**CT413 Port Knocking Project Definition Document**

|  |  |
| --- | --- |
| **Name** | Cathal Corbett |
| **Student** **ID** | 16321973 |
| **Year** | 4th Year |
| **Class** | 4BCT Computer Science |

Table of Contents

[1.1 Introduction 2](#_Toc24103215)

[1.2 Basic Port Knocking Implementation 2](#_Toc24103216)

[1.3 Port Knocking Security Concerns 3](#_Toc24103217)

[1.4 Port Knocking Improvements 4](#_Toc24103218)

[1.5 Proposed Port Knocking Protocol 6](#_Toc24103219)

[1.6 Current Steps to Implementing PK Protocol 9](#_Toc24103220)

[**1.7 References** 9](#_Toc24103221)

## 

# Chapter 1 – Introduction & Literature Review

## 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 internet users. Prior to this proposed protocol the ‘firewall’ was the only mechanism that existed but will only accept network traffic from client IP addresses manually 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. The main aim of PK as a security protocol is to allow firewall rules to be amended dynamically to allow desired traffic access to the server and to authenticate a user wishing to gain access to a port.

## 1.2 Basic Port Knocking 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 [1]. Both the client and server have an agreed ‘secret’ which is a sequence of port numbers representing a single server port number which a client wants access to. The client forms network packets (usually TCP or UDP) with a port number from the specific port sequence contained in each packet. The client makes a connection attempt to the server’s 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. Figure 1 below [2] shows the simplest implementation of the PK protocol.

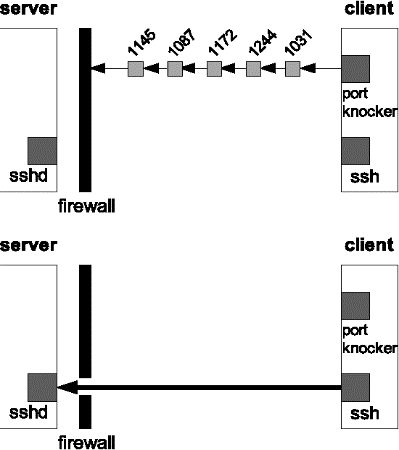


Figure - Basic Port Knocking.

## 1.3 Port Knocking Security Concerns

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

1. Sniffing: an undesired attacker can monitor the network using a tool such as Wireshark and easily extract the PK sequence if unencrypted. 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]
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. [2] [4] [3]
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 any port activity changes with the intention of getting access to the server. [4] [3]
5. Replay Attack: network attack where valid data is transmitted maliciously or fraudulently repeated or delayed. In terms of PK an attack may try to replay a sniffed sequence and gain unauthorized access to a server. [4]
6. Out-of-Order Packet Delivery: is targeted to UDP packets as they were designed for speed and not reliability and the protocol does not guarantee successful delivery of the packet. Packets sent out after one another can become out of order if they take different network paths. This can occur if part of the network experiences downtime/uptime in the middle of sending out a PK sequence, the shortest path needs to be recalculated and can cause the packets to become disordered. Therefore, if the server receives one packet of the packet sequence out of order from the client the PK attempt is redundant. [2] [3]
7. 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. [2]

## 1.4 Port Knocking Improvements

The original PK protocol has been advanced 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 security of 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.

The below table [5] shows how PK has been improved upon from its first existence up to about 2012.

