Concealing Services – Port Knocking

- One of the best ways to protect networks is to deploy firewalls and perimeters that protect sensitive portions of the network.
- However there are certain services (and therefore ports) that are essential to a company's business and therefore must be accessible through the firewall and perimeter structures. For example services such as HTTP, SMTP, SSH, and DNS tend to the most common services publicly accessible on most networks.
- Once a service is publicly accessible, it is only a matter of time before a vulnerability, such as a buffer overflow, is exposed and exploited.
- The point is that once an attacker has the opportunity to interact with a service, they will, with perseverance, find ways in which they can either compromise the system or at least bring it down with a DOS attack.
- The fact is that services with a finite user base do not need to have their ports open at all times. Public services such as SMTP or HTTP are essential and must be kept publicly accessible so as to accept connections from anyone and anywhere. They also do not require authentication.
- Services such as SSH on the other hand authenticate users and will permit only password–bearing users to access the service.
- Port knocking is a unique authentication system because the client sends one or more authentication tokens across a closed port without acknowledgement of receipt.
- The client is therefore authenticated blindly and is not aware whether authentication is taking place, or whether it is successful. This aspect of port knocking is what makes it much more difficult to detect and subvert by an intruder.
- A correctly formatted knock triggers the server to perform an action in accordance to instructions stored in the knock. The server may open or close a port for the client's IP address or perform any other action, such as send email, perform a backup or even shut down.
- Consider the idea that we are able to keep the SSH port (tcp/22) closed, thus making the service inaccessible and protected from exploits, until the service is actually requested by one of our legitimate users.
- The obvious question arises almost immediately: how is this service requested if the port is closed and no connections are possible?
- The answer lies in a technique referred to as "**port knocking**". Port knocking permits a user to request that a port be opened so that the network service can be accessed.
- The request takes the form of a passive sequence of **authentication packets** across closed ports on the server.

- It is possible to send information across closed ports, even if the ports are closed and no network services are listening, because a custom–designed **libpcap** application, or a utility like ***tcpdump*** can be configured to monitor all incoming packets, even if they are dropped by the firewall rules and never reach an application like SSH.

- Once the packets have been received and decoded, it is a simple matter of using a utility like ***iptables*** to open the requested port and then close it again.

- As a simple example, consider two ports A and B, which are both closed, with no associated listening application.

- Assume that we have a covert application that is running and configured to monitor packets arriving at these ports.

- Once these packets have been received, it is relatively straightforward to parse out the filter logs and identify the port sequence (e.g. ABAAABBAA) associated with a client IP.

- This port sequence can be used to encode information. The encoding could be a map between a specific port sequence and piece of information (e.g. ABA = open port 22 for 15 minutes, ABB = close port 22, BAA = close port 22 and do not permit additional connections).

- With port knocking, an association (in the context of a firewall) between an IP address and individual user is no longer necessary.

- The users can identify themselves using their authentication tokens without requiring any ports to be open on the server.

- Port knocking therefore allows a specific user to connect from any IP, rather than any user to connect from a specific IP.

- This is an important distinction, as the number of individual devices from which a user can establish a connection is generally increasing (office computer, home computer, laptop, PDA, cell phone, etc.).

The use of UDP v/s TCP

- There are generally two types of port knocking systems: those that use UDP and those that use TCP. A few implementations use ICMP, or combinations of protocols.
- Both protocols have benefits and neither has significant disadvantages. Port knocking implementations that use TCP make use of the SYN packet to authenticate the client, although the SYN flag is not necessarily required.
- Since the client is free to send sequences of packets, SYN or otherwise, to any remote socket, a combination of ports to which these packets are sent can be used to encode data.
- It is best to use a large number of unused ports that we can allocate for monitoring for the authentication packets.
- If authentication is done through a sequence of port knocks, theoretically all we need is two ports. However, the more ports we have the more complex, and therefore flexible, authentication data can be encoded into the same number of ports.
- If authentication is done through a packet bearing a payload, we only need a single port.
- There are a large number of implementations of port knocking currently available and these will be introduced later, including one that will be used here (Doorman).
- Since the packets are not acknowledged by the server (port knocking is passive), some implementations choose to use UDP as the protocol, which is a more natural choice of protocol when acknowledgement is not required.
- The benefits of TCP are that the TCP header is larger and can accommodate more authenticating information (sequence and acknowledgement numbers provide storage room for 4 bytes each).
- If a lot of TCP packets are sent, it's important for the server to be able to reconstruct their initial order, which can be achieved by using the sequence identification number in each packet.
- TCP systems make use of a part of the three–way handshake, a preamble in the TCP connection establishment, and an initial value for a packet order counter is set.
- TCP port knocking systems may encode information in the destination port values in a sequence of packets, or in the header or data payload of a single packet.
- UDP, on the other hand, has a much smaller header (8 bytes versus 20 bytes) and requires that data be placed in the payload of the packet.
- UDP systems typically send a single packet with authentication data located in the data payload part of the packet.
- It can be argued therefore that UDP is a more practical and elegant way to send information in a single, unacknowledged packet.

