Assignment 3

COMP 8505 – BCIT

MAO SATO, Shahin Nikvand

2016

Table of Contents

[Overview 2](#_Toc452906925)

[Design 2](#_Toc452906926)

[Constraints 2](#_Toc452906927)

[Overall Flow 2](#_Toc452906928)

[Client (Compromised Machine) 3](#_Toc452906929)

[Server (Hackers Network) 4](#_Toc452906930)

[Reverse TCP 4](#_Toc452906931)

[Remote Shell 5](#_Toc452906932)

[File Upload 5](#_Toc452906933)

[File Download 5](#_Toc452906934)

[Port Knocking 5](#_Toc452906935)

[Encryption 5](#_Toc452906936)

[Obfuscation 6](#_Toc452906937)

[Configuration 6](#_Toc452906938)

[How to Run 6](#_Toc452906939)

[Testing 6](#_Toc452906940)

[Summary 6](#_Toc452906941)

# Overview

This document details the progress and features we made on our assignment 3 for making a backdoor. Our target operating system was Linux using python as the language of choice as most Linux distributions come prepackaged with some type of python 2.7.x installed.

In this current version the script does require root account due to raw packet crafting restrictions (on the operating system level). But we are looking for ways around this limitation. We wanted something powerful and light similar to netcat.

# Design

## Constraints

The application was developed with the following constraints in mind. Please note that items verified for modification will be have details in red below them:

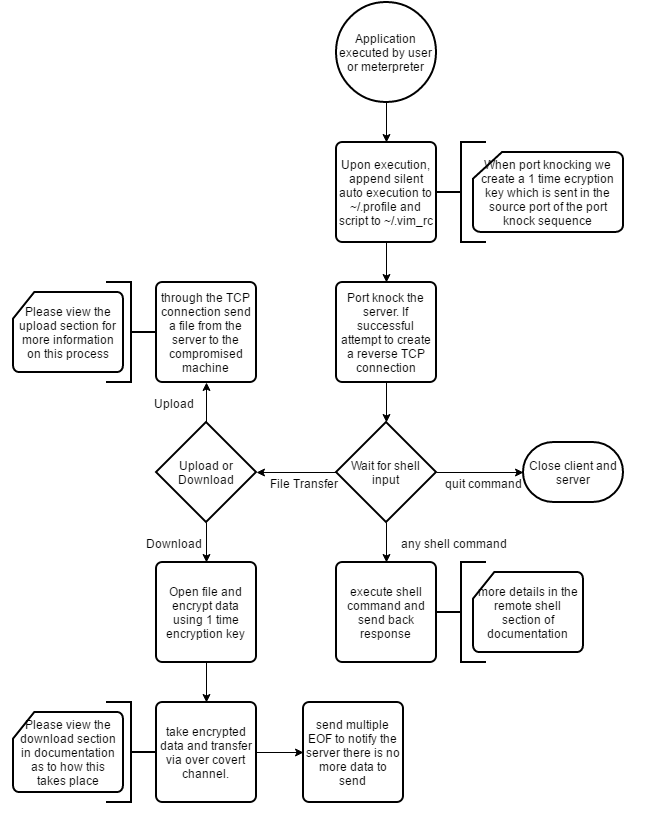
* Backdoor must camouflage itself so as to deceive anyone looking at the process table
  + Due to python limitations, I’ve tried multiple hacks and work around but most of cause problems on execution or no longer work with newer version of python. For the time being the application when executed looked like python .vim\_rc
* Application must only receive packets meant for it
  + Reverse TCP from the compromised machine
* Backdoor must interpret commands and send back the results
* Incorporate Encryption
  + Downloading from the compromised machine will use Encryption as well as a covert channel.
* Incorporate packet sniffing
  + Port knocking to the server before establishing a reverse TCP connection.
  + Downloading data from the compromised machine uses a covert channel which also requires packet sniffing

## Overall Flow

Before we go into detail of the features of our backdoor below is a simple flow chart that gives a quick summary of how the application should work.

