Andromeda's Five Star Custom Packer – Hackers' Tactics Analyzed

By Roy Moshailov

Archived: 2026-04-05 20:06:42 UTC

Introduction

Packer-based malware is malware which is modified in the runtime memory using different and sophisticated compression techniques. Such malware is hard to detect by known malware scanners and anti-virus solutions. In addition, it is a cheap way for hackers to recreate new signatures for the same malware on the fly simply by changing the encryption/packing method. Packers themselves are not malware; attackers use this tactic to obfuscate the code's real intention.

For security solutions to be effective, they will need to augment their solutions with in-memory capabilities in order to monitor/hook the behavior of the malware after unpacking is completed.

This document describes a sophisticated Andromeda/Gamarue Custom Packer. Andromeda first appeared in 2011 and still remains popular. Since the Andromeda attack chain has been described previously, this analysis instead focuses on the packer and deobfuscation, which happens before the malware downloads or executes its next stage malicious payload. The recent version of the custom packer we obtained (originating June 2016), has noteworthy and innovative functionality.

Does Morphisec stop this attack? Of course, even these new tricks can't get past Morphisec, which prevents this attack before it can drop its load.

Technical Analysis: Andromeda/Gamarue Custom Packer

Nowadays most malware employs anti-analysis techniques to make their code harder to analyze by security researchers. Just like legitimate software developers protect their proprietary work, hackers use obfuscation techniques to protect their code from being reverse-engineered or debugged.

The malware sample in our analysis is packed by a custom packer. To be able to get to the actual code, we first need to unpack it.

How can you recognize a packed malware?

- The sample usually comes with a resource section (in this example RC data contains some encrypted content).
- Typically, the compressed file is very large.
- By looking at the import table – It might have only a few imports and many times these include LoadLibrary and/or GetModuleHandleW as those functions are used for the initial unpacking procedure.

- No readable static strings as the strings are encrypted.
- High entropy in sections for higher efficiency of information storage.
- A large portion of the code is inside the .data section (although there are newer versions with code inside text).
- The program has abnormal section sizes, such as .data and .rsrc sections. The *RawDataSize* is lower than *VirtualSize* and usually also the section names themselves may indicate a particular packer.

How to unpack a packed malware

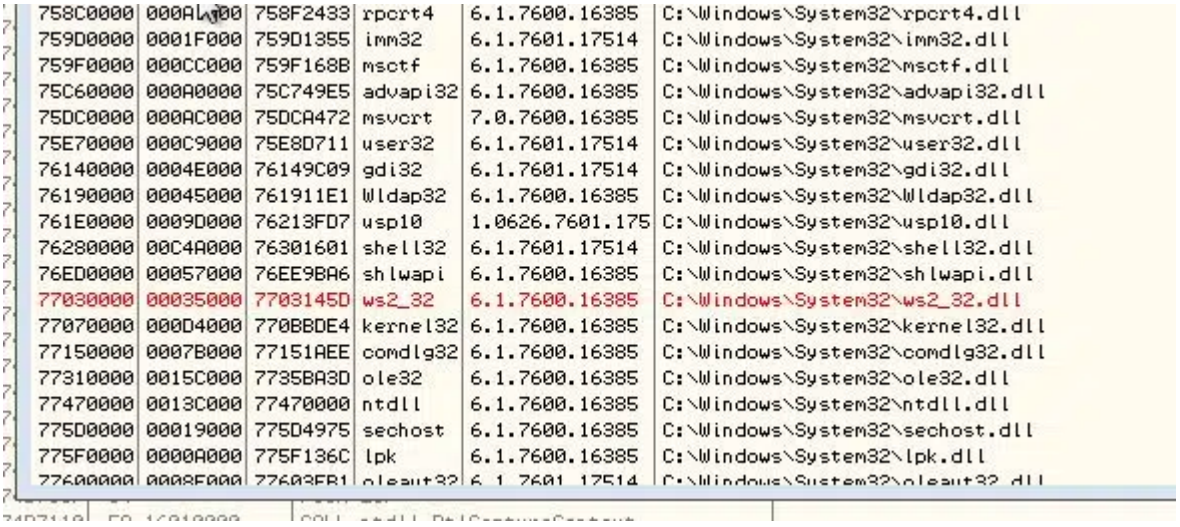
In forensic analysis, there are different ways to handle the unpacking process. While there are automatic tools for different popular packers, it is more difficult to handle custom packers, which require some manual work and a deeper knowledge of the different anti-debugging obstacles. Moreover, custom packers usually also involve stripping off multiple packing layers.

