revbdw7.exe Write-up

Installation

The install_self function is called from address 004015de which is inside the entry function. The install_self function is itself located at address 00401482. The persistence method utilized is that it creates a registry key/value pair for the EXE in the following location: Software\\Microsoft\\Windows\\ CurrentVersion\\Run. This includes the EXE in the list of programs to run upon user login to the system. The encrypted strings are: "lc&!rwmx!|&:=@-/B/W/B/" and "y<9-|@-". These are passed into the function FUN_0040122a which then performs decryption. The first decrypted string is "C:\WINDOWS\ psvcexe.exe". The second decrypted string is "ProcSvc" and it is the value inside the newly added registry key for the backdoor to run. I ran the revolution executable and followed the code in both immunity debugger and ghidra. I came to the function call inside entry of the undefined function at address 00401482. I then stepped through to see the encrypted strings passed in, and then stepped through the decryption function to see the unencrypted outputs for each string that was being decrypted. By locating the function's location and the decryption algorithm and its use, we have found the base location of the persistence method and how it is performed. This is key insight into locating the location of the persistent file to keep the backdoor from reinitializing itself and we now have insight into the beginning of the inner workings of the revolution shell.

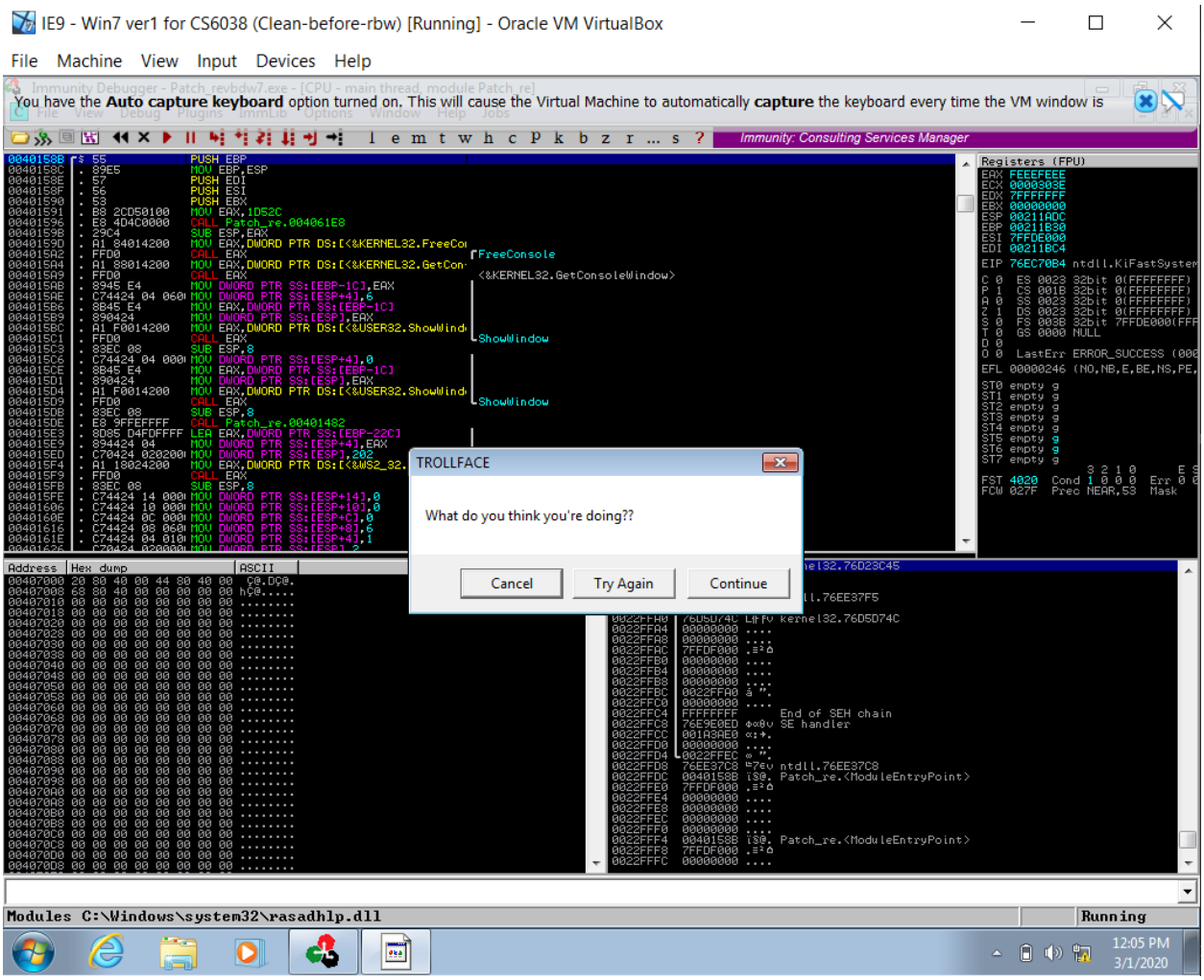
VM Detection

The routine in the GitHub repo main.c first checks the CPUID flag to see if it is set via a call to cpuid_test. Then, depending on the system, we take two different detection routes. If WIN32, then run_command function is used to run systeminfo to find the system manufacturer, and finally the registry_check function is used to check the register values for vmware. If we are not on a Windows-based system, then we call run_command to run dmesg to try and find a hypervisor, and finally we call run_command again to run dmidecode to find system manufacturer info for VMWare. During each one of these custom function calls ("checks"), we increment the vm_score attribute if necessary. After this, we check if the vm_score attribute meets the threshold to determine if the EXE should be stopped. If it does, then this symbolizes that we are likely running on a virtual machine system. In the modified revolution EXE, the vm detection routine is slightly modified. In this modified EXE, we start by making a call to IsDebuggerPresent at address 0040168f. Upon return, we execute a JE short statement which tests the output from IsDebuggerPresent and if it is executed we continue the vm detection by jumping to the address at 004016a4. If this does not jump, we then jump to 004061dd which is a function that empties the registers and returns an exit code to exit the entire program. If we made the jump to 004016a4, we call the user defined function located at address 004013c2. The function located at 004013c2 which itself seems to be for the purpose of vm detection and in turn it also performs two calls to function FUN_004012b7 and this performs the comparisons performed by the run_command in the vm-detection script located at the github repo link. At first this function is called with the following three parameters: "systeminfo | find \"tem Manufacturer\"", "System Manufacturer: \t innotek GmbH", and