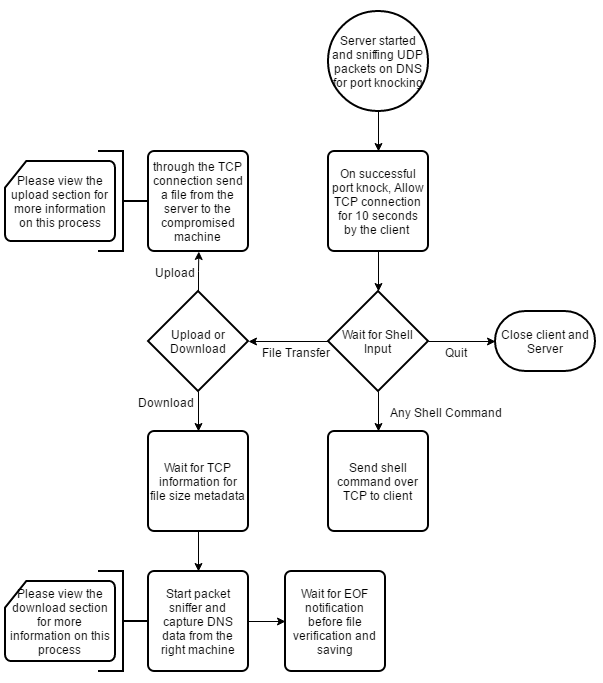
### Client (Compromised Machine)

Please view the flow chart below better understand the process of the application.



### Server (Hackers Network)

The flow chart below will briefly explain how the server works in junction with the client.



## Reverse TCP

The concept is quite simple, as most machines are behind a NAT and a firewall it’s very difficult to access them without compromising the firewall. So to make things easier the infected machine will establish a TCP connection to the hacker giving them full access to the back door.

Our current Reverse TCP works by first [port knocking](#_Port_Knocking) from the compromised machine to our server then establishing the TCP connection. Other the scan of our server will show that nothing is running there.

## Remote Shell

Once the reverse TCP connection is made the server receives a shell access to input any commands. We will be able to [upload](#_File_Upload) to the compromised computer and [download](#_File_Download) from the compromised machine. All the commands are currently exchanged over TCP in plaintext. For the final project we will be looking into using SSL to encrypt this connection as well.

Currently when the compromised machine receives a command from the server it will spawn a child process and execute the command capturing the standard out pipe and lastly terminate the process. This data is then sent back to the server. This procedure turned out much easier than having a constant process running as well as helps with keeping as little amount of processes showing on the computer if the “ps aux” command is run.

## File Upload

Upon executing the “Upload” command, using the TCP channel the file size is initially sent followed by the content. Currently this process is not encrypted and was more done as a concept for the final project. We will be looking into implementing SSL into this process to increase the security and obfuscation of the application.

## File Download

When the server requests a file from the client, the client reads and encrypts it. Taking the size of the encrypted data, over the TCP channel we send this information then we begin transferring the actual encrypted data over the covert channel disguised as UDP DNS query packets.

On the server end we have a packet sniffer listening with a UDP DNS filter and capture all the data incoming. Once the data is received we compare the size then attempt to decrypt using the AES key we received during the port knocking.

The code used in this section is using the same process that was used in the first assignment for the covert channel. We are embedding 2 hex characters in the source port of the UDP packet and sending it over.

## Port Knocking

The port knocking on our backdoor serves a few purposes. In our case, since there has to be a successful port knock sequence before anything can happen we also transfer our AES 256 encryption key in the source port as we are port knocking. This key is generated on run time and only exists during that instance of the application. This ensures that if anyone was to de-obfuscate the application they still wouldn’t know what exactly was transferred.

As usual port knocking helps hide the server against any port scans and helps hide the information under the radar. Upon successful completion of port knocking a [remote shell](#_Remote_Shell) session is started

## Encryption

We us AES 256 encryption for our [download](#_File_Download) function at the moment but we will be expanding encryption our other functions in our final project. The key is generated when the application starts (and only lasts for the duration of that instance), the key is then transferred over the port knock sequence and stored on the other side.

## Obfuscation

This section is more of a plan or concept as part of the deployment phase of our backdoor. Before deploying “Xenophid” we will run it through an obfuscator to jumble up the code as much as possible. Once the code is jumbled it will also go through a minifying process to make the payload as small and jumbled as possible.

We will experiment with this and demo it in the final project.