- In addition, a single UDP packet is always more stealthy than a characteristic sequence of TCP packets to various ports.

- Regardless whether UDP or TCP is used, a port knocking server allocates a set of closed ports and monitors them for specially formatted packets that constitute the port knock.

- The knock plays the role of a personalized trigger – each user may have their own individual knock constructed from their authentication tokens, or other personal information.

Proof of Concept

- The concept of port knocking is not new and several different implementations are available.

- One such implementation carried out the authentication process using a series of SYN packets. The port sequence was used to encode the authentication tokens. One such example was an 8 port knock that contained the following bytes: IP address, port, time and checksum.

- Instead of using the IP address stored in the packet's header, which can be forged, the client inserted its **IP address** into the **knock**.

- The port and time fields specified the port to which the client wanted access and the amount of time to keep the port open.

- The 1–byte checksum was present to permit the client to validate the integrity of the knock. This knock was then encrypted and encoded onto a range of closed server ports.

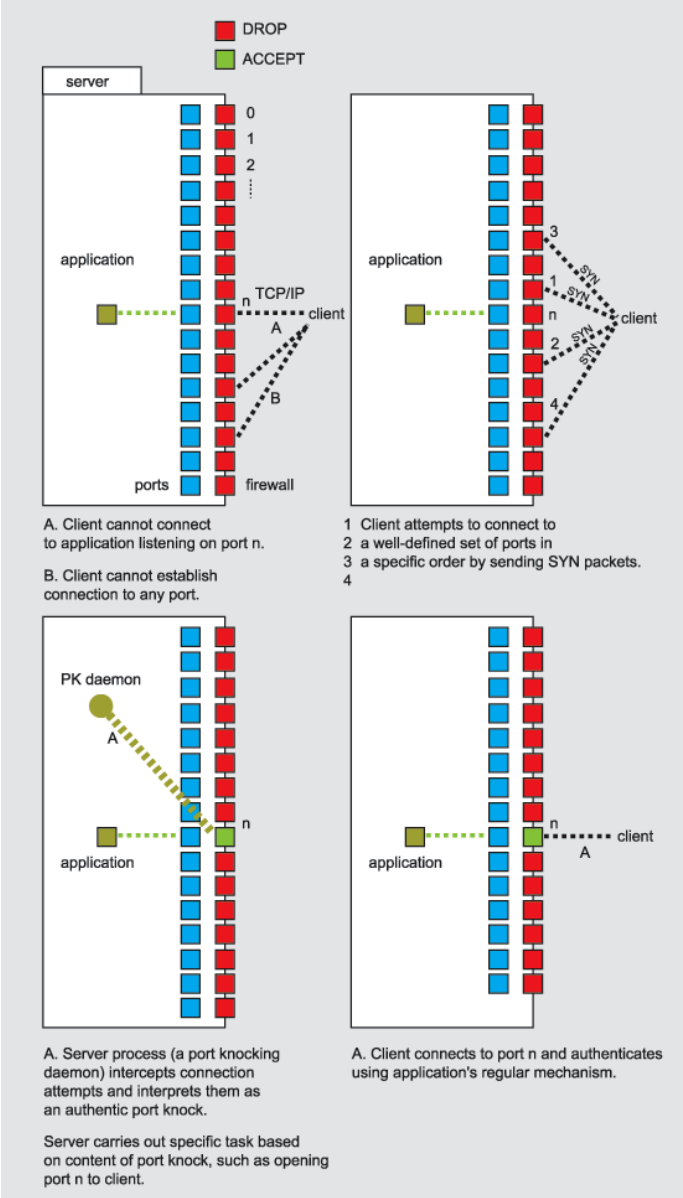
- The server, upon receiving the knock SYN packets, would extract the destination port from each packet, decode and decrypt this port sequence and then act according to the instructions encoded in the knock.

