Special Interest Group in
Security and Information Integrity (SIG^2)
http://www.security.org.sg

SIG^2 Port Knocking Project

Remote Server Management using Dynamic Port Knocking and Forwarding

Authors: Chew Keong TAN (chewkeong@security.org.sg)
Cappella (cappella@security.org.sg)

2 May 2004

Introduction

Port knocking is a technique that can be used to hide services that are running on a hardened server. This is achieved by not opening the service port until a correct sequence of "knock" packets are received by the server. There are currently many implementations of port knocking and most of them require the client to send a fixed pre-defined sequence of port knocks to the server. The problem with this approach is that once the adversary got knowledge of the knock sequence, it would be trivial for him to replay the sequence in order to gain access to the service port.

In this paper, we present the implementation of a port knocking technique that does not require the obscurity of the port-knock sequence. Our approach does not require the clients to send a fixed pre-defined sequence of port knocks to the server. Instead, the port knocking sequence is dynamic and is determined only when the client needs the server to open a connection port.

Motivation

In certain cases, a server may need to run additional services to support tasks like remote administration. For example, SSHD may need to be enabled on a web server to allow uploading of new webpages. However, it may not be desirable to enable additional services on the web server, since these services may be subjected to attacks. Further, any vulnerabilities in these services may be exploited by the adversary to gain unauthorized access to the server.

The port-knocking technique can be used in this situation as a means of dynamically enabling or disabling the additional services as required. For example, if the Administrator needs to upload a new webpage via SCP, he would send a pre-defined sequence of port knocks to the server. Upon receiving the port knocks, the server would open the SSHD port to allow the Administrator to connect. After uploading the files, the Administrator may need to send another sequence of port knocks to close the port. This is depicted in the diagram below.

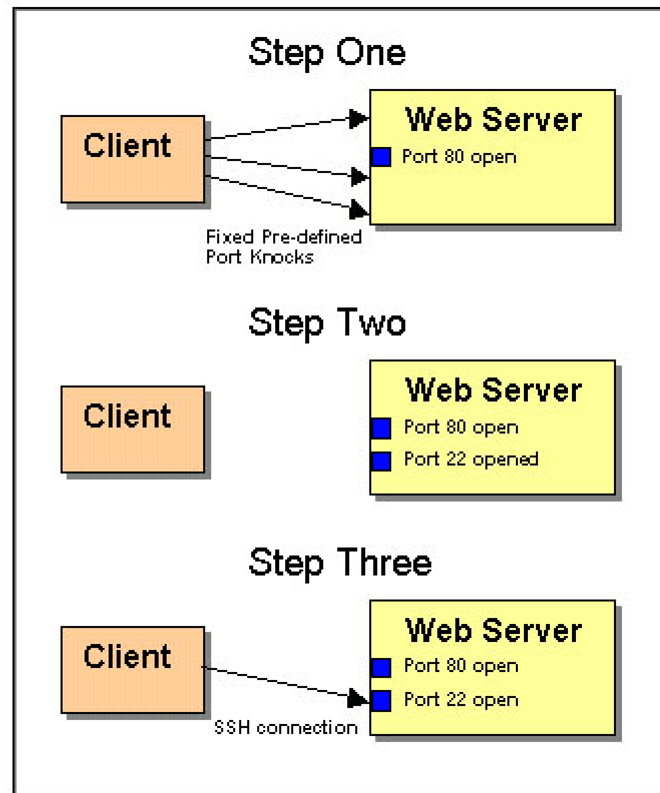


Figure 1 - Opening a "hidden" port by port knocking

Several implementations of port knocking exist for both Linux and Windows. Such implementation typically requires the port knock sequence to be fixed and pre-defined. This approach has the following issues.

1. If the adversary got hold of the port knock sequence by packet sniffing, he would be able to replay the sequence to the server and cause the "hidden" port to be opened.

2. If multiple users need to access the "hidden" port/service, the port-knock sequence must be made known to all of them. This is a problem if there are many users and when one of the users becomes un-trusted, a new port knock sequence would need to be distributed to the remaining users.

3. The service port to be opened is fixed. In the above example, a correctly received knock sequence will cause the server to open port 22 (SSHD). During the period of time the port is open, that particular service will be susceptible to attacks.