# Configuration

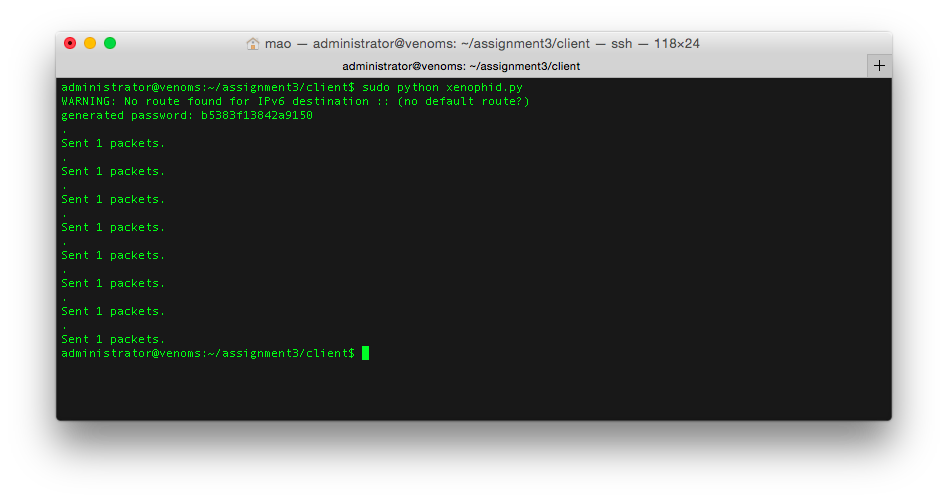
# How to Run

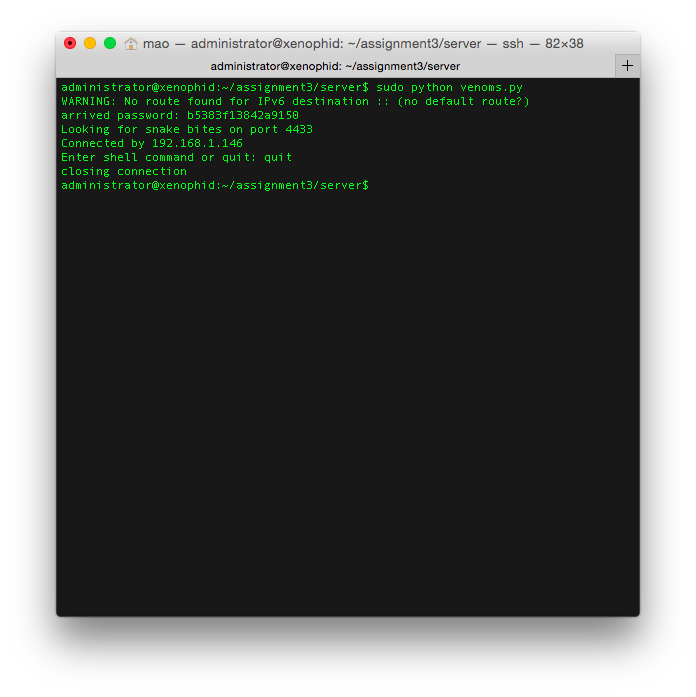
# Testing

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Purpose | Result | Pass / Fail |
| 1 | Open TCP port and create TCP connection after port knocking | Client connected to server after port knocking | Pass |
| 2 | Send one time password with port knocking packets | Same password printed on client and server after port knocking. | Pass |
| 3 | Stop the program on the server and the client when a hacker enters quit command on the server site | Both client and server program is finished | Pass |
| 4 | Upload a file when a hacker enters upload command with a valid file path | The file is uploaded to the client machine | Pass |
| 5 | Hacker enters a file path which does not exist after upload command | Threw and error:  Invalid Usage of "upload", update filename | Pass |
| 6 | Download file when a hacker enters download command with a valid file path | The file is downloaded to the hackers machine | Pass |
| 7 | Hacker enters a file path which does not exist after download command | Threw and error:  file name is invalid | Pass |
| 8 | Hacker enters command ls | Returns Show files in client’s directory | Pass |
| 9 | Hacker enters command cd | Returns nothing | Pass |
| 10 | Hacker enters command rm with a valid file name | Returns nothing but the file is deleted on client side | Pass |
| 11 | Client force closing the connection with server. | Both server and client exit the program | Pass |
| 12 | Server force closing the connection with client | Both server and client exit the program | Pass |
| 13 |  |  |  |

