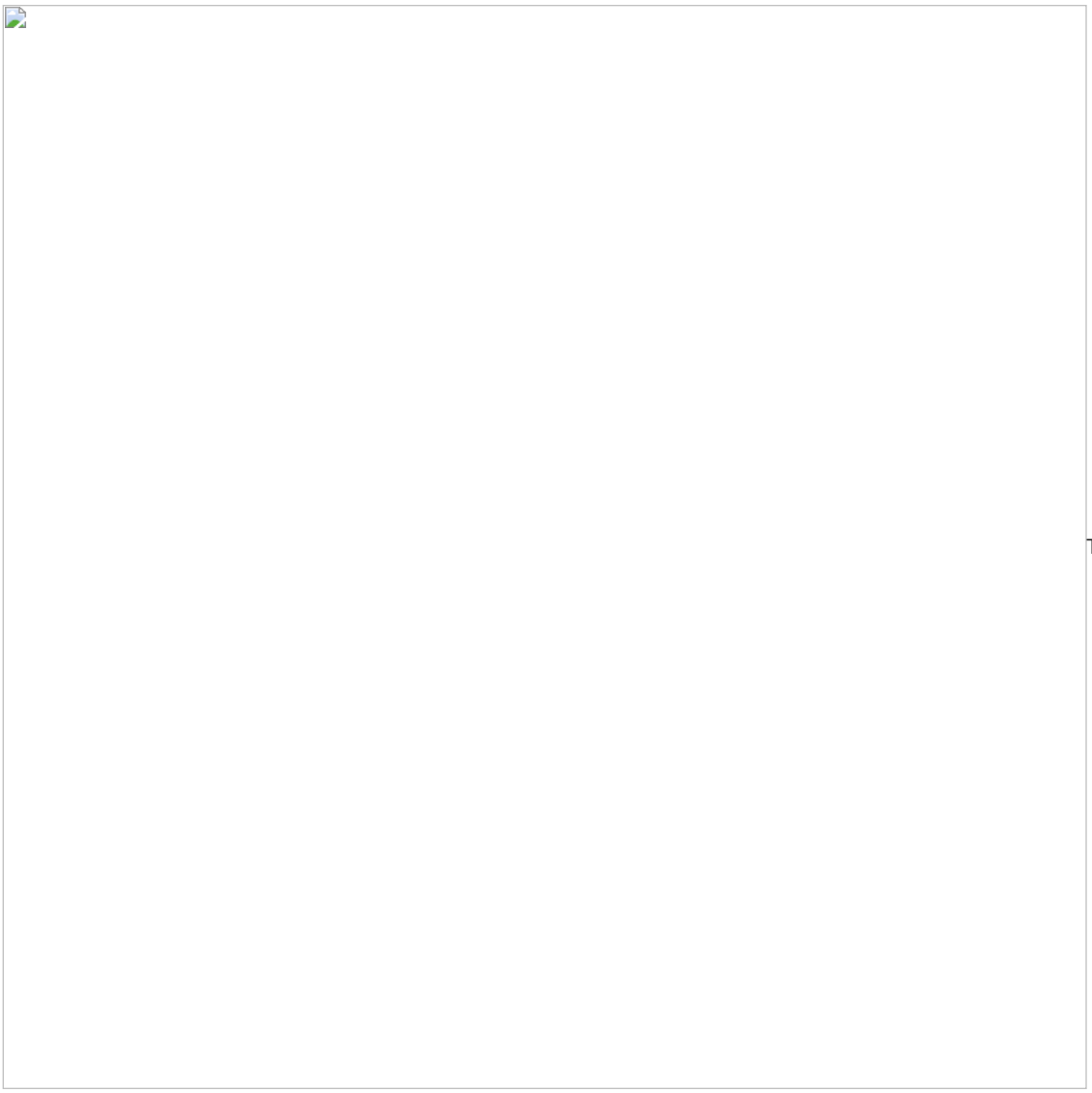
Deobfuscating Emotet’s powershell payload

malfind.com/index.php/2018/07/23/deobfuscating-emotets-powershell-payload/

Emotet is a banking trojan, targeting computer users since around 2014. During that time it has changed its structure a lot. Lately we see massive emotet spam campaigns, using multiple phishing methods to bait users to download and launch a malicious payload, usually in the form of a weaponized Word document.

