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

blog.morphisec.com/andromeda-tactics-analyzed

Posted by Roy Moshailov on March 13, 2017
Find me on:
Twitter

- Tweet
-

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. As the Andromeda attack chain has been described previously, this analysis 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?

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

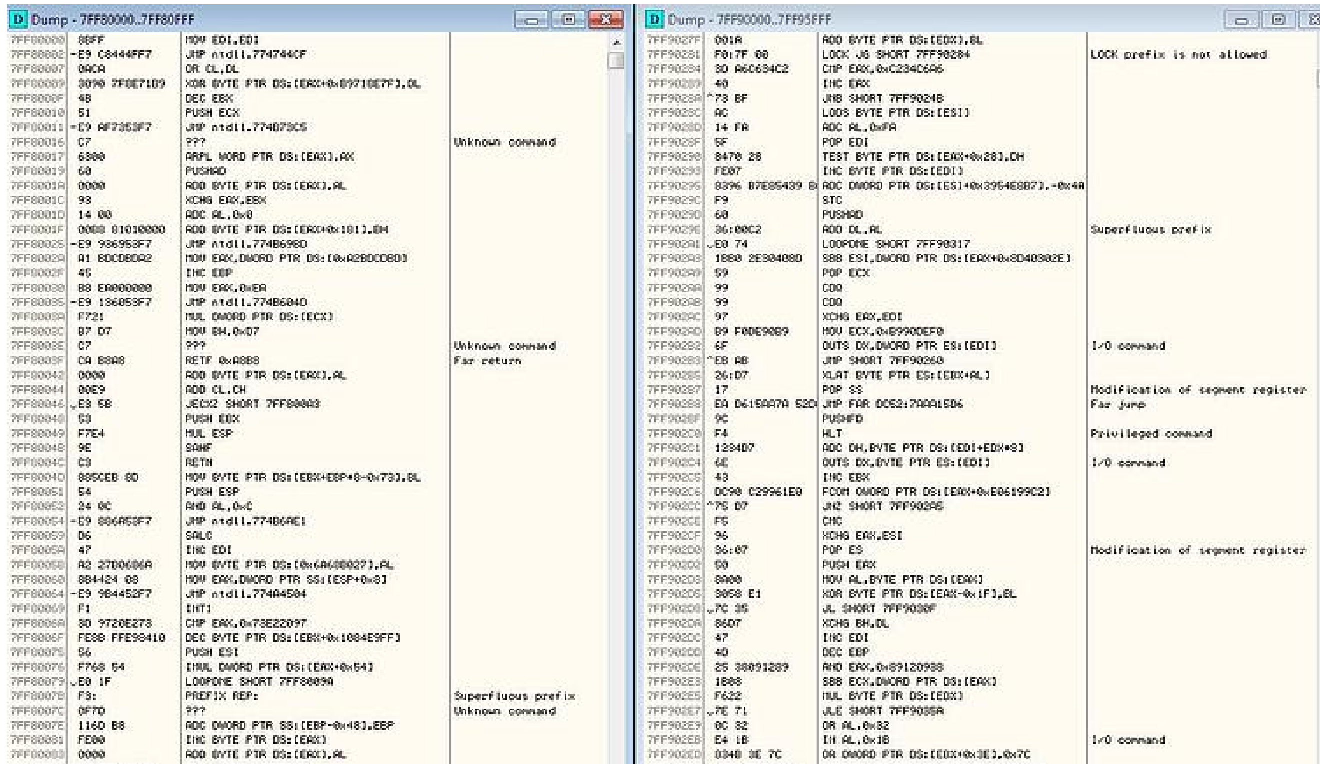
Determining a point in time for which we know the malicious code was already unpacked, we identify the use of ws2_32.dll (responsible for communication API). This means we can assume that the malicious code will start communication after it is unpacked. This is of high probability for downloaders or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