Our Port Knocking Implementation

Our implementation of port knocking is designed to overcome the issues mentioned above. In particular, we do not require the port knock sequence to be fixed and pre-defined. We also

randomize the "hidden" port, so that each correctly received knock sequence will cause to server to open a random port for the client. In other words, the SSH service can be accessed via a different port each time. This significantly reduces the possibility of the adversary finding and attacking the SSH service.

The following sequence of steps gives a high-level overview of how our implementation works.

1. Port-knock client generates a random port-knock sequence, encrypts it using the user's password hash, and sends it to the server in a single MD5 HMAC protected UDP packet. This packet shall be called P1. The client uses this packet to declare to the server the port-knock sequence it will send in step 3.

2. Server decrypts the request packet, validates the MD5 HMAC and waits for the client to send the knock sequence.

3. Client sends the agreed port-knock sequence to the server.

4. Client sends an encrypted and MD5 HMAC protected UDP packet to the server to check the status of the port-knock. This packet shall be called P2.

5. After receiving the correct port-knock packets and packet P2, the server will reply with packet P3. Packet (P3) is encrypted with the user's password hash and contains the server assigned port.

6. After receiving P3, the client can connect to the server via the assigned port.

7. The server will only allow connections to the assigned port by the client who requested the port-knock (identified by IP address). Connections from other IP's will be closed immediately. The server will perform port-forwarding to forward the incoming connection to a predefined IP and PORT.

The following diagram depicts this.

