

Malware Analysis —Manual Unpacking of Redaman

 jonahacks.medium.com/malware-analysis-manual-unpacking-of-redaman-ec1782352cfb

Jon

January 26, 2022



Jon

Jan 26

.

6 min read

In this post, we are looking to manually unpack the sample called Redaman, which is a banking trojan. Some of its capabilities include:

- Monitor browser activity,
- Downloading files to the infected host
- Keylogging activity
- Capture screen shots and record video of the Windows desktop
- Collecting and exfiltrating financial data, specifically targeting Russian banks
- Smart card monitoring
- Shutting down the infected host
- Altering DNS configuration through the Windows host file
- Retrieving clipboard data
- Terminating running processes
- Adding certificates to the Windows store

| [Info from Unit42 Analysis.](#)

What makes this sample unique and an excellent training sample to practice manual unpacking is because this sample performs a fairly simple packing process: PE overwrite and a secondary DLL Injection.

Self-Injection, or in this example the PE Overwrite occurs when the malware allocates a “stub” in itself, transfers to that stub address, allocates that stub area and write whatever malicious content it needs to in there, and then changes the permissions, and then run from that overwritten area.

| [A Better Explanation.](#)

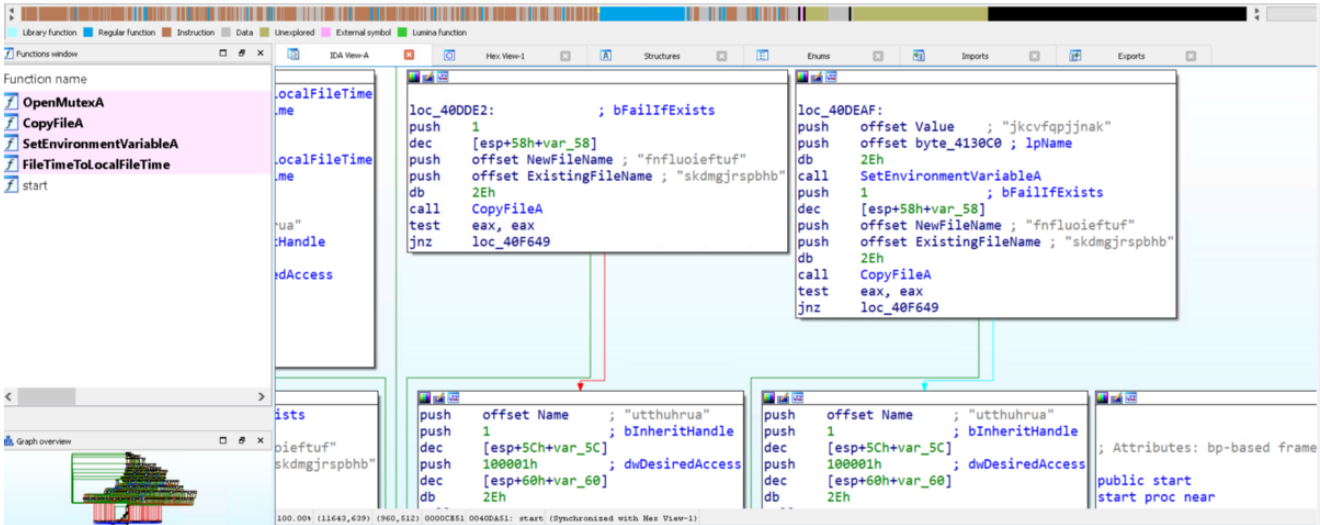
Packed Sample

We can identify this file as packed based on a number of info:

High level entropy on the main file with PEStudio:

property	value
md5	DF725667733410F1A023A76D36FCBD31
sha1	F7DEC59AEF9CC9E5C13827CF7786D05819170F1B
sha256	CEB8EFB3A3EB1085C61BBA4B0A77D1ACA1F7B10511497E1521135F18
md5-without-overlay	n/a
sha1-without-overlay	n/a
sha256-without-overlay	n/a
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00
first-bytes-text	M Z @
file-size	326656 (bytes)
size-without-overlay	n/a
entropy	7.164
imphash	80D3242711EC48AC212E70C55619D01D
signature	n/a
entry-point	50 8D 15 BD 56 44 00 42 89 E2 89 2C 24 89 E5 6A 05 83 C4 B0 66 81 EA
file-version	n/a
description	n/a
file-type	executable
cpu	32-bit
subsystem	GUI
compiler-stamp	0x514C152D (Fri Mar 22 08:24:13 2013)
debugger-stamp	n/a
resources-stamp	empty
exports-stamp	n/a
version-stamp	n/a

