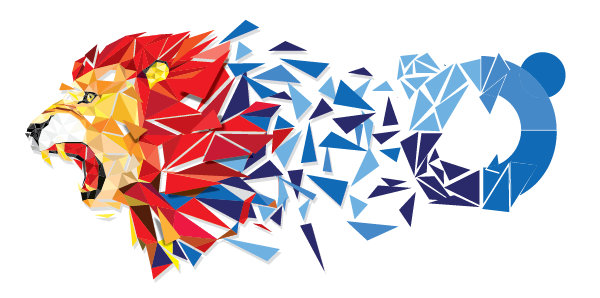
WatchGuard Technologies -- Panda Security



Malware Analysis Report

**Attachment\_57904.vbs**

Ryan Estes

Intrusion Analyst

WatchGuard Technologies

505 Fifth Avenue South Suite 500

Seattle, WA 98104

**T:** 1-217-381-2597 **E:** ryan.estes@watchguard.com

Contents

[Introduction 2](#_Toc67319549)

[Attachment\_57904.vbs 4](#_Toc67319550)

[File Metadata 4](#_Toc67319551)

[Result 4](#_Toc67319552)

[Tools Used 4](#_Toc67319553)

[External Links 4](#_Toc67319554)

[Analysis 5](#_Toc67319555)

# Introduction

On March 18th, 2021, the DNSWatch Tailored Analysis Team received an email from an internal WatchGuard employee who deemed the email as suspicious. The initial email included an attachment with the title *Attachment\_57904*. A DNSWatch Analyst performed an intial assessment of the file in saerch of any malicious indicators or behaviors only to discover that the file was a heavily obfuscated Visual Basic Script (.vbs). Upon this discover, the file was securely passed on to the Panda Attestation Team for further analysis. This report documents this analysis process and how, ultimately, the file was unveiled to be a dropper for a ZLoader variant.

Intentionally Left Blank

# Attachment\_57904.vbs

## File Metadata

|  |  |
| --- | --- |
| **File Metadata** | |
| File Name: | Attachment\_57904.vbs |
| File Size (Bytes): | 1,403,043 |
| MD5: | DE5FD5E5D41D654355B21942E938FFC3 |
| SHA1: | B3898BADB33A62F84EBDB5459BDCB8187DDA1867 |
| SHA256: | A46AA343E090D3EEC942D0E186AACFC3AE5A4E37FF4DD0A27FA9751961E1524C |
| File Type: | VBS |

## Result

**Dropper for ZLoader (Ursnif/Gozi)**

## Tools Used

* PowerShell
* Python
* CFF Explorer
* Notepad++
* Atom

## External Links

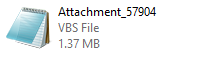
<https://github.com/ryanestes/MalwareAnalysis>

<https://www.virustotal.com/gui/file/94eb81bc58adb976f21344d3eb273c9eb833afbcadd121eb2ad38f1ef07a1f85/detection>

<https://analyze.intezer.com/files/94eb81bc58adb976f21344d3eb273c9eb833afbcadd121eb2ad38f1ef07a1f85>

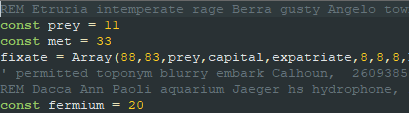
## Analysis

1. The attachment provided in the email (i.e. the malware sample).

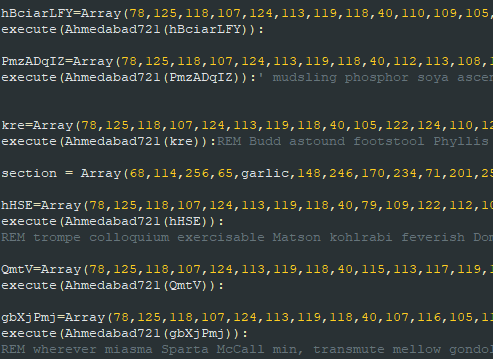


1. What is seen immediately upon opening the file. The file consists of gibberish dictionary words declared as variables, inputted as comments, and large arrays consisted of additional dictionary words and integers. The smaller image below gives a magnified visual.

