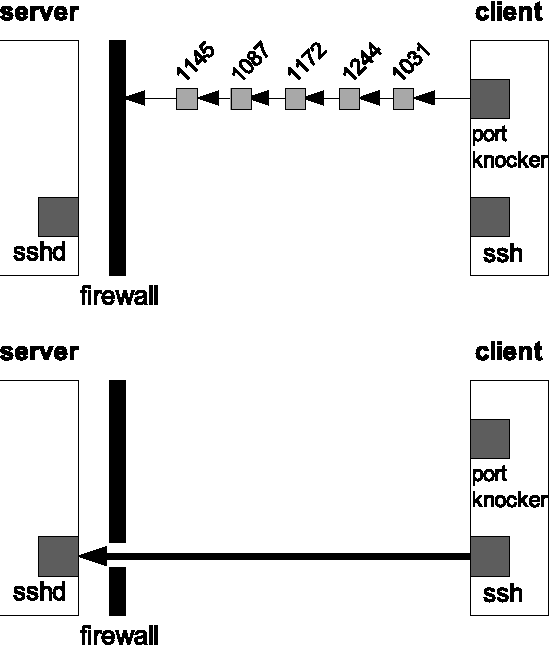
**Port Knocking Project Definition Document**

**1.1 Introduction**

The Port Knocking (PK) protocol has been a research topic for most of the past twenty years. The original implementation, although extremely unsecure, was proposed as an extra layer of security allowing users to protect their devices from untrustworthy users. Prior to this proposed protocol only the ‘firewall’ mechanism existed which will only accept network traffic from the IP addresses configured in the firewall rules. The firewall can be compromised through IP spoofing where an undesired attacker will modify the source IP address of a network packet which is accepted by the firewall rules and may easily gain access to the network. A huge proportion of servers have either no or extremely weak password protection on their devices making it even easier to access a device. IoT devices such as cheap webcams are subject to this behaviour and often silent attacks might never detected by the device owner as the system is rarely monitored by the user or the device might not recognise an undesired attacker. Therefore, an approach should be taken that involves the layering of security protocols along with the firewall to protect a device. PK is one protocol which is proposed to contribute to this layering of security on a device or server.

**1.2 Port Knocking Basic Implementation**

The original PK protocol is a simple concept of externally opening ports on a firewall by generating a sequence of connection attempts on a set of prespecified closed ports. Both the client and server have an agreed ‘secret’ which is a sequence of port numbers representing the server port number which a client wants access to. The client forms network packets (usually UDP or TCP) with a port number from the specific port sequence contained in each packet. The client makes a connection attempt to the servers closed ports. At the server side a PK daemon is monitoring the log files of the firewall or uses a packet capture tool. If the daemon recognises the sequence of packets based on the ‘secret’, the PK daemon will externally modify the firewall rules allowing a connection from the source IP address in the client packets to the server port. Essentially, the server port will become open to the client. If a knocking sequence is not recognised the daemon will ignore the sent packets. The only indication to an outside source a connection attempt has occurred is at the end of the knock sequence by doing a port scan to check if the port expected is opened or closed. Below shows the simplest implementation of the PK protocol.



**1.3 Port Knocking Security Concerns**

It was soon discovered that there were major faults in this proposed security protocol.

1. Sniffing: an undesired attacker can monitor the network using a tool such as Wireshark and easily extract the PK sequence. Therefore, this predefined ‘secret’ could become compromised.
2. Spoofing: an attacker can change the source IP address of packets pretending to be a desired user. This is dangerous if the attacker has previously sniffed out the port sequence and intends to get access to the server pretending to be another client.
3. NAT Attacks: NATing occurs when multiple devices on a network share one or limited public IP address. This has occurred due to the IPv4 address exhaustion. Therefore, when a packet is sent from a client and reaches the router the IP address often changes to a public one which is sent out externally. If a port sequence sent out has gone through NATing on the client side and the server allows access to that public IP address, any device on that network can connect to the client and the server has no way of verifying that the correct device has access.
4. DOS Attacks: occurs when an attacker overloads the server by hitting it with multiple invalid network packets while valid traffic may not make it to the server. If the server is known to be vulnerable this can have serious consequences slowing it down or worst case, causing it to fall over. Brute Force Attack is a type of DOS attack where an attacker hits a sequence of closed ports and doing a port scan to check for ant port activity changes with the intention of getting access to the server.
5. Out-of-Order Packet Delivery: can occur and is targeted to UDP packets as it was designed for speed and not reliability. Therefore, if the server receives one packet of the packet sequence out of order from the client the PK attempt is redundant.
6. Authentication vs Connection: all PK protocols developed identify a PK sequence from a client and allow a connection. However very few protocols have tried to make sure the client is who they say they are and not an undesired attacker.