758C0000	000A1000	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 | Before

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

```
004025A6
004025A6 loc_4025A6:
004025A6 mov     ecx, 30h
004025AB push    dword ptr fs:[ecx] ; Get a pointer to PEB
004025AE pop     [ebp+var_8]
004025B1 mov     eax, [ebp+var_8]
004025B4 mov     edx, 8000003Fh
004025B9 mov     eax, [eax+0Ch]  ; Get PEB->ldr
004025BC add     eax, 14h        ; _PEB_LDR_DATA.InMemoryOrderList.Flink
004025BF mov     [ebp+var_C], eax
004025C2 mov     ecx, [eax]
004025C4 mov     [ebp+var_4], ecx
```

```
004025C7
004025C7 loc_4025C7:
004025C7 lea     ecx, [edx+edx]
004025CA shl     edx, cl
004025CC lea     eax, [edx+13]
004025CF mov     esi, eax
004025D1 mov     ecx, eax
004025D3 shl     esi, cl
004025D5 sub     eax, esi
004025D7 lea     ecx, [eax-2Ah]
004025DA test    ecx, ecx
004025DC jz      short loc_4025E3
```

```
757579FF  8BFF        MOV EDI,EDI                          VirtualAlloc
75757A01  55          PUSH EBP
75757A02  8BEC        MOV EBP,ESP
75757A04  FF75 14     PUSH DWORD PTR SS:[EBP+0x14]
75757A07  FF75 10     PUSH DWORD PTR SS:[EBP+0x10]
75757A0A  FF75 0C     PUSH DWORD PTR SS:[EBP+0xC]
75757A0D  FF75 08     PUSH DWORD PTR SS:[EBP+0x8]
75757A10  6A FF       PUSH -0x1
75757A12  E8 A1FFFFFF CALL KernelBa.VirtualAllocEx
75757A17  5D          POP EBP
75757A18  C2 1000     RETN 0x10
75757A1B  3D 00000100 CMP EAX,0x10000
```

```
0012FCD4  004020BF  CALL to VirtualAlloc
0012FCD8  00000000  Address = NULL
0012FCDC  00002DEA  Size = 2DEA (11754.)
0012FCE0  00001000  AllocationType = MEM_COMMIT
0012FCE4  00000040  Protect = PAGE_EXECUTE_READWRITE
0012FCE8  77070000  kernel32.77070000
0012FCEC  82FAFBDF
```

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.

```
00401590  . 881F          MOV BYTE PTR DS:[EDI],BL
00401592  .v74 05         JE SHORT 7b45c014.00401599
00401594  . 8D0C10        LEA ECX,DWORD PTR DS:[EAX+EDX]
00401597  . 23D1          AND EDX,ECX
00401599  > 8BC8          MOV ECX,EAX
0040159B  . 8D5C12 C1     LEA EBX,DWORD PTR DS:[EDX+EDX-0x3F]
0040159F  . 33CA          XOR ECX,EDX
004015A1  . 0BC1          OR EAX,ECX
004015A3  . 8BD0          MOV EDX,EAX
004015A5  . 83F2 27       XOR EDX,0x27
004015A8  .v74 07         JE SHORT 7b45c014.004015B1
004015AA  . 8BCB          MOV ECX,EBX
004015AC  . 83F1 37       XOR ECX,0x37
004015AF  . 0BD9          OR EBX,ECX
004015B1  > 8BC8          MOV ECX,EAX
004015B3  . 83E1 DC       AND ECX,0xFFFFFFDC
004015B6  . D3F8          SAR EAX,CL
004015B8  . 8BD0          MOV EDX,EAX
004015BA  . 0FAFD3        IMUL EDX,EBX
004015BD  . 85D2          TEST EDX,EDX
004015BF  .v74 07         JE SHORT 7b45c014.004015C8
004015C1  . 8BC8          MOV ECX,EAX
004015C3  . 83F1 CC       XOR ECX,0xFFFFFFCC
004015C6  . D3F8          SAR EAX,CL
004015C8  > 8D0C1B        LEA ECX,DWORD PTR DS:[EBX+EBX]
004015CB  . 85C9          TEST ECX,ECX
004015CD  .v74 07         JE SHORT 7b45c014.004015D6
004015CF  . 8BD3          MOV EDX,EBX
004015D1  . 83F2 2C       XOR EDX,0x2C
004015D4  . 23DA          AND EBX,EDX
004015D6  > 8BD0          MOV EDX,EAX
004015D8  . 8BCB          MOV ECX,EBX
004015DA  . D3FA          SAR EDX,CL
004015DC  . 03D0          ADD EDX,EAX
004015DE  . 8BC3          MOV EAX,EBX
004015E0  . 83F0 21       XOR EAX,0x21
004015E3  . 0BD0          OR EDX,EAX
004015E5  . 8B45 14       MOV EAX,DWORD PTR SS:[EBP+0x14]
004015E8  . 4B            DEC EBX
004015E9  . 47            INC EDI
004015EA  . 48            DEC EAX
004015EB  . 8945 14       MOV DWORD PTR SS:[EBP+0x14],EAX
004015EE  .^0F85 DAFEFFFF JNZ 7b45c014.004014CE
004015F4  > 8BC2          MOV EAX,EDX
004015F6  . F7D8          NEG EAX
004015F8  . C1E0 02       SHL EAX,0x2
004015FB  . 2BC2          SUB EAX,EDX
004015FD  . 8D0CC0        LEA ECX,DWORD PTR DS:[EAX+EAX*8]
00401600  . D3FA          SAR EDX,CL
00401602  . 8D4B 26       LEA ECX,DWORD PTR DS:[EBX+0x26]
00401605  . 85C9          TEST ECX,ECX
00401607  .v74 05         JE SHORT 7b45c014.0040160E
00401609  . BA C1FFFFFF   MOV EDX,-0x3F
0040160E  > 8D045B        LEA EAX,DWORD PTR DS:[EBX+EBX*2]
00401611  . 8D04C0        LEA EAX,DWORD PTR DS:[EAX+EAX*8]
00401614  . D1E0          SHL EAX,1
00401616  .v74 05         JE SHORT 7b45c014.0040161D
00401618  . 8D0C1A        LEA ECX,DWORD PTR DS:[EDX+EBX]
0040161B  . 0BD1          OR EDX,ECX
0040161D  > 8D0452        LEA EAX,DWORD PTR DS:[EDX+EDX*2]
00401620  . 8D04C0        LEA EAX,DWORD PTR DS:[EAX+EAX*8]
00401623  . D1E0          SHL EAX,1
00401625  . 2BC2          SUB EAX,EDX
00401627  .v74 07         JE SHORT 7b45c014.00401630
00401629  . 8BCB          MOV ECX,EBX
0040162B  . 83E1 32       AND ECX,0x32
```