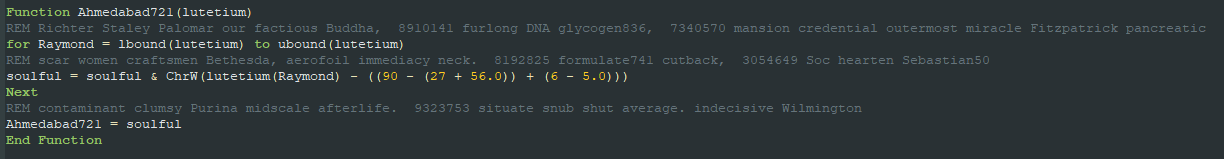


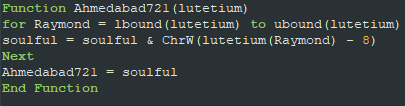


1. Apart from the over 500 arrays defined in this script, there are exactly 16 arrays proceeded by an execute function. Each execute function is passed an argument of another function, Ahmedabad721(), which takes the array as an argument itself.

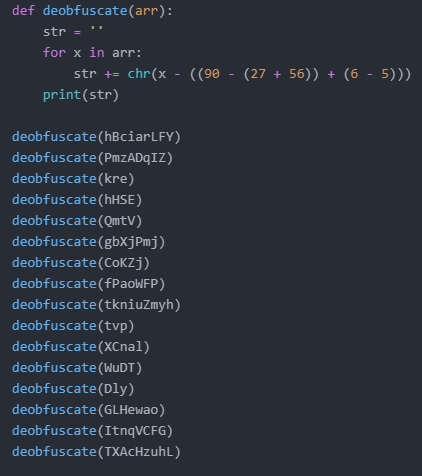


1. In the middle of all the arrays and comments that also contain gibberish dictionary words, the Ahmedabad721 function can be found. After cleaning up the function a little it is shown to be a trivial algorithm to turn these arrays into strings. The small function uses a for loop to iterate through the array that is passed as an argument, subtracts 8 from the number in the array index, converts the decimal into a character using chrW(), concatenates that character to a variable, and then returns the entire concatenated string.





1. After extracting the 16 arrays from the script, a Python script was hacked together to replicate the Ahmedabad721 function being executed dynamically. Listed below, in order, are the functions that are executed with one being in bold – WuDT. It is because, after deobfuscating this function, it was discovered that this function builds another file by using *another* giant array of dictionary words. Each dictionary word within this array aligns with every other array in the original VBS that was not executed. The image below shows the small deobfuscation script and the 16 arrays that were executed in the VBS.



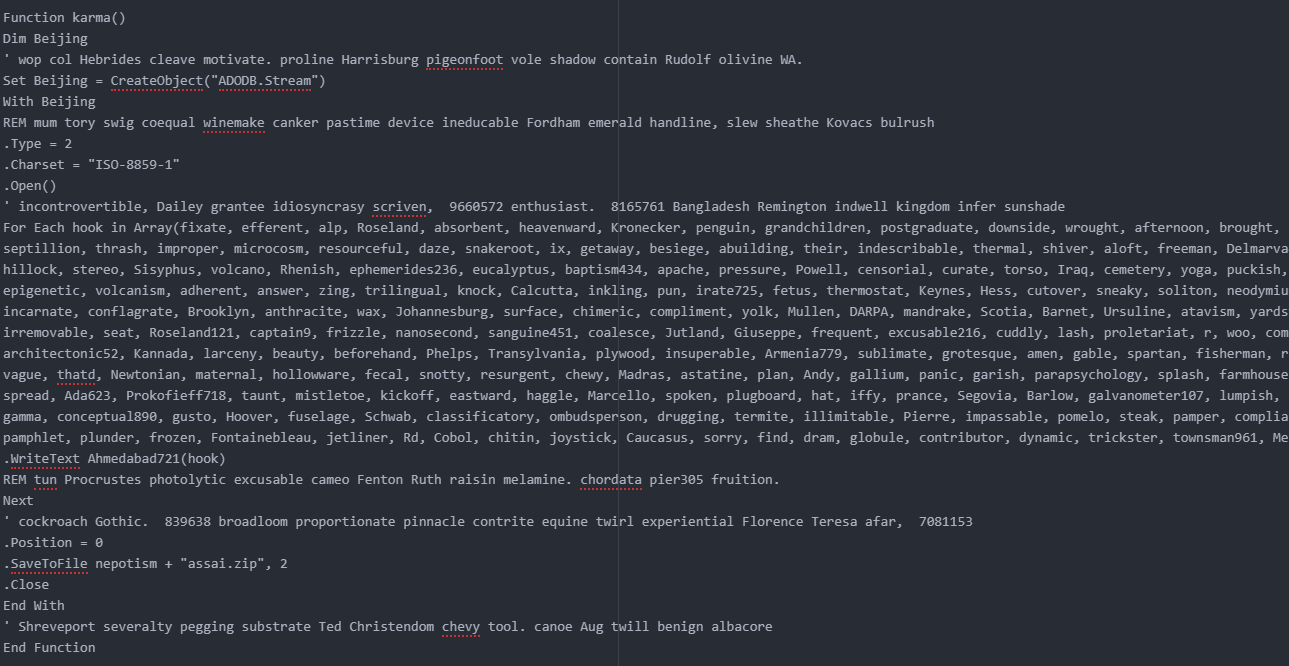
1. The next 4 pictures show how the WuDT array evolves into visual basic code as it is deobfuscated
   1. A snippet of the original WuDT array as it is observed in the original script. Notice how there are dictionary words within the list of integers. The only one seen here is “hogging” which is equal to 18 as defined by a variable embedded within the script.