- This traditional port knocking process is illustrated below.

- Encrypting the port sequence is important to prevent spoofing and man–in–the–middle attacks, although does not address the issue of replay attacks.

- The client can craft packets using a dedicated packet generator like ***hping2***, obviating the need for a dedicated knocker.

- Incoming SYN packets can be logged to a Netfilter log file and thus can be monitored by even the most primitive file handling tools like ***grep***. Alternatively, incoming packets can be monitored directly using ***tcpdump***.



- We will use a **Netfilter**–based utility such as **iptables** to demonstrate a simple proof of concept.
- We will design as system that will allow us to establish an SSH connection to a system, which will initially block all traffic to port 22.
- We will use ***hping2*** to craft and send trigger packets to a server, which will be monitoring for these packets using ***tcpdump***.
- When an appropriately formatted trigger packet is received, the server will open the requested port to the incoming IP for 10 seconds to allow a connection to be initiated.

- After this timeout, the firewall rules will be reset and no new connections will be permitted.

- Since this application would be running on a stand–alone firewall, it would not be unreasonable to have a minimum of rules for this experiment such as:

```
# Firewall configuration written by system-config-securitylevel
# Manual customization of this file is not recommended.
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -i lo -j ACCEPT
-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
-A INPUT -s 0/0 -d 0/0 -p udp -j DROP
-A INPUT -s 0/0 -d 0/0 -p tcp --syn -j DROP
COMMIT
```

- The rule set does not permit new TCP connections or UDP packets to any port, but permits established connections to continue.

- The resulting table would look like:

```
# iptables -L
Chain FORWARD (policy DROP)
target     prot opt source     destination

Chain INPUT (policy DROP)
target     prot opt source     destination
ACCEPT     all -- anywhere   anywhere
ACCEPT     all -- anywhere   anywhere state RELATED,ESTABLISHED
DROP       udp -- anywhere   anywhere
DROP       tcp -- anywhere   anywhere tcp flags:SYN,RST,ACK/SYN

Chain OUTPUT (policy ACCEPT)
target     prot opt source     destination
```

- The trigger for opening a port of our choice will be single TCP SYN packet sent to a port in the range 1000–10999 and have the sequence number in the header set to 17.

- These two conditions are both simplistic and arbitrary and they are used here simply for pedagogical reasons.

- The objective is to make the packets reasonably unusual so that they can be distinguished from regular traffic.

- Using hping2, we send a packet to port 1022 on the server, for reasons that will become apparent shortly.

  hping2 –a 192.168.1.5 –s 1000 –p 1022 –M 17 192.168.1.10 –S

- Now, we will be using the source IP address of the packet along with the destination port to change the firewall settings on the server.

- The firewall will be opened for 10 seconds to the source IP address for connection to the destination port with the leading 10* stripped.

- Thus, the trigger packet above would permit connections from 10.1.17.1 to port 22 (1022 without the leading 10).

- We will accomplish triggering the firewall rule set with the bash script **guard.sh** as follows:

```
#!/bin/bash
# guard.sh
# allow incoming packets
/sbin/iptables -I INPUT -j ACCEPT -i eth0 -p tcp -s $1 --dport $2
sleep 10
# deny incoming packets
/sbin/iptables -D INPUT -j ACCEPT -i eth0 -p tcp -s $1 --dport $2
```

- The command line invocation to open port 22 for 10 seconds for source IP 192.168.1.5, would be as follows:

  # guard 192.168.1.5 22

- The main task now is to get **tcpdump** to trigger the execution of the script as it is scanning for packets.

- This is accomplished by piping **tcpdump** to **xargs** and constructing a **bash** command based on the contents of the output of tcpdump. This script (**trigger**) is shown. Run this as "***sh trigger***".

- This script will be running the server, listening for these trigger packets. The packet capture filter will limit the packets to SYN TCP packets addressed to the port range 1000–10999 with a sequence number 17. The tcpdump command portion the script performs this task.

- Next, the tcpdump output is handled by **sed** to remove everything from the line except the **IP address** and **port number**.

- Thus the line:

15:08:21.960173 IP 192.168.1.5.1000 > 192.168.1.10.1022: S 17:17(0) win 65535

- Will be transformed to

192.168.1.5 22

- Finally, **xargs** is used to construct a background command to call the **guard.sh** bash script.

