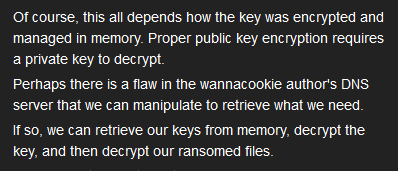
Objective--Recover Alabaster’s Password (Part 4)

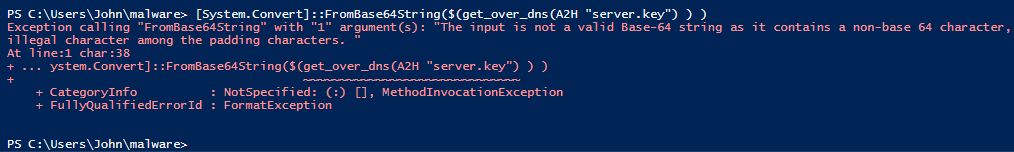
# Searching for a private key

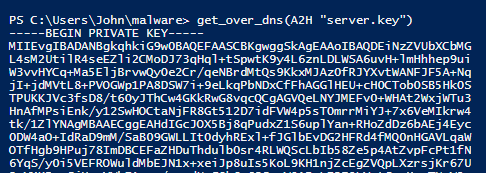
So far, the encryption has been done well. The key was randomly generated (although we haven’t evaluated its quality.) The malware sends an encrypted version of the key to the server and deletes its own copy of the key when it no longer needs it. It keeps a SHA-1 hash of the key so that it can verify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

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


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.  


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”  


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


Bingo! We can save that with get\_over\_dns(A2H “server.key”) | Out-File server.key  


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.  


# 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](https://www.sysadmins.lv/blog-en/how-to-join-certificate-and-private-key-to-a-pkcs12pfx-file.aspx):

*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](https://www.sysadmins.lv/blog-en/how-to-join-certificate-and-private-key-to-a-pkcs12pfx-file.aspx). 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:
   1. create a variable with the path to your new server.pfx  
      $cert\_path = Get-Childitem file\path\server.pfx
   2. 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](https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.x509certificates.x509certificate2.import?view=netframework-4.7.2).   
      $cert.Import($cert\_path, "password", 0)
   3. to decrypt the key, you need to load the file containing the encrypted key (512 bytes) into a variable.
   4. the 512-byte data is in hex, but the Decrypt method wants binary. Use one of the malware’s conversion functions to fix that.
   5. the syntax for the Decrypt function is slightly different as well.  
      $Byte\_key = $cert.PrivateKey.Decrypt($key\_enc, $true)
   6. Convert $Byte\_key to hex, and hand that in.