the integer 36. The second call to the run_command-esque function uses the parameters: “systeminfo | find \”tem Manufacturer””, “System Manufacturer: \t VMWare, Inc.”, and the integer 36. Upon return we perform another test of register values, and if the value is satisfactory we perform another JE short statement to the address 004016BC which initiates the backdoor code by starting the WSAConnect process. Otherwise, we once again jump to the 004061dd function to pop register values and then exit the program due to “vm detection”. I had to change the IsDebuggerPresent call code all the way up to, but not including, the JE short statement. I then added an instruction to move 0 into the EAX register via “MOV EAX, 0” so that the JE short statement would execute and the path of vm detection could continue. I began by decompiling the code in Ghidra. I found the base address of the IsDebuggerPresent function call and then moved over to immunity debugger and followed the execution of the program. I followed into the function calls that perform the vm detection and followed to see the execution of the associated vm detection code. I used ghidra and immunity debugger to follow this execution and to see the parameter values passed into the various tests and functions, specifically address 004012b7 to get the strings for the systeminfo call and associated integer for the character length check. These findings are essential information for our analysis to continue. We can now connect via remote revolution shell to inspect the shell’s various commands with this knowledge of how to bypass the vm detection. We also now know that the malware is able to detect the vm system it’s being run in, and this allows us to determine that the malware might be more sophisticated than imagined.

New Command

The newly added command is the ‘taunt’ command. The call of MessageBoxA resides at address 00402e7e. The message box generated by the taunt command is in a loop, and the only condition that must be met to stop the looping is for the user to click the ‘try again’ button in the message box. The message box’s window is titled ‘TROLLFACE’ and the message within is ‘What do you think you’re doing??’. This is what the user sees upon execution of the taunt command by the adversary:



For this to work, I had to set up my Kali instance to be able to obtain a revolution shell. I made sure each vm was connected to the same host-only network adapter, and then I had to change the iptables for my Kali instance to connect to the revolution shell. I used the following command in a Kali shell to achieve that: 'iptables -t nat -I PREROUTING --dest 192.168.12.71 -p tcp --dport 444 -j REDIRECT --to-ports 8444'. This was to ensure that when the revolution backdoor reached out to 192.168.12.71 ip address that it was redirected to port 8444 of my kali instance to gain access to the windows 7 vm. Yet, before running the malware I also set up a listener in my Kali instance via the following Kali shell command: 'nc -l -p 8444'. After that, I ran the patched EXE from step 4 and got access in my Kali instance via the revolution shell. Once I had the revolution shell access, I then ran the taunt command. Upon the execution of the taunt command the message box was produced in my windows 7 vm. We can now see one of the shell's various commands and determine how to bypass its annoying behavior. We also may gain insight into the various methods that the malware might use to annoy or attack the user.

Patch EXE

The patched EXE is included in the zipped folder and is entitled “Patch_revbdw7.exe”. I went into the revbdw7.exe in immunity debugger and I changed the beginning of the IsDebuggerPresent at address 0040168f and the following lines all the way up until the next JE short statement to NOPs. I then added a line before the referenced JE short statement to mov the value 0 into the EAX register via ‘MOV EAX, 0’. This bypasses the vm detection code and allows the revolution shell to be setup. With this patched EXE we can now bypass the vm detection and really dig deeper with the analysis. We can connect with a remote shell and this is full access to understanding this specific malware.