Checking in IDA, we see that first there is some obfuscation, barely any functions, and only a small amount of analyzed code (blue bar at the top of screenshot)



PE Overwrite

To start we look for where virtual allocation of memory takes place which in this case it is the function VirtualAlloc. The return value for VirtualAlloc is the base address of the allocated region. Which we can find in the EAX register. We put a breakpoint at the return of the function:

•	75B1E970	8BFF	mov edi,edi	VirtualAlloc
•	75B1E972	55	push ebp	
•	75B1E973	8BEC	mov ebp,esp	
•	75B1E975	51	push ecx	
•	75B1E976	51	push ecx	
•	75B1E977	8B45 0C	mov eax,dword ptr ss:[ebp+C]	
•	75B1E97A	8945 F8	mov dword ptr ss:[ebp-8],eax	
•	75B1E97D	8B45 08	mov eax,dword ptr ss:[ebp+8]	
•	75B1E980	8945 FC	mov dword ptr ss:[ebp-4],eax	
•	75B1E983	56	push esi	
•	75B1E984	85C0	test eax,eax	
•	75B1E986	✓ 74 0C	je kernelbase.75B1E994	
•	75B1E988	3B05 9856BD75	cmp eax,dword ptr ds:[75BD5698]	
•	75B1E98E	✓ 0F82 41870300	jb kernelbase.75B570D5	
•	75B1E994	FF75 14	push dword ptr ss:[ebp+14]	
•	75B1E997	8B45 10	mov eax,dword ptr ss:[ebp+10]	
•	75B1E99A	33F6	xor esi,esi	
•	75B1E99C	83E0 C0	and eax,FFFFFFC0	
•	75B1E99F	50	push eax	
•	75B1E9A0	8D45 F8	lea eax,dword ptr ss:[ebp-8]	
•	75B1E9A3	50	push eax	
•	75B1E9A4	56	push esi	
•	75B1E9A5	8D45 FC	lea eax,dword ptr ss:[ebp-4]	
•	75B1E9A8	50	push eax	
•	75B1E9A9	6A FF	push FFFFFFFF	
•	75B1E9AB	FF15 3887BD75	call dword ptr ds:[<&ZwAllocateVirtualMe	
•	75B1E9B1	85C0	test eax,eax	
•	75B1E9B3	✓ 78 0A	js kernelbase.75B1E9BF	
•	75B1E9B5	8B75 FC	mov esi,dword ptr ss:[ebp-4]	
•	75B1E9B8	8BC6	mov eax,esi	
•	75B1E9BA	5E	pop esi	
•	75B1E9BB	C9	leave	
•	75B1E9BC	C2 1000	ret 10	

This will help us to see how many times and where memory is being virtually allocated.

We also want to add a breakpoint at the entry of VirtualProtect, this is where the protections and access is changed. The first argument to VirtualProtect will be the address to the memory section which protections will be changed. It needs to change the protections to get the permission to write

75B1DF17	CC	int3	
75B1DF18	CC	int3	
75B1DF19	CC	int3	
75B1DF1A	CC	int3	
75B1DF1B	CC	int3	
75B1DF1C	CC	int3	
75B1DF1D	CC	int3	
75B1DF1E	CC	int3	
75B1DF1F	CC	int3	
75B1DF20	8BFF	mov edi,edi	VirtualProtect
75B1DF22	55	push ebp	
75B1DF23	8BEC	mov ebp,esp	
75B1DF25	51	push ecx	
75B1DF26	51	push ecx	
75B1DF27	8B45 0C	mov eax,dword ptr ss:[ebp+C]	
75B1DF2A	56	push esi	
75B1DF2B	FF75 14	push dword ptr ss:[ebp+14]	

Now we run the debugger until we hit our second breakpoint (First one is always on the entry point of the file).

