Final Project

CSE3810: Fundamentals of Cyber Defense

Joshua Connolly

Summary

Deceptiport is a cyber-deception tool built to detect and act on attacker reconnaissance and enumeration. The tool is written in Python and requires no modules other than those built-in. Deceptiport is designed to be run in a Linux environment as it uses IPTables to block traffic from the attacker's IP address.

The tool opens ports to act like listening services and waits for someone to try to connect to that service. It tracks how many different services an IP address has connected to and if the amount of services connected to exceeds the maximum number of allowed connections, it drops all traffic from that IP address.

Motivation

The idea for this project came from Black Hills Infosec's Active Defense and Cyber Deception course. In the course, they presented a tool called portspoof that was written in bash. The script below is the original portspoof script included in the course.

```bash
#!/bin/bash

while [ 1 ]
do
    IP=`nc -lvp 1025 2>&1 1> /dev/null | grep from | awk '{print $3;}' | tr -d "[]"`
    echo $IP
    iptables -A INPUT -p tcp -s $IP -j DROP
done
```

I wanted to expand on this to make it a little more useful and user friendly. As it stands, the script sets up a netcat listener on port 1025. When a connection is made, it strips out the IP address of the connection and passes it to iptables. If anyone connects to port 1025, they automatically get all traffic dropped by the firewall.

To improve on this tool, I wanted to be able to open multiple ports. This allows the blue team to sprinkle fake services throughout the machine while keeping real services running. Since attackers many times enumerate multiple open ports during reconnaissance, I wanted the script to add the suspected attacker's IP address to the firewall only if the attacker connected to multiple services, since most users don't need to connect to multiple services. I wanted the number of services before the firewall creates a rule for the IP address to be up to the user. An easy to use and very familiar method of passing command line arguments to the script enables the user to not only customize which ports to spoof, but also the maximum connections allowed.

Details

I wanted to abide by specific design considerations for this program. First, I wanted it to be easy to use. Next, I wanted to be able to open multiple ports and have those ports be actively listening at the same time. Finally, I wanted there to be a threshold for the user to set that dictated when someone could be deemed an attacker before being IP banned from the server.

Ease of use

In order to be easy to use, I wanted the user to be able to specify what ports the user wanted to create as fake services, and I wanted the user to be able to select how many services they deem as an unreasonable amount warranting the attacker being added to a firewall rule. In order to do this, I decided command line arguments would be a solid choice. I chose the flag -p for ports, since this is already a common flag for tools such as ssh and nmap. For maximum connections, I chose -m. I used the getopt module to parse the options and arguments.

```
try:
    opts, args = getopt.getopt(sys.argv[1:], "hp:m:")
except getopt.GetoptError:
    print "./deceptiport.py -p <ports> -m <max connections>"
    sys.exit(2)

for opt, arg in opts:
    if opt == '-h':
        usage()
    elif opt in '-p':
        ports = arg.split(",")
    elif opt in '-m':
        MAXCONN = arg
if len(opts) == 0:
    usage()
```

Above is the portion of the script responsible for the command line arguments. There are three flags, and a usage function. The usage function simply prints information on how to use the script as well as an example.

Multiple Ports

The next design consideration is for the user to be able to open multiple ports. This seemed fairly cut and dry when I began the project, but it was a little more complicated. I wanted each service to have its own set of variables to track its own state. In order to do this, I had to create a class. Each service object contains its own set of functions and variables to handle the state of the fake service.

In order for each service to run concurrently, threading was necessary. Each port the user listed in the command line argument is iterated over and passed to the createService class to instantiate a service object.

```python
for port in ports:
    t = threading.Thread(target=createService, args=(port,MAXCONN))
    t.start()
```

Each service is responsible for binding to the port it was given. The class constructor sets up the socket as an IPv4 TCP stream and creates class variables. It also binds to the port issued to it from the main function that is instantiating the service object. Finally, it enters a listen loop and continuously listens for connections using a custom written listen function for the class.

```python
def __init__(self, port, MAXCONN):
        try:
            self.MAXCONN = int(MAXCONN)
            self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.port = int(port)
            self.portConnections = []

            print "Opening port: " + str(self.port)
            self.sock.bind(('0.0.0.0', self.port))

            while True:
                self.listen(self.sock)

        except IOError as e:
            if e.errno == 13:
                print "You need elevated privileges to open port " + str(port)
                print "Try using sudo if you're a sudoer!"
            else:
                print e
```

In addition to setting the service up, the constructor also does some basic exception handling. It is very common for someone using this script to forget that ports below 1024 require root to open, so this exception handling catches error 13, permission denied, and prints a message to remind the user to use sudo for those ports.

