Introduction of a PE file extractor for various situations

Reference of the second structure of the second struct

May 25, 2022 • tool, malware

During a malware analysis, you may encounter the situation where a next stage payload is loaded or injected into another process. When this is the case, usually a raw PE file gets decrypted in memory that is used to build the memory module. The trick is to find the procedure which decrypts the raw file with a debugger and dump the memory region which contains this payload. This allows you to easily extract the original PE file from the dump for further analysis. Thus, you usually don't need to perform contorts like dumping the memory module of the payload and rebuild its import address table.

When I don't extract a payload by hand, I've always used the combination of a debugger and <u>PEExtract</u>. In most cases, PEExtract works correctly and grabs the PE payloads. However, it has a few shortcomings like no support for signed PE files and its development also stopped in 2007. That's why I've created <u>pe_extract.py</u> to overcome those issues. While there are already scripts like this (e.g. <u>pe-carv</u>), I've added a few improvements and features:

- Multiple file scan support (e.g. for automatically created memory dumps)
- Skip likely incomplete page sized PEs (for automatically created memory dumps)
- · Support for XORed PE files

Use case 1 - Extract payload(s) from a manual memory dump

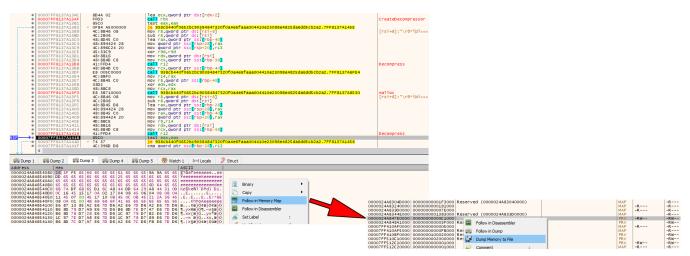
Typical usage: python pe_extract.py <FilePathToDump> (--extract-xored)

As an example, let's take a Cobalt Strike loader I dubbed **FlyingTurtleLoader** after its internal name **FlyingTurtle**. The initial 64-bit DLL is a loader for a first stage EXE which in turn is the final loader for the Cobalt Strike beacon. Each stage is base64 encoded, MSZIP compressed and the first stage additionally XOR encrypted.

Malware (SHA-256):

938cb440f0652bc90384847320f0a4e6faaa004410e23098e4825da6dd5cb2a2

As initially described, when you analyze a loader or multi-stage malware sample, there's usually a raw PE file written to a memory buffer at some point. The following <u>x64dbg</u> screenshots show the **XOR encrypted first stage EXE payload** in the Dump 3 window:



At this point, you can already go to the memory region that contains the encrypted payload (*Follow in Memory Map*) and save it to disk (*Dump Memory to File*). You don't even need to find the final routine which decrypts the payload (XOR bytes with 0x65), as this can be done by pe_extract.py . You just need to run the script with the --extract-xored argument and it extracts the decrypted final loader. According to its PDB path and the empty Cobalt Strike config data, this is a test tool:

C:\test\FlyingTurtle\x64\Release\FlyingTurtle.pdb

Use case 2 - Extract payload(s) from on-disk file(s)

Typical usage: python pe_extract.py <FilePathToDump> --extract-xored (--extract-overlays)

Sometimes, there is malware that keeps one or more payloads unencrypted in its PE sections. Or the payloads are encrypted with a simple XOR algorithm. When this is the case, you can easily extract them and make a initial assessment what the purpose of the malware might be by looking at them. For PE extration, this is the best case because the embedded files can be pulled out reliably.

Again, as an example, we use a Cobalt Strike loader I dubbed **K32Loader** according to this specific API function that it uses called K32GetProcessImageFileNamew. It disguises itself as a legitimate looking Windows file and keeps other legit signed files along with the actual Cobalt Strike beacon loader in its resources section. The legit files do not serve any purpose except to make the malware look less suspicious. The final beacon loader DLL is run by a shellcode created with the help of <u>sRDI</u>. The final loader contains the AES encrypted Cobalt Strike beacon.

