RESOLVING THE AUTHENTICATION-CONNECTION ASSOCIATION PROBLEM IN PORT KNOCKING AND SINGLE PACKET AUTHORIZATION BASED DYNAMIC FIREWALLS

By

Muhammad Tariq

A thesis submitted to the faculty of Information Security Department, Military College of Signals, National University of Sciences and Technology, Pakistan in partial fulfillment of the requirements for the degree of MS in Information Security

September 2008

ABSTRACT

Port Knocking and Single Packet Authorization (SPA) based dynamic firewalls authenticate remote users at firewall-level for ensuring authorized access to potentially vulnerable network services. Despite being around for quite some time, both passive authorization techniques still suffer from a crucial problem. The authentication-connection association problem, which allows an attacker to connect to a protected server on behalf of a valid client, after the client has successfully authenticated with the firewall but before he establishes the TCP connection with the protected server. A novel design has been proposed in this work that resolves this problem by encoding nonces in suitable fields of selected packets in transit between client and server. The proposed design is incorporable into the existing architectures of both passive authorization techniques and keeps the previously made enhancements to these systems intact. The proposed design has been implemented in Java by modifying an existing open-source port knocking system, JPortKnock. The performance evaluation has been carried out on the basis of various parameters like processing overhead, robustness and stealthiness. To measure the processing overhead incurred by incorporating the proposed design into existing systems, the ability of processing different numbers of simultaneous authentication request packets of JPortKnock and the proposed system has been examined. Results have shown that the processing overhead, which is crucial for passive authorization systems, incurred by incorporating the proposed design into JPortKnock remains less than 1% which is marginal.

To my loving parents…

ACKNOWLEDGMENTS

All praise to Almighty Allah who has showered invaluable blessings on me throughout my life. He has blessed me with all the strength and spirit to complete this research work.

I am extremely grateful to my parents, my sister and brothers who have always shown complete trust in me and provided me the constant support and encouragement in all the hard times throughout my life. I would have to write theses for many lifetimes to match the amount of love and support that they offer me.

I wish to express my sincere gratitude to my supervisor AVM (R) Dr. M. Shamim Baig for his continuous support during the thesis. I had a lot of extensive discussions with him which always resulted in innovative ideas. His profound knowledge, stimulating suggestions, encouragement and critical, but valuable, remarks led me to do a good research. I am highly indebted to him for his patience in countless reviews and for his contribution of time and energy.

I gratefully acknowledge the help and guidance provided to me by my thesis committee members Lt. Col. Attiq Ahmad, Lt. Col. Mofassir-ul-Haque and Air. Cdre. Tahir Mahmood Khalid. I owe them sincere thanks for their personal supervision, advice and valuable guidance, without which completion of this research work would not have been possible.

I also owe very special thanks to Mr. Muhammad Tariq Saeed and Sqn. Ldr. Liaquat Ali Khan for giving me their valuable advices as well as practical help whenever I needed it.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table **Caption** **Page**

Chapter 1

INTRODUCTION

1.1 Introduction

Ever since the inception of Internet, more and more computers and networks are becoming a part of it every day. This increase in the networks being connected to the Internet is bringing with it the increase in the number of vulnerable services offered by these networks. Despite the advancements in the world of information security in past few years, we have not been able to significantly reduce the number of security compromise incidents reported every year, rather they've been on exponential rise [1].

Authentication is a process of allowing only the authorized user to access the services running over a server and restricting all the rest. Generally the authentication processes are integrated with the services allowing the services to decide themselves which user should be allowed to connect and which not. This approach has not been able to solve the problems completely in the past because when authentication systems are implemented as a part of services to be protected, it obliquely allows the attackers to interact directly with those services by either bypassing or compromising the authentication mechanisms of vulnerable services.

The idea of 'Port Knocking' [2] proposed to integrate the authentication process with firewalls. Later the idea of 'Single Packet Authorization' [3] came into the picture as a variant of port knocking. Port knocking and SPA are passive authorization techniques that are used to communicate authentication information across a pre-specified set of

closed ports. These techniques offer a blockade in front of an attacker before he could interact with the service to be protected.

In the past, main emphasis has remained on strengthening the authentication procedures of the port knocking. Though port knocking and SPA have matured with the passage of time, still there exist some problems in these systems that could not be fixed so far, one such problem is the lack of association between the authentication process and the follow-on TCP connection to be established [4]. This problem is actually a vulnerability in both passive authorization techniques that allows an attacker to stop the transmission from the client when he has successfully authenticated with the firewall and connect to the protected service on behalf of that client.

The authentication-connection association problem is the most crucial problem still existing in both of the passive authorization techniques but the literature so far surveyed reveals that not much attention has been paid to this problem in the past. It is unproblematic for an attacker in adequate position to exploit this vulnerability and the results of this exploitation are total bypassing of the authentication at firewall making the entire concept useless.

This thesis addresses the authentication-connection association problem and proposes a design to resolve the authentication-connection association problem in both port knocking and SPA. The design can be easily incorporated into existing systems.

1.2 Problem Statement

Despite being around for quite some time now, both of the passive authorization techniques, Port Knocking and SPA, still suffer from a major problem; the lack of

association between the authentication process and the follow-on TCP connection to be established. It is possible for an attacker to hijack a successful authentication by blocking further transmissions from a client and assuming its identity to a server after authentication with firewall has completed, but before a TCP connection has been opened. A solution that would keep the simplicity of these schemes unharmed and still provide effective resolution to this problem is required.

1.3 Overall Objective

The basic object of this dissertation is to address the authentication-connection association problem in the dynamic firewalls based on passive authorization techniques. This problem allows an attacker to bypass the firewall authentication mechanism by hijacking a successful authentication of a valid client and connect to the protected service on behalf of that client. The objective is to propose a design that would resolve this problem without compromising the strengths of existing architectures and that would be incorporable in the existing architectures of both port knocking and single packet authorization systems.

1.4 Thesis Organization

This dissertation is organized as given. Chapter 2 gives a detailed review of firewall technology and the notion of user authentication at firewalls. Chapter 3 presents an in depth review of the two passive authorization techniques used to build dynamically reconfigurable firewalls capable of performing user authentication at network-level. Chapter 4 focuses on the main research topic, the authentication-connection association problem, prior work done to resolve this problem and our proposed approach at an abstract level. Chapter 5 presents the specifications of our proposed design in detail with the perspective of incorporating our design in the existing architectures of passive

authorization based dynamic firewalls, the underlying architectural assumptions and implementation concerns. Chapter 6 presents the performance evaluation of our design on the basis of various parameters. Lastly, chapter 7 concludes the research and highlights the future work, which can be done to carry forward the efforts of enhancing firewalls.

1.5 Summary

This chapter provides a comprehensive introduction to the addressed research work. A brief introduction, problem statement and overall objective of the research work have been discussed. Towards the end of the chapter, overall thesis organization is provided to help the reader in a thorough manner.

Chapter 2

BACKGROUND

2.1 Introduction

The Internet was born in 1969 as ARPANET, which was a small community of trusted computers, but it kept on growing exponentially after that. November 2, 1988 was the day in the history of Internet that changed it forever. It was the day when 'The Morris Worm' surprised the researchers and made them realize that the Internet is no more a small community of trusted entities. Thirty-nine years after the inception of Internet, it has become an established fact that Internet is a hostile place. Despite the advancements in the world of information security over the time, the number of security compromise incidents reported every year has been on exponential rise [1]. On the other hand, the security professionals have also kept on reacting to the situations accordingly and the game is still on.

There are two main obstacles introduced by the security professionals in front of the attackers. First is the process of authentication that is usually built into the service that is to be protected. Second is the use of firewalls to discriminate between legitimate and illegitimate traffic. In Sections 2.2 and 2.3, we look at both of these security measures along with their weaknesses. Section 2.4 will explain how these two techniques are merged by the concept of passive authorization at firewalls to achieve greater benefits.

2.2 Authentication

Authentication is the process of determining whether an entity is, in fact, who or what it claims to be. In public and private computer networks, authentication is usually done

using usernames and passwords. Knowledge of the username and the associated passwords is assumed to guarantee the authenticity of an entity. Authentication is used to allow only the legitimate users to access the services running over a server and restricting all the rest.

Generally the authentication processes are integrated with the services to be protected. This allows the services to decide themselves which user should be allowed to connect and which not. Though this approach has proved to be effective, it has not been able to solve the problems completely in the past. Many commonly used network services are insecure by design having weak or no security. Also some network services are large complicated systems in which flaws keep on emerging periodically, even in the authentication systems of these services, allowing the attackers to gain unauthorized access to those services. A common approach used to limit access from unauthorized users to the protected services is to use firewalls.

2.3 Firewalls

A firewall [5-7] is a system that secures a network, protecting it from access by unauthorized users. A firewall makes it possible to filter incoming and outgoing traffic that flows through a network. It can use one or more sets of “rules” to inspect the network packets as they come in or go out of a network and either allows the traffic through or blocks it. The rules of a firewall can inspect one or more characteristics of the packets, including source and destination IP addresses, source and destination port numbers, underlying network protocol types (TCP, UDP, ICMP etc) and, to a very limited extent, the type of application layer protocols (SMTP, FTP, SSH, HTTP etc) being used. Stateful firewalls keep track of which connections are opened through the

firewall and will only allow traffic through which either matches an existing connection or opens a new one.

Firewalls can greatly enhance the security of a host or a network. They can be used to protect and insulate the applications, services and machines of the internal network from unwanted traffic coming in from the public Internet. Firewalls also limit or disable access from hosts of the internal network to services of the public Internet.

There are two basic ways to create firewall rule-sets: “inclusive” or “exclusive”. An exclusive firewall allows all traffic through except for the traffic matching the rule-set. On the other hand, an inclusive firewall is configured in all-drop mode and it only allows traffic matching the rules through and blocks everything else. Inclusive firewalls are generally safer than exclusive firewalls because they significantly reduce the risk of allowing unwanted traffic to pass through the firewall but they are not as easy to build and deploy as they may seem to be.

As stated earlier, most modern firewalls are designed to filter packets or validate protocol semantics, and they are very good at doing this. However, behind the packets and protocol messages visible to firewalls are users and programs; firewalls have little knowledge of these and consequently they aren’t very good at filtering based on the users and programs responsible for network traffic. This can be viewed as two separate problems

Firewalls can limit which network services can be reached from outside. However, it may also be necessary to limit which remote users can connect to those services. A basic

assumption, made by many modern firewalls, is that trusted users only connect from small sets of trusted hosts with specific network addresses; they implement user filtering by blocking incoming packets with source addresses not in these sets. Unfortunately, the source addresses on incoming packets give very little information about the user who sent them; malicious users can spoof trusted hosts, and trusted users can connect from un-trusted machines. Since many trusted hosts may have dynamic IP addresses assigned using DHCP, opening a firewall to one trusted host may virtually be opening it to thousands of IP addresses, making it easier for an attacker to find an address to spoof or a machine with a trusted address to hijack. Adjusting the set of trusted IP addresses in a firewall requires manual reconfiguration by an administrator.