The listen function mentioned above doesn’t do much, but it is still important. First, it takes the socket object and listens for anyone connecting to it. When someone connects, it accepts the connection and saves the IP address of the remote machine in a class variable. It then calls the handleConnection function.

The handleConnection function is where a lot of the action takes place. In order for the script to appropriately respond to a unique IP address connecting to too many services, a global variable of type dictionary was used. This dictionary contains the IP address as the key, and an attacker object as the value. The attacker class is very simple. It contains some simple getters and setters to track the IP address and the number of connections that IP currently has made. This needed to be global in order for each thread to be able to access it, as a class function would only scope to that specific service object.

```python
class Attacker:

    def __init__(self, ip):
        self.ip = ip
        self.connections = 0

    def addConnection(self):
        self.connections += 1

    def getConnections(self):
        return self.connections

    def getIP(self):
        return self.ip
```

The attacker class tracks the number of connections that specific IP has made. There was the possibility that a legitimate user tried to connect to a service multiple times, so to avoid having multiple connection attempts on the same service count against someone, a class scoped list was created to track the IP addresses locally per service. If the service had already had a connection from that IP, it does not increment the attacker.connections variable.

```python
if addr[0] not in activeConnections:
    activeConnections[addr[0]] = Attacker(addr[0])

if addr[0] not in self.portConnections:
    self.portConnections.append(addr[0])
    activeConnections[addr[0]].addConnection()
```

The code above is basically saying "if we don't have an attacker object for this IP address, create one, and if the service has not seen this IP address, add it to our IP list and increment the connection counter". This bit of logic is what is used to make sure we track individual IP connections and only count connections to a service once, even if the user tries to connect multiple times.

Dropping Traffic

Tracking the IP address of each connection and how many services they connect to is great, but we need some way to act on the information. Given the maximum number of services that we allow connections to from an IP address, we can make that decision. In the connectionHandler function after a connection is handled, we need to make sure the connection doesn't hit our limit. If the number of connections contained in the attacker object associated with the IP address of the connection is greater than or equal to the maximum allowable connections, we call the blockIP function.

```python
def blockIP(self, addr):
    cmd = "iptables -A INPUT -p tcp -s " + addr[0] + " -j DROP"     # Forms the IP tables rule based on the attacker's IP
    print cmd

    ##### Commented out for testing! Please uncomment when in production #####
    os.system(cmd)
```

The blockIP function is responsible for forming the iptables rule and enacting it via a system function call.

Future Improvements

Thread-safe

The service objects are concurrently running in threads and access a global variable. This can lead to a race condition where the connections variable of an attacker object is incremented at the same time. The danger in this happening is minimal, as it would be rare to receive two connections on different ports from the same IP address at the same time, and it would also only benefit the attacker by allowing one additional connection. Regardless, it is bad practice and a thread-safe approach should be considered.

Clean up

Some of the code is a mess as certain ideas during development did not pan out. This resulted is variables being passed to a function when class scoped variables are being used instead of the passed variables. Additionally, there may be several unused variables and functions that are depreciated.

Additional Deception

My original goal for this project was to create several services that would act like an actual service but simply lie to the user. The whole idea was to lie to an nmap service scan so that the attacker would believe he or she had found a vulnerable service and waste time attempting to exploit that service. Unfortunately, there is a lot of tests that a service undergoes to determine its version and for this project's scope, I decided to leave this for another day. I decided this after basically writing my own FTP service in Python, so it was a difficult portion of the project to put off.

User warning

In this version of deceptiport, the user can specify a maximum number of connections greater that the number of ports they open. This would effectively render the blockIP function useless. A feature should be added to alert the user when they are specifying a maximum connections number greater than the number of services being spoofed.

Conclusion

When an attacker conducts reconnaissance of a target, they do something that a normal user wouldn’t: They connect to multiple ports on the machine to determine what services are running. With this behavior in mind, we can stop attackers from being able to enumerate a machine by creating false services for the attacker to connect to and blacklisting their IP address once we’ve determined they’ve done something out of the norm, such as connecting to too many services.

References

[1] Black Hills Information Security: https://www.blackhillsinfosec.com/active-defense-cyber-deception-intro/