The screenshot shows a debugger window with the following assembly code:

```

75B1E98C  C2 1000  ret 10
75B1E98F  8BC8    mov ecx, eax
75B1E991  E8 4A29FEFF  call kernelbase.75B01310
75B1E996  EB F0    jmp kernelbase.75B1E9B8
75B1E998  CC      int3
75B1E999  CC      int3
75B1E99A  CC      int3
75B1E99B  CC      int3
75B1E99C  CC      int3
75B1E99D  CC      int3
75B1E99E  CC      int3
75B1E99F  CC      int3
75B1E9A0  CC      int3
75B1E9A1  CC      int3
75B1E9A2  CC      int3
75B1E9A3  CC      int3
75B1E9A4  CC      int3
75B1E9A5  CC      int3
75B1E9A6  CC      int3
75B1E9A7  CC      int3
75B1E9A8  CC      int3
75B1E9A9  CC      int3
75B1E9AA  CC      int3
75B1E9AB  CC      int3
75B1E9AC  CC      int3
75B1E9AD  CC      int3
75B1E9AE  CC      int3
75B1E9AF  CC      int3
75B1E9B0  8BFF    mov edi, edi
75B1E9B2  55      push ebp
75B1E9B4  8BEC    mov ebp, esp

```

The memory dump shows the following data:

Address	Hex	ASCII
00030000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00030010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00030020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00030030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00030040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00030050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00030060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00030070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

From the screenshot we hit the breakpoint, we right click on the address in EAX and follow in dump. We can see that at address 00030000 there is a large amount of zeros where VirtualAlloc has allocated space.

Continuing on we hit the return of VirtualAlloc again at address 021B0000. So we know that VirtualAlloc is used at address 00030000 and 021B0000. Our next hit is the entry of VirtualProtect:

The screenshot shows a debugger window with assembly code for the VirtualProtect function. The EAX register is highlighted, showing the value 00400000. Below the assembly code, a memory dump is visible, showing the contents of the EAX register at address 021B0000. The dump shows the hex values 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00, which correspond to the ASCII string "MZ.....yy.....".

