

Objective--Recover Alabaster's Password (Part 3)

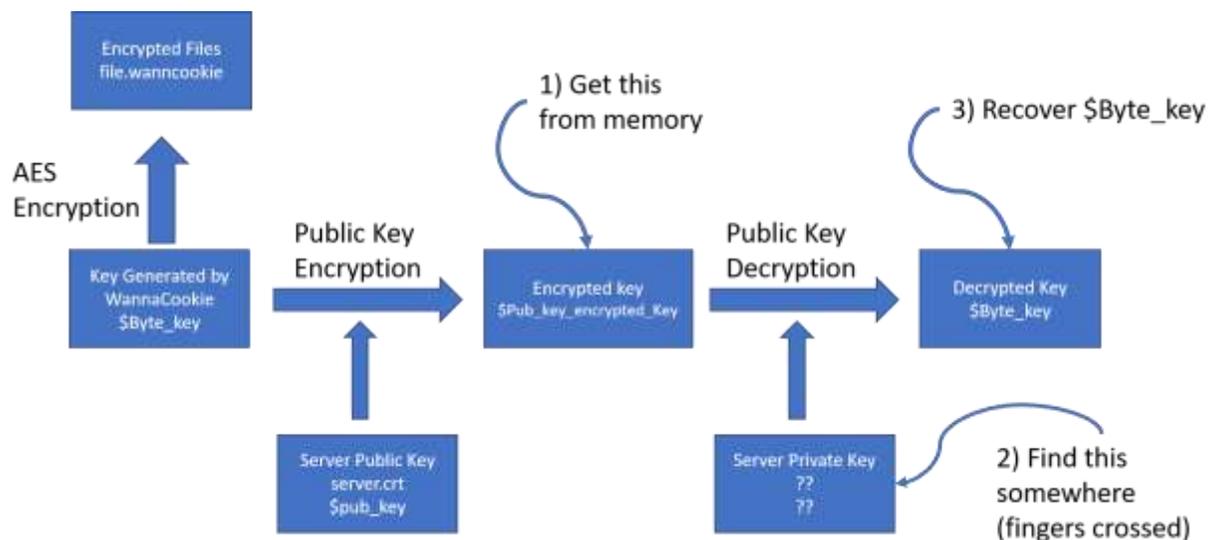
Searching for a solution in the memory dump

Code analysis has taught us that we need a key (`$Byte_key`) in order to decrypt Alabaster's files that WannaCookie encrypted with AES. However, that key was deleted from memory. The malware encrypted the key using the server's public key and sent it to the server. The code did not clear/erase the encrypted version of the key from memory. If we can find the encrypted version of `$Byte_key`, `$Pub_key_encrypted_Key`, and the companion private key to the server's public key we can recover `$Byte_key`. This is what Shinnny Upatree is telling us to do.

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.



Although the malware deleted the key we need (`$Byte_key`), it encrypted it with the server's public key and sent it to the server. Since the server has the private key that matches the public key, it can decrypt `$Byte_key` and save it for safekeeping. Farther along in the code (line 245, then 235), you can see where the server will return the unencrypted key to the malware once the ransom has been paid.

Alabaster's zip file

Once we download the zip file from Alabaster, we see that it contains the encrypted version of his password database (alabaster_passwords.elfdb.wannacookie) and the dump of the memory from the WannaCookie process on his computer (powershell.exe_181109_104716.dmp).

Data (D:) > HolidayHack2018 > Lessons > forensic_artifacts

| Name | Date modified |
|---|--------------------|
|  alabaster_passwords.elfdb.wannacookie | 12/26/2018 9:11 AM |
|  powershell.exe_181109_104716.dmp | 12/26/2018 9:12 AM |

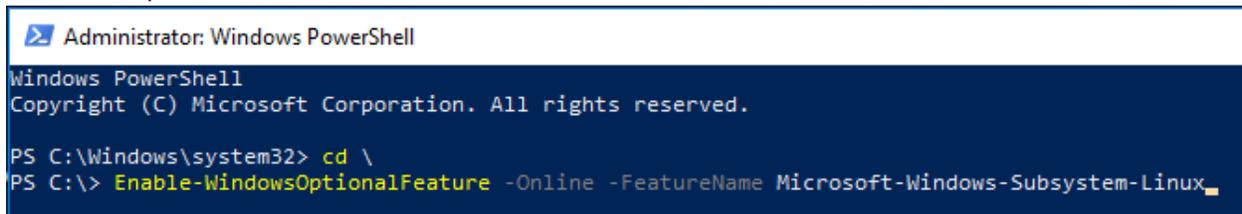
Sure enough, the encrypted database had an elfdb extension and WannaCookie appended its extension.

