TECHINCAL ANALYSIS ON VBS PACKED MALWARE SAMPLE 1 - DIFFUSEGRAVITY

This paper goes into detail about the technical details gone into unpacking and analyzing vbs based samples and the methods I used in order to make the analysis easier and more efficient. This paper is both for documentation and for future reference

By Philip C. Okoh

Introduction

VB6 compiler can be treated as a packer because of its translation from what is known as P code into assembly code. This makes static analysis of it impossible unless manual unpacking is done which will be implemented using a debugger such as, x64dbg. For reference, see **A.1** of the appendix for what an unpacked version VB6 looks like in IDA pro.

Unpacking Procedure

This sample was utilizing allocated memory to write an at the same time change its protection rights a means to start a new execution. This is a typical approach and can easily be captured through the usage of setting break points on the following APIs. IsDebuggerPresent, VirtualAlloc, and VirtualProtect, CreateProcessInternalW and or CreateProcessInternalA.

IsDebuggerPresent is typical because of the usage of anti-debugging in malware such as these. Setting a break point at this function can allow us to zero out its return value thus negating its effects.

This sample called upon VirtualAlloc more than 10 times, all while returning the same address all except the very last one. This was very tedious, and I recommend the following approach.

- Set a break point solely on CreateProcessInternal(A/W) and wait until its execution.
- Search the executable for strings that would be included within every PE executable. Because it is known that a PE executable is to be written inside an allocated memory the words “This program cannot” should be enough as this is included in the header of every type of PE executable.
- Finding the start address of this allocated memory can then be dumped to file and analyzed. Debugging should be stopped unless further dynamic analysis would want to be done on the new executed file.

Static Analysis - Unpacked Sample

Sample is compiled with Borland Delphi compiler version 2.25. Based around the entropy it was determined that the resource section was packed in some way. The sample also included VirtualAlloc, Sleep, HeapAlloc and GetProcessAddress and LoadLibrary. The last two indicate that some dynamic linking is done along with static linkage.

It could be determined that the resource section was being unpacked with an unknown algorithm. In these situations, it is best to allow the program to handle this itself and will be noted during debugging.

Dynamic Analysis - Debugger

Moving on from static analysis I decided to set the following API breakpoints. The following are standard. Breakpoints are set on CreateFile(A/W), CreateProcessInternal(A/W), MapViewofFile, VirtualAlloc, VirtualProtect, CreateFileMapping(A/W), WriteFile and ResumeThread. Through these APIs the following behavior was found.

The windows executables were being used for execution:

- Powershell.exe
- Mshta.exe
- Regsvr32.exe

See **Appendix A.2 - A.4** for more background information on the previously stated windows executables.

Techniques and Tactics

Diffuse gravity utilizes a lot common but good techniques that can keep it from being detected especially from an unsuspecting user. For its abilities I categorize it as a malware which is a lay of the land. It utilizes trusted tools provided by Windows to do most of the heavy lifting.

Network Communication Hiding

Regsvr32.exe is utilized by this malware as a means of communications with outside networks. It utilizes its network capabilities to send TCP communications outbound through http. Although no analytical data was captured, it was observed that several IP address that had no official Microsoft affiliation were being used. With a program pertaining to one such as regsvr32.exe any network communications would have regular occurring IP addresses communicated with, but log analysis showed irregular and an abundance of non-windows IP addresses.

Log analysis using Procmon has shown that the application regsvr32.exe has been looking through the HKCU\Software\cnatdatkc registry directory. This was created and has components on it that can restart the entire process with a .htm file.

The Procmon log file will be included in the **Appendix A.5**.

Shell Code Running and Code Obfuscation

The primary function of this malware acts like a loader of payloads which are then executed by more trusted software such as PowerShell. One such example is the shell code found in a PowerShell script file and executed. This PowerShell script can be found in **Appendix A.6** but it should be noted that this file has been de-obfuscated, and the original was in the base64 encoding, an encoding used a lot throughout this sample.

Analysis of the shell code proved to be difficult. Using tools such as scdbg proved to be ineffective as the output did not make much sense. The shell code can be found in **Appendix**

A.7. In addition to the process tree, which can be found in **Appendix A.8**, it is fair to say that this shell code is what might be injected in the regsvr32.exe file. Also, with the prior mentioned log files and the odd behavior displayed by regsvr32.exe this too backs up this claim.

While base64 is the main obfuscation technique here it is not the only one as self-created algorithms have been formed to keep detection low. This will be talked about in relationship to this malware's persistence solution.

Persistence

Diffusegravity utilizes the registry to gain persistence after reboot, executing a very obfuscated JavaScript file which utilizes its own XOR algorithm to decrypt the data. This file can be found in **Appendix A.9**. The data is decrypted to a PowerShell script which again utilizes base64 to hide its contents. This data is then Invoked to execute by PowerShell. The decoding of this data reveals to be the setup and shell code for which my analysis makes me to believe is being injected into regsvr32.exe and causing it to run and contact outside IP addresses.