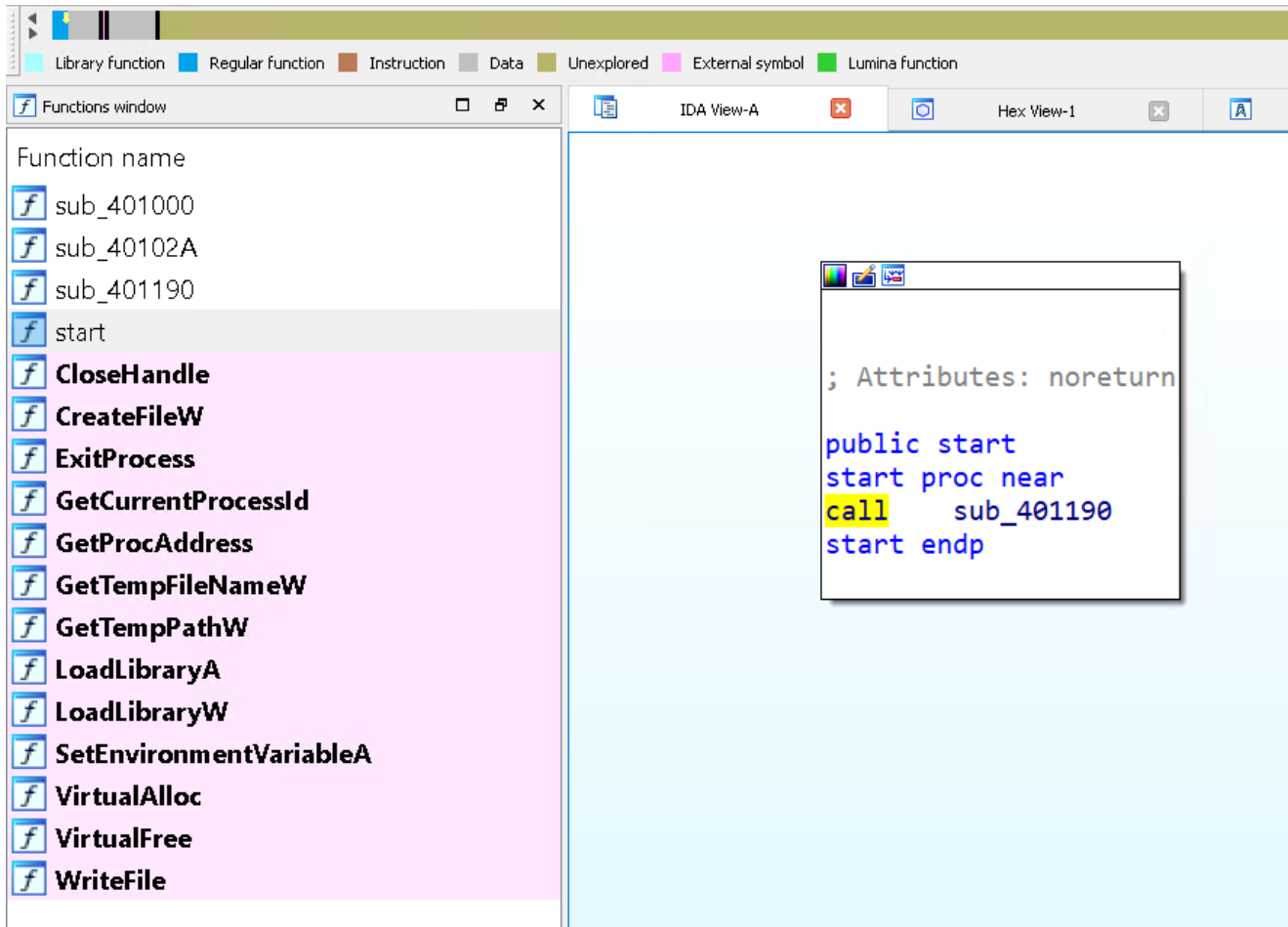
Checking the first argument passed to VirtualProtect in the EAX register we can automatically see that instead of zeros we now have what looks to be an exe (the MZ or hex 4D 5A gives it away). At this point we now have gotten to the point in the malware where not only has the main payload been unpacked but now it is ready to have its permissions and access changed, we now right click on the dump and choose the “Follow in Memory Map”

Address	Size	Info	Content	Type	Protection	Initial
003DA000	00026000	Reserved (00200000)		PRV		-RW--
00400000	00001000	radaman.exe		IMG	-R---	ERWC-
00401000	00011000	".text"	Executable code	IMG	ERWC-	ERWC-
00412000	00001000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00413000	00003000	".data"	Initialized data	IMG	-RW--	ERWC-
00416000	0003C000	".rsrc"	Resources	IMG	-RW--	ERWC-
00460000	000C7000	\\Device\\HarddiskVolume2\\Windows\\Sys		MAP	-R---	-R---
00530000	00006000			PRV	-RW--	-RW--
00536000	0000A000	Reserved (00530000)		PRV		-RW--
00540000	000FC000	Reserved		PRV		-RW--
0063C000	00004000	Thread 714 Stack		PRV	-RW-G	-RW--
00660000	00010000			PRV	-RW--	-RW--
00670000	000F0000	Reserved (00660000)		PRV		-RW--
00760000	00035000	Reserved		PRV		-RW--
00795000	0000B000			PRV	-RW-G	-RW--
007A0000	000FC000	Reserved		PRV		-RW--
0089C000	00004000	Thread 6A4 Stack		PRV	-RW-G	-RW--
008A0000	00035000	Reserved		PRV		-RW--
008D5000	0000B000			PRV	-RW-G	-RW--
008E0000	000FD000	Reserved		PRV		-RW--
009DD000	00003000	Thread CEC Stack		PRV	-RW-G	-RW--
009E0000	0000A000			MAP	-R---	-R---
009EA000	001F6000	Reserved (009E0000)		MAP		-R---
00BE0000	00181000			MAP	-R---	-R---
00D70000	00053000			MAP	-R---	-R---
00DC3000	013AE000	Reserved (00D70000)		MAP		-R---
021A0000	00001000			PRV	ERW--	ERW--
021B0000	0002B000			PRV	-RW--	-RW--
022A0000	00003000			PRV	-RW--	-RW--
022A3000	0000D000	Reserved (022A0000)		PRV		-RW--

In memory map we can see that at the address where the exe is loaded, (021B0000) that location has read and write protections. We now dump out that location and examine it.