**1.4 Port Knocking Improvements**

The original PK protocol has been improved throughout the years improving upon the above security concerns. Such improvements include encryption of data packet contents, single-packet knocking, single-packet authentication, use of IPsec to create a VPN tunnel, use of steganography, hashing to spoof client IP addresses in the PK sequence, timestamp synchronisation to address the out-of-order delivery and the by-passing of the firewall by opening a port to all network traffic where the PK daemon will exists. The PK protocol using the above techniques has improved immensely but can still be compromised and many of the protocols adapted use a large amount of compute power to ensure a secure connection exists.

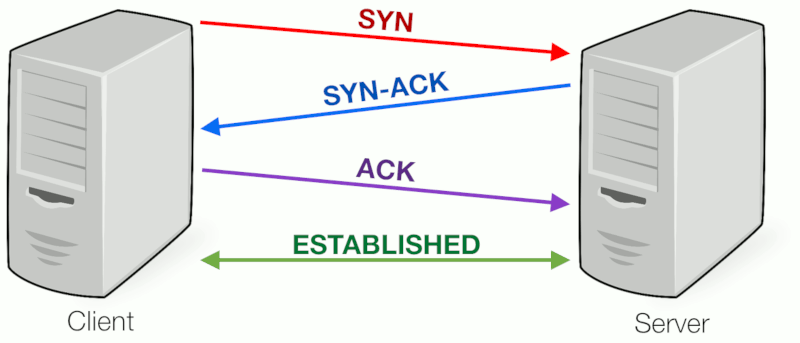
**1.5 Proposed Port Knocking Protocol**

//Paragraph on symmetric, asymmetric encryption and digital certificates.

//Paragraph on TCP vs UDP.

//PK buffer or queue?

For my proposed protocol, UDP packets will be used. The reason for this is that a TCP connection requires the three-way handshake for data to be sent between the client and server. Below shows the handshake required for a TCP connection.



In addition to this a TCP protocol will always send back an acknowledgement packet back to the client to say it successfully received a packet. The requirement for the server to send acknowledgements significantly increase the time for a PK connection to occur. Not only this but the considerable amount of packet sending, and acknowledgements can lose the main benefit of PK: being a stealth protocol. If an attacker monitoring the network sees numerous connection attempts to closed ports and acknowledgements it will give off an idea that the ‘undetectable’ security measure may exist. Sending UDP packets does not require this initial connection or acknowledgement packets reducing the time to make a PK connection. Additional UPD packets were designed for fast transmission. The protocol might only be compromised if an attacker sees several UDP packets going to a closed port but with the huge amount of network traffic that exists nowadays this should be unlikely.

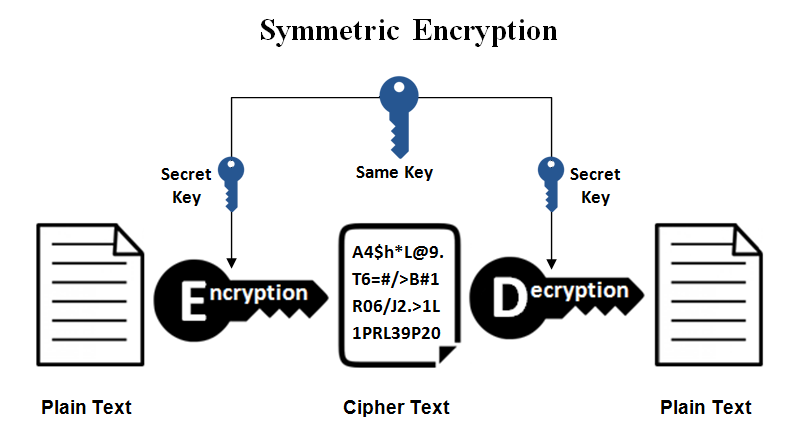
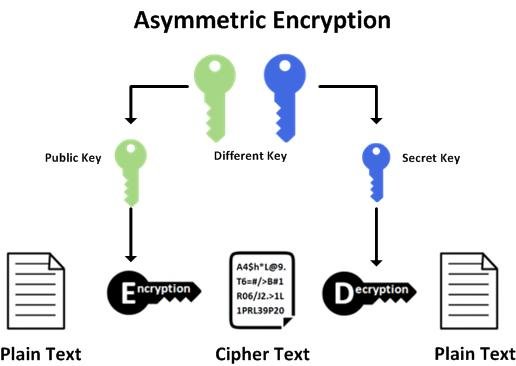
Because UDP packets tend to deliver out of order, this problem will be addressed by synchronising both the client and server to a timestamp protocol (NTP) or giving each packet in the PK attempt a sequence number. One implementation of using PK with NTP timestamp only gave accuracy to the minute. Such an accuracy would prove useless in distinguishing out of order packets. To make such a protocol is secure the timestamp would need to be accurate to at least milliseconds. This would ultimately fix the problem of out of order packets. Therefore, if the PK daemon at the server sees a packet being sent with a timestamp/sequence number older than the previous packet, such a packet and all incoming packets will be dropped silently until the proper sequence is resent. For this to occur a time interval will exist client side where it will resend the packet sequence again if a connection is unsuccessful.