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

EAX address shellcode start = 0x003D0000

004020E5	. 68 F1204000	PUSH 7b45c014.004020F1	
004020EA	. A1 609E4000	MOV EAX,DWORD PTR DS:[0x409E60]	
004020EF	. 50	PUSH EAX	
004020F0	. C3	RETN	unpacking shellcode
004020F1	. 5E	POP ESI	
004020F2	. C3	RETN	

Registers (3DNo
EAX 003D0000
ECX 00000000
EDX FFFFFF5E
EBX 00000000
ESP 0012FCD4

Address	Hex dump	Disassembly	Comment
003D0000	F7D9	NEG ECX	
003D0002	68 312BEE29	PUSH 0x29EE2B31	
003D0007	6BC0 70	IMUL EAX,EAX,0x70	
003D000A	EB 02	JMP SHORT 003D000E	
003D000C	BB AA68192C	MOV EBX,0x2C1968AA	
003D0011	0000	ADD BYTE PTR DS:[EAX],AL	
003D0013	01DF	ADD EDI,EBX	
003D0015	B9 38000000	MOV ECX,0x38	
003D001A	EB 01	JMP SHORT 003D001D	
003D001C	7A E8	JPE SHORT 003D0006	
003D001E	0100	ADD DWORD PTR DS:[EAX],EAX	
003D0020	0000	ADD BYTE PTR DS:[EAX],AL	
003D0022	92	XCHG EAX,EDX	
003D0023	B8 26000000	MOV EAX,0x26	
003D0028	832C24 85	SUB DWORD PTR SS:[ESP],-0x7B	
003D002C	05 E9000000	ADD EAX,0xE9	
003D0031	83D2 27	ADC EDX,0x27	
003D0034	E8 0A000000	CALL 003D0043	
003D0039	89DF	MOV EDI,EBX	
003D003B	E9 2D1E0000	JMP 003D1E6D	
003D0040	83D2 78	ADC EDX,0x78	

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

Resources

http://resources.infosecinstitute.com/andromeda-bot-analysis/#article

http://www.iosrjournals.org/iosr-jce/papers/Vol16-issue1/Version-1/L016117177.pdf

https://www.botconf.eu/wp-content/uploads/2015/12/OK-P07-Jose-Esparza-Travelling-to-the-far-side-of-Andromeda-2.pdf

Beat hackers at their own game!

Read our free ebook

"Moving Target Attacks vs. Moving Target Defense"

DOWNLOAD NOW!

Contact SalesInquire via Azure