

Objective--Recover Alabaster's Password (Part 4)

Searching for a private key

So far, the encryption has been done well. The key was randomly generated (although we haven't evaluated its quality.) The malware sends an encrypted version of the key to the server and deletes its own copy of the key when it no longer needs it. It keeps a SHA-1 hash of the key so that it can verify that the server has returned the correct key when the victim pays the ransom.

However, Shinnny Upatree thinks there may be a flaw in the DNS server that will allow us to retrieve the private key.

Of course, this all depends how the key was encrypted and managed in memory. Proper public key encryption requires a private key to decrypt.

Perhaps there is a flaw in the wannacookie author's DNS server that we can manipulate to retrieve what we need.

If so, we can retrieve our keys from memory, decrypt the key, and then decrypt our ransomed files.

Let's take a look at the line in the malware that retrieved the public key.

```
$pub_key = [System.Convert]::FromBase64String($(get_over_dns("7365727665722E637274")))
```

Remember that the code has functions to convert data between different formats. We can use the malware's H2A function to read the value the function submitted to get_over_dns.

```
PS C:\Users\John\malware> H2A "7365727665722E637274"  
server.crt
```

Maybe we can ask for "server.key". Rather than convert ASCII to hex, we can put this in the command instead of converting it separately: A2H "server.key"

```
PS C:\Users\John\malware> [System.Convert]::FromBase64String($(get_over_dns(A2H "server.key") ) )  
Exception calling "FromBase64String" with "1" argument(s): "The input is not a valid Base-64 string as it contains a non-base 64 character,  
illegal character among the padding characters."  
At line:1 char:38  
+ ... [System.Convert]::FromBase64String($(get_over_dns(A2H "server.key") ) )  
+ ~~~~~  
+ CategoryInfo          : NotSpecified: (1) [], MethodInvocationException  
+ FullyQualifiedErrorId : FormatException  
  
PS C:\Users\John\malware>
```

We are receiving something, but it is failing the conversion to base64. Perhaps we should do it piece by piece.

```
PS C:\Users\John\malware> get_over_dns(A2H "server.key")
-----BEGIN PRIVATE KEY-----
MIIEvgIBADANBgkqhkiG9w0BAQEFAASCBAKggSgAgEAAoIBAQDEiNzZVubXCbMG
L4sM2Uti1R4seEZ1i2CMoDJ73qHq1+tSpwtK9y4L6znLDLW5A6uvH+1mHhhep9ui
W3vvHYCq+Ma5E1jBrvwQy0e2Cr/qeNBrdMtQs9KkxMJAz0fRJYXvtWANFJF5A+Nq
jI+jdMVtL8+PVOGwp1PA8DSW7i+9eLkqPbNDxCfFhAGG1HEU+cHOCTob0S85Hk0S
TPUKKJVc3fsD8/t60yJThCw4GKkRwG8vqcQCgAGVQeLNYJMEFv0+WHAt2WxjWtu3
HnAfMPsiEnk/y12SwHOCtaNjFR8Gt512D7idFWw4p5sT0mrrMiYJ+7x6VeMIkrw4
tk/12LYNAGMBAAECggEAHdIGcJ0X5Bj8qPudxZ1S6up1Yan+RHoZdDz6bAEj4Eyc
ODW4a0+IdRaD9mM/SaB09GwLLIt0dyhREx1+fJG1bEvDG2HFRd4fMQ0nHGAVLqaw
OTfHgb9HPuj78ImDBCEFaZHDuThdu1b0sr4RLWQScLbIb58Ze5p4AtZvpFcPt1fN
6YqS/y0i5VEFR0Wu1dMbEJN1x+xeiJp8uIs5KoL9KH1njZcEgZVQpLXzrsjKr67U
6YqS/y0i5VEFR0Wu1dMbEJN1x+xeiJp8uIs5KoL9KH1njZcEgZVQpLXzrsjKr67U
6YqS/y0i5VEFR0Wu1dMbEJN1x+xeiJp8uIs5KoL9KH1njZcEgZVQpLXzrsjKr67U
```

Bingo! We can save that with `get_over_dns(A2H "server.key") | Out-File server.key`

```
PS C:\Users\John\malware> get_over_dns(A2H "server.key") | Out-File server.key
PS C:\Users\John\malware> |
```

Now we have the encrypted key, the 512-byte hex string we recovered from memory, and the private key. We may need the public key so let's grab a copy of that.

```
PS C:\Users\John\malware> get_over_dns(A2H "server.crt") | Out-File server.crt
PS C:\Users\John\malware> |
```

Decrypting the key

We have everything we need to decrypt Alabaster's key. It isn't easy, however. Here's a quote from [this link](#):

Unfortunately there are no universal tool for all cases. This really depends on an application that was used for key file generation. For example a key file created by OpenSSL is not compatible with certutil and pvk2pfx. A key created by makecert is compatible with pvk2pfx only and so on.

Both our private and public keys are in base64 text. It would seem they should be easily transportable between Windows and Linux, but little things get in the way.

1. Text encoding varies. Linux and openssl use ASCII or UTF-8, while Windows tends to use UTF-16.
2. Line endings vary. Linux and openssl use `\n`, while Windows uses `\r\n` to mark the end of a line.
3. Headers vary. openssl requires headers like "-----BEGIN CERTIFICATE-----" while Windows sometimes omits them. The `server.crt` file we downloaded does not have headers, for example.

Since the malware is written in PowerShell, assume that the key should be decrypted using Windows tools. I could not find Windows methods that allowed decryption with only the private key (openssl does.) Instead we must combine the private and public keys into one file in PFX (also known as PKCS-12) format. Since the new file will contain the private key, Windows will want you to protect it with a password; just pick a simple password you can remember. I used "password".

Hand in

- 1) Combine the private and public keys (`server.key` and `server.crt`) into a new file called `server.pfx` using the procedure found at [this link](#). Use the procedure for `certutil.exe`, which is found on Windows by default. Include the modifier “ExtendedProperties” (without quotes) at the end of your `certutil` command. See the help using `certutil -MergePFX help`. Hand in a copy of the `server.pfx` file, and the password you used when you created it.
- 2) Decrypt the key using the function the malware used for encryption, `Pub_Key_Enc`, with some changes:
 - a. create a variable with the path to your new `server.pfx`
`$cert_path = Get-Childitem file\path\server.pfx`
 - b. to import `server.pfx`, we need to use a slightly different syntax. The import function works differently depending on what that parameters are. We are using the version “Import(String, String, X509KeyStorageFlags)” from [here](#).
`$cert.Import($cert_path, "password", 0)`
 - c. to decrypt the key, you need to load the file containing the encrypted key (512 bytes) into a variable.
 - d. the 512-byte data is in hex, but the `Decrypt` method wants binary. Use one of the malware’s conversion functions to fix that.
 - e. the syntax for the `Decrypt` function is slightly different as well.
`$Byte_key = $cert.PrivateKey.Decrypt($key_enc, $true)`
 - f. Convert `$Byte_key` to hex, and hand that in.