Although users can be accurately linked to IP addresses within a local network, it is difficult to limit the services with which those users are allowed to communicate. Firewalls generally attempt to filter outbound traffic by restricting the ports to which users can connect: for example, disallowing outbound connections to anything except TCP ports 80 (HTTP), 443 (HTTPS), and 20 and 21 (FTP). Unfortunately, this isn't particularly effective: non-standard services may be running on these allowed ports. Whereas application-layer firewalls can easily filter traffic that doesn't match the expected protocol for a port, it is much more difficult to detect disallowed applications that tunnel traffic through standard protocols on standard ports. For instance, tunneling various protocols through port 80, normally used for unencrypted WWW traffic, has become quite common, and encryption renders most application layer filters useless. Also, standard protocols can run on standard ports and still be used for unauthorized purposes. Restricting network access to only authorized local users and programs has the potential to alleviate these problems, but information about the users and applications

that generated or will receive network traffic is usually only available at the source or destination hosts themselves, and isn't necessarily reliable.

2.4 User Authentication at Firewalls

As explained in previous sections, the authentication procedures integrated with the services to be protected do not provide sufficient security and are susceptible to being compromised in many ways. Also, Limiting sources of network connections using firewalls is not a perfect solution as firewalls perform packet filtration on the basis of source addresses of incoming packets that are likely to be spoofed easily by the attackers. Moreover, not all authorized users have predictable IP addresses and limiting access based on IP addresses would not be feasible in such a case. No defensive measure is perfect. Security software can be disabled or bypassed by exploiting software or configuration vulnerabilities. For this reason, security depends on the principle of defense in depth; the principle that security comes in layers, where the defeat of one layer doesn't leave everything vulnerable and the attackers must bypass multiple layers to reach anything important.

What is needed is a mechanism to integrate user authentication process with firewalls. Such a mechanism can be constructed with the addition of an intelligent module with firewall that would validate the authenticity of remote users and dynamically modify firewall rule-sets according to per-user access policies. This dynamic firewall [8] would serve as an extra layer of security in front of attackers that they must penetrate in order to connect to the protected service. Such a mechanism would make the protected service invisible to port scanning/probing, would provide good enough security to those commonly used vulnerable network services having weak or no security and would even

provide sufficient resistance against the attackers knowing zero-day vulnerabilities in the protected service. Also, such a mechanism would be required to provide strong user authentication at firewalls, should be interoperable with existing network protocols and devices, should be simple enough so it might not itself become vulnerable to being compromised and should put low demands on the network and processor resources.

Port knocking and Single Packet Authorization are two techniques used to construct such a dynamically reconfigurable inclusive rule-set based firewall, able to provide strong user-level authentication. The detailed working of these two schemes is presented in the Chapter 3.

2.5 Summary

User authentication and firewalls are the two main obstacles used by the security professionals to present blockade in front of malicious attackers. This chapter described the concepts of authentication and firewalls in their strengths and limitations. Conventionally, user authentication is integrated with the services to be protected. This chapter explained that this practice does not provide sufficient security and is susceptible to being compromised in many ways. This chapter built the foundation for the concept of integrating user authentication process with firewalls, which is the basic idea of passive authorization techniques of port knocking and single packet authorization systems. Chapter 3 describes these passive authorization techniques in detail.

Chapter 3

PASSIVE AUTHORIZATION TECHNIQUES

3.1 Introduction

Passive authorization schemes are becoming increasingly important for securing networked services. Among a long list of benefits that passive authorization provides, the most important is its ability to limit the scope of zero-day vulnerabilities. This chapter describes the concepts of passive authorization techniques of Port Knocking and Single Packet Authorization in weaknesses and strengths of these systems.

3.2 Port Knocking

The term ‘port knocking’ was introduced by Martin Krzywinski [9,10] but this concept was introduced by Barham *et al.* [11]. Port knocking [12] is a network-based client-server communication mechanism that is used for transmitting the authentication information across closed firewall ports to remotely modify access permissions of the firewall. The standard port knocking mechanism is based on generating a set of connection attempts on a set of predefined closed ports on the firewall. These connection attempts are analyzed by the port knocking server, and once the correct sequence is identified, the firewall rule-set is dynamically modified to add a new rule that allows the client to connect to a target port. The crude idea was vulnerable to many problems [4,13,14] but with the passage of time the idea of port knocking kept on maturing by the addition challenge-response in authentication phase [4, 15], mingling with passive OS fingerprinting [16], cryptographic [17], and steganographic [6] enhancements to the concept.

Let us understand the basic working of port knocking in detail. At the server side, the firewall is set in ALL-DROP stance. This means that the firewall would drop all the packets that arrive on its interfaces and no response would be sent out, not even the ICMP_PORT_UNREACHABLE message that is sent when a packet is normally rejected/dropped by a firewall. The significance of this is to make to make it impossible for an attacker to determine whether a machine exists at that address or not, making it difficult for him to notice the presence of a listening host.

The next step is to find a way to connect to the protected server through the firewall. A port knocking daemon sits on the firewall and watches packets as they are dropped, waiting for a sequence of packets arriving at a predetermined set of ports in correct order. The client who wishes to connect to the server sends a series of TCP SYN or UDP packets to the predetermined firewall ports in order, for example: ports 222, 111, 333, 777. The firewall will receive these packets and drop them silently; however, the port knocking daemon will see these incoming packets and recognize the valid ‘knock’ on the ports. On recognizing a valid knock, the daemon would modify firewall rules to allow access to the client who knocked, based on his IP address as shown in Figure 3.1.

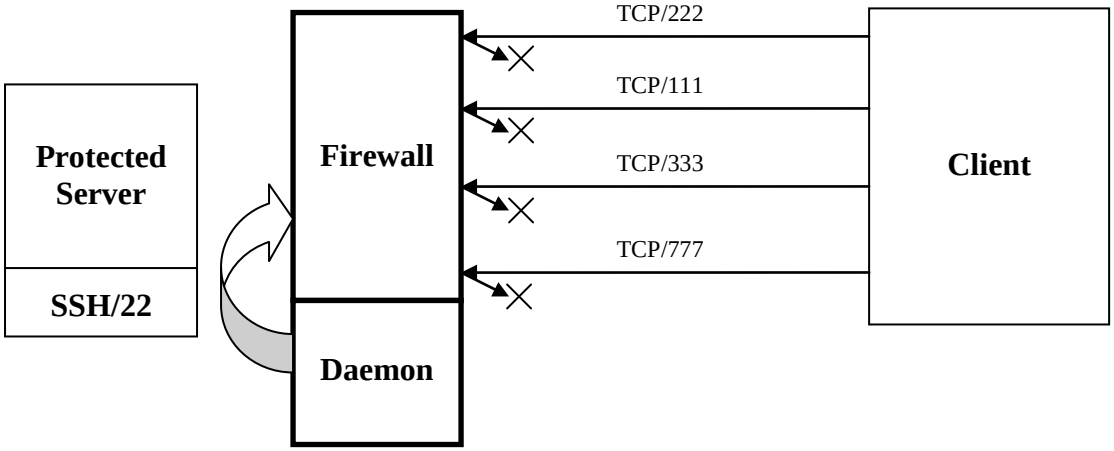

Figure 3. 1: A typical example of port knocking

The client would than be able to connect the protected service, e.g. SSH, and establish a normal TCP connection as shown in Figure 3.2. Another knock sequence, e.g. on ports 200, 190, 180, 170, can be used to close the opened port at the firewall. Most good implementations, however, simply close the port automatically after a given amount of time has elapsed and the associated TCP connection has been terminated by either communicating entity.

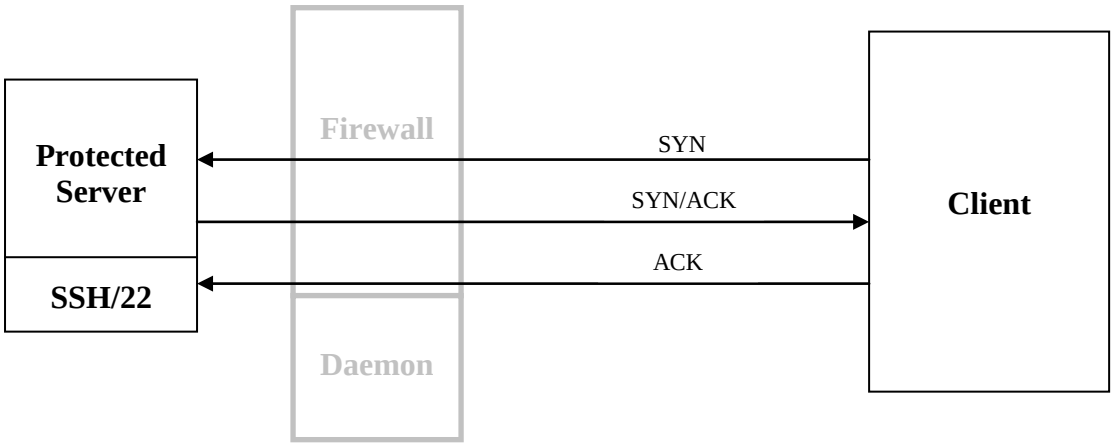

Figure 3. 2: An established TCP Connection after a valid knock sequence

As the server's firewall is set to drop all packets, so a scanning or probing [23] attacker will have no clue whether or not the server exists, let alone what services it is running. Also, by watching the incoming traffic for a predetermined knock, which essentially acts as a secret key known only to the trusted users to use the protected server, the port knocking daemon is able to authenticate the user at the other end before allowing them to connect to any potentially vulnerable services. More than thirty implementations [30] of port knocking are available at [10].

3.2.1 Strengths of Port Knocking

Port knocking provides a lot of benefits over conventional methods used for remote network authentication.

3.2.1.1 Strong Authentication

One of the key features of port knocking is it provides a stealthy method of authentication and information transfer to a networked machine that has no open ports. It is not possible to determine successfully whether the machine is listening for knock sequences by using port probes. For an external attacker having no idea about the port knock sequence, even the simplest of sequences would require a massive brute force effort in order to be discovered. A three-knock simple TCP sequence (e.g. port 1111, 2222, 3333) would require an attacker without prior knowledge of the sequence to test every combination of three ports in the range 1-65535, and then to scan each port in between to see if anything had opened. That equates to approximately 65535^4 packets in order to obtain and detect a single successful opening. This is made even more impractical when knock attempt-limiting is used to stop brute force attacks, longer and more complex sequences are used and cryptographic hashes are used as part of the knock.

3.2.1.2 Stealth

Because information is flowing in the form of connection attempts rather than in typical packet data payload, without knowing that this system is in place it would be unlikely that the use of this authentication method would be detected by monitoring traffic. To minimize the risk of a functional sequence being constructed by the intercepting party, the information content containing the remote IP of the sequence can be encrypted.