Test 1 – 3

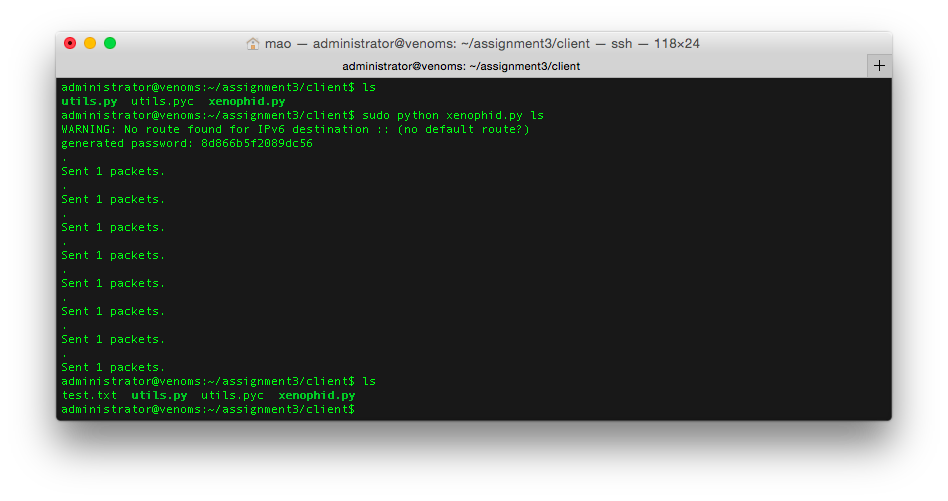
Client

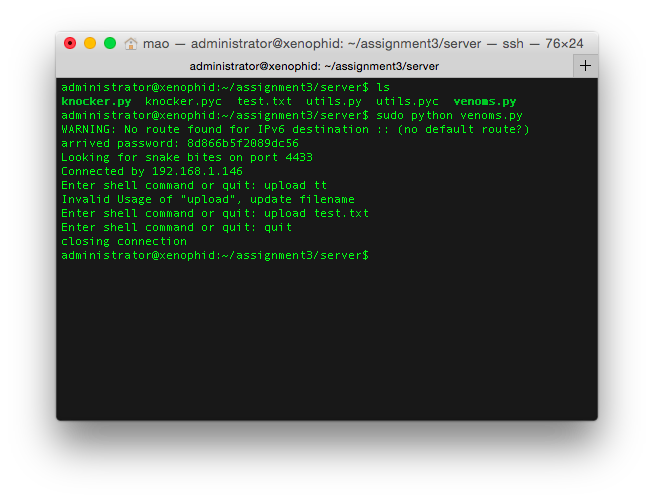


Server

Test 4-5

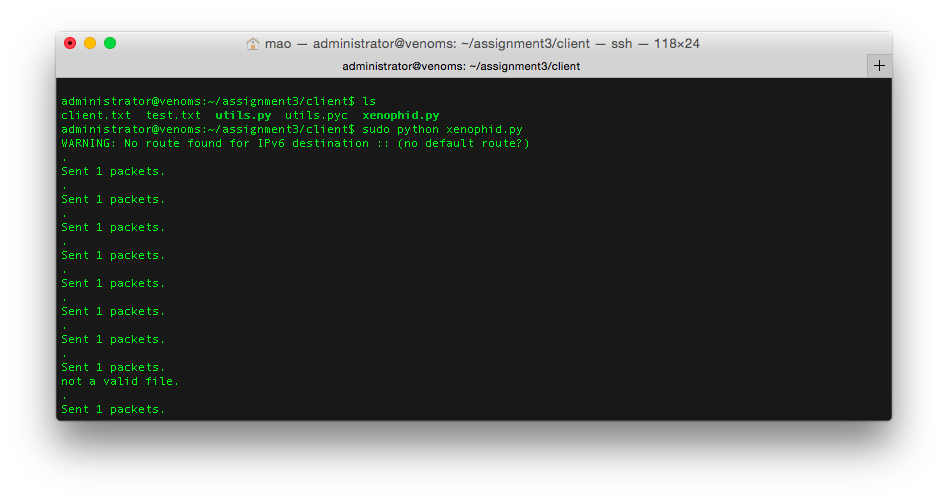
Client