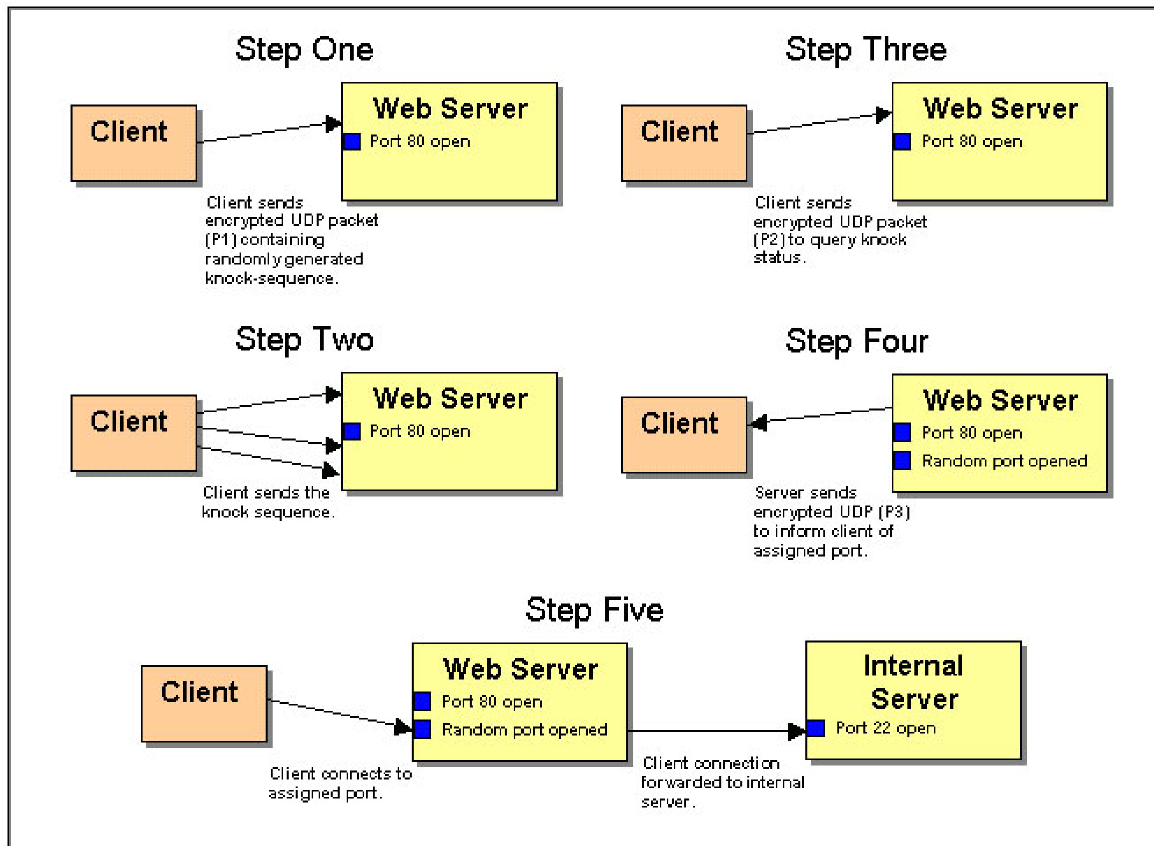


Figure 2 - Our Dynamic Port-Knocking Implementation

Design Considerations

1. **Encryption**

   The UDP packets (P1, P2 and P3) are encrypted using the user's password hash. This means that the server must have a copy of each user's password hash to perform decryption. Compromise of the server will also lead to the disclosure of the hashes and will allow the adversary to forge packets P1, P2 and P3. Hence, the file that stores all users' password hashes must be well protected on the server.

   The advantage of using the user's password hash for encryption is that there is no need to store any encryption keys on the client.

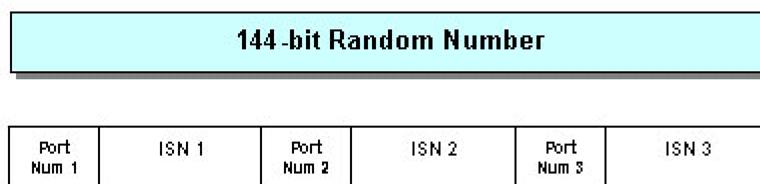
2. **Dynamic Knock Sequences**

   Our port-knock implementation does not require the client to send a fixed pre-defined sequence of port knocks to the server. Instead, the client would randomly generate the knock sequences and declare them to the server using packet P1. The port-knocks in

our implementation are a sequence of TCP SYN packets with specific Destination Port Numbers and Initial Sequence Numbers (ISNs). When the user runs our port-knock client program, three destination port numbers and three ISNs will be randomly generated. These will be encrypted and sent to the server using packet P1. Subsequently, three SYN packets with the chosen destination port numbers and ISNs will be sent to the server as port-knocks.

The decision to use three SYN packets is a trade-off between the ease of guessing the knock sequence and the difficulty of sending large number of SYN packets in sequence. When the SYN packets travel through the Internet, there is no guarantee that they will arrive in the sequence they were sent. Hence, we decide not to rely on sending a large number of SYN packets to make the random knock sequence difficult to guess.

We make use of the ISN field in each SYN packet to store a 4-byte (32-bit) random number that must match with the ones that are sent in the encrypted P1 packet. Three SYN knocks will give us 12 bytes (96 bits) of randomness. This is combined with the randomly chosen destination port number of each SYN knock, giving us a total of 12 bytes + (3 * 2-byte destination port number) = 18 bytes (144 bits) of randomness. This is shown in the following diagram.



3. **Replay Attacks**

The adversary can use replay attacks against our port-knock implementation by capturing and replaying all the UDP packets (P1, P2, and P3) and the associated TCP knock sequences. Encryption of the UDP packets prevents the adversary from forging P1, P2 and P3 without knowledge of the user's passwords, but this does not prevent him from replaying previous captured packets.

To prevent this attack from happening, we include a TimeStamp in packets P1, P2 and P3. After the server has successfully decrypted packet P1, P2 and P3, the received TimeStamp value will be compared against the last TimeStamp value received from the client. If the received TimeStamp value is the same or older than the last value, the request will be rejected.

4. **Computation Resource Starvation Attacks**

Our port-knock implementation listens for all incoming packets in promiscuous mode. Hence, there is the danger that any port-scans on the server would generate additional load due to the processing being performed by our port-knock daemon. In addition,

our implementation also requires the decryption of UDP packets P1, P2 and P3. Hence, it might be possible for the adversary to craft P1, P2 or P3 packets and sent them to our server in an attempt to cause computation resource starvation.

To protect the server against such attacks, our implementation performs several checks to weed out invalid packets as early as possible in the processing loop. In addition, we permit each client to have at most one outstanding port-knock request at any one time. This prevents a user from generating an excessive number of port-knock requests. In addition, the daemon will only process P1, P2 and P3 packets that are arriving for a particular destination port. This does not mean that the daemon listens on a particular UDP port, but rather, the daemon listens promiscuously and will only respond to UDP packets with a particular UDP destination port number.

