2.wrzhost.com
kidshot.ijustcantbelieve.com
callspoof.ijustcantbelieve.com
celebrityhotnews.net
justinbieberoncam.ijustcantbelieve.com

Many of these records appear to be old by at least a year, and are probably not used in the attack.

Once the local files are encrypted, the user is presented with a ransom notice:

Conclusion

As adversaries continue to invest resources on packing we should expect to see more examples of how these tricks will still slow down the detection pipeline.

SentinelOne detects this specific attack at the initial exploit phase, before the ransomeware is even unpacked.

To learn more about how SentinelOne can help you prevent these kinds of attacks, please click here to download SentinelOne’s Technical Overview.