Emotet’s chain of infection

First user receives a fake e-mail, trying to persuade him to click on the link, where the weaponized doc is being downloaded. Document is then trying to trick user to enable content and allow macros in order to launch embedded VBA code. VBA is obfuscated. We can also deobfuscate it, but in the end it launches a powershell command. Let’s skip VBA deobuscation today, as I want to focus on powershell. We can obtain powershell command launched by VBA code without deobfuscation, by using any sandbox with powershell auditing.



Typical Emotet document

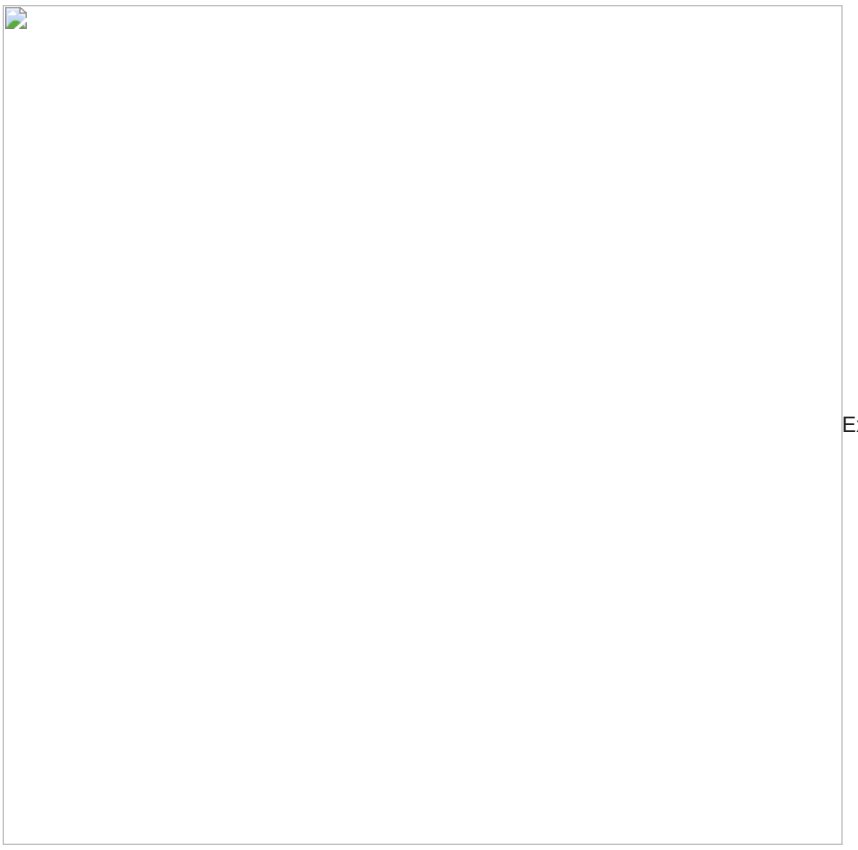
The powershell code itself is obfuscated as well. The problem with just launching it in the virtual environment is that we probably won't see every network IoC this way. Of course there are ways to do it (just block dns requests, and malware should try every fail-over domain), but in my opinion if there is time to do it – it is always better to deobfuscate code to better understand it.

Obfuscation is a way to make a malicious code unreadable. It has two purposes. First to trick antivirus signatures, second to make analysis of the code harder and more time-consuming.

In this post, I want to show three ways of obfuscation used by Emotet malware since December 2017.