5. **Connection from Unauthorized IP Addresses**

   Instead of opening a fixed port for the client after receiving the correct knock sequences, our implementation opens a randomly chosen port. This has an advantage since it makes it harder for the adversary to guess the correct open port. The daemon also perform IP address checks to prevent connections to the assigned port from unauthorized IP addresses. In addition, our implementation also integrates with iptables (Linux) and pktfilter (Win32) to further block connections from unauthorized IP addresses.

   Forwarding of connection from the random port to the actual service port is done in our daemon code. This makes it possible to run SSHD on 127.0.0.1 instead of running it on the public IP interface. Our port-knocking daemon can then be used to forward connections from the random port to the SSHD service.

Screen Dump

The following is a screen dump of our port-knock client.

```
[root@localhost sig2knockc]# ./sig2knockc 10.0.0.5 1001
SIG^2 KnockC Version 0.1 Copyright (c) 2004 SIG^2 (www.security.org.sg)
Linux Coding by Chew Keong TAN

User Name: foobar
Password:

Knock? (1 - Port = 15075, ISN = 54841860)
Knock? (2 - Port =  4174, ISN = E2AA5A03)
Knock? (3 - Port = 44809, ISN = 902AC754)

Can I come in?  Waiting for response.

Door is open at 10.0.0.5 Port 58201 for 30 seconds.
30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12,
11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
```

Screen dump of the port-knock daemon is shown below.

```
[d:\coding\sig2knockd\release]sig2knockd 2
SIG^2 KnockD Version 0.1 Copyright (c) 2004 SIG^2 (www.security.org.sg)
Win32 Coding by Chew Keong TAN

IP Address = 10.0.0.5

UDP_PORT                    = 1001
FORWARD_TO_IP               = 127.0.0.1
FORWARD_TO_PORT             = 22
SINGLECONN_PORTOPEN_TIME    = 30
PKTFILTER_INTERFACE         =

Opening \Device\NPF_{E7349A73-466E-4476-84CC-588608E93B58}
Server started.

2004-05-02 14:39.38 Server started.
2004-05-02 14:39.50 Port request received from foobar. (10.0.0.3, 10.0.0.3)
2004-05-02 14:39.53 Correct knock sequence received from foobar (10.0.0.3,
10.0.0.3).
2004-05-02 14:39.55 Binded port 16646 for foobar. (10.0.0.3, 10.0.0.3)
2004-05-02 14:40.19 Port request received from foobar. (10.0.0.30,
10.0.0.30)
2004-05-02 14:40.23 Correct knock sequence received from foobar (10.0.0.30,
10.0.0.30).
2004-05-02 14:40.25 Binded port 1380 for foobar. (10.0.0.30, 10.0.0.30)
2004-05-02 14:40.25 Port 16646 for 10.0.0.3 timeout and closed.
2004-05-02 14:40.39 Port request received from foobar. (10.0.0.30,
10.0.0.30)
2004-05-02 14:40.42 Correct knock sequence received from foobar (10.0.0.30,
10.0.0.30).
2004-05-02 14:40.44 Binded port 58201 for foobar. (10.0.0.30, 10.0.0.30)
2004-05-02 14:40.55 Port 1380 for 10.0.0.30 timeout and closed.
2004-05-02 14:41.14 Port 58201 for 10.0.0.30 timeout and closed.
```

Conclusion

In this paper, we presented our port-knocking implementation that does not require the client to send a fixed pre-defined port-knock sequence. Our implementation also uses dynamic port forwarding where each client is assigned a random port number to access the same service. Further work is still possible in this area, for example, the implementation of port-knock TCP wrappers that do not require the Administrator to pre-configure allowed IP addresses.

Acknowledgement

The authors would like to thank Mr. Aloysius Cheang for reviewing this paper, and the SIG^2 G-TEC Lab (http://www.security.org.sg/webdocs/g-tec.html) for supporting our research.

References

1. Krzywinski, M. 2003. Port Knocking: Network Authentication Across Closed Ports. SysAdmin Magazine 12: 12-17.
2. CMN, SAdoor A non listening remote shell and execution server
3. FX, cd00r.c

REVIEWED BY
Aloysius Cheang, CISA, CISSP, GCIH
President, SIG^2
aloysius@security.org.sg

APPROVED FOR DISSEMINATION BY
Aloysius Cheang, CISA, CISSP, GCIH
President, SIG^2
aloysius@security.org.sg