Chris Davis' powerdump.py script works fine in an Ubuntu VM. [In the talk](#) he uses Windows 10 and the Windows Subsystem for Linux (WSL). It is really nice to switch back and forth between Windows and Linux command shells in Windows but be careful. In a recent Sacred Cash Cow Tipping Contest (2017?) at Black Hills Information Security, they escaped antivirus detection by jumping to WSL and executing malware there. I see no problems with using WSL on a protected machine, though.

Installing Linux on Windows 10

This [link is to an article by Microsoft](#) with instructions on installing Linux on Windows 10, or WSL as Microsoft calls it. You have the choice of several different distributions. For this lab I chose Ubuntu 18.04 LTS.

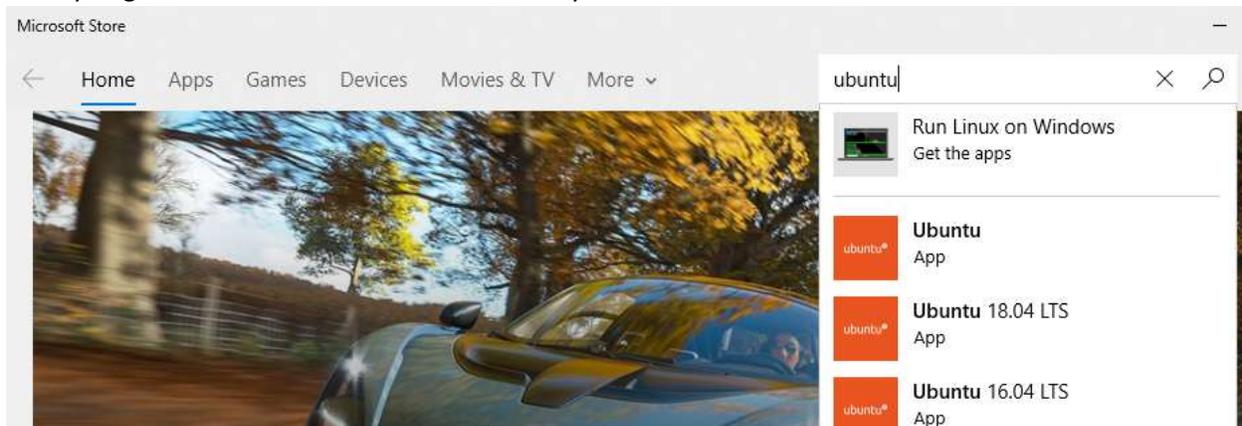
The first step is to execute a PowerShell command as Administrator.



```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> cd \
PS C:\> Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

Then you go to the Microsoft Store and choose your version. Even Kali is available.



There are more steps after that, but they are not difficult and are well documented.

Installing Power_dump

This is a [link to the Git Hub repository](https://github.com/chrisjd20/power_dump) for Chris' software. The installation is easy.

```
git clone https://github.com/chrisjd20/power_dump.git
```

```
john@DESKTOP-UR71QBS:/mnt/c/Users/John$ git clone https://github.com/chrisjd20/power_dump.git
Cloning into 'power_dump'...
remote: Enumerating objects: 60, done.
remote: Counting objects: 100% (60/60), done.
remote: Compressing objects: 100% (46/46), done.
remote: Total 60 (delta 24), reused 36 (delta 10), pack-reused 0
Unpacking objects: 100% (60/60), done.
john@DESKTOP-UR71QBS:/mnt/c/Users/John$
```

The default installation of Python on the distribution I used is python3, and power_dump.py did not run well for me in python3. We can add version 2 of Python easily.

```
sudo apt install python
```

Before we run power_dump.py, let's recall the variable table we made before. We will need to know the content type and length of the variables we are searching for. We can find hex strings (or strings of hex) but we won't be able to find byte arrays (or binary) with this tool.

| <u>Variable</u> | <u>type</u> | <u>length</u> | <u>purpose</u> |
|-------------------------|---------------------------|---------------|--|
| \$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 array of | 512 | \$b_k key encrypted with server's public cert |
| \$cookie_id | strings | 16 | \$cookie_id[15] is a string len 20, the rest are empty |
| \$date_time | string array of | 39 | date and time |
| \$future_cookies | strings | variable | file paths to be encrypted, all *.elfdb files |

We drop into BASH from PowerShell, as Chris did.