The Packer – Detailed

Looking at *Andromeda's top-layer packer*, we start by noticing an interesting, relatively high entropy in one of the sections (e.g. entropy of .rsrc is 7.376) which gives us the first indication that it is a packer.

RCData	15135	unknown	x	21781	6971E39C83931265F565B711800F340E	7.376	English Australia

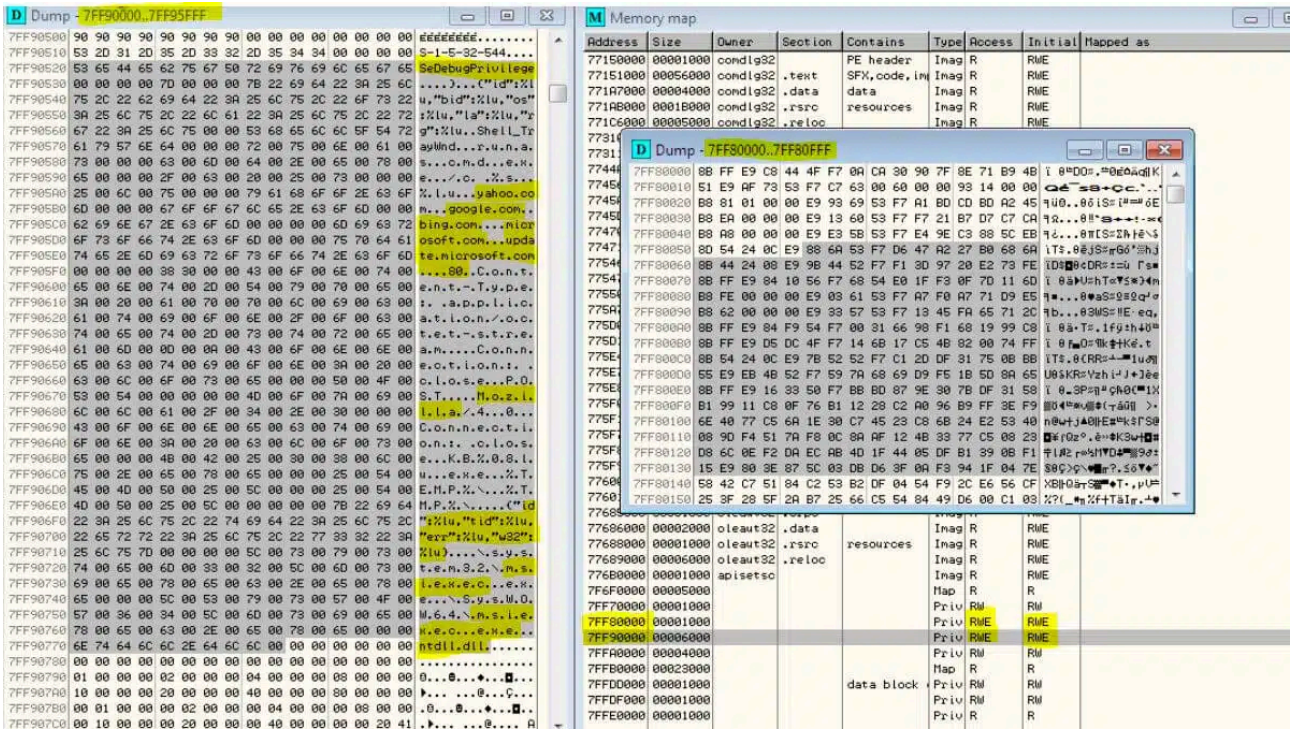
Determining a point in time for which we know the malicious code was already unpacked, we identify the use of ws2_32.dll (responsible for communication API). This means we can assume that the malicious code will start communication after it is unpacked. This is of high probability for downloaders or C&C based malware.