3.2.1.3 Default-Drop Stance

When a port knock is successfully used to open a port, the firewall rules are generally only opened to the IP that supplied the correct knock. This is similar to only allowing a certain IP white-list to access a service but is also more dynamic. An authorized user

situated anywhere in the world would be able to open the port he is interested in to only the IP that he is using without needing help from the server administrator. He would also be able to "close" the port once he had finished, or the system could be set up to use a timeout mechanism, to ensure that once he changes IP's, only the IP's necessary are left able to contact the server. Even if somebody did manage to guess, steal or sniff the port knock and successfully use it to gain access to a port, the usual port security mechanisms are still in place, along with whatever service authentication was running on the opened ports.

3.2.1.4 Simplicity

The software required, either at the server or client end, is minimal and can in fact be implemented as simply as a shell script for the server or a Windows batch file and a standard Windows command line utility for the client. Overhead in terms of traffic, CPU and memory consumption is at an absolute minimum. Port knock daemons also tend to be so simple that any sort of vulnerability is obvious and the code is very easily auditable.

3.2.1.5 Customizability

The system is completely customizable and not limited to opening specific ports or, indeed, opening ports at all. Usually a knock sequence description is tied with an action, such as running a shell script, so when a specific sequence is detected by the port knock daemon, the relevant shell script is run. This could add firewall rules to open ports or do anything else that was possible in a shell script. Many port knocks can be used on a single machine to perform many different actions, such as opening or closing different ports.

3.2.1.6 Protection against Zero-day Vulnerabilities

Due to the fact that the ports appear closed at all times until a user knowing the correct knock uses it, port knocking can help cut down not only on brute force password attacks and their associated log spam but also protocol vulnerability exploits. If an exploit was discovered that could compromise SSH daemons in their default configuration, having a port knock on that SSH port could mean that the SSH daemon may not be compromised in the time before it was updated. Only authorized users would have the knock and therefore only authorized users would be able to contact the SSH server in any way. Thus, random attempts on SSH servers by worms and viruses trying to exploit the vulnerability would not reach the vulnerable SSH server at all, giving the administrator a chance to update or patch the software. Although not a complete protection, port knocking would certainly be another level of defense against random attacks and, properly implemented, could even stop determined, targeted attacks.

3.2.2 Problems with Port Knocking

During the survey of existing port knocking systems, quite a few problems have been identified in the existing systems [4,13,19].

3.2.2.1 The authentication-connection association problem

The lack of association between authentication process and follow-on TCP connection to be established is the most crucial problem in port knocking systems. It is possible for an attacker to hijack a successful authentication by blocking further transmissions from a client and assuming its identity to a server after authentication has completed, but before a connection has been opened. This problem is explained in detail in chapter 5.

3.2.2.2 Out-of-order packet delivery

Proper decoding of port-knock sequences by most port knocking servers is dependent on the order of arrival of the knock packets. During busy network conditions on Internet backbone routers, the probability of out-of-order packet delivery becomes fairly high. All the systems that have been surveyed, only deGraaph [4] and Barham [11] made serious attempts to deal with this problem.

3.2.2.3 Network Address Translators (NATs)

The problem of working across a NAT is serious problem in port knocking systems. If traffic from a client passes through a NAT [24] to a server, and the client's (private) IP address is encoded in the authentication token, then the authentication exchange, if successful, will result in the correct port being opened to the incorrect client address. If the public IP address is encoded in the token, then the port will be opened to all hosts sharing the same public address. If the client's address is not encoded in the token at all, then the public address from the packet headers would likely be used, leading to the same problem.

3.2.2.4 Susceptibility to Denial-of-service (DOS) attacks

There are several denial-of-service attacks possible against port knocking servers. An attacker could prevent a client from authenticating by sending packets with the client’s source address to random ports on the server while the client is trying to authenticate; if any of these packets went to ports being monitored by the port knocking server, then the client’s sequence would be corrupted and authentication would fail. Also, an attacker could affect a resource-consumption attack against a known port knocking server by sending packets with random forged source addresses to random ports. A port scan from a single source at a sufficiently high rate may also be sufficient to overload a server’s

processing resources, particularly if the server uses a computationally-intensive cryptographic protocol.

3.2.2.5 Replay Attacks

Port knocking systems that use plain-text authentication are vulnerable to replay attacks as captured tokens can be replayed by an attacker, in MITM position, to connect on behalf of a valid client. Port knocking systems using either cryptographic or one-time tokens could provide sufficiently strong authentication and are immune to this problem.

3.3 Single Packet Authorization

Single Packet Authorization (SPA) [20] is a network-based client-server communication mechanism that is used for transmitting the authentication information across closed firewall ports to remotely modify access permissions of the firewall, like port knocking. But unlike port knocking, instead of encoding authentication information in a series of port numbers, it encodes it in the payload of a single UDP datagram. This datagram is analyzed by the daemon running at firewall, and once the correct credentials are verified, the firewall rule-set is dynamically modified to add a new rule that allows the client to connect to a target port, as shown in Figures 3.3 and 3.4.

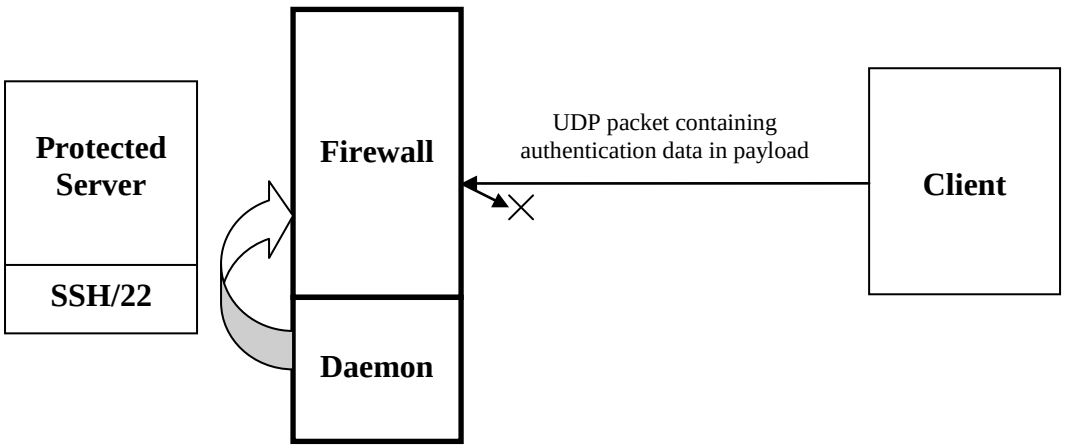

Figure 3. 3: A typical single packet authorization example

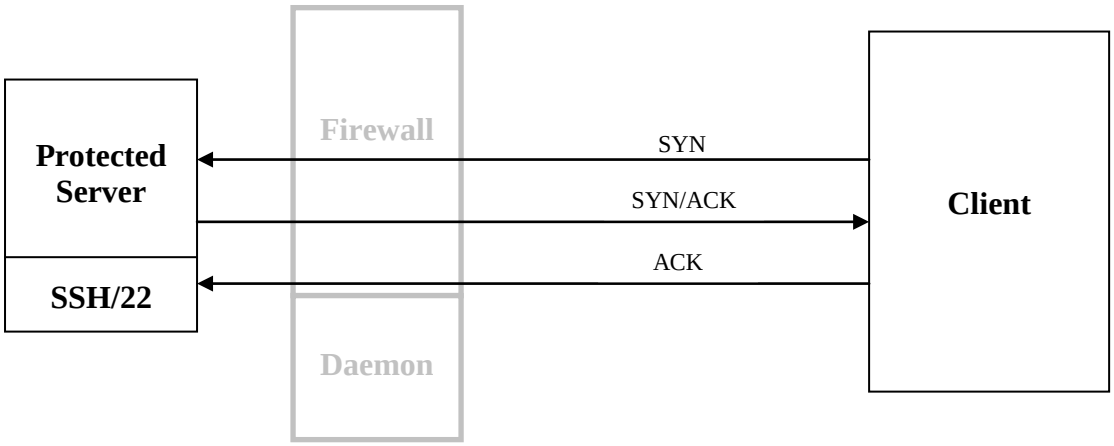

Figure 3. 4: An established TCP connection after successful SPA authentication

SPA retains most of the benefits of port knocking and fixes many of its limitations. The concept of authentication using a single packet was introduced by Barham but the term 'single packet authorization' (SPA) was introduced by MadHat [11,21]. The first publicly available SPA implementation was released in May 2005 as a piece of software called fwknop. Fwknop was originally created in 2004 as the first port knocking implementation to combine passive OS fingerprinting and port knocking, but later it changed to an implementation of SPA [16,22].

Single Packet Authorization mandates a similar architecture to port knocking. Both have client and server (daemon) components, the daemon maintains control of a default-drop packet filter and monitors packets passively. However, Single Packet Authorization moves the data transmission to where it belongs - in the application layer. This implies that instead of being able to send only a few bytes of data per packet, as in the case of port knocking, SPA is able to send up to the minimum MTU worth of data (1,500 bytes on Ethernet networks) between the client and the server in each packet. This far outstrips

the data transmission rate possible with port knocking, and having easy access to this amount of packet data opens up a huge range of possibilities.

3.3.1 Strengths of Single Packet Authorization

As stated earlier as well, Single Packet Authorization retains the benefits of port knocking. In addition SPA provides a lot of advantages over port knocking.

3.3.1.1 Typical Network Service

An SPA server can be written as a normal network service on an open port. Since UDP services are not required to respond to messages that they receive, and the protocol does not automatically generate any response, a non-responding UDP service on an open port on a system that silently drops unexpected packets is indistinguishable from a closed port to a port scan. An SPA server can therefore be written as a normal network service, without needing to resort to packet sniffers or any platform-specific mechanisms.

3.3.1.2 Optimal Resource Allocation

NATs and stateful firewalls between SPA clients and servers will only have to allocate resources for at most one logical connection, rather than one for each knock as required with port knocking.

3.3.1.3 Larger Authentication Data Transfer

Single Packet Authorization messages can communicate much more information because the packet payload is used rather than just the packet headers to transmit data. Port knocking schemes typically encode information as sequences of connections to ports, and the sequence itself represents the information being transmitted. Due to the fact that the port fields of the TCP and UDP headers are only 16 bits wide, each individual packet of a port knock sequence can only communicate two bytes of information. It should be

noted that other packet header fields could also be used within a port knocking scheme in order to increase the amount of data that can be transmitted, but any conceivable variation of this will not be able to communicate nearly as much data as any method that makes use of the payload portion of IP packets.

3.3.1.4 Faster Authentication

Single Packet Authorization is much faster. Port knocking schemes must artificially introduce time delays between successive knock packets because there is no notion of a connection with reliable in-order delivery. Packets that arrive out of sequence would break any shared or encrypted port knock sequence agreement between the client and server. By contrast, SPA only requires a single packet to communicate all desired access or command information.