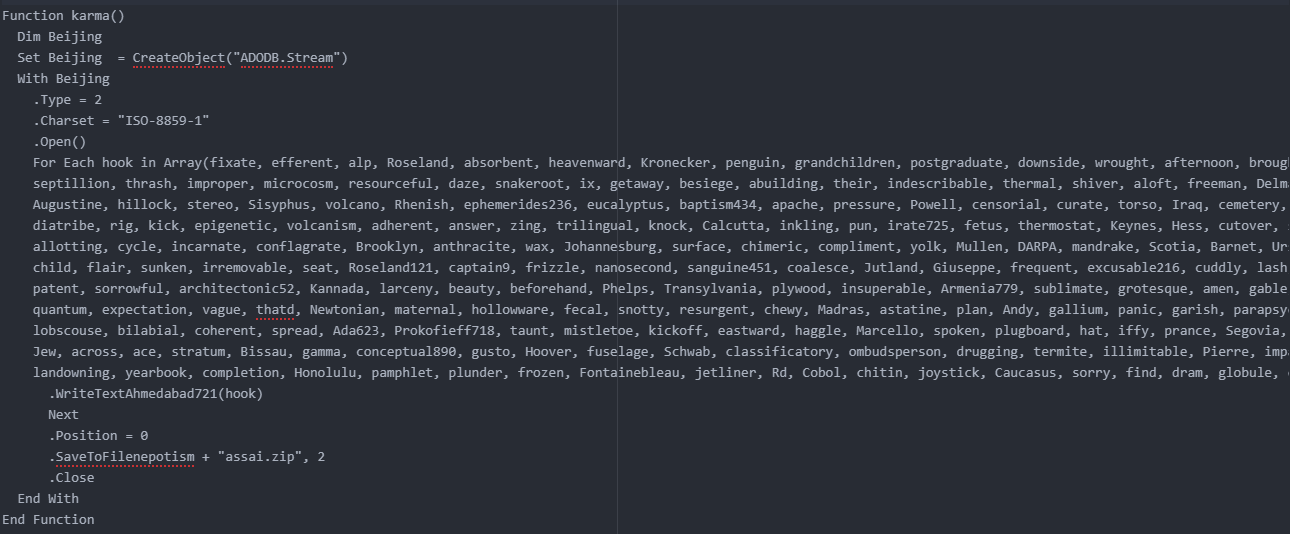
* 1. The array with the string variables substituted with their respective string variables.



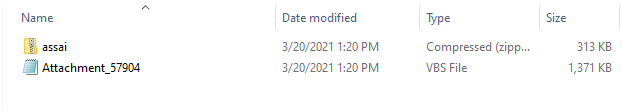
* 1. The WuDT array after it is ran through the Ahmedabad721 function. Notice how there is further obfuscation by using comments of gibberish dictionary words. Also notice how the array is a large list of different dictionary words, each corresponding to an entirely different array found within the original script.