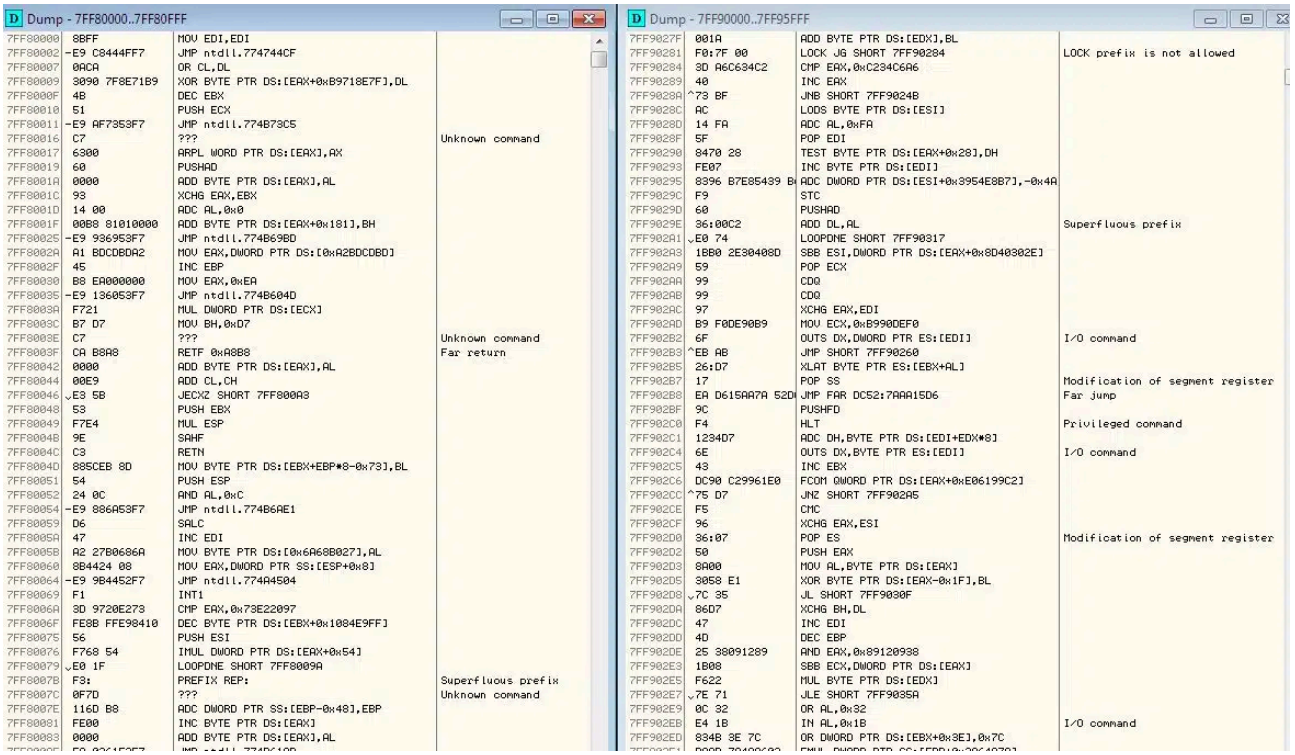
758C0000	000A[illegible]00	758F2433	rpcrt4	6.1.7600.16385	C:\Windows\System32\rpcrt4.dll
759D0000	0001F000	759D1355	imm32	6.1.7601.17514	C:\Windows\System32\imm32.dll
759F0000	000CC000	759F168B	msctf	6.1.7600.16385	C:\Windows\System32\msctf.dll
75C60000	000A0000	75C749E5	advapi32	6.1.7600.16385	C:\Windows\System32\advapi32.dll
75DC0000	000AC000	75DCA472	msvcrt	7.0.7600.16385	C:\Windows\System32\msvcrt.dll
75E70000	000C9000	75E8D711	user32	6.1.7601.17514	C:\Windows\System32\user32.dll
76140000	0004E000	76149C09	gdi32	6.1.7601.17514	C:\Windows\System32\gdi32.dll
76190000	00045000	761911E1	Wldap32	6.1.7600.16385	C:\Windows\System32\Wldap32.dll
761E0000	0009D000	76213FD7	usp10	1.0626.7601.175	C:\Windows\System32\usp10.dll
76280000	00C4A000	76301601	shell32	6.1.7601.17514	C:\Windows\System32\shell32.dll
76ED0000	00057000	76EE9BA6	shlwapi	6.1.7600.16385	C:\Windows\System32\shlwapi.dll
77030000	00035000	7703145D	ws2_32	6.1.7600.16385	C:\Windows\System32\ws2_32.dll
77070000	000D4000	770BBDE4	kernel32	6.1.7600.16385	C:\Windows\System32\kernel32.dll
77150000	0007B000	77151AEE	comdlg32	6.1.7600.16385	C:\Windows\System32\comdlg32.dll
77310000	0015C000	7735BA3D	ole32	6.1.7600.16385	C:\Windows\System32\ole32.dll
77470000	0013C000	77470000	ntdll	6.1.7600.16385	C:\Windows\System32\ntdll.dll
775D0000	00019000	775D4975	sechost	6.1.7600.16385	C:\Windows\System32\sechost.dll
775F0000	0000A000	775F136C	lpk	6.1.7600.16385	C:\Windows\System32\lpk.dll
77600000	0008F000	77603FB1	oleaut32	6.1.7601.17514	C:\Windows\System32\oleaut32.dll