3.3.1.5 Replay Attack Prevention

Replay attacks are easily be thwarted by including random data within SPA messages and then tracking MD5 sums on the server side. This is much cleaner than methods employed by port knocking schemes that involve relatively complicated state sharing mechanisms that require time synchronization or successive iterations of a hashing function.

3.3.1.6 Resilience against Spoofing

An attacker cannot easily break the SPA scheme be simply connecting to ports that may be used to communicate authorization messages. In port knocking schemes, if an attacker simply spoofs duplicate packets from the same IP that initiates a legitimate knock sequence to the same target IP, then the sequence will effectively be broken by the attacker.

3.3.1.7 Smaller Network Footprint

SPA has a much smaller network footprint in terms of alarms that may be generated by an intrusion detection system that monitors the authorization messages. By contrast, a port knocking sequence could easily be detected and interpreted as a port scan by any IDS that have its port scan detection thresholds set low enough.

3.3.1.8 Freedom to Use Any IP Protocol

SPA can utilize any IP protocol; even those without any concept of a "port". The FWKNOP SPA implementation includes support for sending authorization messages over ICMP.

3.3.2 Problems with Single Packet Authorization

Single packet authorization resolves most of the problems existing in port knocking systems. The most crucial problem that remains unresolved in SPA as well is the authentication-connection association problem [19]. Literature survey has shown that no attempt has been made to resolve this problem in SPA. Chapter 5 discusses this problem in detail.

Attempts have been made in SPA to partially resolve the problem of working across NATs and some work has also been found trying to develop immunity in SPA against the denial-of-service attacks. These two problems still need some serious attention to be paid, though.

SPA is inherently immune to the problem of out-of-order delivery of packets from which port knocking suffers because SPA uses only a single packet for transmitting the

authentication data. Different measures have been presented by different implementations of SPA to resolve the replay-attack problem. SPA is also immune to the problem of using shared global secrets among more than one client.

3.4 Summary

This chapter has presented the concepts of both passive authorization techniques of Port Knocking and Single Packet Authorization in detail and weaknesses and strengths of these systems. This chapter has also established the fact that the most crucial problem still persisting in both of these techniques is the authentication-connection association problem. What adds to the severity of this problem is the ease with which it can be exploited, and a successful exploitation of this problem leads to complete bypassing of authentication at firewall. Chapter 4 presents the technical details of this problem.

Chapter 4

THE AUTHENTICATION-CONNECTION ASSOCIATION PROBLEM

4.1 Introduction

Passive authorization techniques of port knocking and single packet authorization offer a very lightweight extra layer of security that is capable of providing dynamic firewall reconfiguration and ensuring strong user authentication at firewalls. This layer of security also makes port scanning/probing impossible for the attackers. It also serves as a strong blockade in front of those attackers possessing the scripts for exploiting the vulnerabilities of the protected service, and even against zero-day exploits. But despite of being such powerful mechanisms, both port knocking and single packet authorization suffer from a crucial vulnerability; the authentication-connection association problem. This chapter describes the authentication-connection association problem and the prior work that has been done in this regard to resolve this problem.

4.2 Problem Description

Both port knocking and single packet authorization schemes do not logically associate their authentication exchanges with the connections that are subsequently opened. This disjoint between the authentication phase and the subsequent TCP connection establishment phase is shown in figure 4.1 and 4.2. This disjoint can be exploited by the attackers to gain access to the protected server on behalf of a valid client, by hijacking the opened port on the firewall as shown in figure 4.3. What adds to the severity of the issue is the fact that this vulnerability is very easy to exploit for an attacker in adequate position and no attention has been paid in the past to fix this problem.

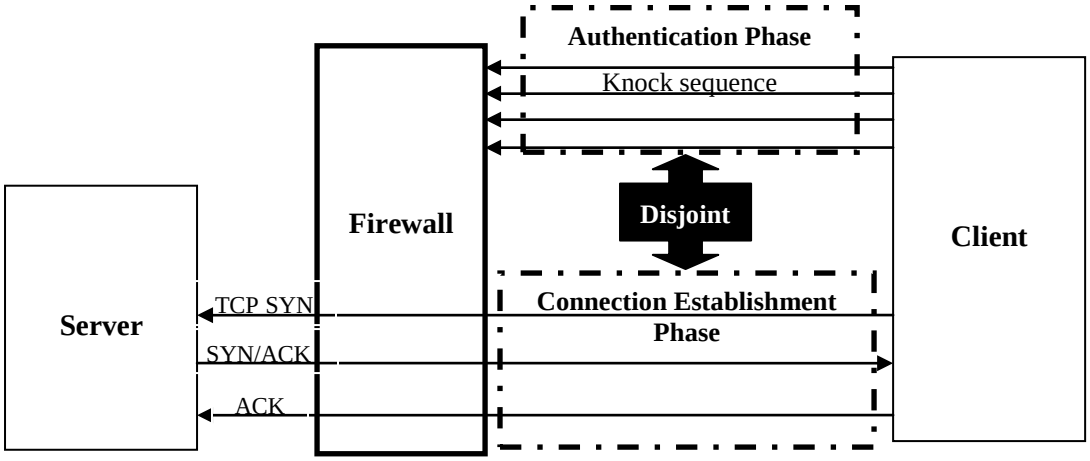

Figure 4. 1: The disjoint among authentication-connection phases in port knocking

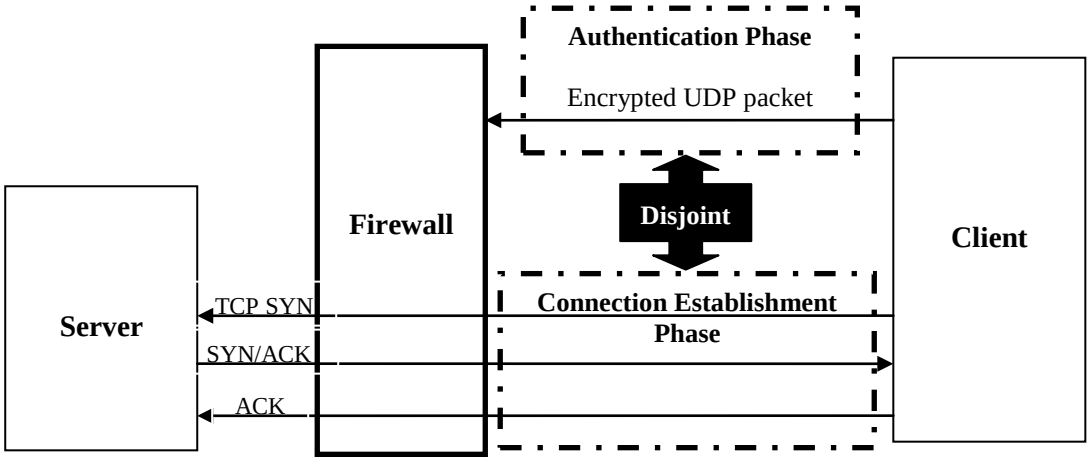

Figure 4. 2: The disjoint among authentication-connection phases in SPA

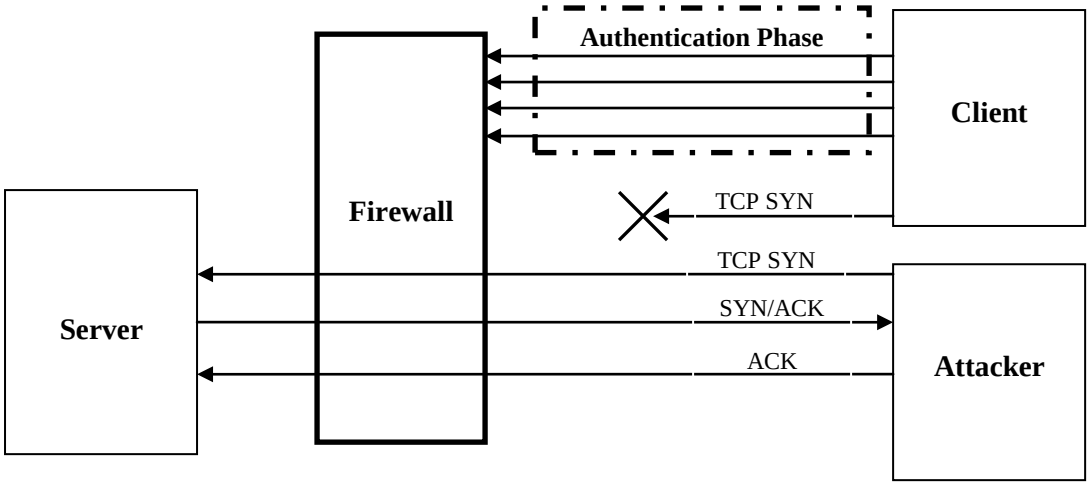

Figure 4. 3: Exploitation in port knocking

An attacker who is able to observe, at least, the connections being established between a client and the server and the typical information associated with these connections, can trivially exploit this vulnerability of port knocking and SPA. The typical information associated with a connection is the source and destination IP addresses of the entities in that connection, source and destination port numbers, the type of underlying network protocol in use (TCP, UDP, ICMP, etc) and optionally the underlying application layer protocol in use like HTTP, SMTP, SSH, FTP etc. An attacker in such a position, after waiting for the authentication process between the client and firewall (daemon) to complete, can take the client down either by launching a DOS attack or by any other means and impersonate the legitimate client to connect to the protected server on behalf of that client, as shown in figure 4.4.