1. String replace method

This method uses multiple powershell's "replace" operators to swap a bunch of junk strings with characters that in the end produce a valid powershell code



Example 1. Code obfuscated with replace string method

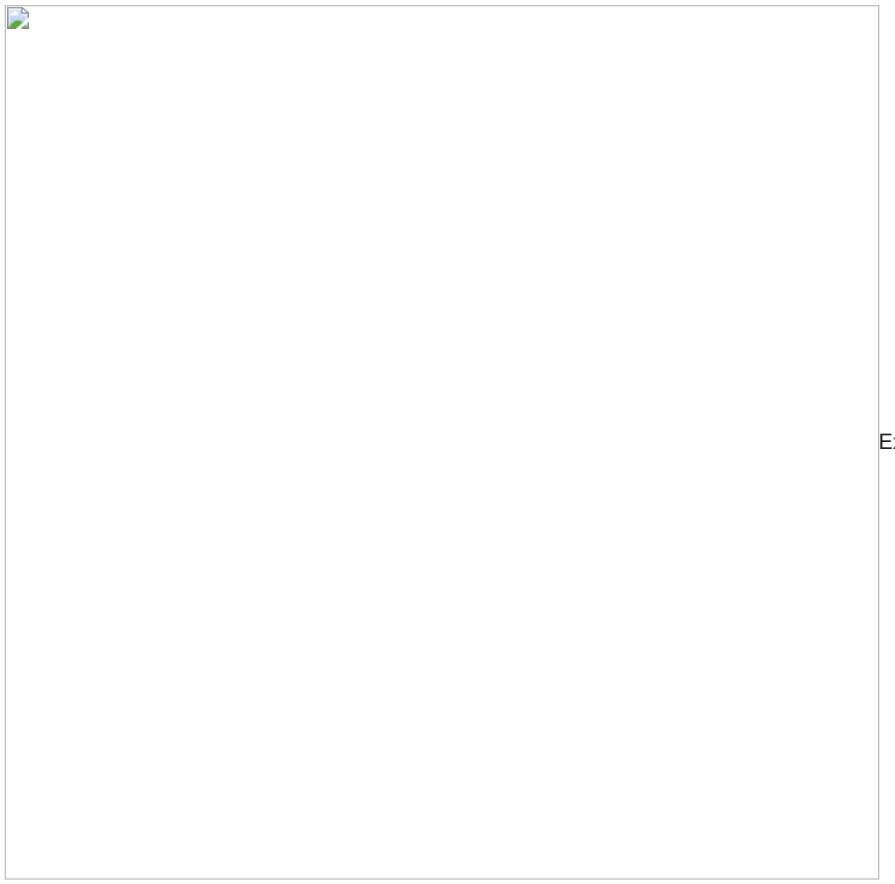
Of course you can deobfuscate it manually in any text editor, just by replacing every string with its equivalent or you can speed up a process with correct regular expression. In the end you can put this regular expression in the python script and automate it completely. There are just few things to consider when implementing it in python:

- String concatenations. These little '+' can mess up with our regexp, so they have to be handled first
- Char type projection – sometimes for additional obfuscation, strings to be replaced are not typed directly to the powershell code, but they are converted from int to char. We have to handle that as well
- Replacing one part of the code can "generate" new replace operators – this is because "junk string" can be in the middle of replace operator (for example: -replFgJace, where FgJ is a string to be replaced with empty string). For this reason it is best to put regexp in the loop and perform replace operation as long as there is something to replace

Deobfuscated code from example 1

2. String compression

This method is quite simple as it uses powershell's built-in class DeflateStream to decompress and execute a compressed stream.



Example 2. Decompress string obfuscation method

The easiest way to deobfuscate this is to use powershell to simply decompress the string. **Just remember to remove command between first two parenthesis – its a an obfuscated Invoke-Expression cmdlet that will execute the code on your computer! Also, always use a safe (possibly disconnected from the network, unless you know what you are doing), virtualized environment when dealing with malicious code**.