|  |  |  |  |
| --- | --- | --- | --- |
| **Port Knocking Project** | **Summary** | **Strength** | **Weakness** |
| Basic Port Knocking | First introduction of port knocking concept | Firewall rules used to open/close a port | Replay packet. Scanning. Out of order packet delivery. |
| Advanced Port Knocking Suite | DES used in packet sequence | First use of encryption with knocking packet | Could be slow due to encryption/decryption |
| Barricade | ICMP echo request as knocking packet | Simple to implement | Password in ICMP packet can be sniffed |
| Cryptknock | Encrypt knocking sequence in one packet | Difficult to replay the knocking packet by sniffing with TCP dump | Data read by Lipcpap where it configures under monitoring state |
| Doorman | Single UDP packet knocking | Uses MD5 hash | Rainbow table can crack MCD5 hash table |
| Knockd | Combine UDP & TCP packet knocking | Able to use UDP & TCP packets | Same implementation of basic port knocking |
| Sig2Knock | Randomize the port sequence | Overcomes port scanning | Difficult to implement |
| Pasmal | Encrypted ion packed for TCP and ICMP | Able to use ICMP & TCP packets | Encryption may slow down performance |
| Portkey | TCP ports from 1 - 65535 can be used | Many TCP ports can be used for knocking | Only supports IP table-based firewall |
| Cryptography of Knocking [3] | Port knocking with cryptography | More secure than basic port knocking | Only supports IP table-based firewall |
| Cerberus by Dana Epp | ICMP packet sent to knocking server | Applied special ICMP ping packet | Only supports IP table-based firewall |
| Port Knocking with Single Packet Authorization [3] | Single packet used as an authentication mechanism | Authentication packet encrypted & difficult to replay | Out of order packet delivery is not discussed |
| One Time Knocking Framework using SPA and IPSec [6] | Enhanced SPA by using IP Sec | Knocking password sent to smartphone by RNG server | IPSec with firewall rules is difficult. Complex system. |
| Network Security using Hybrid Port Knocking [7] | Combination of cryptography, steganography & mutual authentication | Difficult to replay | Overhead increased due to cryptography & steganography |
| Advanced PK Authentication Scheme with QRC using AES [8] | QRC spoofs IP address | Port scans difficult. Hard to replicate IP address. | The complexity of the protocol may slow performance. |

In most recent times there have been papers proposed to further improve the protocol.

Covert Communication Using Port Knocking [9] proposes a new covert channel for stealthy communication. The channel uses Least Significant Bit steganography and Tariq PK to hide data. Tariq PK was first proposed in Network Security using Hybrid Port Knocking [7] which used both steganography and encryption. Additionally, GnuPG encryption is applied before hiding the data in the packet to add another layer of protection. Using Peak Signal to Noise Ratio the communication efficiency has been tested and the channel can achieve 152 bps as a maximum transmission rate. Despite the increased overhead due to steganography and encryption the protocol defends against DOS attacks, spoofed packets and TCP replay attacks.

sKnock (scalableKnock) [4] is a one-way authentication protocol that requires the client to send a UDP authentication packet before opening a connection to the server behind a firewall. The packet contains the client’s certificate and the port number of the server it requires connection to. The privacy of the client is protected by asymmetrically encrypting packet contents with a freshly chosen ephemeral key derived from the server’s public key using ECDH encryption. To reduce the number of packets involved in the authentication to just one UDP packet, ECC public keys are used which are lengths of up to 256 bits. This results in a packet size of 800 bytes keeping within the network MTU size of 1500 bytes. RSA public keys couldn’t be used due to lengths of 2048 bits which would have exceeded the MTU size. This MTU size will have to be taken into consideration in the proposed protocol.

Another proposed PK protocol uses Network Time Protocol (NTP) synchronization [3]. A knock sequence is less venerable to replay and brute force attacks if its lifespan is shorter. Therefore, the client and server share the same time by both sending a synchronization request at knock daemon start-up. The protocol states that the NTP time() function only gives granularity of a second which might not be good enough when we take into account the how fast a network can transfer packets. Upon further research I have found that NTP can now achieve granularity of tens of a millisecond over public internet and becomes even more efficient in a LAN [10]. Clock skew can occur due to asymmetric routing and network congestion, so it is important to implement repeated NTP queries to synchronize the client and server every so often.

The simple port knocking method [5] proposes to remove the added complexity of integrating a firewall and instead leave open a random listening port where the PK sequence can be sent over an SSH connection allowing the client to connect straight to the server. The port number from SSH is predefined and the common port 22 will not be used. The proposed method has a very simple implementation while defending against replay attacks and port scanning.

Certificate-Based Port Knocking [11] allows both client and server to hold a digital certificate by some certificate authority. Using this type of asymmetric encryption, the client’s identity is authenticated even before the communication begins. This addresses the problem of the shared PK secret to be invalidated at the firewall in conventional PK methods.

Secure Port Knock-Tunnelling [12] protects against NAT and DOS attacks. DOS attacking is solved when a PK sequence is successfully sent the firewall opens one port for the client and triggers a VPN connection. However, in the sending of the knocking sequence stage UPD packets are sent. To avoid the problem of out of order delivery a UDP packet is only sent every 10 seconds. Therefore, the process of four knocks will take at least 40 seconds with the buffer flushing automatically if the packets do not arrive in time. Having to wait this amount of time for a client-server to occur makes the protocol ineffective. Incorporating a timeout on the client side I believe would be a better solution if it happened that packets became out of order or lost.

## 1.5 Proposed Port Knocking Protocol

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. Figure 2 shows the handshake required for a TCP connection.