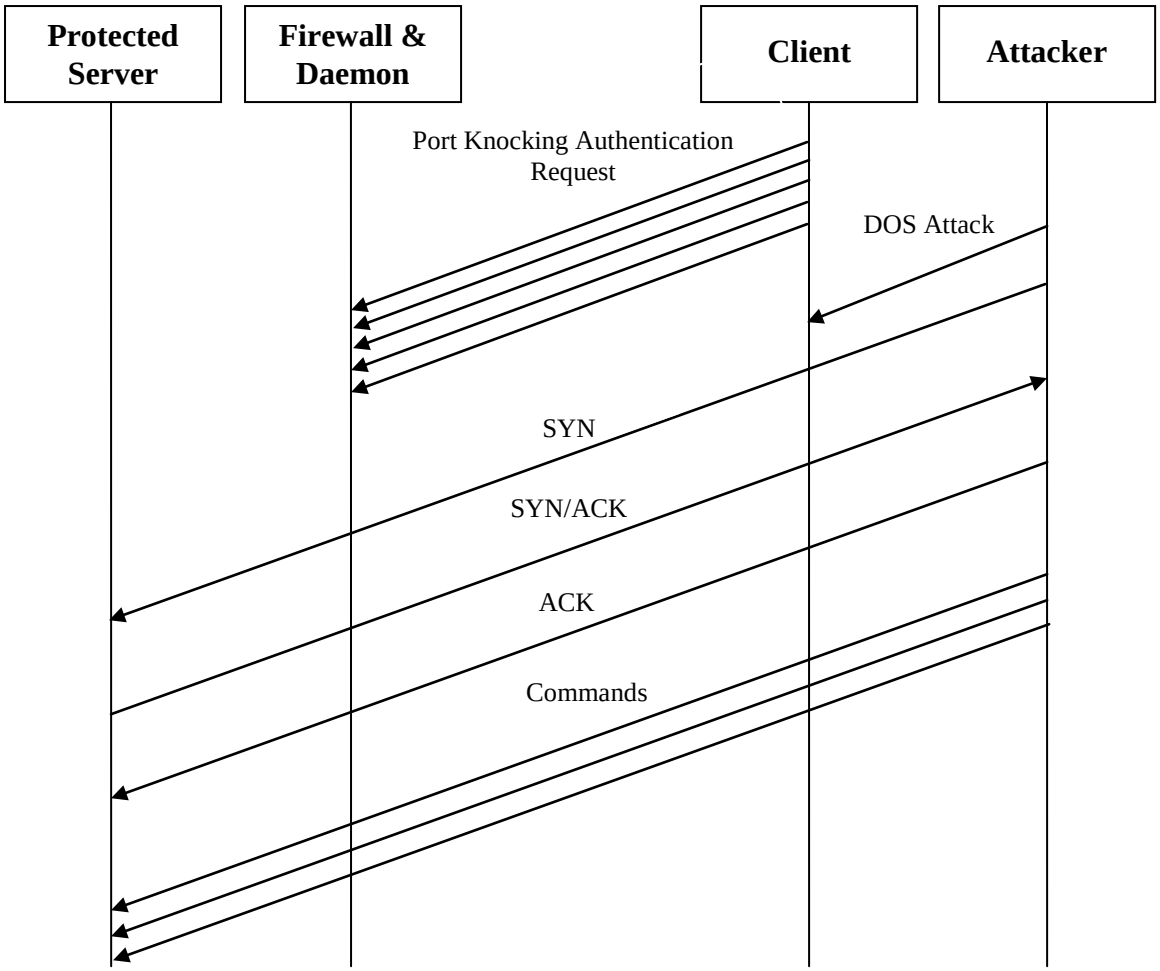

Figure 4. 4: A typical order of events for exploitation in port knocking

All that becomes easier for the attacker if he is in the private network of the client behind a NAT, sharing a single public IP address with that client and all the other systems within that network. In such a case, after waiting for the authentication process to complete, the attacker just needs to send the TCP connection establishment request to the server before the client, the attacker doesn't even need to take the client down in such scenario and he can connect to the server at that the behalf of that client.

Before proceeding, it is important to differentiate between the authentication-connection association problem and session-hijacking [26,27]. Former is the method of hijacking the subsequent TCP connection to be established impersonating as a valid client, after the

authentication with the firewall has completed but before the clients sends the request for TCP connection establishment. Whereas the latter is a method of attack where an attacker takes control over an established TCP session to execute his commands on the server on behalf of the authenticated client in the session. The scope of this thesis is limited to the resolution of authentication-connection association problem only under the specific set of assumptions in which attacker can exploit this problem. Session hijacking is a different problem and is out of scope of this dissertation.

4.3 Prior Work

Not much work is available in the literature to resolve the authentication-connection association problem. This problem is also referred to as ‘race attacks’ in the literature. Except Tan and Cappella’s system [11], all other solutions proposed to resolve this problem are in the form of suggestions to be implemented as future extensions for the work of various authors. Most of the previously proposed solutions either compromise the basic objectives of passive authorization systems i.e. simplicity and lightweight, or they are unfeasible to be implemented in real time systems. Tan and Cappella proposed a partial solution to the authentication-connection association problem using a 5-step procedure. In their system, the server informs the client in an encrypted UDP packet about the random port at which the service would be available for that client and then expects the connection attempt on that random port. Their technique does not prevent the attackers from locating the random port by scanning or discovering it by blocking the client after successful authentication and sending the TCP connection request on every port. Their technique is also susceptible to the loss of stealth, which is a basic necessity of passive authorization techniques, because the server generates a response to send the port number to the client and that may allow attackers to notice the existence of the server.

DeGraaf [1] proposed three design considerations for resolving this problem as future extensions to his work. First is to introduce a secondary wrapper server within the architecture which, on successful authentication of client, would tunnel the post authentication-connection to the actual server. This idea adds too much complexity at the server end compromising the basic requirement of simplicity and lightweight. The second idea is the same as that of Tan's system [11] where the server sends the random port number to the client and expects the connection attempt for the service on that random port. His third idea is to negotiate the ISN number to be used in the subsequent connection during the authentication phase but this idea is unfeasible to be implemented in real time systems as it would require that client and server are implemented in operating system's kernel space.

Jeanquier [2] proposed to wrap the post-authentication connections within encrypted sessions but the added cryptographic operations required for doing that will overload the server, compromise the basic challenges of passive authorization schemes. Jeanquier also proposed an idea similar to DeGraaf's third idea with similar disadvantages.

All of the ideas discussed above are either computationally intensive, require the clients to have unrealistic or unfeasible privileges from operating system kernels or they add excess of complexity on top of the initially simple concepts of port knocking and SPA.

4.4 Summary

This chapter presented the technical details of the authentication-connection association problem. It acquainted the reader with the techniques an attacker can use to trivially exploit the authentication-connection association problem and completely bypass this

strong security mechanism. Towards the end of this chapter, prior work that has been done to resolve this problem was presented, clearly establishing the fact that not much work has been done to resolve this most crucial vulnerability of passive authorization systems.

Chapter 5

PROPOSED DESIGN

5.1 Introduction

A major challenge in proposing a solution to the authentication-connection association problem is to devise such a design that would meet three crucial goals; the proposed design should not compromise the simplicity of passive authorization schemes and keep these architectures lightweight; the proposed design should be feasible with implementation perspective; and the design should be easily incorporable into the existing implementations of port knocking and SPA.

This chapter proposes such a solution by sharing something as simple as a nonce between client and server. The basic idea is to exchange an encrypted nonce between the client and the firewall during the authentication phase and then encode that nonce in a suitable [28] IP or TCP option field in the SYN packet of the subsequent TCP connection establishment phase, to associate the two phases. After a successful authentication, the firewall will only accept that connection attempt which would be carrying that nonce in its first request packet. The connection establishment request packets of the attackers without the nonce, shared during the authentication phase, would be dropped by the firewall. The proposed methodology is discussed in detail in subsequent sections.

5.2 Assumptions

Before we proceed to our proposed design, let us list down the assumptions that underlay our proposed architecture for resolution of the authentication-connection association problem.

5.2.1 Authentication-Connection Association Problem and Session Hijacking

This dissertation is only concerned with resolving the authentication-connection association problem. Session hijacking is a separate problem and is out of the scope of this dissertation. The problem of session hijacking has received some attention in the past and session hijacking has been made difficult for the attackers by some measures, for example complexity has been added to the procedure of ISN number generation, replacement of hubs with switches in networks making sniffing difficult etc. But the authentication-connection association problem in passive authorization schemes is still very easy to exploit for an attacker. This dissertation presents a solution to the authentication-connection association problem that would make it impossible for an attacker to bypass the firewall authentication and the only way left for the attacker to connect to the server on behalf of a valid client would be to hijack an established TCP connection.

5.2.2 Capabilities of the Attacker

In our threat model, we assume that the attacker possesses the power to craft and send any type of packets containing any information and these are not blocked by ingress or egress filters.

5.2.3 Limitations of the Attacker and Justifications

In our threat model, the attacker is not able to stop or modify the packets in transit between the client and server. It is also assumed that it is hard for an attacker to sniff the ISN (Initial Sequence Number) and ACK (Acknowledgement Number) from packets in transit between client and server. However he can observe on-the-fly the connections between client and server as soon as they are established and the typical information associated with these connections like source and destination IP addresses of packets, source and destination port

numbers, underlying network protocol (TCP, UDP, ICMP) and application-level protocols being used (e.g. SMTP, FTP, etc).

It is important to note that for an attacker, it is trivial to exploit the authentication-connection association problem even if he does not have the access to the complete information contained in the packets in transit between the client and server. Information about source and destination IP addresses and port numbers, and the authority of observing connections made between client and server on-the-fly is sufficient for an attacker to easily bypass the authentication at firewall and connect to the protected server masquerading as a valid client. An example of an attacker in such a position could be of one having user-level access to a router, firewall or a proxy server near the client or to the log files generated by routers and firewalls on-the-fly. In other words, this dissertation assumes that the attacker is not able to sniff the packet-level information but he is able to get all the connection-associated information. It is worth mentioning that all those solutions that have been proposed to resolve the problem of session hijacking, by making the process of ISN number generation process more random and complex to make is difficult for an attacker to predict, work under similar assumptions.

Though it is not impractical, but to be able to sniff all of the information from the packets in transit between a client and server is a bit too strong position to be assumed for an attacker, when fixing the problems for the systems like port knocking and SPA. Port knocking and SPA are intended to provide only authentication at firewall and they provide confidentiality, using encryption, only to the authentication requests sent to the firewall. These systems are not meant to provide confidentiality and integrity to the connections that are to be established after successful authentication with the firewall.

For an attacker in a position to sniff all the information during transmission between a client and server, even session hijacking is not a problem. No attempt is made in this dissertation to prevent attacks after the connections have been opened, such as TCP session hijacking. If such attacks are a concern, than a system that provides both authentication and confidentiality of connections, such as IPsec or TLS, should be used instead of port knocking or SPA.

Also, traffic sniffing has been made difficult for attacker over the time. Even for an attacker residing within the switched network of victim client, it is not easy to sniff client's traffic unless he has compromised the server of organization through which all of the traffic is passing, or he uses ARP spoofing for which he needs to be in the subnet of client. For an attacker outside the client's network, traffic sniffing is even harder.

5.3 Proposed Approach

If the authentication-connection association problem is analyzed, an obvious solution for resolution of this problem, that is simple, lightweight, feasible with implementation perspective and easily incorporable in the existing systems, can be proposed by sharing some encrypted information during the authentication phase between the client and firewall (daemon) and than carry this information to the TCP connection establishment phase somehow in order to associate the two phases. This dissertation proposes such a design that meets all of the previously mentioned challenges by sharing something as simple as a nonce between client and the firewall during the authentication phase and resolves the authentication-connection association problem.

The basic idea is to exchange an encrypted nonce between the client and the firewall during the authentication phase and then encode that nonce in a suitable IP or TCP options field in

the SYN packet of the subsequent TCP connection establishment phase, to associate the two phases. After a successful authentication, the firewall would accept only those connection attempts that would be carrying the nonce, shared during the authentication phase, in unencrypted form in its first request packet. The connection establishment request packets of the attackers without the nonce would be dropped by the firewall.