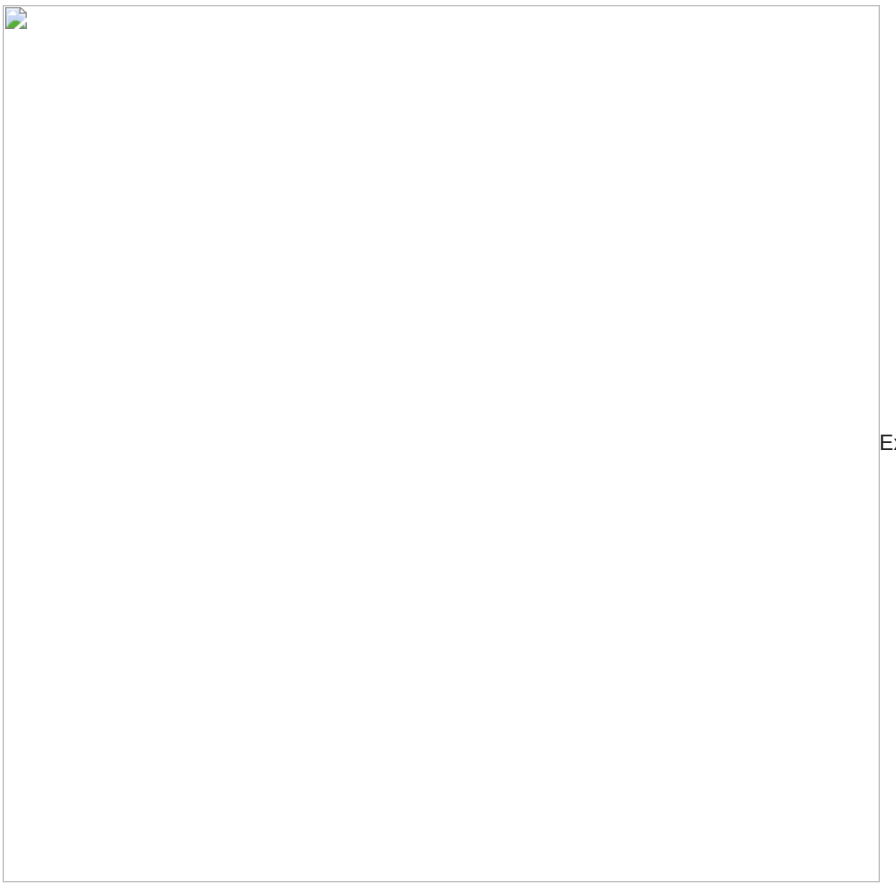
Decompression method

deobfuscation in powershell

But what if we'd like to have a portable python script that can deal with this type of deobfuscation? If we look at MSDN documentation, then we will see that DeflateStream class follows RFC 1951 Deflate data format specification, and can actually be decompressed by using zlib library. There is one catch: zlib's decompress method by default expects correct zlib file header, which DeflateStream does not have, as it is not a file but a stream. To force zlib to decompress a stream we can either add a header to it or simply pass a *-zlib.MAX_WBITS* (there is a minus at the beginning!) argument to decompress function. zlib.MAX_WBITS (which is 15) argument with a negative value informs decompress function that it should skip header bits.