```
john@DESKTOP-UR71QBS: /mnt/c/Users/John
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

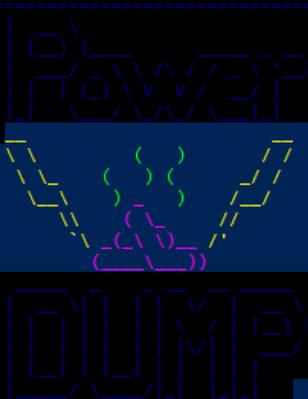
PS C:\Users\John> bash
john@DESKTOP-UR71QBS:/mnt/c/Users/John$
```

Then we start Power_dump. On my machine, power_dump.py is in ~/power_dump/, and my malware is in ~/malware. With the current working directory set to ~/malware, we can start Power_dump with

```
python ../power_dump/power_dump.py
john@DESKTOP-UR71QBS:/mnt/c/Users/John$ cd malware
john@DESKTOP-UR71QBS:/mnt/c/Users/John/malware$ python ../power_dump/power_dump.py
```

Now we are ready to start. From here on the procedure follows Chris' talk very closely.

```
john@DESKTOP-UR71Q8S: /mnt/c/Users/John/malware$ python ../power_dump/power_dump.py
```



```
Dumps PowerShell From Memory
=====
1. Load PowerShell Memory Dump File
2. Process PowerShell Memory Dump
3. Search/Dump Powershell Scripts
4. Search/Dump Stored PS Variables
e. Exit
: 1
```

Just to check, we make sure that we are in the correct directory and the dump is available.

```
=====
1. Load PowerShell Memory Dump File
2. Process PowerShell Memory Dump
3. Search/Dump Powershell Scripts
4. Search/Dump Stored PS Variables
e. Exit
: 1

===== Load Dump Menu =====
COMMAND | ARGUMENT | Explanation
=====|=====|=====
ld | /path/to/file.name | load mem dump
ls | ../directory/path | list files
B | | back to menu
===== Loaded File: =====
█

: ls

===== Listing of ./ =====
Dir - powershell_var_script_dump
File - .last_memory_processed.pickle 16447880
File - 32byte_values.txt 164
File - 40byte_values.txt 40
File - 512byte_values.txt 512
File - alabaster_passwords.elfdb.wannacookie 16420
File - CHOCOLATE_CHIP_COOKIE_RECIPE.docm 113540
File - CHOCOLATE_CHIP_COOKIE_RECIPE.zip 110699
File - dropper.ps1 516
File - firstStage.ps1 496
File - forensic_artifacts.zip 123326040
File - malware-nomin.ps1 21090
File - malware.ps1 16034
File - max-version.ps1 21090
File - powershell.exe_181109_104716.dmp 427762187
Enter to Continue...
```

We load the dump file that Alabaster gave us.

```
===== Load Dump Menu =====
COMMAND | ARGUMENT | Explanation
=====|=====|=====
ld      | /path/to/file.name | load mem dump
ls      | ../directory/path | list files
B       | | back to menu
===== Loaded File: =====
█
=====
: ld powershell.exe_181109_104716.dmp

===== Load Dump Menu =====
COMMAND | ARGUMENT | Explanation
=====|=====|=====
ld      | /path/to/file.name | load mem dump
ls      | ../directory/path | list files
B       | | back to menu
===== Loaded File: =====
powershell.exe_181109_104716.dmp 427762187
=====
: B
```

We process it and save the processed version.

```
===== Main Menu =====
Memory Dump: powershell.exe_181109_104716.dmp
Loaded      : True
Processed   : False
=====
1. Load PowerShell Memory Dump File
2. Process PowerShell Memory Dump
3. Search/Dump Powershell Scripts
4. Search/Dump Stored PS Variables
e. Exit
: 2
[i] Please wait, processing memory dump...
[+] Found 65 script blocks!
[+] Found some Powershell variable names to work with...
[+] Found 10947 possible variables stored in memory
Would you like to save this processed data for quick proces
: y

Successfully Processed Memory Dump!

Press Enter to Continue...
```

First search was with the hex string regex that Chris used. It finds 196 possible values.
 matches ``^[a-fA-F0-9]+\$``

```

===== Filters =====
1| MATCHES bool(re.search(r"^[a-fA-F0-9]+$",variable_values))

[i] 196 powershell Variable Values found!
===== Search/Dump PS Variable Values =====
COMMAND | ARGUMENT | Explanation
-----|-----|-----
print | print [all|num] | print specific or all Variables
dump | dump [all|num] | dump specific or all Variables
contains | contains [ascii_string] | Variable Values must contain string
matches | matches "[python_regex]" | match python regex inside quotes
len | len [>|<|>=|<=|==] [bt_size] | Variables length >,<,>=,<= size
clear | clear [all|num] | clear all or specific filter num
=====
  
```

Narrow the field by adding a search for length 16. Only \$cookie_id could match here. The type of \$Byte_key is byte array (binary), so strings won't find it. We found nothing at all.