Section 5.4 and 5.5 describe, at an abstract level, our proposed design with the perspective of effective incorporation into port knocking and single packet authorization systems separately. Section 5.6 explains the implementation aspects of our proposed design.

5.4 Incorporation into Port Knocking

The evolution of port knocking over the time has further resulted into two variants of port knocking systems. One category of port knocking systems integrate challenge-response schemes [1,11] to the basic concept in order to gain certain benefits, during the authentication phase. The second category of systems stick with the actual idea of port knocking to preserve stealthiness, they do not require the firewall to generate any kind of response after receiving the authentication request packet/knock sequence from the client. In sections 5.4.1 and 5.4.2, we discuss our proposed design with perspective of incorporating it in each of the above mentioned categories of port knocking systems.

5.4.1 For Challenge-Response based Port Knocking Systems

As stated earlier, the notion of challenge-response already exists in a few port knocking systems. This section discusses the strategy for effectively incorporating our design in such systems. Communication protocol of our design is shown in Figure 5.1.

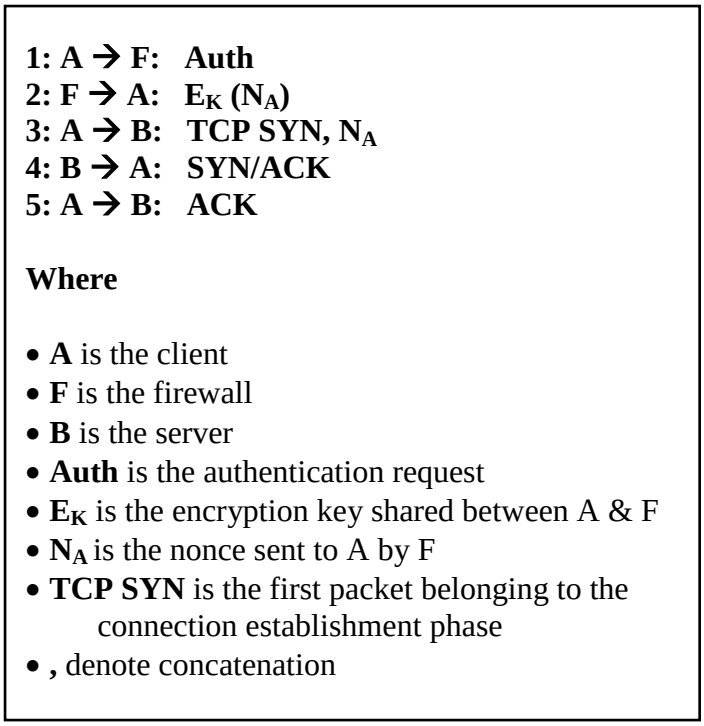

Figure 5. 1: Proposed communication protocol for challenge-response based port knocking systems

In the communication protocol suggested in Figure 5.1, **A** sends the authentication request to **F**, using multiple packets in case of port knocking and using a single packet in case of SPA. **F** checks the validity of the request, if the request is not valid no response is sent and the packet is dropped. In response to a valid request, a 64-bit nonce is randomly generated, encrypted using a shared symmetric key and is sent to **A** in the payload of the UDP packet used by challenge-response based port knocking systems to send information to the client. This response serves as an acknowledgement confirming **A** that the pre-specified port on **F** has been opened. **A** decrypts the nonce, encodes it in an appropriate IP or TCP options field in the SYN packet of the subsequent TCP connection and sends is to **S** through the open port. The SYN packet is allowed through **F** to **S** as it carries that nonce. **S** sends back the SYN/ACK request normally, **C** Sends an ACK as a response and the TCP session is established successfully and the client can than execute his commands on the protected server. Every connection request after successful authentication at firewall would be

dropped if it does not contain the nonce shared between firewall and the client. A typical illustration of the proposed communication protocol for challenge-response based port knocking systems is shown in Figure 5.2.

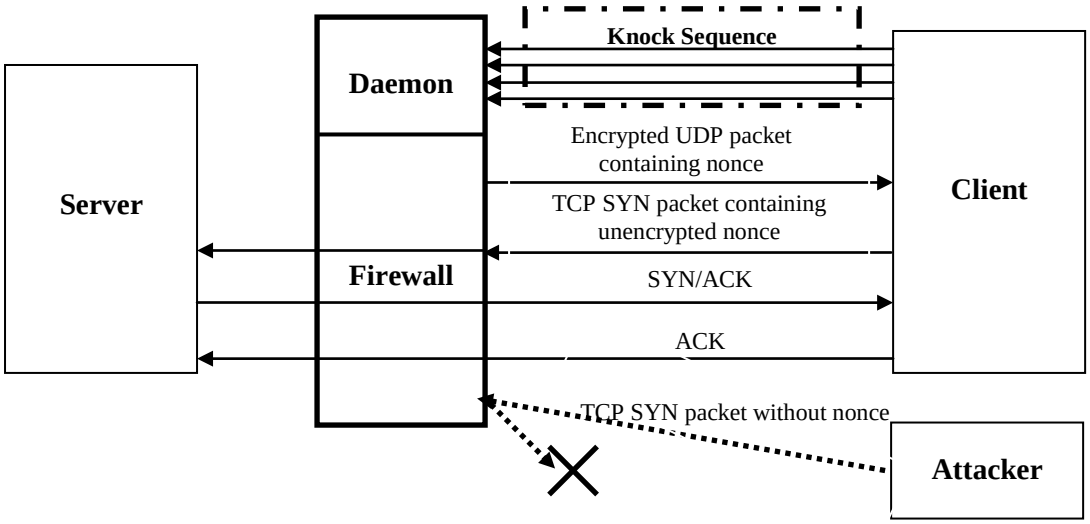

Figure 5. 2: Incorporating the proposed design into challenge-response based port knocking systems

It is important to mention that the client does not encrypt the nonce that he sends encoded in the SYN packet of the actual TCP connection. Even if the attacker is able to sniff that unencrypted nonce somehow, it would not work for him as the client's packet would have reached the server to initiate the session before the attacker crafts a packet and attaches that nonce to it, because we assumed that the attacker is not able to stop or modify a packet in transit between client and server. The attacker can block all the traffic from client though, by launching a DOS attack but in that case he will have to rely upon that encrypted nonce sent by the server and decrypt it to gain any benefit. The attacker may also take the client down after the client has sent the SYN packet, sniff that unencrypted nonce of that packet, send a spoofed RST packet to reset the client's session and initiate his own connection along with a valid nonce. To thwart such attempts, the firewall should allow only one connection attempt per successful authentication.

5.4.2 For Non-Challenge-Response based Port Knocking Systems

The problem with the previous approach is that it requires the firewall to generate a response to the authentication request of a client and majority of Port Knocking and SPA implementations do not follow this scheme. It is also not recommended in systems like port knocking and SPA as it may result in loss of stealth confirming to attackers about the existence of a listening server.

To incorporate our design with those port knocking and SPA schemes that do not rely on challenge-response during the authentication with the firewall, we propose a modification in the previous approach, shown in Figure 5.3.

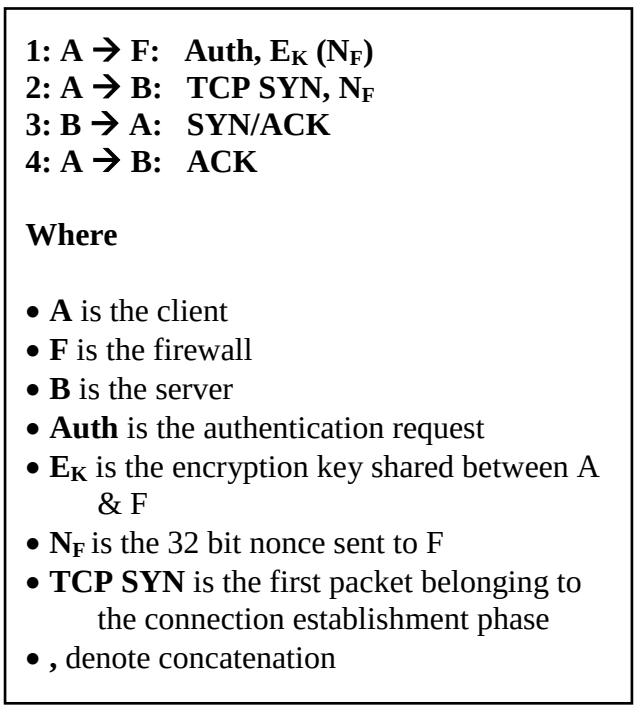

Figure 5. 3: Proposed communication protocol for standard port knocking and SPA systems

In the communication protocol shown in figure 5.3, **A,** instead of **F**, generates a random nonce and encrypt it using the shared symmetric key. Because of increased size due to encryption, the nonce is divided in smaller parts and is sent along with the knock sequence

by encoding in suitable fields, as shown in figure 5.4, or it can be made a part of normal authentication data. After validating the request, parts of nonce are collected and merged by **F** and the pre-specified port is opened for **A** by **F**. **A** will then send unencrypted nonce to **B** by encoding it in the SYN packet of the subsequent TCP connection and the rest will go on smoothly as already explained in the previous section.

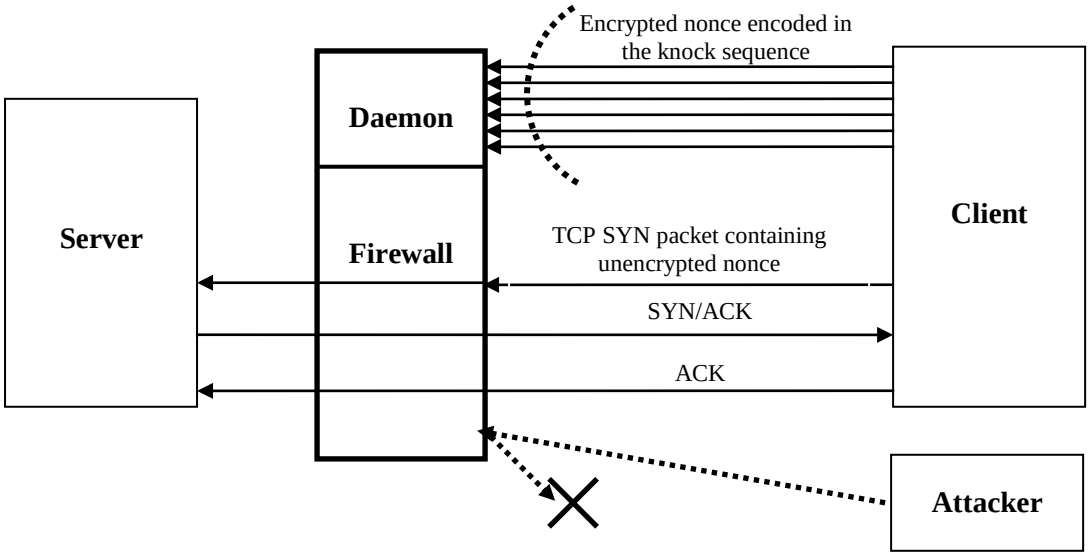

Figure 5. 4: Incorporating the proposed design into standard port knocking systems

This scheme prevents against the susceptibility to the loss of stealth, to which the challenge-response based port knocking systems are considered vulnerable. Some of the available port knocking systems already employ encryption to the authentication data and transmit this data to the firewall by dividing it in parts and encoding these parts in suitable fields of packets in the knock sequence. In such systems, the unencrypted nonce can simply be made a part of authentication data. Those port knocking systems that authenticate users by sending connection attempts to the pre-specified ports on the firewall and do not add additional authentication credentials to these packets, they either need to be enhanced in order to perform the functionality of encrypting, dividing and encoding of nonce in suitable

fields of knock packets, or a separate UDP packet containing the encrypted nonce can be sent, on a pre-specified port of firewall, during the knock sequence.