The proposed protocol will use multiple packets instead of a single packet PK. If a single packet becomes intercepted and decrypted the whole message is compromised. Sending multiple packets reduces the chances of the whole message becoming established. If one packet in a multiple packet sequence is captured the whole message is not compromised.

This protocol will not use secure VPN tunnelling through IPsec protocol due to the huge overhead to implement such a structure.

Server side the daemon will closely monitor traffic and will silently DROP or ACCEPT packets based on the incoming packet sequence. If the server sent a DENY packet when an unsuccessful PK sequence is sent the client would be notified of the unsuccessful sequence and would not be able gain access to the required port. This might indicate to an attacker a PK protocol is guarding the device. Using DROP will never give such a notification back out onto the network securing the secrecy of the PK protocol.

Many of the previous PK protocols implemented are not scalable. One main reason for this is due to the use of symmetric cryptography where only one client and one server share the same public key. Client-side data contents are encrypted into the network packet using the public key and are decrypted server-side using the same public key. Firstly, the PK protocol can only scale to one server and one client using the above approach and secondly, if the public key becomes compromised an attacker can use the key to intercept and decrypt the PK sequence on the network allowing it to access the server. Therefore, I intend to use asymmetric cryptography which allows multiple clients access one server making a more scalable solution. Asymmetric encryption is based off using a public key which is publicly known and can be used by any device and a private key which is known by only the server. Here the public key is used to encrypt the message client-side and server-side the message is decrypted using the private key it owns. Therefore, if a message becomes intercepted on the network by an attacker, the public key, which it may have obtained, will not be able decrypt the message as only the private key has such functionality. Ciphertext is the data (PK sequence in our case) that has undergone encryption and will be put into the network packets. Server-side it will be the PK daemon that will decrypt the packets in the buffer.



PK protocols are dependent on the robustness of the PK daemon server-side. The failure of a daemon will cause port access to be denied to all IP addresses which is an undesirable single point of failure. The protocol implemented will restart the PK daemon if failure occurs. Can more than one instance of a PK daemon be made to deal with traffic?

On the server side instead of having the client connect to a port for ‘x’ minutes we could have a switching of ports at the server side to the one specific client. Therefore, if a normal connection would be 30 minutes, we could implement our solution to change port every 5 minutes making the client to do 6 connection attempts to the server in that 30 minutes. One solution like the way a client randomly chooses a port when it establishes a connection with a server using ephemeral port range. Here any port in the 65,000 range can be chosen. The client chooses a random port for each packet in the PK sequence (ignoring the most common ports). Each port number will be encrypted into the UDP using the public key and sent to the server. Server-side the packets are decrypted, and a log will be kept of each port number if a success port knocking sequence is sent. Now the servers port can be opened which corresponds to the PK sequence by amending the firewall rules. After every approx. 5 minutes the server firewall rule will be amended to open the random port in the first packet, the connecting port with close to the client IP so it can re-establish a connection to the newly opened port. Therefore, for every n packet sequence we can achieve n+1 connection change to different server ports.

The programming language used to implement the above protocol will need to be able apply sockets so any C type language would be best. Therefore, Java will be used as it is a well-established language with huge support and has well documented APIs around sockets.

**1.6 Notes**

We will want to check using a command line argument if a connection has been established to the specified port after a sequence has been sent from the client that sent the sequence. If we can obtain this information, we can close the port after ‘x’ number of seconds to that certain client IP address, so it is not open to attackers. Remember that we cannot close web/mail ports so if someone is trying to access these, we will not be able to close them and hope the firewall rules will DROP this connection attempt.

Exercise: opened a listening port on 500. ‘Netstat -an showed’ 0.0.0.0:500 and 0.0.0.0:0 (listening to all IP addresses whereas 127.0.0.1:500 opens to connections from localhost). Ok so, next ‘telnet localhost 500’ allowed me to connect to the port 500 on my local machine. But a new thread was created showing 127.0.0.1:500 and 127.0.0.1:63373 ESTABLISHED. Also showed this vice versa on the next line (because 2 localhost connections on machine). Therefore, a client will pick a random port (0->2^16 range) when connecting to a server. Multiple clients can connect to one server port. The client port is based on the ephemeral port range which is available ports at client-side at that specific time.

Brute force attacks: PK is a very good protocol to defend against such attacks. On a device we have 2^16 (65536) ports. If we have a 3 packet sequence an attacker must try every 3-port sequence in that port range and do a port scan between each attack to check if there has been a change in any port activity. Average case will take 141 trillion packets!! (65535^3/2).

Port knocking on a listening port, not an established one!!! Will not have to integrate with firewall then.

**1.7 Steps to Implementing PK Protocol**

* First set up a simple client-server socket application on Eclipse framework with a sending message from client to server.
* Incorporate sending a UDP into the application.
* Send UDP packets with port sequence from client to single open server port.
* Get server to recognise a successful or unsuccessful knocking sequence.
* Incorporate firewall in front of server.