As shown in the image below, there are two unnamed modules with RWE (read write execute) access rights – those are indicators for the unpacked executable shellcode (the code will write and execute from the same location).

Additionally, we can see now strings which are typical to Andromeda.



It is noticeable that those modules are still not a PE file (do not start with PE header) – those are executable shellcodes.



We also notice that the code starting from the entry point of the executable was modified, which reminds us of Process Hollowing/ **RunPE** techniques.

Offset	Name	Value	Value
108	Magic	10B	NT32
10A	Linker Ver. (Major)	6	
10B	Linker Ver. (Minor)	0	
10C	Size of Code	7000	
110	Size of Initialized Data	10000	
114	Size of Uninitialized Data	0	
118	Entry Point	7198	
11C	Base of Code	1000	
120	Base of Data	8000	
124	Image Base	400000	
128	Section Alignment	1000	
12C	File Alignment	1000	
130	OS Ver. (Major)	4	Windows 95 / NT 4.0
132	OS Ver. (Minor)	0	
134	Image Ver. (Major)	0	
136	Image Ver. (Minor)	0	
138	Subsystem Ver. (Major)	4	
13A	Subsystem Ver. Minor)	0	
13C	Win32 Version Value	0	

After

```
00407198  . 55              PUSH EBP
00407199  . 8BEC            MOV EBP,ESP
0040719B  . 6A FF           PUSH -0x1
0040719D  . 68 F8804000     PUSH 7b45c014.004080F0
004071A2  . 68 D0724000     PUSH <JMP.&MSVCRT._longjmpex>            SE handler installation
004071A7  . 64:A1 0000000   MOV EAX,DWORD PTR FS:[0]
004071AD  . 50              PUSH EAX
004071AE  . 64:8925 00000   MOV DWORD PTR FS:[0],ESP
004071B5  . 83EC 20         SUB ESP,0x20
004071B8  . 53              PUSH EBX
004071B9  . 56              PUSH ESI
004071BA  . 57              PUSH EDI
004071BB  . 8965 E8         MOV DWORD PTR SS:[EBP-0x18],ESP
004071BE  . 8365 FC 00      AND DWORD PTR SS:[EBP-0x4],0x0
004071C2  . 6A 01           PUSH 0x1
004071C4  . FF15 B8804000   CALL DWORD PTR DS:[<&MSVCRT.__set_app_t  msvcrt.__set_app_type
004071CA  . 59              POP ECX
004071CB  . 830D 80FE4000   OR DWORD PTR DS:[0x40FE80],0xFFFFFFFF
004071D2  . 830D 84FE4000   OR DWORD PTR DS:[0x40FE84],0xFFFFFFFF
004071D9  . FF15 E9714000   CALL DWORD PTR DS:[0x4071E9]             7b45c014.00402520
004071DF  . 8B0D 7CFE4000   MOV ECX,DWORD PTR DS:[0x40FE7C]
004071E5  . 8908            MOV DWORD PTR DS:[EAX],ECX
004071E7  . FF15 20254000   CALL DWORD PTR DS:[0x402520]
004071ED    F0              DB F0
004071EE    5A              DB 5A                                    CHAR 'Z'
004071EF    B2              DB B2
004071F0    DF              DB DF
004071F1    8F              DB 8F
004071F2    E9              DB E9
004071F3    00              DB 00
004071F4  . 88A1 BC804000   OR BYTE PTR DS:[ECX+<&MSVCRT._adjust_fd
004071FA  . 8B00            MOV EAX,DWORD PTR DS:[EAX]
004071FC  . A3 88FE4000     MOV DWORD PTR DS:[0x40FE88],EAX
00407201  . E8 C3000000     CALL 7b45c014.004072C9
00407206  . 833D 509E4000   CMP DWORD PTR DS:[0x409E50],0x0
0040720D  .v75 0C           JNZ SHORT 7b45c014.0040721B
0040720F  . 68 C6724000     PUSH 7b45c014.004072C6