5.5 Incorporation into Single Packet Authorization

This section discusses the strategy for effectively incorporating our design in single packet authorization. In SPA, like most of the port knocking systems, the SPA server does not generate any response to the authentication request of the client. Thus the communication protocol shown in Figure 5.3 is equally applicable to SPA based dynamic firewalls. A typical illustration of the proposed communication protocol for single packet authorization systems is shown in Figure 5.5.

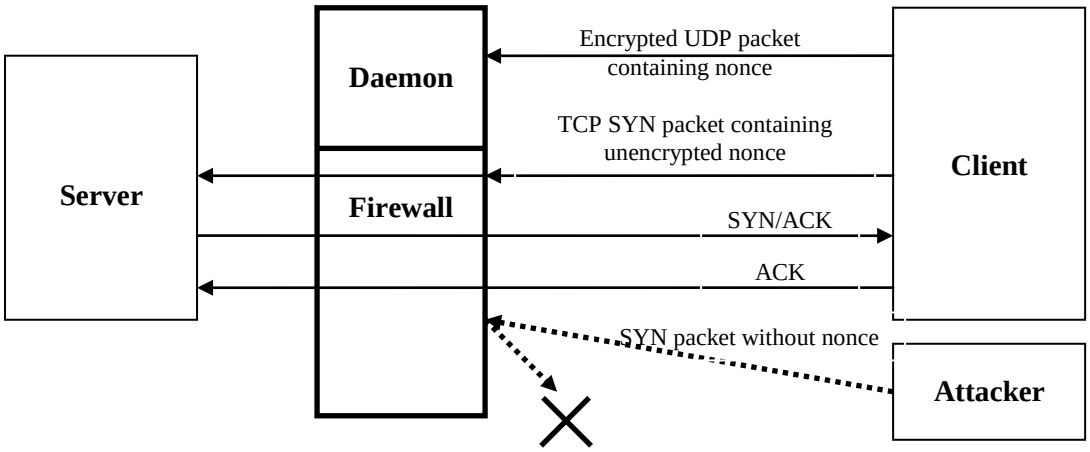

Figure 5. 5: Incorporating the proposed design into SPA

In SPA, the nonce has to be randomly generated by the client. That nonce has to be made a part of encrypted UDP packet used to transmit authentication data from client to the firewall. The daemon, on successful validation of the client's request, will open the pre-specified firewall port and expect to receive the nonce in unencrypted form in the upcoming TCP connection-establishment request. Connection request packets of the attackers, hoping to exploit the authentication-connection association problem, without that

nonce will be dropped by the firewall. The connection request of the actual client will be allowed through the firewall to the protected server.

5.6 Implementation Aspects

In both port knocking and SPA, the clients use a program that acts as a wrapper over the service-client. This wrapper authenticates the client with firewall and keeps the entire procedure transparent from the service-client. For incorporating any of the two designs we have proposed, the wrapper would require some extension to its capabilities in order to act as a proxy for the service-client. All the traffic of the service-client would be tunneled through that proxy to allow the wrapper to encode nonce in the SYN packet of the connection to be established. The wrapper would also recalculate and update the checksum of the SYN packet. It is important to mention that this extension in the capabilities of the wrapper does not require any support from the operating system kernel and adds no complexity on the server-side, unlike most of previously proposed solutions.

For encoding the nonce in the SYN packet of the TCP connection to be established, IP Timestamp and TCP echo option are two suitable fields as proposed in [17]. Size of both fields is 36 bytes each and any of them can easily accommodate an 8-byte nonce. Note that if the timestamp field is used for transmission of nonce, the timestamp buffer has to be flagged as full, so that intermediate routers do not manipulate the timestamp. To ensure end-to-end reliability in transmitting the packet, it is not recommended using the payload of the SYN packet to transmit the nonce because some routers trim the payload attached to the SYN packets and intrusion detection and prevention systems filter SYN packets with large payloads.

For maximum assimilation, it is recommended in this dissertation to make the nonce, used in proposed scheme to associate the authentication and subsequent connection establishment phases, a part of normal authentication data in all the existing systems of port knocking and single packet authorization. This also helps keeping the simplicity of these architectures unharmed and as a result assures security. For all those systems where it is not feasible to make that nonce a part of normal authentication data, perhaps because of no notion of authentication data in some of port knocking systems, it has been suggested to generate an additional UDP packet, make the encrypted nonce a part of its payload and send it to the destined entity falling in either categories mentioned in 5.3.1 or 5.3.2. In that case, AES should be used for encryption with 128 bit of key length. The size of nonce to be used should be atleast 32-bits but we recommend using 64-bit randomly generated nonce to ensure good security.

5.7 Summary

This chapter has presented detailed technical description of the design that has been proposed in this dissertation to resolve the authentication-connection association problem. We clearly defined the threat model in the start of this chapter and also presented detailed justifications for the assumptions that were taken in this regard. Then we explained how the proposed design can be incorporated into two forms of port knocking and single packet authorization schemes. Towards the end of the chapter, we discussed the implementation aspects of our proposed design along with certain recommendations.

Chapter 6

PERFORMANCE EVALUATION

6.1 Introduction

This chapter presents implementation aspects of the proposed system, performance evaluation and the results obtained from the performance evaluation of the proposed system. The performance evaluation of the proposed system has been carried out on the basis of various parameters like processing overhead, robustness and stealthiness. These are the parameters that have been commonly used to evaluate the passive authorization systems.

6.2 Implementation

The proposed design has been implemented by modifying an existing open-source port knocking system, JPortKnock [29]. JPortKnock is an implementation of port knocking in Java available for both Linux and Windows platforms. We have implemented our system by incorporating the proposed design in JPortKnock on Microsoft Windows Platform. The implementation has been done in Java using JDK1.4. The test platform used for implementation and performance evaluation of the proposed system was a dual Intel Xeon machine running at 2.8 GHz with a 4 MHz bus, 512 kB L2 Cache and 1 GB of RAM. The source code of the two main classes of the simulation is attached as appendix A.

6.3 Processing Overhead

Most of the problems that have been resolved in port knocking and single packet authorization based dynamic firewalls are at the cost of increased overhead on these

systems. The more features are added to these systems, the more they become vulnerable to DOS attacks. Hence a primary concern while enhancing these systems is that the overhead imposed by the proposed enhancements at the server side remain marginal.

To measure the processing overhead incurred by incorporating our proposed design into the existing systems, the ability of processing different numbers of simultaneous authentication request packets of an existing and the proposed system was tested and the deviation in corresponding delay was observed. Existing system used was the standard JPortKnock System and proposed system was the version of JPortKnock in which our proposed design was incorporated. The results obtained after the experiments conducted during the simulation phase are shown in Table 6.1.

Table 1: Performance comparison of the proposed system

Auth. Requests Per Second	Delay in milliseconds			Percentage Performance Overhead of Proposed System	
	Theoretical Results (X)	Existing System (Y)	Proposed System (Z)	Compared to Theoretical Results (DT)	Compared to Existing System (DE)
1	0.344	0.344	0.347	0.872 %	0.872 %
2	0.688	0.691	0.698	1.453 %	1.013 %
4	1.376	1.383	1.396	1.453 %	0.940 %
8	2.752	2.763	2.788	1.308 %	0.905 %
16	5.504	5.523	5.565	1.108 %	0.760 %
32	11.008	11.041	11.112	0.945 %	0.643 %

The percentage processing overhead imposed by the existing and proposed systems was calculated using the following formulas

$$DT = \{(Z-X)/X\}*100$$

$$DE = \{(Z-Y)/Y\}*100$$

The percentage processing overhead imposed by our proposed design, compared to the theoretical results and the results obtained from the existing system, is shown in Figure 6.1.

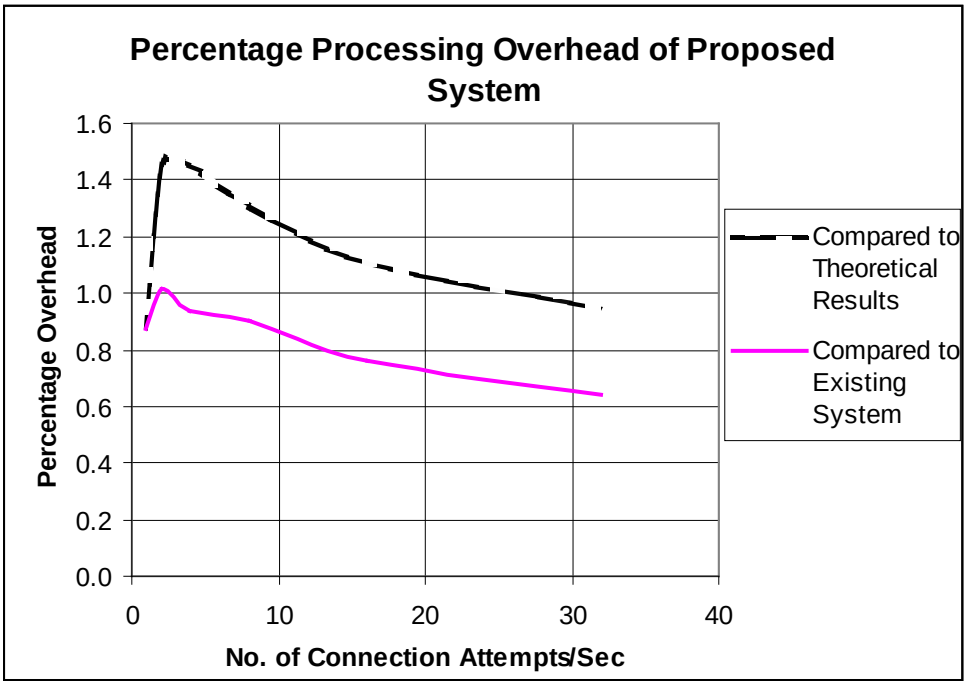

Figure 6. 1: Percentage processing overhead of proposed system

The graph shows that the processing overhead of the proposed system when compared to the existing system remains less than 1% on average. The processing overhead of the proposed design when compared to the theoretical results also does not exceeds 1.5%. It is also evident from the graph that the processing overhead keeps on decreasing with the increase of load on the server.