Unpacked File

Immediately after opening the “unpacked” file we notice that is indeed packed again based on IDA.



There are not enough functions and small amount of analyzed code by IDA. Looking at the few functions that are available we can start to see some interesting actions taking place.

```
lpBuffer= dword ptr -4
```

```
push    ebp
mov     ebp, esp
add     esp, 0FFFFFFBE4h
push    ebx
mov     edx, 1B8C88EEh ; A key? because its being used in the loop below
lea    eax, unk_403000 ; Strange function
mov     ecx, 29CD6h   ; Counter or length?
```

```
loc_4011AA:
xor     [eax], dl
rol     edx, 7
inc     eax
dec     ecx
jnz    short loc_4011AA
```

```
mov     [ebp+dwSize], 33800h
push    4 ; flProtect
push    3000h ; flAllocationType
push    [ebp+dwSize] ; dwSize
push    0 ; lpAddress
call    VirtualAlloc
```

loc_4011AA looks to be a loop. The key is moved to EDX and XORed with a byte from unk_403000 then rotated left. Then theres some decreasing and increasing happening and then there is a conditional jnz which moves the code along only if not being equal to zero.

This is most likely the encryption or encoding algorithm used.

```
mov     [ebp+dwSize], 33800h
push    4             ; flProtect
push    3000h        ; flAllocationType
push    [ebp+dwSize] ; dwSize
push    0            ; lpAddress
call    VirtualAlloc
test    eax, eax
jz      loc_4014E7
```

```
mov     [ebp+lpBuffer], eax
mov     byte_42CD10, 6Eh ; 'n'
mov     byte_42CD11, 74h ; 't'
mov     byte_42CD12, 64h ; 'd'
mov     byte_42CD13, 6Ch ; 'l'
mov     byte_42CD14, 6Ch ; 'l'
mov     byte_42CD15, 2Eh ; '.'
mov     byte_42CD16, 64h ; 'd'
mov     byte_42CD17, 6Ch ; 'l'
mov     byte_42CD18, 6Ch ; 'l'
mov     byte_42CD19, 0
push    offset byte_42CD10 ; lpLibFileName
call    LoadLibraryA
test    eax, eax
jz      loc_4014E7
```

Following along we can see that it is loading DLLs into a buffer. LoadLibraryA is called which provides a return to a handle that can be used in GetProcAddress below:


```
mov     byte_42CD1C, 52h ; 'R'
mov     byte_42CD1D, 74h ; 't'
mov     byte_42CD1E, 6Ch ; 'l'
mov     byte_42CD1F, 44h ; 'D'
mov     byte_42CD20, 65h ; 'e'
mov     byte_42CD21, 63h ; 'c'
mov     byte_42CD22, 6Fh ; 'o'
mov     byte_42CD23, 6Dh ; 'm'
mov     byte_42CD24, 70h ; 'p'
mov     byte_42CD25, 72h ; 'r'
mov     byte_42CD26, 65h ; 'e'
mov     byte_42CD27, 73h ; 's'
mov     byte_42CD28, 73h ; 's'
mov     byte_42CD29, 42h ; 'B'
mov     byte_42CD2A, 75h ; 'u'
mov     byte_42CD2B, 66h ; 'f'
mov     byte_42CD2C, 66h ; 'f'
mov     byte_42CD2D, 65h ; 'e'
mov     byte_42CD2E, 72h ; 'r'
mov     byte_42CD2F, 0
push   offset byte_42CD1C ; lpProcName
push   eax                ; hModule
call   GetProcAddress
test   eax, eax
jz     loc_4014E7
```

