my_output.exe Write-up

- The jar file will install the malware in the current directory at the time of installation. I decompiled the jar in Ghidra to see the writing of the my_output.exe executable and the invocation of the save function with no input parameter thus indicating the current directory.

- The malware attempts to reach out to the following ip address and port: ip -> 192.468.177.201, port -> 25637. I took the my_output.exe executable obtained after running java -jar final.jar, and I ran the executable within the windows 7 VM. Before running the executable though I opened up wireshark in the Kali VM to capture the traffic on the host-only network adapter that the windows 7 and Kali VMs shared. Within the traffic captured by wireshark I was able to locate the attempt by the malware to reach out to the mentioned ip address and port.

- The following commands are the numeric commands within the my_output.exe backdoor and their corresponding commands from the previously used revolution backdoor:
  - 1 = taunt
  - 2 = shell
  - 3 = systeminfo
  - 4 = pwd
  - 5 = move
  - 6 = copy
  - 7 = exit
  - 8 = cd

- First, I ran the command java -jar final.jar command within the Kali VM to obtain the my_output.exe executable. To be able to obtain the backdoor shell within the Kali VM, the default gateway had to be changed to reflect the ip that the malware tries to reach out to (as seen in question 2) as the Kali ip address. After this configuration change, I ran a netcat instance within the Kali vm to listen on the port that the malware tries to reach out to (as seen in question 2) so that the shell could be obtained. Once the netcat listener was running, I ran the my_output.exe in ImmunityDebugger and obtained the shell within the Kali VM. I then was able to test each numerical command. I verified that the commands were indeed present in the executable via decompilation within Ghidra. Within the executable, I found the switch for each command as the following cases, respectively: switchD_004019ee::caseD_1, switchD_004019ee::caseD_1, switchD_004019ee::caseD_2, switchD_004019ee::caseD_3, switchD_004019ee::caseD_5, switchD_004019ee::caseD_6, switchD_004019ee::caseD_7, and switchD_004019ee::caseD_8. Through testing each of these commands, I was able to discern their counterpart from the previously used revolution backdoor. Most were obvious through their output; however, a few commands' actions were more easily traceable through watching the execution on the stack within ImmunityDebugger.

- The registry value that the malware uses to register itself for startup on reboot is: BleachAV. Once again, decompilation of the my_output.exe executable led to the discovery of this value. The registry key corresponding to "Software\\Microsoft\\Windows\\CurrentVersion\\Run" was found in FUN_004011d0 and this key is the location of persistent programs at startup of the system. I then found the function FUN_0040115a which is called multiple times and the input parameter is an encrypted string upon calls to said function. Reversing the operations of the FUN_0040115a function with the inputs from FUN_004011d0 yields the answer of BleachAV. The input that yielded this result is the encrypted string: “e4O9i3}L”.

- The backdoor installs a copy of itself at the following location: C:\Program Files\Internet Explorer\iebav.exe. The methods utilized for determining this location included the functions discussed in my answer to question 4 which are obtained through decompilation in Ghidra. The encrypted string that was passed into the FUN_0040115a function I found to be “k:z$r6or9V AE4O)zlC(OrCO(Glu46rOrzEOs9.KOIO” and when decrypted this reads the location where the malware copies itself.

- The encoding of the ip address is done via the FUN_0040109a function, and the input parameters to this function are a hexadecimal value array and location for the storage of the value. The method FUN_0040109a runs for a total of 4 iterations, and the same operations are performed each time. The hexadecimal value at the index of index*2 is left-shifted by 8. Then an OR operation is performed with that result and the hexadecimal value at the index of index*2+1. Next, an XOR operation is performed with that result and the hexadecimal value 0x55 and right shifted by 4. The last operation performed is an AND operation with that result and the hexadecimal value 0xff. The output for each iteration is one of the four ip octets and it is appended to an array for the octets. After completion of the FUN_0040109a function, the values in the octet array are appended together and this will form the ip address. I decompiled the my_output.exe executable in Ghidra and stepping through the entry function showed the process of the ip address generation and the proceeding inet_addr call. This led me to inspect the FUN_0040109a function after suspecting that it produced the input parameter used for the inet_addr call which led to me figuring out the method by which the ip address is generated.