3. ASCII codes array

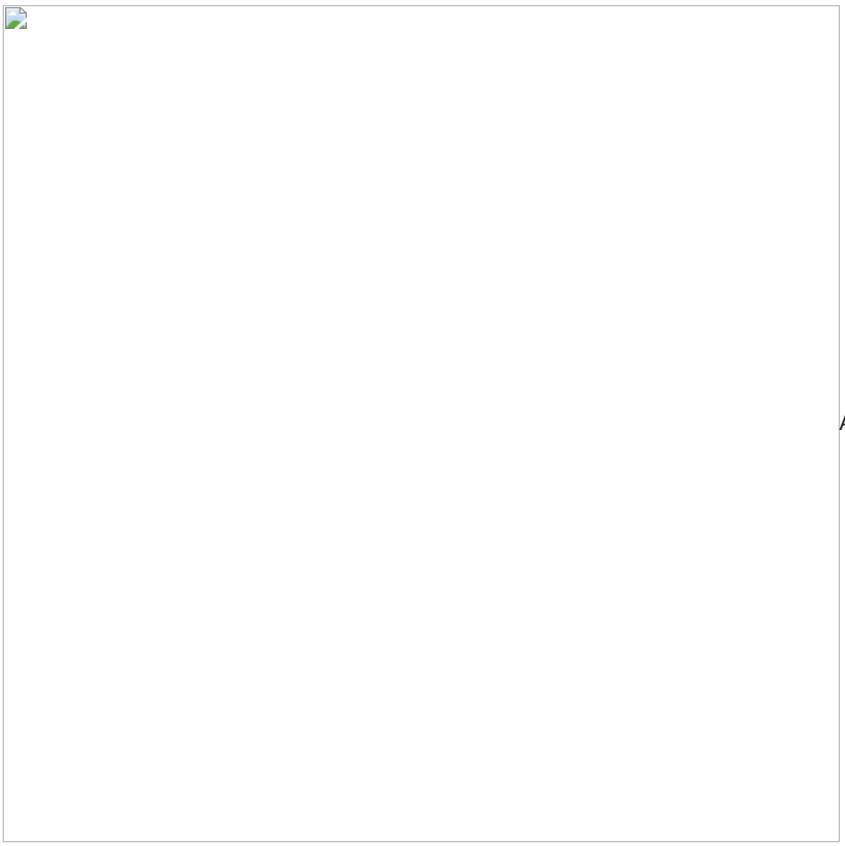
How does the computer represents strings? Well that is simple, as numbers. But numbers are much harder to read for human than strings, so these numbers are later changed to strings by every program. But if obfuscation's goal is to make code harder to read, then why don't use this trick to hide a true purpose of malicious code? This is the third obfuscation method I will present.



Example 3. Ascii code array obfuscation method

On the example above we can see a long string, with a lot of numbers in it. If you are familiar with ASCII codes, you will probable recognize them instantly. If not then your hint should be a type projection after a pipe that converts every given string from table first to int then to char. Method presented in example 3, also uses a split operator, that splits a string by a given separator to further obfuscate the code. I saw samples where a pure char array is used instead of a string that had to be split.

To deobfuscate this in python simply use similar split method (found in re library), and then map numbers to chars by using chr() function.



Ascii array with split method deobfuscation in python

A little more about the code

So now we deobfuscated the code, what we can gain from it? We can clearly see that this is a simple dropper, that uses WebClient class to connect to hardcoded domains, download a binary to %TEMP% directory and then launch it. The break instruction combined with try-catch clause assures that this script will connect to the domains provided until a download operation is completed successfully. So if it gets a binary from the first domain on the list, we will never see others in dynamic analysis. This is why deobfuscation is important.

