Graded Practical 3 – Take-Home Portion

Garret Naegle

CSE 4990/6990 Software Reverse Engineering

March 21, 2014

Introduction
This report analyzes the malware sample that Mandiant attributed to the APT1 hacking group. The report will outline information about the malware sample in regards to how the malware can be patched in Immunity Debugger so that it uses a local command and control server. The report will then go through the steps taken to reverse engineer that commands the sample supports and implement them in a command and control server. The report will finally discuss how the commands were tested.

Patching the Malware Sample
The malware sample was patched in Immunity Debugger using a plugin called OllyDumpEx [1]. After putting the proper OllyDumpEx DLL in the Immunity Debugger program directory so that Immunity Debugger could access it, I proceeded to edit the original malware sample. The process of patching the sample began with editing an available section of the hex dump of the data section of the malware (0x00404360) to include the "127.0.0.1" string. Next, the references to the original command and control server IP address were looked up and found at locations 0x004010B6, 0x004019ED, 0x00401C41, and 0x00401DA5. Using this information, these locations were modified so that they point at 0x00404360.

Use of Command and Control Server
The command and control server is started by executing the CCInterface.py file using Python. This will launch the user interface window where a user can issue commands to the client and another window that will show messages from the server about the requests it is handling and other debugging and status information from the server. Because these two windows are running as different processes, closing one window does not always close the other, so it might be necessary to close the two windows individually.

Process of Creating Server through Reverse Engineering
The process of reverse engineering the sample so that the command and control server could be developed started with some static analysis. This began near the start of the WinMain function at 0x004010BB where there is a function that takes in an IP address, a string that resembles a request string, and a character pointer. At the start of this function, there is a call to a small function located at 0x00401BC5 that takes in a character pointer and an integer. It was clear that this function generated a string of random uppercase letters with its length provided by the integer argument. This was seen based on looping of the rand() call, the modulus 26 of this random number and the adding of the modulus result to the letter "A". Upon looking further at the function at 0x004040BB, it was found that many WinINet functions were being called. Based on the progression of these functions, it was found that a GET request was being sent to the IP address string with the random character string appended to the end of the request string argument as the request string. It was then found that the character pointer argument is used to store the response from the GET request. This showed the kind of GET requests the command and control server would need to handle.

Continuing with static analysis, another function is found at 0x004010CB in WinMain. This function was found to look for the presence of "<!--" and "--!>". When these were found in a string that was passed to the function, the string between "<!--" and "--!>" was parsed and passed to a function at 0x00402154. The purpose of this function was not clear and would be

better analyzed using dynamic analysis. A debugger was then used to jump to the beginning of this function and modify the data to include a string that can be passed to the function. After stepping through the program, it became evident that this function was decoding the string from base 64. A string was also found in the data of the program that illustrated the modified base 64 order. This function and the modified base 64 string indicated that the server would need to have functionality to decode and encode strings into this format.

Going back to WinMain, it is found that if this decoded string is less than seven characters, an error is thrown. The sample then looks for the string "######" at the beginning of the decoded string and if found, the sample sleeps for 2235 milliseconds. This shows a way for the server to tell the sample to sleep. When "######" is not found, the sample enters into a loop that did not seem to have a clear purpose. After this loop, there seemed to be a number of conditional statements that were looking for certain letters and based on the letter found, the sample would perform an action. It appeared that after all of these actions, a string was passed to the same function. This seemed like a good time to dig deeper with some additional dynamic analysis and to start developing a server to build off of.

This base server was created using Python's HTTPServer [2] and BaseHTTPRequestHandler [3] modules and was configured to handle the GET requests by serving a static string that was encoded using the modified base 64. This encoding was done using a function that was written as part of the command and control server and took advantage of Python's built-in base64 [4] module. Based on the appendix of the Mandiant APT1 report [5] on the sample, there is a "hello" command that sends a beacon packet. There also seemed to be a case that handled a "hello" command in the conditional statements in WinMain. Because of this, the "hello" command was chosen as the static string that the server would serve for GET requests. After getting the server up and running, some dynamic analysis was done by stepping through the execution of WinMain and the function that sends a GET request, ensuring that the sample was able to connect to the server, issue the request, receive the response, and decode it correctly. This went smoothly and the sample continued onto the steps following the decoding. The sample reached the point where it makes sure that the response string is at least seven characters long and failed because "hello" is only five characters. Dynamic analysis continued by trying to add padding characters to meet the length requirement. The size of the padding started with two characters so that the minimum length would be met. With two padding characters, it would not continue on to where the "hello" command seemed like it should have, so the number was increased. It became evident that the previously mentioned loop that did not seem to have a clear purpose was getting the seventh character in the response string, so that it could go through the conditional statements and be routed to its correct location. This indicated that there needed to be six padding characters. The "@" character was arbitrarily chosen as the padding character. Upon adding the proper amount of padding, the proper location was reached when the "hello" command was issued.

Next, the function at 0x00401312 that is called after a command is routed through WinMain was examined. It resembled the previously found function that handled sending GET requests in that it was composed of a series of WinINet functions. It was easily seen that this function is the POST version of the earlier GET function and sends the data requested in the GET request.