- The port that the malware reaches out to is encoded via the FUN_00401070 function when called with the input parameter of decimal value 9572, or hexadecimal value 0x2564. Within this function, that value is left-shifted by 8 then an OR operation is performed with a right-shifted 9572. The value obtained by this step is 2450469 in decimal and this value is returned by the function then an AND operation is performed with this output value and 0xffff in hexadecimal to obtain the port number. Once again, I decompiled the my_output.exe executable in Ghidra and stepped through the entry function. I was able to pinpoint the origin of the call to the FUN_00401070 function and through analysis of the method via reverse engineering determine it was responsible for the port number’s generation.

- The encoding of the registry value and persistent file path are encoded in the same way. A pointer that points to a hardcoded string within the executable is used as an input parameter to the FUN_0040115a function previously mentioned, and another input parameter passed is the location of where to save the decrypted string. The code within FUN_0040115a will then go character by character through the encrypted string and find the first occurrence of said character within the string stored at location 00404014. Whatever the index is from that occurrence will be used to access a character from a different string stored at location 00404060. Whatever character is indexed into in the second string will be the outputted decrypted character for the original encrypted string. A non-occurrence in the string located at 00404014 will result in the original encrypted character being used in the decrypted string. Hence, the ‘:’ character remaining as ‘:’ in the final decrypted string for the persistent file path. I decompiled the my_output.exe executable in Ghidra and stepping through the entry function led me to the calling of the FUN_0040115a function at 0040120e for the persistent file path and for the registry value at 0040127a and this led me to dig through the FUN_0040115a function and reverse its operations to determine what exactly was happening.

- Yara signature composed:

```
rule final_jar{
meta:
date = "April 2020

strings:
$first_key_string = "k:z$r6or9V AE4O)zIC(OrCO( Glu46rOrzEOs9.KOlO"
$second_key_string = "e4O9i3}L"
$third_key_string = "Software\\Microsoft\\Windows\\CurrentVersion\\Run"
$fourth_key_string = { e8 47 ff ff ff }
$fifth_key_string = { e8 db fe ff ff }
$sixth_key_string = { e8 02 fd ff ff }
$seventh_key_string = { e8 b3 fc ff ff }
$eigth_key_string = { d1 55 d7 d5 d6 45 d1 c5 }
$ninth_key_string = { 7d 65 6b 52 47 41 6a 61 49 66 4a 2f 44 4e 59 24 67 58 76 5c 4d 4c 71
64 6d 68 74 30 62 48 63 5a 78 38 50 46 7a 26 4b 79 77 70 51 35 37 57 39 73 69 53 4f 6e 6f
33 45 54 2d 34 56 43 36 75 32 72 29 28 55 2e 7b 6c 42 31 00 00 00 00 }
$tenth_key_string = { 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f 50 51 52 53 54 55 56 57
58 59 5a 30 31 32 33 34 35 36 37 38 39 5c 2d 2e 2f 26 24 7b 7d 28 29 61 62 63 64 65 66 67
68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76 77 78 79 7a 00 }
condition:
all of them
}
```

- The decryption code inside the Java is found within the AppHelp.class file and specifically within the function called run_void. Within run_void is where the extraction and decryption of data from within myimg.png occurs to produce the my_output.exe file. To find this I imported the final.jar into ghidra via batch import and after decompiling the Final.class file I traced the execution to the instantiation of a new AppHelp object and after decompiling the AppHelp.class file I then found the run_void call. Within this call, a stream is opened to read from myimg.png to write to my_output.exe and within is the decryption method to decrypt the data before writing to the executable. The python script that performs similar extraction from a similarly constructed png file can be found within the zip folder submitted for this assignment. The script is entitled ‘extract_png_payload.py’.

- The executable extracted from also-important.png was obtained via execution of the ‘extract_png_payload.py’ script submitted in the zip folder and mentioned in question 10’s answer. The input_file variable within the script was changed to read “also-important.png” and the output_file variable was changed to write to “also-important_extracted_payload.exe” rather than the myimg.png related attribute values used for the myimg PNG originally in question 10. The executable extracted from the also-important.png is included in the zip folder and is entitled ‘also-important_extracted_payload.exe’.