Next it pushes into a buffer RTLDecompressBuffer which decompresses the buffer which is in this case: NTDLL.DLL

```
mov     byte_42CD32, 44h ; 'D'
mov     byte_42CD33, 6Ch ; 'l'
mov     byte_42CD34, 6Ch ; 'l'
mov     byte_42CD35, 47h ; 'G'
mov     byte_42CD36, 65h ; 'e'
mov     byte_42CD37, 74h ; 't'
mov     byte_42CD38, 43h ; 'C'
mov     byte_42CD39, 6Ch ; 'l'
mov     byte_42CD3A, 61h ; 'a'
mov     byte_42CD3B, 73h ; 's'
mov     byte_42CD3C, 73h ; 's'
mov     byte_42CD3D, 4Fh ; 'O'
mov     byte_42CD3E, 62h ; 'b'
mov     byte_42CD3F, 6Ah ; 'j'
mov     byte_42CD40, 65h ; 'e'
mov     byte_42CD41, 63h ; 'c'
mov     byte_42CD42, 74h ; 't'
mov     byte_42CD43, 0
push   offset byte_42CD32 ; lpProcName
push   eax                ; hModule
call   GetProcAddress
test   eax, eax
jz     loc_4014E7
```

Next called up is DLLGetGlassObject of NTDLL.DLL and then a call to GetProcAddress.

We then see that EAX which holds RTLDecompressBuffer is moved to EDX and then called again. Looking at the documentation for [RTLDecompressBuffer](#), the parameters are:

- [in] which is 102h
- [Out] Buffer which is [ebp+lpBuffer]
- [in] which is [ebp+dwSize]
- [in] buffer that contains the data in ECX which holds unk_403000 (encryption method)
- [in] which is the length 29CD6h
- [out] which is the return stored at EAX

This result is then cmp with itself and if it meets the conditional it continues on.

```
mov    edx, eax    ; EAX holds RTLDecompressBuffer, now in EDX
lea    eax, [ebp+dwSize]
lea    ecx, unk_403000
push   eax
push   29CD6h      ; Counter length again
push   ecx
push   [ebp+dwSize]
push   [ebp+lpBuffer]
push   102h
call   edx        ; RTLDecompressBuffer called
test   eax, eax
jnz    loc_4014E7
```

```
call   sub_40102A  ; Loads Kernel32.dll and calls RTLDecompressBuffer
lea    eax, [ebp+Buffer]
push   eax        ; lpBuffer
push   104h       ; nBufferLength
call   GetTempPathW
test   eax, eax
jz     loc_4014E7
```

sub_40102A loads KERNEL32.DLL and calls RTLDecompressBuffer in the same way NTDLL.DLL is loaded in. Then we start to see the formation of a temp file

```
lea    eax, [ebp+TempFileName]
push   eax           ; lpTempFileName
push   0             ; uUnique
push   0             ; lpPrefixString
lea    eax, [ebp+Buffer]
push   eax           ; lpPathName
call   GetTempFileNameW
test   eax, eax
jz     loc_4014E7
```

```
push   0             ; hTemplateFile
push   6             ; dwFlagsAndAttributes
push   2             ; dwCreationDisposition
push   0             ; lpSecurityAttributes
push   0             ; dwShareMode
push   40000000h     ; dwDesiredAccess
lea    eax, [ebp+TempFileName]
push   eax           ; lpFileName
call   CreateFileW
mov    ebx, eax
inc    eax
jz     loc_4014E7
```

The malware uses GetTempFileNameW, creates the file with CreateFileW, writes to the file using WriteFile and then loads the file as a DLL using LoadLibraryA. (Partially Pictured)