The server was then modified so that it included a POST request handler and checked for the correct request string with the 10 random characters. It was clear that there should be thread or process handling because the HTTP server's serve_forever() function would block execution and would not allow the user to issue commands to the server. There was an attempt to set up the server in its own thread, but when using the raw_input() function to get the command the user wished to issue, it appeared to block all threads and the server thread would stop responding until the raw_input() ended. The server and the user interface were then split up into separate Python files and the server was started as its own process by the user interface.

The rest of the commands were found by referencing the Mandiant documentation [5] and tracing the conditional statements in WinMain to see where they lead. It was found that the "pslist" command lists all the processes on the infected machine because the command leads to a function that steps through the currently running processes and formats them into a table based on process name and PID. It was also found that the "cmd.exe" and "xcmd.exe" commands create a reverse shell. This was found by looking at the function that these commands lead to. This function was discovered to create two threads. The first opens a pipe and sends the data that is written to that pipe to the command and control server. The second thread receives data from the command and control server and writes it to a pipe. With the "cmd.exe" command, the function copies the executable from the system directory into the temporary directory and renames it "Google.exe". It then goes on to start the executable with the input stream connected to the pipe that reads from the server and with the output stream connected to the pipe that sends to the server. With the "xcmd.exe" command, the function just runs the executable with the same input/output stream configuration. It appears as though it then enters a loop that ends when the executable is closed and then returns to WinMain to get additional commands. The "d" command was found to download a file from a server and save it to the infected machine. This was found by analyzing the function that the command leads to. The first portion of the function is similar to the function discussed earlier that sends a GET request in that it sends a GET request to download the file. After receiving the response from this request, the contents of the downloaded file are written to a local file and a success message is sent to the command and control server. This is also an "s" command that sleeps for a specified number of minutes. There is also the "exit" command that simply tells the sample to exit. If an invalid command is issued, an error message is sent to the command and control server.

After all the commands had been identified, the POST request handler of the server needed to better shaped so that it properly received the output from the commands. The output from the "cmd.exe" commands in the user interface was also changed such that it looked and behaved more like a real shell program. The GET request handler was also modified so that when a command is not being issued, the server simply tells the sample to sleep by sending a command with the "######" string. While analyzing the download command, some logic was added to the GET request handler so that it could also be used for serving files to be downloaded.

Testing the Commands

The "exit" command was tested by issuing the "exit" command and then verifying that the sample closed and then the server and user interface closed along with it.

Testing was done for the “hello” command by issuing the command and making sure the sample replied with a “hello” string. The testing also included sending multiple “hello” commands in a row.

The “d” command was tested by issuing the command using an address of a server that exists, a file that exists on the server, and a valid local path for the file to be saved. It was also tested using one or more invalid arguments for the server address, file on the server, and local save path to verify that the server would handle the output correctly.

The “pslist” command was tested by issuing the command and checking the output against the real list of running processes on the infected machine to verify that they are identical.

To test the “cmd.exe” command, the command was issued and the unique prompt that is only used with the “cmd.exe” commands is verified to behave the same as the normal prompt. Then additional Command Prompt commands such as “cd”, “mkdir”, and “echo” are used to test the functionality of the reverse shell. Incorrect commands were also issued to the reverse shell to better understand its behavior. If these commands behave the same as they would in a normal Command Prompt window, then the “cmd.exe” command is behaving as expected. There seems to be an issue when the “exit” command is issued to the reverse shell. Even though it can be seen in Task Manager that the “Google.exe” process is closed when the exit command is issued, the sample does not seem to acknowledge it closing. The sample gets stuck in the loop waiting for cmd.exe to complete and stops responding to commands until it is restarted.

The “xcmd.exe” command was not able to be tested. An “xcmd.exe” executable [6] was obtained from the Internet, but it appears that it requires command line arguments in order to execute and the sample does not provide any arguments to it. This means that when the sample tries to start it, xcmd.exe starts, outputs a help message, and then closes. It is possible the xcmd.exe that was obtained was the incorrect executable or version, and if the correct xcmd.exe was used, then the sample would be able to execute it.

The “s” command was tested by issuing the command with an amount of time to sleep and then observing if the sample sends or receives commands and messages during this amount of time.

In addition to all the valid commands that exist in the sample and the server, invalid commands were also issued to test that they were handled.

Conclusion

During this inspection, the sample was patched to use a local server, the sample’s commands were identified and reverse engineered so that a command and control server could be written to issue commands, and the commands from the server were tested with the sample to verify correct functionality.

References

[1] “OllyDumpEx”. Retrieved March 14, 2014. Available: http://low-priority.appspot.com/ollydumpex/

[2] "BaseHTTPServer". Retrieved March 21, 2014. Available: http://docs.python.org/2/library/basehttpserver.html

[3] “BaseHTTPServer”. Retrieved March 21, 2014. Available: https://wiki.python.org/moin/BaseHttpServer

[4] “base64”. Retrieved March 21, 2014. Available: http://docs.python.org/2/library/base64.html

[5] Mandiant, “APT1 Appendix C: The Malware Arsenal”. Retrieved March 21, 2014. Available: http://intelreport.mandiant.com/

[6] “xCmd”. Retrieved March 21, 2014. Available: http://feldkir.ch/xcmd.htm.

Sikorski, M., & Honig, A. (2012). Practical malware analysis: The hands-on guide to dissecting malicious software. San Francisco: No Starch Press.