00407214  . FF15 C0804000   CALL DWORD PTR DS:[<&MSVCRT.__setusema  msvcrt.__setusermatherr
0040721A  . 59              POP ECX
0040721B  > E8 94000000     CALL 7b45c014.004072B4
00407220  . 68 0C904000     PUSH 7b45c014.0040900C
00407225  . 68 08904000     PUSH 7b45c014.00409008
0040722A  . E8 7F000000     CALL <JMP.&MSVCRT._getwche>
0040722F  . A1 74FE4000     MOV EAX,DWORD PTR DS:[0x40FE74]
00407234  . 8945 D8         MOV DWORD PTR SS:[EBP-0x28],EAX
00407237  . 8D45 D8         LEA EAX,DWORD PTR SS:[EBP-0x28]
```

Before

```
00407198  $ 55              PUSH EBP
00407199  . 0000            ADD BYTE PTR DS:[EAX],AL
0040719B  . 0000            ADD BYTE PTR DS:[EAX],AL
0040719D  . 0000            ADD BYTE PTR DS:[EAX],AL
0040719F  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071A1  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071A3  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071A5  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071A7  . 0000            ADD BYTE PTR DS:[EAX],AL
004071A9  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071AB  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071AD  . 0000            ADD BYTE PTR DS:[EAX],AL
004071AF  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071B1  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071B3  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071B5  . 0000            ADD BYTE PTR DS:[EAX],AL
004071B7  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071B9  . 0000            ADD BYTE PTR DS:[EAX],AL
004071BB  . 0000            ADD BYTE PTR DS:[EAX],AL
004071BD  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071BF  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071C1  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071C3  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071C5  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071C7  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071C9  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071CB  . 0000            ADD BYTE PTR DS:[EAX],AL
004071CD  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071CF  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071D1  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071D3  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071D5  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071D7  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071D9  . 0000            ADD BYTE PTR DS:[EAX],AL
004071DB  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071DD  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071DF  . 0000            ADD BYTE PTR DS:[EAX],AL
004071E1  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071E3  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071E5  . 0000            ADD BYTE PTR DS:[EAX],AL
004071E7  . 0000            ADD BYTE PTR DS:[EAX],AL
004071E9  ? 0000            ADD BYTE PTR DS:[EAX],AL
004071EB  ? 0000            ADD BYTE PTR DS:[EAX],AL
```