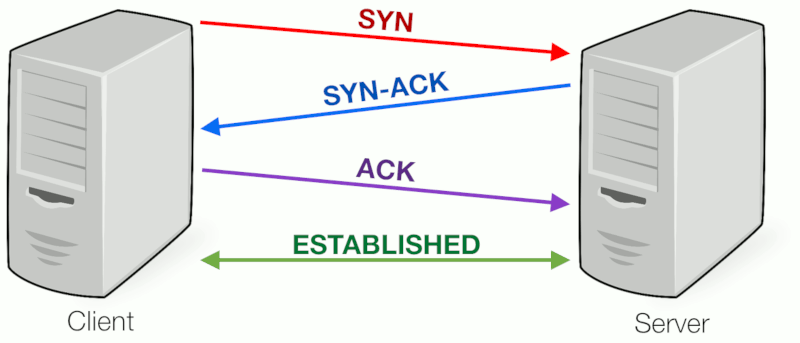


Figure - Client Server TCP Connection.

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 increases 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 protocol may exist. Sending UDP packets does not require this initial connection or acknowledgement packets reducing the time to make a PK connection. Additionally, 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.

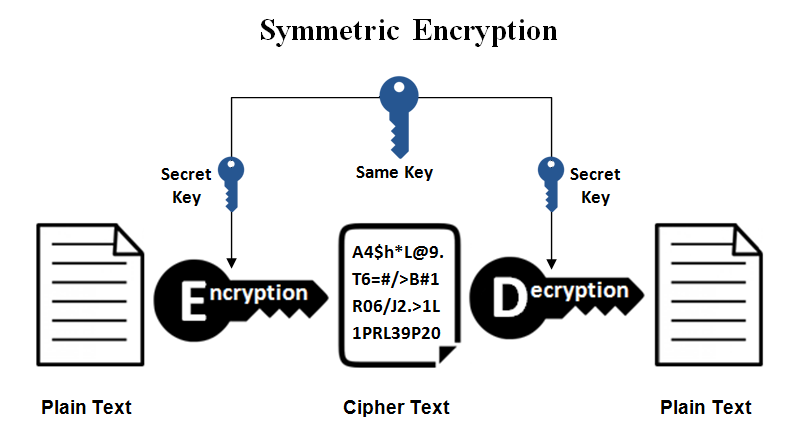
Because UDP packets tend to deliver out of order, this problem will be addressed by synchronising both the client and server to a timestamp protocol, NTP or similar time protocol, or giving each packet in the PK attempt a sequence number. One implementation of using PK with NTP timestamp only gave accuracy to the minute [3]. Such an accuracy would prove useless in distinguishing out of order packet delivery. To make such a protocol 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. Additionally, sending multiple packets reduces brute force attacks from undesired hackers. On a device we have 65536 (216) ports. If we have a three 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 the attacker will need to send 141 (655353/2) trillion packets!

This protocol will not use secure VPN tunnelling through IPsec protocol due to the huge overhead to implement such a structure. [6]

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 in Figure 4 - Symmetric Encryption. Figure 4 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 in Figure 3 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 in the below figures is the data, the PK sequence in our case and other data inserted into the packet, 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.



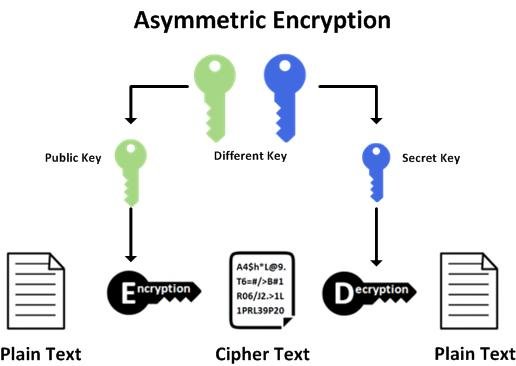


Figure - Asymmetric Encryption.

Figure - Symmetric Encryption.

PK protocols are dependent on the robustness of the PK daemon server-side [1]. 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. Perhaps more than one instance of a PK daemon be made to deal with traffic or could make the daemon scalable if there are increased packet numbers being sent to a server due to DOS attacks?

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 is 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 PK 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. Additionally, many opensource Java libraries exist for many the features needed for the protocol exist including packet capturing and iptables [13].

Most of the protocol will be coded on a Windows environment using Eclipse framework as it is easy to use. For a working demo the programme will be ran on a Linux environment, perhaps using a Raspberry Pi or spinning up a virtual machine, as iptables will be used as the firewall which is not available for Linux.