Conclusion

My analysis has not brought on a true conclusion. Because of my lack of evidence with examining the injected shell code it is hard to say exactly what is happening within the regsvr32 executable and what is being either downloaded or further executed. One thing that can be stated is that this sample is a Trojan and loader. It disguises itself as a clock with an icon as such. What makes this Trojan so dangerous is how it hides itself. Upon full execution of this malware itself deletes itself and its last shell code utilizes the tools which are provided by the system to execute what is further needed to be done. Even after a system restart with the help of the registry and without the need of the original loader it can restart the programs mshta, powershell and regsvr32 to do the commands it needs to do. This malware could be a botnet, ransomware dropper or even espionage code. The speculation should be saved from this paper as further analysis is necessary to determine the validity of these statements.

Furthermore, I would categorize this as semi-fileless as its only disk fingerprint is the registry but otherwise exists entirely within memory. The usage of trusted programs will make detection of this very hard for most and almost impossible for those who aren't looking for it specifically. A completely autonomous malware which travels from windows program to windows program undetected is a very dangerous thing.

Indicators of Compromise I.O.C

Original Packed Payload

fc4f695752f8eb20b17689e60a7161a43665fa3455dc379aeb2a251838eb4da6

Unpacked Payload

ef8970501e077afabde3142d2ff30d82253bfce50cc1b07812f64ccab85af720

Registry Created

- HKCU\Software\cnatdatkc
- HKCU\Software\cnatdatkc\xobkepd
- HKCU\Software\cnatdatkc\dxcn
- HKCU\Software\cnatdatkc\plwf
- HKCU\Software\cnatdatkc\ppee
- HKCU\Software\cnatdatkc\mrfsocc
- HKCU\Software\cnatdatkc\gpmtife

Registry Modified

- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run

Appendix

Section A

A.1

```
.text:00401106                 align 4
.text:00401108
.text:00401108                 public start
.text:00401108 start:
.text:00401108                 push    offset dword_403990
.text:0040110D                 call    ThunRTMain
.text:0040110D ; ---------------------------------------------------------------------------
.text:00401112                 dw 0
.text:00401114                 align 8
.text:00401118                 dd 30h, 40h, 0
.text:00401124                 dd 0F90328D4h, 419420C9h, 0EF30D892h, 7ACDBC51h, 0
.text:00401138                 dd 10000h, 400000h, 40C0D8h, 6C616E41h, 435F676Fh, 6B636F6Ch
.text:00401138                 dd 2F0B100h, 0
.text:00401158 dword_401158    dd 31CCFFh, 8546A706h, 823036B9h, 0FCACA24Dh, 0D3BEEDBFh
.text:00401158                                         ; DATA XREF: .text:00403A7C↓o
.text:00401158                 dd 313061BAh, 137B3396h, 8D7C8647h, 0A81372D4h, 0AD4F3AABh
```

A.2

Powershell: https://docs.microsoft.com/en-us/powershell/

A.3

Mshta.exe: https://www.mcafee.com/blogs/other-blogs/mcafee-labs/what-is-mshta-how-can-it-be-used-and-how-to-protect-against-it/

A.4

Regsvr32.exe: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/regsvr32

A.5

https://github.com/ByridianBlack/Malware-Reports/blob/main/DiffuseGravity/diffuseGravityLogFile.PML

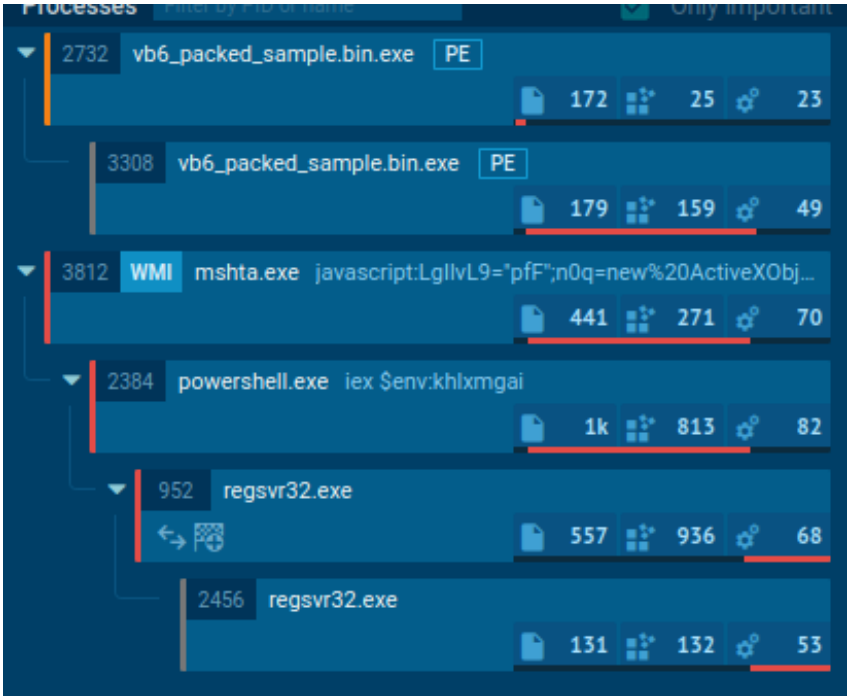
A.6

https://github.com/ByridianBlack/Malware-Reports/blob/main/DiffuseGravity/decoded_data.ps1

A.7

https://github.com/ByridianBlack/Malware-Reports/blob/main/DiffuseGravity/extracted.ps1

A.8



A.9

https://github.com/ByridianBlack/Malware-Reports/blob/main/DiffuseGravity/registry_data.js