RunPE techniques are designed to evade AV mitigation methods.

Here are RunPE characteristics, as described in an Andromeda Bot Analysis by Infosec Institute:

- Unpack or decrypt the original EXE file in memory.
- Call CreateProcess on a target EXE using the CREATE_SUSPENDED flag. This maps the executable into memory and it's ready to execute, but the entry point hasn't executed yet.
- Next, Call GetThreadContext on the main thread of the newly created process. The returned thread context will have the state of all general-purpose registers. The EBX register holds a pointer to the Process

Environment Block (PEB), and the EAX register holds a pointer to the entry point of the innocent application. In the PEB structure, at an offset of eight bytes, is the base address of the process image.

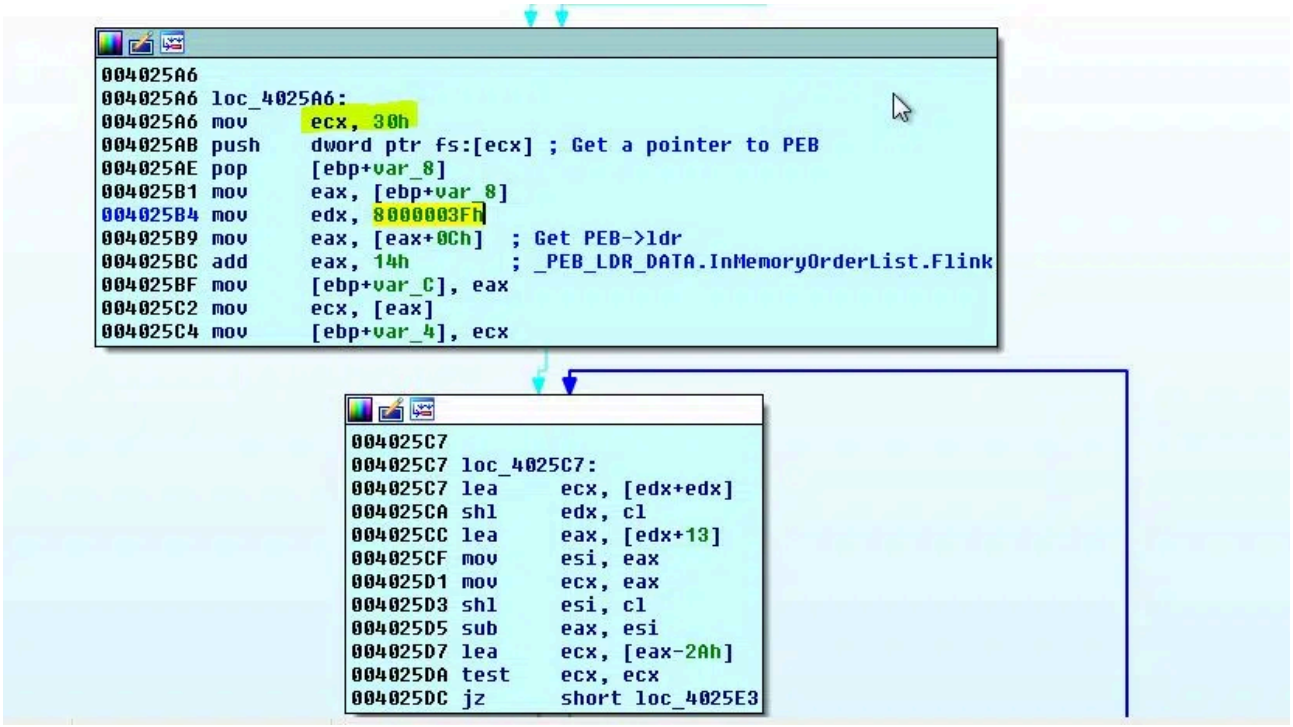
- Call NtUnmapViewOfSection to unmap and free up the virtual address space used by the new process.
- Call VirtualAllocEx to re-allocate the memory in the process' address space to the correct size (the size of the new EXE).
- Call WriteProcessMemory to write the PE headers and each section of the new EXE (unpacked in Step 1) to the virtual address location they expect to be (calling VirtualProtextEx to set the protection flags that each section needs).
- The loader writes the new base address into the PEB and calls SetThreadContext to point EAX to the new entry point.
- Finally, the loader resumes the main thread of the target process with ResumeThread and the windows PE loader will do its magic. The executable is now mapped into memory without ever touching the disk.