Ideally it would be preferred to build the application over localhost on Windows 10 environment. However, to integrate the iptables firewall a Linux environment is required. Secondly, incorporating a traffic sniffing tool over localhost is difficult as traffic needs to be redirected to the Wi-Fi router for the packets to be picked up. This can be done by incorporating a loopback via command prompt. Therefore, it seems more logical to leave this part for the time being and to keep testing on the Windows environment so traffic will be sent to a listening port on the server.

To capture network traffic at a specific port I have decided to use the open source Java library for capturing, crafting and sending packets, ‘Pcap4J’ [14]. The library and its dependencies need to be added as class paths. Alternatively, a build tool such as Maven, which is available in Eclipse, can also be used.

For unit testing purposes, JUnit will be used. JUnit5 allows for developer-side testing particularly for Java 8 [15].

## 1.6 Current 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 (iptables) in front of server on Linux environment (incomplete). [13]
* Incorporate network packet sniffing tool to detect port sequence on network. [14]
* To be continued…

# Chapter 2 - Technical Review & Previous Methodologies

# Chapter 3 – Implementation & Technical Issues

# Chapter 4 – Results, Evaluation & Final Deliverables

# Chapter 5 - Conclusion

# 1.7 References

|  |  |
| --- | --- |
| [1] | Wikipedia, “Port Knocking,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Port\_knocking. [Accessed 03 11 2019]. |
| [2] | R. . deGraaf, J. . Aycock and M. . Jacobson, “Improved port knocking with strong authentication,” , 2005. [Online]. Available: https://acsac.org/2005/papers/156.pdf. [Accessed 3 11 2019]. |
| [3] | T. Popeea, V. Olteanu, L. Gheorghe and R. Rughinis, “Extension of a port knocking client-server architecture with NTP synchronization,” in *10th Roedunet International Conference (RoEduNet)*, Iasi, Romania, 2011. |
| [4] | D. Sel, S. H. Totakura and G. Carle, “sKnock: Port-Knocking for Masses,” in *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, 2016. |
| [5] | F. H. M. Ali, R. . Yunos and M. A. M. Alias, “Simple port knocking method: Against TCP replay attack and port scanning,” , 2012. [Online]. Available: https://ieeexplore.ieee.org/document/6246118. [Accessed 3 11 2019]. |
| [6] | J.-H. . Liew, S. . Lee, I. . Ong, H.-J. . Lee and H. . Lim, “One-Time Knocking framework using SPA and IPsec,” , 2010. [Online]. Available: http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.ieee-000005529780. [Accessed 3 11 2019]. |
| [7] | H. Al-Bahadili and A. H. Hadi, “Network Security Using Hybrid Port Knocking,” in *2011 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC)*, Udaipur, India, 2011. |
| [8] | V. . Srivastava, A. K. Keshri, A. D. Roy, V. K. Chaurasiya and R. . Gupta, “Advanced port knocking authentication scheme with QRC using AES,” , 2011. [Online]. Available: https://ieeexplore.ieee.org/document/5958506. [Accessed 3 11 2019]. |
| [9] | K. Mariam, A. Hadi and A. Hudaib, “Covert Communication Using Port Knocking,” in *IEEE Software, Computer Science, Princess Sumaya University of Technology Amman*, Jordan, 2016. |
| [10] | Wikipedia, “Network Time Protocol,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Network\_Time\_Protocol. [Accessed 07 11 2019]. |
| [11] | B. Mahbooba and M. Schukat, “Digital certificate-based port knocking for connected embedded systems,” in *2017 28th Irish Signals and Systems Conference (ISSC)*, Killarney, Ireland, 2017. |
| [12] | P. . Mehran, E. A. Reza and B. . Laleh, “SPKT: Secure Port Knock-Tunneling, an enhanced port security authentication mechanism,” *Information Sciences,* vol. , no. , pp. 145-149, 2012. |
| [13] | “Java library for IPTables in LaunchPad,” 09 07 2011. [Online]. Available: https://launchpad.net/libiptables-java. [Accessed 04 11 2019]. |
| [14] | K. Yamada, “Pcap4J – A Java livrary for capturing, crafting, and sending packets,” [Online]. Available: https://www.pcap4j.org/. [Accessed 03 11 2019]. |
| [15] | The JUnit Team, “JUnit 5,” 2019. [Online]. Available: https://junit.org/junit5/. [Accessed 07 11 2019]. |

**Client side**

Developed using Java JDK 8.