- When a trigger packet is sent from the client to the server, tcpdump will print the packet header, sed will transform the output and **xargs** will call **guard.sh** with the resulting text as parameters.

- A port will be then opened to permit the client to connect. This port will be closed in 10 seconds, which should be enough of time for the client to establish a connection. A listing of the firewall rules would show something like:

```
# iptables -L
(...)
Chain INPUT (policy DROP)
target     prot opt source        destination
ACCEPT     tcp -- 192.168.1.5     anywhere       tcp dpt:ssh
ACCEPT     all -- anywhere        anywhere
ACCEPT     all -- anywhere        anywhere       state
RELATED,ESTABLISHED
DROP       udp -- anywhere        anywhere
DROP       tcp -- anywhere        anywhere       tcp
flags:SYN,RST,ACK/SYN
(...)
```

- The existing rule (...state RELATED, ESTABLISHED) that maintains stateful connections and accepts packets associated with established connections, will ensure that the connection is maintained even after the port access is removed.

- Clearly, this simple proof–of–concept shows that no knowledge of network programming or the TCP/IP stack is necessary to implement port knocking in its simplest form.

- However, the trigger packet is trivial to replay and offers no protection against forgery.

- The example simply serves to illustrate that one need not do a lot of work to significantly enhance the security of a service such as **ssh** by maintaining connectivity from anywhere without opening the SSH port to everyone.

Port Knocking using "doorman"

- There are a number of more sophisticated implementations of port knocking and Bruce Ward's **Doorman** takes a different and more elegant approach to the canonical port knocking specification that was just presented.
- Doorman uses a single UDP trigger packet, supports a password file, includes anti–replay measures, and provides out–of–the–box compatibility with a variety of firewalls, including Netfilter.
- Instead of sending authentication information by connection attempts to blocked ports, Doorman uses a UDP packet containing 4 strings for authentication and access control.
- The payload consists of a MD5 hash of a shared secret, created using as a key a combination of the requested port, user ID and a random number.
- When the Doorman daemon receives this type of packet, it verifies the validity of authenticating information by looking up the secret phrase for the corresponding user/group ID found in the packet and creating its own version of the MD5 hash.
- It then compares this hash with the one found in the packet and permits access to the specified port if they match.

Doorman Configuration

- The Doorman application is provided to you. The compilation and installation instructions can be found in the INSTALL and README files.
- The service binary is **doormand** and the knocker script is **knock**.
- The **doormand** daemon listens on a particular port for UDP packets with a specific payload. The UDP port is defined in configuration file **/usr/local/etc/doorman/doormand.cf**.
- The EXAMPLE file found in the same folder after installation should be used as the basis for our configuration.
- By default Doorman uses **UDP port 1001**, which need not be changed unless the port is used for some other application.
- Obviously by using a different port the chances of Doorman knocks being discovered by a potential intruder are reduced.
- Doorman requires a password file **/usr/local/etc/doormand/guestlist** to enumerate who can trigger the firewall to open for connections. A sample is provided with the package.
- Let us assume we want two users to have access to SSH: **foo** and **bar**. The other fields in the guestlist file contain the password (stored in plain text), ports that a user can request to be opened and network addresses from which the user can connect.

- The next step is to start Doorman using –D to view the debug messages:

  # /usr/local/sbin/doorman –D

- At this point Doorman is ready to receive authentication packets on the server (192.168.1.5).

- A trigger packet is sent from the client (192.168.1.10) using Doorman's knock client. For this example, the user is identified as “foo” and the request is that Doorman open port 22.

  # /usr/local/bin/knock –g foo –p 1033 –s foobar 192.168.1.5 22

- Following the trigger packet, doormand will produce an output that indicates that the packet has been intercepted and the requested port has been opened in the firewall to permit the client to connect to port 22.