Malware (SHA-256):

2016258b9aea66a204a4374aeef2d5f7a0c6857ee92491a12440ce8487aaf938 (Sample 1) 6344b05fe37649d87617e5ba26cd90a3d9b4bff28904df89a6b9028265c9db65 (Sample 2) e8eb5597550ba347114a67cb5173c389aeb3addff8f2f5eaefb634e18508526a (Sample 3)

The sample's embedded files are described in the following table:

Sample	Stages	Initial DLL masked as	Initial DLL signed files	1st stage DLL masked as	1st stage signed files	Final DLL loader name
1	2	Windows MsMpRes.dll	Windows MsMpRes.dll	Not masked (internal name final- load.dll)	-	Protections- Remover.dll
2	1	Windows userenv.dll	Windows MsMpRes.dll	->	->	final- load.dll
3	2	Windows userenv.dll	Sophos ICManagement.dll. DetectionFeedback.dll	Not masked	Windows MessaqinqDataModel2.dll, midimap.dll	astraGem.dll

The following EXE Explorer screenshot shows the embedded files in sample 3:

Headers	Sections	Directories	Exports	Imports	Resource	ces Strings	Load Co	nfig Debug	Exceptions Hex V	iew
BINARY	(*							Property	Value	
100								Name	200	
200								Type	"BINARY"	
····· 300								Language	1033	
								Code page	0	
								RVA	0x000CDC38	
								Offset	0x000C9038	
								Size	1533321 B	
								Entropy	6.746	
								Entropy MD5	6.746 3616DF9AF4C634	484C36138C
2,508,985 esource View										484C36138C
	wer	203 0405	0607 0809	0A0B 0C0D	0E0F 012	23456789ABC				
	wer 2001 02			0A0B 0C0D 81C1 800B		23456789ABC YI ÈH Á	DEF			
esource Viev	wer 0001 02 0 E800 00	00 0059	4989 C848	81C1 800B	0000 è.		DEF			
esource Viev oxooooo	wer 0001 02 0 E800 00 0 BA00 10 0 0000 50	000 0059 09D E149 548 89E6	4989 C848 81C0 8065 4883 E4F0	81C1 800B 1700 41B9 4883 EC30	0000 è. 0400 º. C744	ҮІ ЀН А́ . а́І À еА /Н ӕН ӓठ҄Н ì	DEF 1 0ÇD			
esource View 0x00000 0x00001 0x00002 0x00003	Wer 8001 02 10 E800 00 10 BA00 10 10 0000 50 10 2420 05	000 0059 09D E149 548 89E6 500 0000	4989 C848 81C0 8065 4883 E4F0 E805 0000	81C1 800B 1700 41B9 4883 EC30 0048 89F4	0000 è. 0400 º. C744 5EC3 \$	ҮІ ЀН А́ . а́І À еА /Н ӕН ӓठ҃Н Ì ѐН	DEF 1 0ÇD ô^Â			
esource View	wer 2001 02 2001 02 2000 00 2000 56 2420 05 2420 05 2400 00	000 0059 09D E149 548 89E6 500 0000 524 0844	4989 C848 81C0 8065 4883 E4F0 E805 0000 894C 2420	81C1 800B 1700 41B9 4883 EC30 0048 89F4 4C89 4424	0000 è. 0400 º. C744 5EC3 \$ 1889 H	ҮІ ЀН А́ . а́І À еА /Н а:Н äðH ì èH \\$.D L\$ L D	DEF 1 0¢D ô^Ã \$.			
esource View 9x98089 9x98081 9x98982 9x88983 9x88884 9x88884 9x88885	Wer	000 0059 09D E149 548 89E6 500 0000 524 0844 055 5657	4989 C848 81C0 8065 4883 E4F0 E805 0000 894C 2420 4154 4155	81C1 8008 1700 4189 4883 EC30 0048 89F4 4C89 4424 4156 4157	0000 è. 0400 º. C744 5EC3 \$ 1889 H 488D T\$	YI ÈH Á . áI À eA /H æH äðH Ì èH \\$.D L\$ L D .UVWATAUAVA	DEF 1 0¢D ô^Ã \$. WH			
executive executive 0x00000 0x00001 0x00002 0x00002 0x00003 0x00004 0x00005 0x00005	wer 2001 02 2001 02 2001 02 2000 02 0000 56 00 2420 05 00 2420 05 00 4889 50 00 5424 16 00 5424 16 00 5424 16	000 0059 09D E149 548 89E6 500 0000 524 0844 055 5657 048 81EC	4989 C848 81C0 8065 4883 E4F0 E805 0000 894C 2420 4154 4155 6001 0000	81C1 8008 1700 4189 4883 EC30 0048 89F4 4C89 4424 4156 4157 4533 EDC7	0000 è. 0400 º. C744 5EC3 \$ 1889 H 488D T\$ 45D0 1\$	ҮІ ЀН А́ . а́І À еА /Н а:Н äðH ì èH \\$.D L\$ L D	DEF 1 0ÇD ô^Ã \$. WH ÇEĐ			