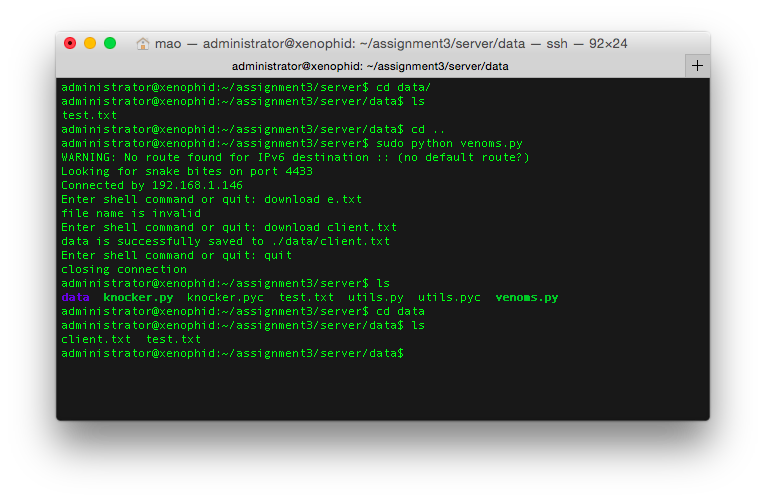


Server

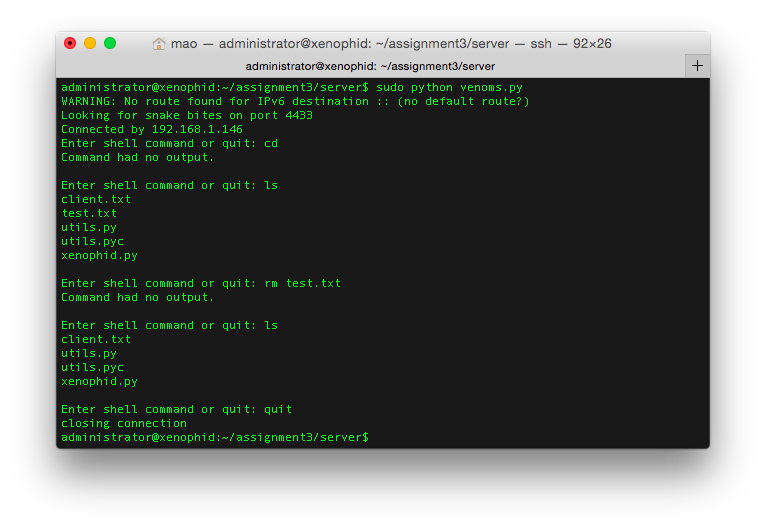
Test 6-7

Client



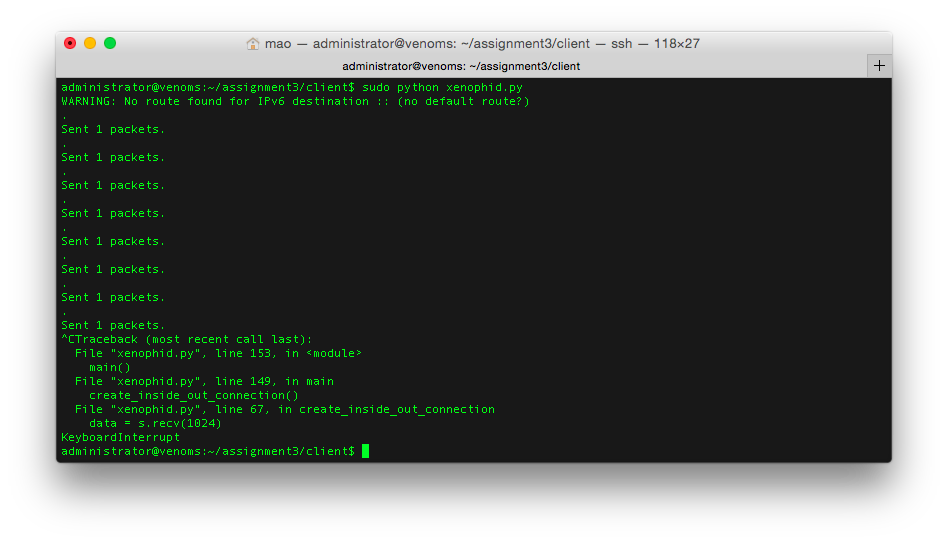
Server

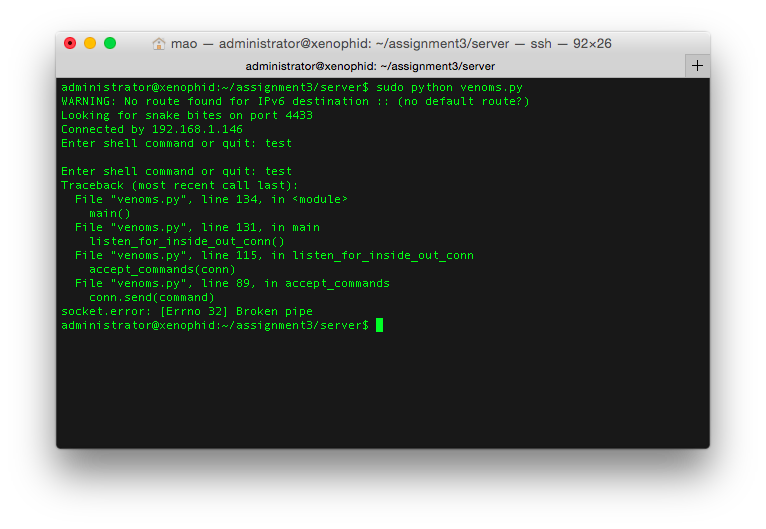