Eclipse IDE.

Version control using git with two branches – master and development.

UDP packets used.

Timestamp and random port encoded in each packet.

128-bit AES symmetric key generated every time packet is sent.

2048-bit asymmetric RSA keypair (public and private key) generated once and used to encrypt the AES key. 2048-bit RSA is safe until 2030 - <https://en.wikipedia.org/wiki/Key_size>.

Encoding of message with AES key and the AES key was RSA public key encrypted and sent in the packet. Recommended because RSA decryption is too slow when message is large.

Perhaps use Elliptic-curve cryptography instead of RSA. Seems to be much faster - <https://www.linkedin.com/pulse/comparing-ecc-vs-rsa-ott-sarv/>.

Connection attempt after using ping and telnet. Also, possible to set up socket connection after knock sequence is sent.

Want to incorporate NTP time synchronization.

Testing using JUnit 4.

**Server side**

Ubuntu 16.04 run on Oracle Virtual Box.

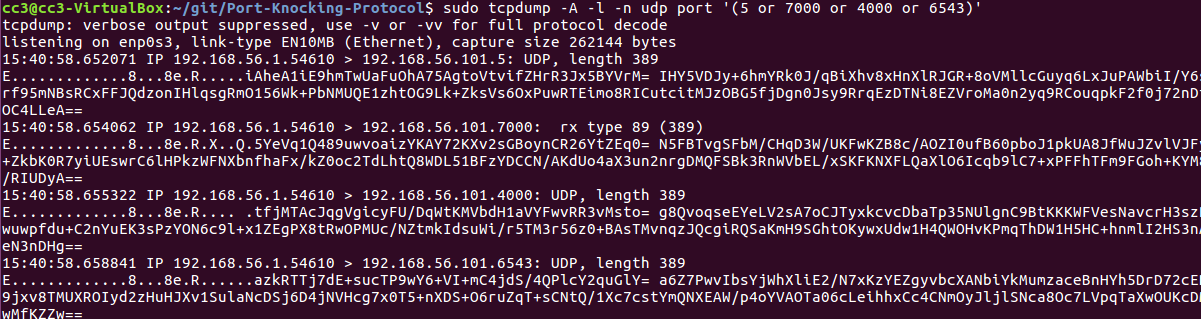
Developed using Java JDK 8.

Eclipse IDE.

Version control using git with two branches – master and development.

Tcpdump used for packet capture on specified interface.

Java parsing of ASCII tcpdump text to extract source, destination and data information.



<https://www.tcpdump.org/manpages/tcpdump.1.html>

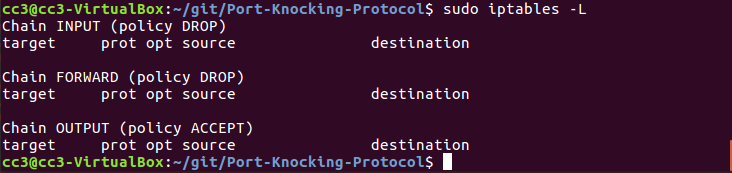
-A flag: Prig each packet (without link level header) in ASCII.

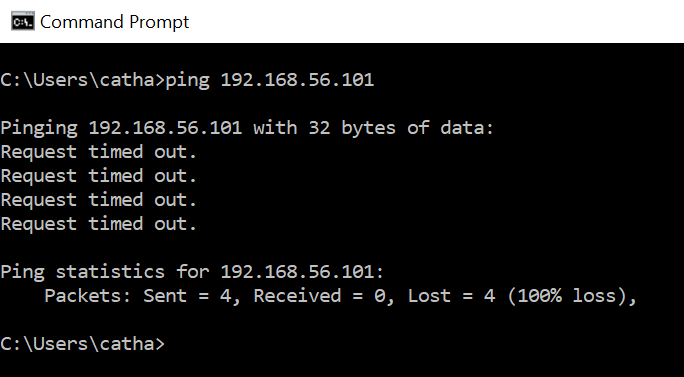
-l flag: flushes tcpdump messages as soon as they arrive. By default, tcpdump buffers messages and does not output messages until the buffer fills.