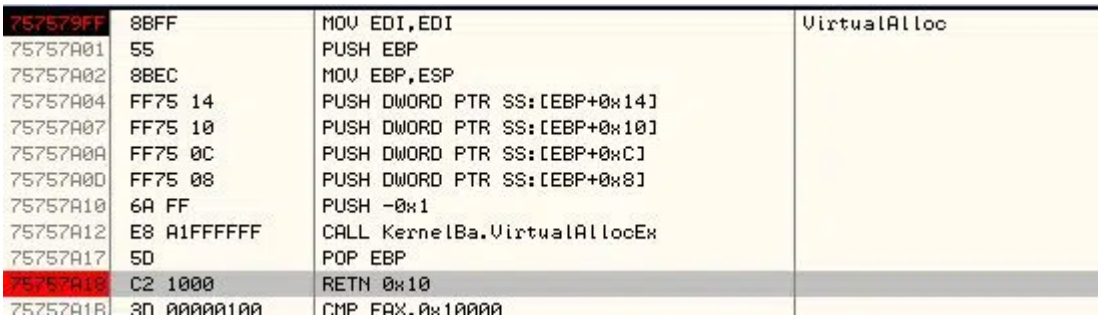
Also in our case, the packer decrypts the executable memory space and replaces previously encrypted memory with the functional code. The packer also updates the entry point to the new functional code start.

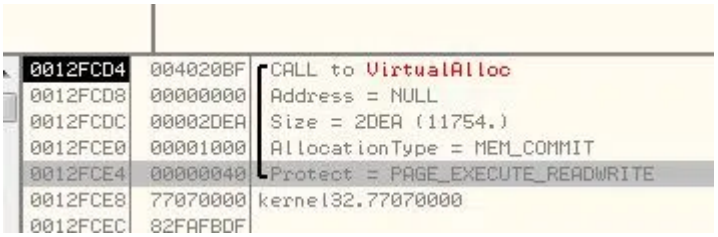
Forensic analysts will usually stop at this stage and dump the first layer decrypted code for further static analysis using different tools like IDA.

Based on the resemblance to RunPE methodology, we will execute the malware again, although now we set a breakpoint on VirtualAlloc functions (used to allocate memory). Other similar functions are VirtualAlloc, VirtualAllocEx, or ZwAllocateVirtualMemory – also part of the Process Hollowing/RunPE method) called to reserve some RWX memory.