- The debug output will indicate that Doorman has invoked the helper script “**iptables_add**” to achieve this:

```
May  4 16:33:30 milliways doormand[16504]:
   notice: Doorman V0.81 starting; listening on eth0 192.168.1.5, UDP
   port 1033
May  4 16:33:37 milliways doormand[16504]:
   info: knock from 192.168.1.10 :
   22 foo 0417266052 2ee0d96e063f2bc4ac7beac7104a9726
May  4 16:33:37 milliways doormand[16504]:
   debug: knock from 192.168.1.10 was valid.
May  4 16:33:37 milliways doormand[16506]:
   debug: open a secondary pcap: 'tcp and dst port 22 and src
   192.168.1.10 and dst 192.168.1.5'
May  4 16:33:37 milliways doormand[16506]:
   debug: run script:
   '/usr/local/etc/doormand/iptables_add eth0 192.168.1.5 0
   192.168.1.10 22'
May  4 16:33:37 milliways doormand[16506]:
   debug: output from script: '0'

.....
.....
May  4 16:48:17 milliways doormand[16506]:
   info: connection established, foo@192.168.1.10:34904 ->
   sshd(pid 16510)
```

- As soon as Doorman intercepts the above packet, it will invoke the necessary iptables command to open a door through the firewall. A listing of the firewall rules will produce something similar to:

```
(...)
Chain INPUT (policy DROP)
target     prot opt source       destination
ACCEPT     tcp -- 192.168.1.10   192.168.1.5    tcp dpt:ssh
ACCEPT     all -- anywhere     anywhere
ACCEPT     all -- anywhere     anywhere      state RELATED,ESTABLISHED
DROP       udp -- anywhere     anywhere
DROP       tcp -- anywhere     anywhere      tcp flags:SYN,RST,ACK/SYN
(...)
```

- Doorman will wait for 10 seconds (**waitfor** parameter in doorman.cf) before closing the firewall and returning it to its previous state.

- When the SSH connection is initiated, Doorman will detect the SYN packet and refine the firewall rule to limit the client's source ports from which packets will be permitted. Doorman will accept further packets only from the same port from which the SYN packet was sent.

- At this stage, the firewall contains the new refined rule:

```
(...)
Chain INPUT (policy DROP)
target     prot opt source       destination
ACCEPT     tcp -- 192.168.1.10    192.168.1.5    tcp spt:34904 dpt:ssh
(...)
```

- Later on, when the ssh session is terminated, Doorman will detect this and delete the refined firewall rule. The server will no longer accept packets from port 34904.

Other Implementations and Advanced Extensions

- The following table provides information on several other port knocking implementations with a variety of extensions and features:

project	language	Authentication Method					Access Method		
		protocol	header	payload	OTP	misc.	firewall rules	arbitrary command	persistent state
cd00r	C	TCP	▲			launches inetd		▲	
SAd00r	C	UDP TCP ICMP[1]	▲	▲		final authentication packet stores encrypted system command		▲	
Doorman	C	UDP		▲	▲		▲		▲
knockd	C	UDP TCP[1]	▲				▲	▲	
pasmal	C	UDP TCP ICMP	▲		▲	intrusion detection, smoke screen, webadmin	▲	▲	
tumbler	Perl/Java	UDP		▲			▲	▲	
fw/knop	C	TCP/UDP /ICMP	▲	▲		OS fingerprinting	▲	▲	

- Most port knocking client/server implementations will share certain characteristics such as authentication for example. The client and server must agree upon the form of the authentication. This could be the format of the port sequence, packet data payload, or both.
- Since validated authentication can trigger any event on the port knocking server, a mapping between such events and knocks is defined. In some implementations all knocks are mapped onto either port opening or closing, and in others knocks can be associated with arbitrary system commands.
- Knock authentication and implementation can have many forms:
- **Size**
    - The knock should be as compact as possible.

    - For knocks in the form of a packet sequence this is particularly important since fewer connection attempts mean fewer potential transmission errors and lower chance of the knock being intercepted and discovered.

    - The knock could be mapped onto as large a port range as possible (e.g. 32,768) to maximize the information content for a given number of ports.

    - For knocks composed of a single packet with a data payload, the practical limit is the average MTU (1500 bytes), which is much larger than the practical amount of information that can be encoded by a sequence of ports.

- **Encryption**

    - The knock, or data within the knock, must be encrypted to limit tainting and replay attacks. In the very least, the secret used in the knock should be encrypted.

- **Checksum**

    - This provides an additional layer of validation for packet combination knocks.

    - Once the port sequence is decoded and decrypted, if one of the values is reserved as a checksum, the contents of the knock can be verified.

    - The checksum is an additional level of protection against forged knocks.

- **One–time knock (OTK)**

    - If a knock is tagged with an index, the server need only keep track of which knock indices it has received from a certain client.