Invoke-Expression

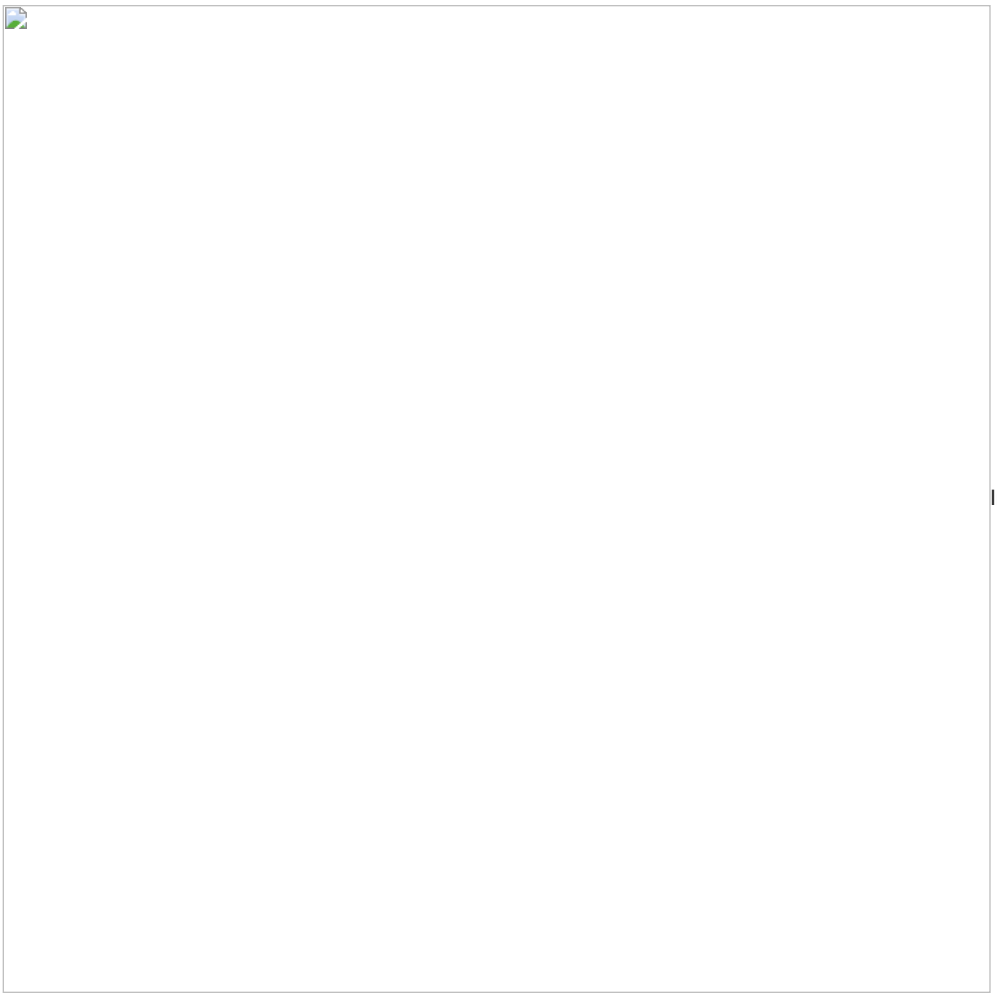
Many obfuscated powershell scripts (not only from Emotet) are using Invoke-Expression cmdlet to run an obfuscated string as a code. This is very important when we are working with powershell malicious code in the windows console, because missed invoke-expression cmdlet will launch a code instead of just displaying it. Therefore it is always important to look for disguised Invoke-Expression cmdlets. Why disguised? Because they are not always easy to spot. Firstly, powershell allows for usage of aliases for long commands. So for example built-in alias for Invoke-Expression is “iex”. But this is not the end! Powershell also allows to concatenate strings and use them as cmdlets, and strings can be stored in variables. You see the problem?

Let’s return to example with DeflateString compression. there is a following line at the beginning of the script:

```
$vERBOsepreFErEncE.tOStRIng()[1,3]+'X'-JoIn''
```

It takes a value of a powershell’s built-in variable $verbosepreference, converts it to string, takes 2nd and 4th char, concatenates it with ‘X’ and concatenates them all together to one string using join operator.

What is the default value of $verbosepreference? It turns out it is ‘SilentlyContinue’. Second and forth chars of this string are, you guessed it, ‘i’ and ‘e’. When we concatenate them with ‘x’ we receive ‘iex’ – alias of Invoke-Expression cmdlet. Creepy? Kinda. this kind of tricks in powershell are very popular among malware developers.



Invoke-Expression obfuscation example

Homework: Can you spot an Invoke-Expression cmdlet in third example (ASCII table)?

Deobfuscation script for Emotet

I put my deobfuscation script for Emotet on GitHub. You can use it and modify it as you wish. For now it automatically detects and deobfuscates all obfuscation methods described in this post.

https://github.com/lasq88/deobfuscate/