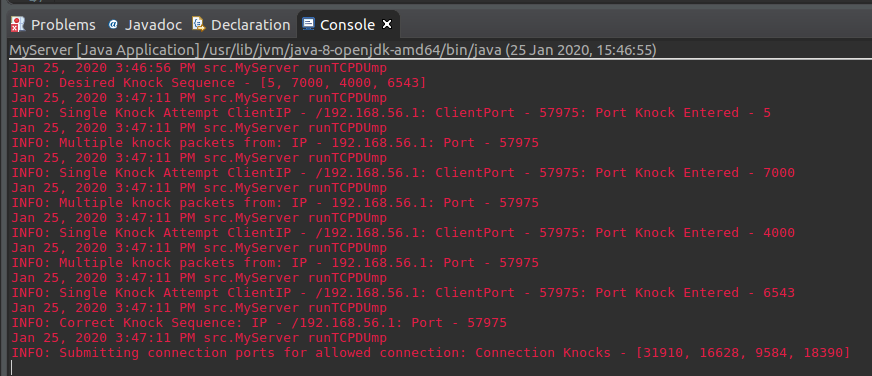
-n flag: don’t convert addresses to names.

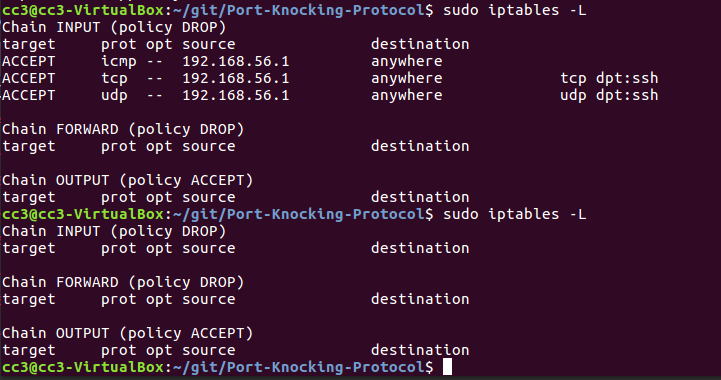
Iptables amended using random ports encoded in client UDP packets.

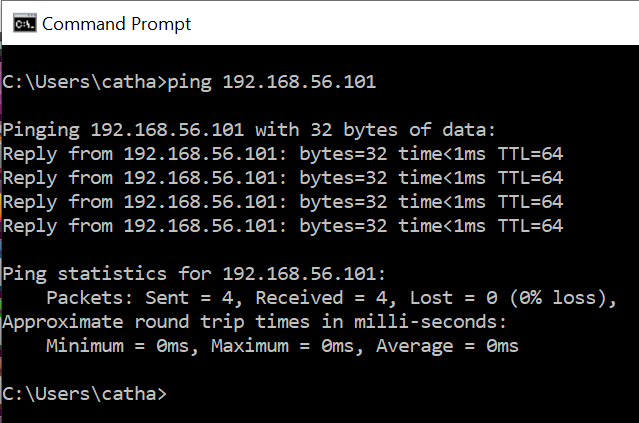
Iptables INPUT rule configured to DROP packets on server startup.











Timestamp and random port decoded in each packet.

Decoding of AES key using RSA private key and message decoded using AES key. Recommended because RSA decryption is too slow when message is large.

Sorting of UDP packets based on late packet arrival. Errors thrown if UDP packet time is in future or out of order.

Using HashMap to store key-value attributes. The key is an AttemptKnockingSequence object storing the client IP address and port the UDP packet is sent from. The value is an array list of SingleKnock objects which represent a single knock in the knock sequence from the client storing the port number of the knock, timestamp and the random connection knock.

Server contains a log file to record all messages, errors etc.

**Technical Difficulties**

Before any port knocking occurs between client and server 2 sources of information beet to be agreed on by bother parties -

* RSA keys to be shared to the correct parties. The server is the only party to hold the private key and is never shared. Public key can be distributed freely.
* Port knocking sequence agreed on.

Finding an appropriate packet capture tool. Looked into Pcap4j and jNetPcap but too many dependencies. Wanted maven projects set up, issues with dll files, installing jar files, setting build and class paths and the libraries haven’t been maintained in some time.

Issues running Linux commands in a Java programme especially when it came to iptables. BuildProcess and Runtime.getRuntime().exec() are two commands that allow you to run Linux commands in a Java program.

Server must be run as root user due to privileges associated with iptables and tcpdump.

Problem with Java library System.currentTimeMillis() as the program records the time of the operating system. Ubuntu clock time was not synced properly with real time and was throwing errors that packets had a time in the future.

Problems decrypting ASCII tcpdump text due to extra 28-character padding that was added onto the message.