    - The client must increment the index for each new knock in order to be considered valid.

- **Means of transport**

    - The knock can be either a sequence of packets, a single packet with authentication tokens in the payload, or both.

    - Multi–packet knocks require very little effort to implement and subvert the distinction between headers and data payload (in this case the header is the data payload).

    - Traditionally, if someone is sniffing traffic for passwords or other useful information, they are likely to ignore packets that lack a data component. On the other hand, when a knock is constructed in a single packet, transmission may be more reliable and the data encoding is more flexible.

- **One–Time Passwords (OTP)**
    - This is a powerful means to combat replay attacks.
    - For example, Doorman implements an anti–replay strategy that incorporates a random number that can be used only once in the authentication string.
    - Another technique would be to run a cryptographic hash function on the previous password to calculate the next password.
    - In this way replay attacks are easily detected, because the backdoor would receive an old or stale OTP.

- There are other interesting features that could be incorporated into the design, such as an intrusion detection mechanism into the backdoor.
- **"Pasmal"** (written by James Meehan) has two interesting features that implement intrusion detection.
- One is an intrusion detection module which will prevent further connections from an IP when port scans or repeated failed knock attempts are detected.
- Another is a smoke screen feature, in which after the client has sent an initial authenticating knock, the remaining knock packets are interspersed among a large number of other packets set by the client.
- In this scheme, the client and server mutually decide which packets of the stream are part of the knock.
- Pasmal also features a web front–end for configuration.
- Another implementation, **"SAdoor"**, takes a unique approach to how the client communicates the command to be executed to the server.
- SAdoor encrypts the command in a command packet, which is the last packet of the authenticating packet sequence.
- Many of the implementations use various aspects of packets for authentication, including header and data payload to store this information.
- **fwknop** implements an additional layer of authentication based on operating system fingerprinting. By examining a variety of TCP option values found in a TCP header, fwknop can match a combination of options to a specific operating system.
- Thus, the port knocking system can include the OS from which the packet was sent as a filter.

Applications of Port Knocking

- Port knocking can be used in a number of ways, for both, enhancing security or subverting it.
- In some cases, port knocking provides security features not found in other systems. One such application is the maintenance of near–line servers, which appear off–line (no open ports) but to which connections, mediated through a port knocking layer, is possible.
- Such servers could be placed on the Internet, or within a subnet on a local network, and could act as gateways to sensitive data.
- Probably the most practical aspect of port knocking is to extend the time–to–patch without direct impact on security. By protecting a high–profile service like SSH, the administrator, who has likely more responsibilities than time, can check up on a server less frequently.
- In addition to enhancing security, port knocking can be used in malicious ways to provide undetectable backdoors.
- A backdoor so protected would not be detectable by a scan of local ports and properly installed these would be very difficult to detect and remove.
- Also note that these types of backdoors could easily be incorporated into applications and distributed.

Port–knocking Detection

- It is not possible to completely conceal a system protected by port knocking. Eventually, packet traffic must travel to or from the system, either as part of the authentication phase or as part of the connection traffic.
- Sniffers may therefore identify and detect your system by detecting any outgoing traffic, detecting incoming knocks, or detecting established connections (port scans)
- Because of the variety of port knocking methods and the relative novelty of the method, anyone wishing to identify, intercept and possibly exploit a port knocking server will have to be extremely persistent in intercepting and decoding traffic.
- Depending on the implementation, the window of opportunity to connect after a knock is validated is very narrow.
- Doorman, for example, immediately tightens the firewall rule after receiving the first SYN packet from the client.

- Therefore, for an attacker to target a system protected with port knocking it is more worthwhile to expend energies exploring standard attack vectors, such as session hijacking, then to attempt to punch a hole through a well–implemented port knocking system.

- Because port knocking is an additional layer of passive security, bringing it down by overwhelming the port knocking daemon with a DoS attack does not leave the server exposed.

- If the port knocking layer is knocked out, the server will remain inaccessible to incoming traffic until the layer can be reset. While inconvenient, this is a desirable characteristic of a security layer.

- Also note that port knocking is meant to conceal and protect network services which should require their own authentication. It may be argued that a port knocking system protecting an SSH server does not require a challenge–response phase because the SSH server implements this.

- Thus, if the SSH service is kept reasonably up–to–date, the additional port knocking layer provides significantly more protection and eliminates thread from all but the most persistent and skilled attackers.