```

===== Filters =====
1| MATCHES bool(re.search(r"^[a-fA-F0-9]+$",variable_values))
2| LENGTH len(variable_values) == 16

[i] 0 powershell Variable Values found!
===== Search/Dump PS Variable Values =====
COMMAND | ARGUMENT | Explanation
-----|-----|-----
print | print [all|num] | print specific or all Variables
dump | dump [all|num] | dump specific or all Variables
contains | contains [ascii_string] | Variable Values must contain string
matches | matches "[python_regex]" | match python regex inside quotes
len | len [>|<|>=|<=|==] [bt_size] | Variables length >,<,>=,<= size
clear | clear [all|num] | clear all or specific filter num
=====
  
```

Next up is len == 32. Before we can enter that, we have to clear the old len==16 line.
 clear 2 The length filter is number 2 in the screenshot; if we forget to clear it we will be looking for len == 16 and len == 32 at the same time.
 This could match \$Hex_key, but that variable was cleared. We do find five strings that match; dump them and save to 32byte_alues.txt. Note: Remember to change the file name so that the next search does not overwrite it.

```

===== Filters =====
1| MATCHES bool(re.search(r"^[a-fA-F0-9]+$",variable_values))
2| LENGTH len(variable_values) == 32

[i] 5 powershell Variable Values found!
===== Search/Dump PS Variable Values =====
COMMAND | ARGUMENT | Explanation
-----|-----|-----
print | print [all|num] | print specific or all Variables
dump | dump [all|num] | dump specific or all Variables
contains | contains [ascii_string] | Variable Values must contain string
matches | matches "[python_regex]" | match python regex inside quotes
len | len [>|<|>=|<=|==] [bt_size] | Variables length >,<,>=,<= size
clear | clear [all|num] | clear all or specific filter num
=====
.
  
```

A search for length 40 finds one string. Chances are, that is the SHA-1 hash of the key, \$Key_Hash. It may not help us but save it as 40byte-values.txt.

```

: len == 40
:
===== Filters =====
1] MATCHES bool(re.search(r"^[a-fA-F0-9]+$",variable_values))
2] LENGTH len(variable_values) == 40
:
[i] 1 powershell Variable Values found!
===== Search/Dump PS Variable Values =====
COMMAND | ARGUMENT | Explanation
-----|-----|-----
print | print [all|num] | print specific or all Variables
dump | dump [all|num] | dump specific or all Variables
contains | contains [ascii_string] | Variable Values must contain string
matches | matches "[python_regex]" | match python regex inside quotes
len | len [>|<|>=|<=|==] [bt_size] | Variables length >,<,>=,<= size
clear | clear [all|num] | clear all or specific filter num
=====

```

Finally, a search for len == 512 finds one string. Most likely it is the encrypted version of our key, \$Pub_key_encrypted_Key. This is what we were looking for. I saved it as 512byte-values.txt

```

===== Filters =====
1] MATCHES bool(re.search(r"^[a-fA-F0-9]+$",variable_values))
2] LENGTH len(variable_values) == 512
:
[i] 1 powershell Variable Values found!
===== Search/Dump PS Variable Values =====
COMMAND | ARGUMENT | Explanation
-----|-----|-----
print | print [all|num] | print specific or all Variables
dump | dump [all|num] | dump specific or all Variables
contains | contains [ascii_string] | Variable Values must contain string
matches | matches "[python_regex]" | match python regex inside quotes
len | len [>|<|>=|<=|==] [bt_size] | Variables length >,<,>=,<= size
clear | clear [all|num] | clear all or specific filter num
=====

```

Back in PowerShell we find that the string is indeed 512 bytes long.

```

PS C:\Users\John> $a = "3cf903522e1a3966805b90e7f7dd51dc7969c73cfb1663a75a56ebf4aa4a1849d1949005437dc44b8464dca05680d531
b7a971672d87b24b7a6d672d1d811e6c34f42b2f8d7f2b43aab698b537d2df2f401c2a09fbc24c5833d2c5861139c4b4d3147abb55e671d0cac709d1
cfe86860b6417bf019789950d0bf8d83218a56e69309a2bb17dcede7abfffd065ee0491b379be44029ca4321e60407d44e6e381691dae5e551cb2354
727ac257d977722188a946c75a295e714b668109d75c00100b94861678ea16f8b79b756e45776d29268af1720bc49995217d814ffd1e4b6edce9ee57
976f9ab398f9a8479cf911d7d47681a77152563906a2c29c6d12f971"
PS C:\Users\John> $a.length
512

```

We have the encrypted key now, although we can only prove that by decrypting it to get the key.

Hand in

We have the encrypted key, \$Pub_key_encrypted_Key, from memory. If we can find the server's private key, we can decrypt it. One line in Shinnny Upatree's discussion may be critical, "Perhaps there is a flaw in the wannacookie author's DNS server that we can manipulate to retrieve what we need."

- 1) Get the server, erhaetfanu.com, to give you the private key. The line from the malware that grabbed the public key may prove helpful.