And finally a buffer with the string "host 000000000000" before the code ends. The zeros are probably changed to some unique ID that the malware uses to send back to a C&C server.

```

mov     byte_42CD46, 68h ; 'h'
mov     byte_42CD47, 6Fh ; 'o'
mov     byte_42CD48, 73h ; 's'
mov     byte_42CD49, 74h ; 't'
mov     byte_42CD4A, 20h ; ' '
mov     byte_42CD4B, 30h ; '0'
mov     byte_42CD4C, 30h ; '0'
mov     byte_42CD4D, 30h ; '0'
mov     byte_42CD4E, 30h ; '0'
mov     byte_42CD4F, 30h ; '0'
mov     byte_42CD50, 30h ; '0'
mov     byte_42CD51, 30h ; '0'
mov     byte_42CD52, 30h ; '0'
mov     byte_42CD53, 30h ; '0'
mov     byte_42CD54, 30h ; '0'
mov     byte_42CD55, 30h ; '0'
mov     byte_42CD56, 30h ; '0'
mov     byte_42CD57, 0
push    0
push    offset byte_42CD46
push    0
push    0
call   eax

```

That's all we can get out of IDA so now we move to the debugger and use the same methods to find the payload DLL

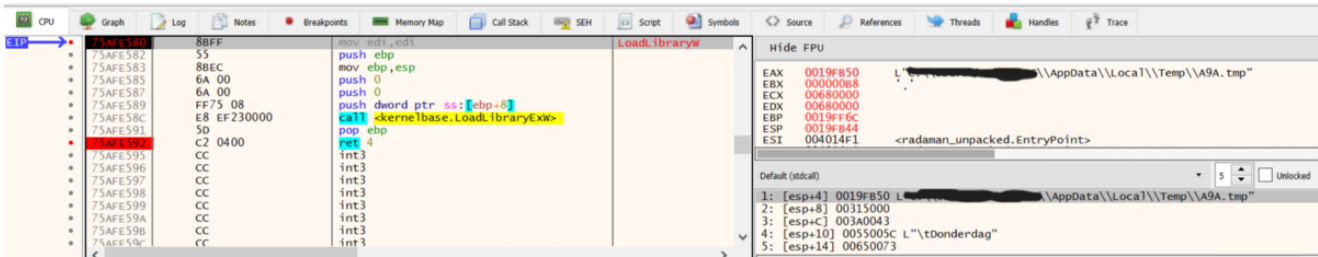
Unpacking the “Unpacked” File

Since we know the next step of this malware is to perform a DLL injection, we can put a breakpoint at LoadLibraryW (not LoadLibraryA)...

'A' stands for ASCII and 'W' stands for byte string and the 'A' calls are just the wrappers around the 'W' ones so placing the breakpoint at the LoadLibraryW will hit all the load DLL calls.

| [Source](#)

and from there we can see the path where the DLL will be dropped.



Checking that location we can find the file and checking in PESTudio we can see that it is a DLL (file maybe hidden).

property	value
md5	BA09C5888E93D7F81B6E65F260962DE4
sha1	D4CF1157B3AF4207803CDA74FD8300E920E3CCF3
sha256	D5CCC140D73A5E76154AA15B2015FCD0F022298825430F02B408C38CDC48F79B
md5-without-overlay	n/a
sha1-without-overlay	n/a
sha256-without-overlay	n/a
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 C
first-bytes-text	M Z @
file-size	200704 (bytes)
size-without-overlay	n/a
entropy	7.450
imphash	D3B0A68EC2185264A5C5A26F84A23AC5
signature	n/a
entry-point	60 31 D2 55 54 5D 83 C4 A4 B9 B3 98 08 00 BA E2 BB 08 00 E8 2A 00 00 00 00 00 00 00 00 00
file-version	n/a
description	n/a
file-type	dynamic-link-library
cpu	32-bit
subsystem	GUI
compiler-stamp	0x58246200 (Thu Nov 10 12:03:12 2016)
debugger-stamp	empty
resources-stamp	empty
exports-stamp	empty

Conclusion

So to wrap things up, we successfully unpacked the initial Redaman file using VirtualAlloc and VirtualProtect, we then discovered the encryption algorithm it uses, and finally unpacked once again with LoadLibraryW to find the payload DLL.

Thanks for reading.

Resources:

[Russian Language Malspam Pushing Redaman Banking Malware](#)

[Unpacking Redaman Malware & Basics of Self-Injection Packers — ft. OALabs](#)

[Unpacking Redaman Malware & Basics of Self-Injection Packers — ft. OALabs \(Video\)](#)