The resources named 100 and 300 are the signed ICManagement.dll and DetectionFeedback.dll files from Sophos. The resource named 200 is the reflective loader shellcode with the embedded Cobalt Strike loader named astraGem.dll. This file contains the additional signed Windows files MessagingDataModel2.dll and midimap.dll in its resource section.

When we use pe_extract.py on each file, we get all the unencrypted embedded files except for the Cobalt Strike beacon as it's encrypted.

Beacon domains:

dns.minimephotos[.]co.uk
orchardstanks[.]com
bellennium[.]com
energy-sciences[.]org

Use case 3 - Extract payload(s) from automatically created memory dumps

Typical usage: python pe_extract.py <FolderPathToDumps> (--extract-overlays) (--extract-all)

Nowadays, more and more sandboxes contain the ability to scan for malware in memory. Usually, this is done by dumping memory images to disk and scan those for any malware patterns. Based on the quality of the mechanism that triggers the dump procedure, you have a bigger or smaller amount of dump files that hopefully contain one or more (decrypted) payloads. One such a sandbox is Virustotal's **Zenbox** that is capable of creating memory dumps during a sample analysis.

As an example, we use a Matanbuchus sample.

Malware (SHA-256):

d9e6395917a1d1103c40f710310de0cf64c370d167def378e9b88f3af247a1b0

It's a **signed MSI** file disguised as a Symantec Protection Engine installer and contains two files. The first file named **notify.vbs** shows a fake error message when run. The second file named **main.dll** is a signed Matanbuchus loader disguised as a Visual Studio installer. The signatures are as follows (MSI file on the left, **main.dll** on the right):

Digital Signature Details	5	?	×	Digital Signature Details	?	×
	nature Information gnature is OK.			General Advanced Digital Signature Information This digital signature is OK.		
Signer information — Name: E-mail: Signing time:	SERVICE ABOVE LTD Not available Not available	View Certifica	əte	Signer information Name: Dream Key Comics Inc. E-mail: diadelfom@hotmail.com Signing time: Monday, May 23, 2022 1:30:0	04 AM ew Certificate	
Countersignatures	E-mail address:	Timestamp Details		Countersignatures Name of signer: E-mail address: Timesta	mp Details	
			ОК		0	к

The following Virutotal screenshot shows the option to download the memdump of this file:

2	① 2 security vendors and no sandboxes flagged this file as malicious				$_{\odot}$ \simeq \pm 12
 756 ? Community ✓ Score 	d9e6395917a1d1103c40f710310de0cf64c370d167def378e9b88f3af247a1b0 C:Windows\Installer3ff9b.msi checks-network-adapters direct-cpu-clock-access malware msi runtime-modules sign	ed		22-05-25 08:16:12 UTC nours ago	MSI
DETECTION	DETAILS RELATIONS BEHAVIOR CONTENT SUBMISSION	s commu	NITY		
Screenshots ①					
Image: Second secon			Memdump		earch similar behavior Q
Behavior Tags ①					
checks-usb-bus detect-	debug-environment long-sleeps		7		
Network Communic	ation ①				
HTTP Requests + http://azureteleme	etry.xyz/cAUtfkUDaptk/ZRSeiy/requets/index.php				
DNS Resolutions					
+ statsazure.xyz					
+ arc.msn.com					
 + azuretelemetry.xy 	Z				

Unfortunately, this feature is limited to enterprise accounts.

In this extraction case, just provide the folder path which contains the memory dumps as an argument and pe_extract.py scans each file for embedded PEs. When we do that, we get five extracted DLLs. From the FDMP files, we have two versions of main.dll with the signature information cut. From the SDMP files, we have an additional version of main.dll with a cut signature and two versions of the main Matanbuchus module.

The first main module file is the raw PE decrypted during the loading routine (see introduction). It contains the usual Matanbuchus strings in cleartext:

Agent.ADNJ Agent.Matanbuchus

B:\Loader\Matanbuchus\Main module\Belial project\MatanbuchusLoader\MatanbuchusLoaderFiles\Matanbuchus\json.hpp

The second main module file is a memory dump of the raw PE that contains additional (decrypted) strings:

azuretelemetry.xyz /cAUtfkUDaptk/ZRSeiy/requets/index.php statsazure.xyz 23.227.196.227 87.236.146.125 icLJkdnBDX qmG Running exe Starting the exe with parameters High start exe RunDll32 & Execute Regsvr32 & Execute Run CMD in memory Run PS in memory MemLoadDllMain || MemLoadExe MemLoadShellCode MemLoadShellCode #2 Running dll in memory #2 (DllRegisterServer) Running dll in memory #3 (DllInstall(Install)) Running dll in memory #3 (DllInstall(Unstall)) Crypt update & Bots upgrade Uninstall Gp Ρk vМ Vs bΝ Jb NSeyDX Los wP6 cBF Vz 3m7x ELj E06 Q6X6 tW acG 3CEk DS2x Fto f1da 3fe11 zkC7

These strings were obfuscated at compile time. They get only revealed when the string decryption procedures are executed during the (C++) initialization phase. You can see the decrypted strings in the .data section when you set a breakpoint on DllMain and run the raw PE sample in a debugger. This obfuscation method was introduced several years ago in a project called <u>ADVobfuscator</u>.

We can see a few additions by examining those strings and comparing them to the <u>previous version</u>. A new memory shellcode loading mechanism (<u>MemLoadShellCode #2</u>) appears to be the most obvious new feature.

We can also see some strings that were decrypted using a different method:

Agriel v1.4.0 DAN03 %02X-%02X-%02X-%02X-%02X-%02X %USERDOMAIN% User Admin kernel32 IsWow64Process 32 Bit 64 Bit %LOGONSERVER% POST HTTP/1.1 Host: User-Agent: Windows-Update-Agent/11.0.10011.16384 Client-Protocol/2.0 Content-Length: Content-Type: application/x-www-form-urlencoded Accept-Language: en-US

Matanbuchus domains:

statsazure[.]xyz
azuretelemetry[.]xyz

Conclusion

With pe_extract.py you have a tool that can save you some time during a malware analysis session and make things easier. It speeds up the analysis process a bit and can help make a first assessment of an unkown malware. It can also be useful to create Yara rules, especially when the retrieved PE file comes from a memory dump and contains decrypted data. This information is a goldmine for in-memory detection signatures.

Script download

pe.extract.py can be found on my Github page: pe_extract