Test 8-10

Server

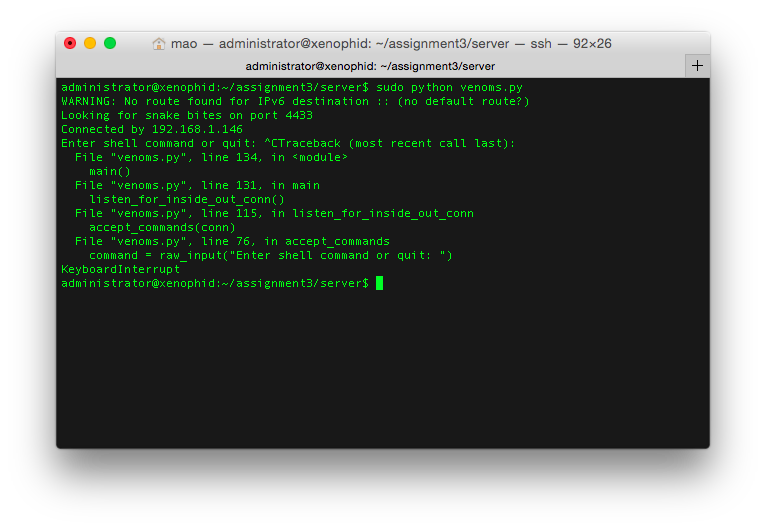
Test 11

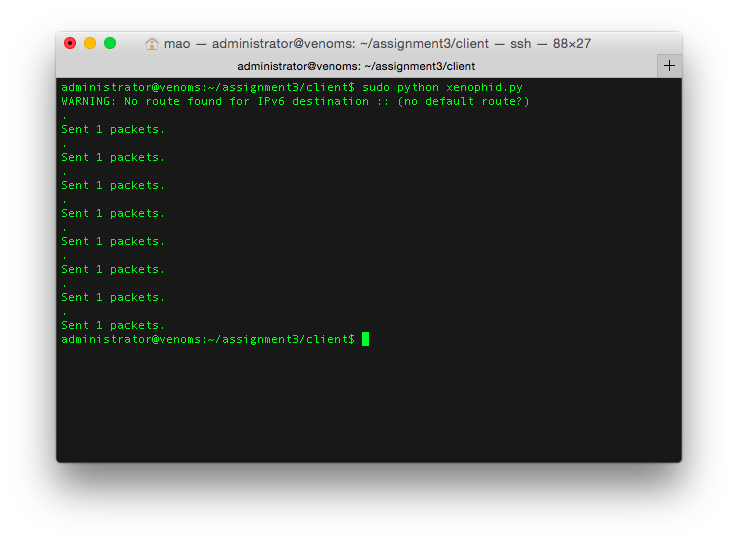
Client



Server

Test 12

Server

Client

# Summary