We get the VirtualAlloc function from PEB->Kernel32.EAT



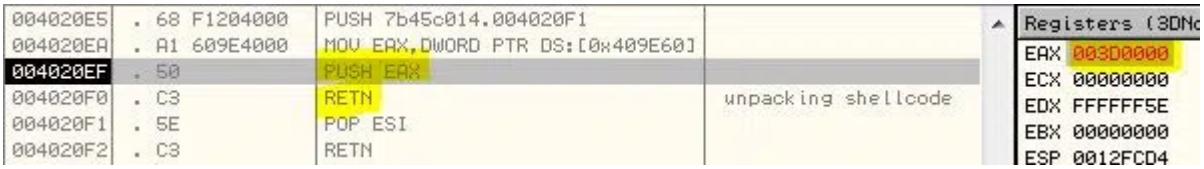


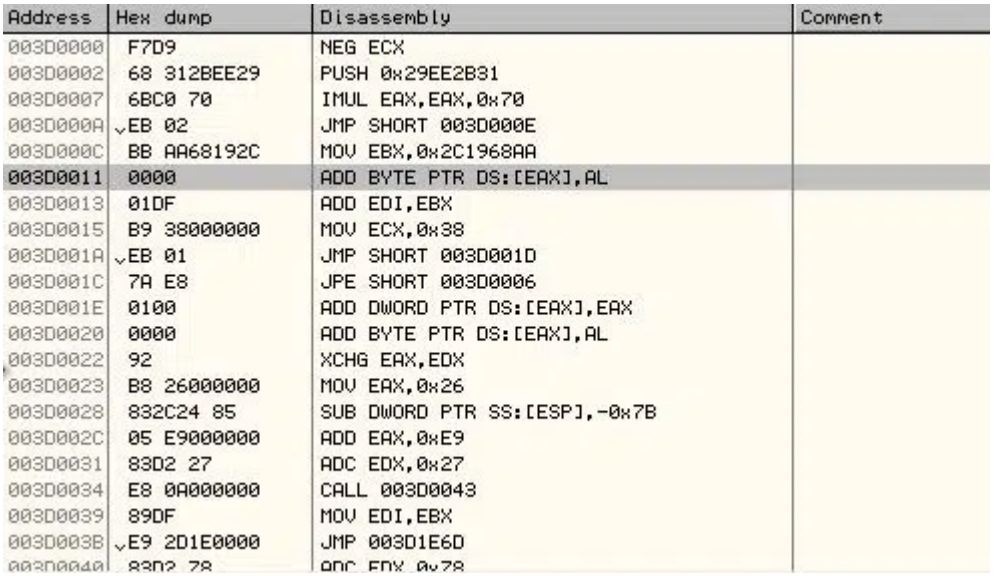


After identifying the RWE buffer address, we set a memory breakpoint on write to this buffer -> the written code is actually the unpacker/decode function.

After the unpacking function finishes execution, its execution is redirected to the first shellcode:

EAX address shellcode start = 0x003D0000





From inside the shellcode, VirtualAlloc is called again. We set a memory breakpoint on write to the new buffer one more time, get a new PE and are redirected to a second shellcode, the unpacked PE.

From this stage on, we get the regular Andromeda Loader which is described in detail by the Avast Threat Intelligence Team.

Conclusion

This article describes a single custom packer for Andromeda, one of the most popular malware delivery frameworks. Packers are a major concern for current security solutions. Packers allow attackers to penetrate network solutions, file scanning solutions and, in many cases, behavior or AI based solutions.

The use of custom packers will only increase, as will the need for in-memory solutions that can block these types of attacks.

A number of popular sandbox dynamic scanning services have some basic in-memory defenses, however these impose severe performance penalties. Moreover, they frequently are not even effective as many packers, such as in our case, include techniques to identify sandbox environments. Morphisec's Moving Target Defense based technology wins the malware packer battle without monitoring, hooking or using any other methods that affect endpoint performance.

Hash: 7b45c0141cca16fc14d4c81c653d4f22eb282cbbc4f913c9e830acf6e9d12b86

About the author

Roy Moshailov

Source: http://blog.morphisec.com/andromeda-tactics-analyzed