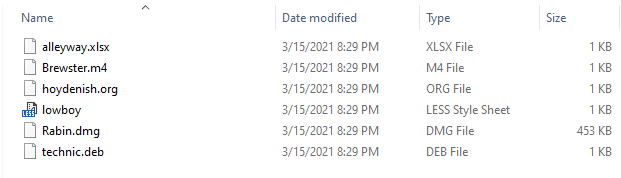
* 1. After cleaning up the final obfuscation in the resulting function.



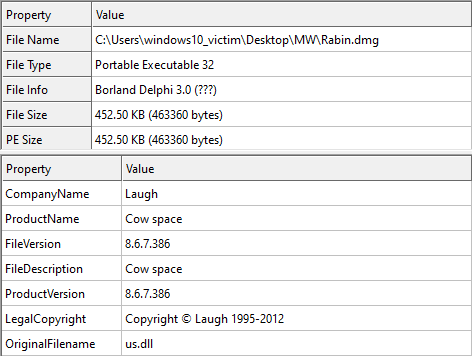
1. Knowing that the WuDT function is the only one that produces another file, one can assume this is the ultimate malicious final being dropped. Since this function only creates a file, the other execute functions (not WuDT) can be commented out and the original VBS can be ran to produce the resulting file this function produces. Ultimately, this produced an entire zip folder named “assai.zip”.



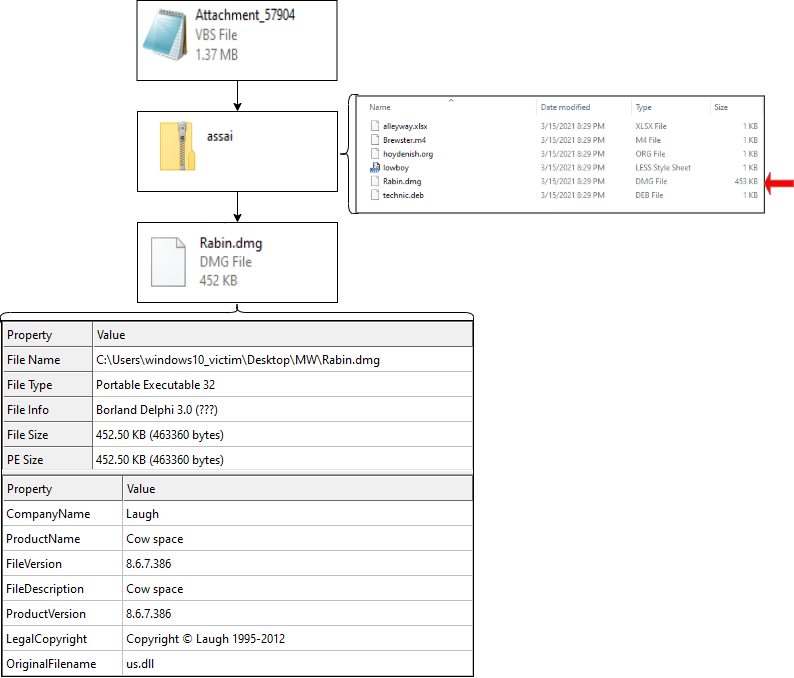
1. Unzipping the assai.zip folder results in a myriad of different files. All of them are 1 KB except the file named “Rabin.dmg”. Observing the 1 KB shows that each one contains exactly one line of an arbitrary string that is meaningless. In other words, further obfuscation.



1. Speaking of further obfuscation, the Rabin.dmg, an alleged macOS installer, is actually a 32-bit DLL originally named “us.dll”.



1. Therefore, the original VBS roughly uses this procedure:



1. The Rabin.dmg file is, in fact, a variant of ZLoader and Ursnif/Gozi

|  |  |
| --- | --- |
| **File Metadata** | |
| File Name: | Rabin.dmg |
| File Size (Bytes): | 463,360 |
| MD5: | B1FC7DC75445A016588402757FDD6FF6 |
| SHA1: | 12AA8A932E6711BECA796F67E717523D6794DE9E |
| SHA256: | 94EB81BC58ADB976F21344D3EB273C9EB833AFBCADD121EB2AD38F1EF07A1F85 |
| File Type: | 32-bit DLL |

End Analysis