The results clearly show that the overhead imposed by our proposed design over the existing system is very marginal as compared to the criticality of the problem of lack of association between the authentication process and the follow-on TCP connection to be established, in both port knocking and single packet authorization.

6.4 Robustness

Robustness is defined as the degree to which a system or component can still function in the presence of partial failures or other adverse, invalid, or abnormal conditions. Under robustness we analyze the effects of various intermediate devices, like firewalls, intrusion detection and prevention systems, proxies and NATs, on our proposed design.

Some firewalls potentially trim the data attached to the payload of a SYN packet passing through it. Similarly, many IDSs and IPSs deem the SYN packets with payload as malicious. To deal with these issues, we have proposed to encode unencrypted nonce in suitable IP or TCP options field, during the actual TCP connection establishment phase, in both of the implementation alternatives of our design. We do not use the payload of the SYN packet to transmit the nonce in unencrypted form. Using the IP Timestamp field to encode data with the Timestamp buffer flagged as full, the blockades of firewalls, intrusion detection and prevention systems are evaded.

The ability of working across NATs/proxies has been a significant issue for port knocking and single packet authorization schemes. Some systems have partially got rid of this issue by adding various innovations to the concepts of passive authorization techniques but most of them still suffer. In the approach that has been proposed for challenge-response based port knocking systems, if the attacker and the client are behind a common NAT, the attacker can take the client down after the client has sent the authentication request to the firewall and receive the challenge packet containing encrypted nonce on behalf of that client. But since the nonce is encrypted, it would be of no use for the attacker because to connect to the server on behalf of that client, the attacker needs to decrypt that nonce. The design that has been proposed for standard port knocking and single packet authorization

systems is totally immune to this issue because in that the nonce generation, encryption and its transmission along with the authentication request is to be done by the client. Under our initial assumptions, it is impossible for an attacker to sniff the encrypted or unencrypted nonce from the packets sent by the client to the firewall and server, thus preventing against the exploitation of the authentication-connection association problem.

6.5 Stealthiness

Stealthiness is a characteristic that is defined as the ease with which an attacker can discover a hidden service. Primarily in both passive authorization techniques, the server sends no response until a valid passphrase/knock sequence is received. But some port knocking architectures that rely on challenge-response during the authentication phase are susceptible to the loss of stealth. Our first approach is proposed to be incorporated into such systems; it is as susceptible to the loss of stealth as those challenge-response based port knocking systems are. But even in these systems, our scheme uses encrypted nonce to be sent by the firewall to the client, it is still impossible for an attacker to get any benefit from our design. However, our second approach has to be incorporated into completely non-responsive implementations of port knocking and SPA, this approach does not compromise the stealthiness of these systems. Use of encryption in both of the proposed implementation alternatives helps in ensuring stealth, against those attackers having limited capability of sniffing the information in transit between the client and firewall.

6.6 Summary

This chapter presented implementation details of the proposed system. Performance evaluation of the proposed system has been carried out using the parameters of the processing overhead, robustness and stealthiness. We showed that the processing

overhead imposed by incorporating the proposed design into passive authorization techniques remains marginal and do not compromise the stealthiness of these schemes. We also showed that the proposed system is robust enough to evade the blockades from which most of the existing port knocking and single packet authorization systems suffer.

Chapter 7

CONCLUSION

7.1 Overview

No security mechanism is perfect; same is the case with firewalls. According to Marcus Ranum, the only perfect firewall is a pair of wire cutters applied to all the wires connecting a computer system to a network. But there is always a room for improvement in everything. Static firewalls are only able to filter packets on the basis of network addresses. The advent of passive authorization technologies, port knocking and single packet authorization, provided the opportunity to enhance the firewall technology by adding the features of strong user authentication and dynamic reconfiguration to the firewalls.

Despite of being very powerful concepts, almost all of the implementations of both port knocking and single packet authorization suffer from the problem of lack of association between the authentication process and the subsequent TCP connection establishment phase. It is such a crucial problem that it allows the attackers to bypass the user authentication process and connect to the protected service on behalf of a valid client, making the entire concept of firewall-level authentication useless. What adds to the severity of this problem is the fact that it is fairly easy to exploit.

This dissertation has analyzed authentication-connection association problem deeply and has proposed a simple and lightweight design that makes it impossible for the attackers to exploit this problem. Two approaches have been devised to incorporate the proposed design into the existing architectures of port knocking and SPA systems without compromising their existing strengths. The proposed approach has been evaluated on the

basis of various parameters like robustness, stealthiness and security. For the purpose of performance evaluation, the capabilities of an existing open-source port knocking system, JPortKnock, have been extended. Simulation results have shown that the processing overhead, which is crucial for passive authorization systems, incurred by incorporating this design into the existing systems is marginal.

7.2 Future Work

Though the techniques of port knocking and single packet authorization have matured enough over the time, still there exist some problems in both of these passive authorization techniques that provide the scope for further research.

7.2.1 The NAT Problem

The problem of working across a NAT is serious issue in passive authorization techniques. When a valid client is behind a NAT, the port opened by the firewall for that client, after successful authentication, to access the protected service is virtually for the whole client’s network sharing a single public IP. If traffic from a client passes through a NAT to a server, and the client's (private) IP address is encoded in the authentication token, then the authentication exchange, if successful, will result in the correct port being opened to the incorrect client address. If the public IP address is encoded in the token, then the port will be opened to all hosts sharing the same public address. If the client's address is not encoded in the token at all, then the public address from the packet headers would likely be used, leading to the same problem. Fixation of this problem is a potential area for the future research in these systems.

7.2.2 Susceptibility to DOS Attacks

The second area having potential for future research in passive authorization systems is their susceptibility to denial-of-service attacks. An attacker could prevent a client from authenticating by sending packets with the client's source address to random ports on the server while the client is trying to authenticate; if any of these packets went to ports being monitored by the port knocking server, then the client's sequence would be corrupted and authentication would fail. Also, an attacker could affect a resource-consumption attack against a known port knocking server by sending packets with random forged source addresses to random ports. A port scan from a single source at a sufficiently high rate may also be sufficient to overload a server's processing resources, particularly if the server uses a computationally-intensive cryptographic protocol.

7.2.3 Utility of Asymmetric Key Cryptography

Except for one implementation of SPA that uses GPG-rings, the rest of port knocking and SPA systems rely on encryption based on symmetric key. The feasibility of public-key algorithms can also be checked in these passive authorization techniques, particulars of such design need investigation.

BIBILIOGRAPHY

[1] Computer Emergency Response Team, "Vulnerability Remediation Statistics", *Computer Emergency Response Team, Carnegie Mellon University,* 1995-2008. [Online]. Available: http://www.cert.org/stats/fullstats.html. [Accessed: Jan. 10, 2008].

[2] P. Lunsford and E. C. Wright, "Closed Port Authentication with Port Knocking", *in Proceedings of the American Society for Engineering Education Annual Conference & Exposition,* Portland, OR, June 2005, pp. 1747-1754.

[3] Michael Rash, "Single Packet Authorization", *The Linux Journal,* April 2007. [Online]. Available: http://www.linuxjournal.com/article/9565. [Accessed: Dec 15, 2007].

[4] R. deGraaf, J. Aycock and M. Jacobson, "Improved Port Knocking with Strong Authentication" *in IEEE/ACM Proceedings of the 21st Annual Computer Security Applications Conference,* Tucson, Arizona, December 2005, pp. 409-418.

[5] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin, *"Firewalls and Internet Security: Repelling the Wily Hacker"*, Addison-Wesley 2nd Edition, February 2003.

[6] Ido Dubrawsky, "Firewall Evolution - Deep Packet Inspection", *SecurityFocus.com,* December 2005. [Online]. Available: http://www.securityfocus.com/infocus/1716 [Accessed: Oct 21, 2007].

[7] Steven M. Bellovin, "Distributed Firewalls", *USENIX ;login:*, November 1999, pp. 39-47.

[8] Gian Luca Volpato and Christian Grimm, "Firewalls Issues Overview", *OGF.org,* August 2006. [Online]. Available: http://www.ogf.org/documents/GFD.83.pdf. [Accessed: Nov 11, 2007].

[9] M. Krzywinski, "Port knocking: Network authentication across closed ports" *SysAdmin Magazine,* vol. 12, no. 6, June 2003, pp 12–17.

[10] Martin Krzywinski, "Port Knocking", *PortKnocking.org,* December 2007. [Online]. Available: http://www.portknocking.org. [Accessed: Jan 17, 2008].

[11] P. Barham, S. Hand, R. Isaacs, P. Jardetzky, R. Mortier, and T. Roscoe, "Techniques for Lightweight Concealment and Authentication in IP Networks," *Intel Research Berkely,* Tech. Rep. IRB-TR-02-009, July 2002. [Online]. Available: http://www.intel-research.net/Publications/Berkeley/012720031106_111.pdf. [Accessed: June 27, 2007].

[12] Dawn Isabel, "Port Knocking: Beyond the Basics", *SANS,* May 2005. [Online]. Available:http://www.sans.org/reading_room/papers/download.php?id=1634&c=85ab23c98c7386048aba1c3d37a91486. [Accessed: Apr 24, 2007].

[13] A. Manzanares, J. Marquez, J. Tapiador and J. Castro, “Attacks on Port Knocking Authentication Mechanism”, *in Computational Science and its Applications – ICCSA, Lecture Notes in Computer Science,* Springer Berlin/Heidelberg 2005, vol. 3483, pp. 1292-1300.

[14] A. Narayanan, “A critique of port knocking”, *Newsforge*, August 2004. [Online]. Available: http://www.linux.com/articles/37888. [Accessed: Sep 29, 2007].

[15] C. K. TAN and Cappella, “Remote Server Management using Dynamic Port Knocking and Forwarding”, *in Special Interest Group in Security and Information Integrity,* May 2004. [Online]. Available: http://www.security.org.sg/code/sig2portknock.pdf. [Accessed: Nov23, 2007].

[16] Michael Rash, “Combining port knocking and passive OS fingerprinting with fwknop”, *USENIX ;login:*, December 2004. [Online]. Available: http://www.usenix.org/publications/login/2004-12/pdfs/fwknop.pdf. [Accessed: Jan 22, 2007].

[17] David Worth, “COK - Cryptographic One-Time Knocking”, *BlackHat USA,* Dec 2004. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-worth-up.pdf. [Accessed: Nov 15, 2007].

[18] E. Vasserman, N. Hopper, J. Laxson and J. Tyra, “SILENTKNOCK: Practical, Provably Undetectable Authentication”, *in European Symposium on Research in Computer Security*, Lecture Notes in Computer Science, Springer-Berlin/Heidelberg, Dresden Germany, September 2007, pp. 122-138.

[19] S. Jeanquier, “An Analysis of Port Knocking and Single Packet Authorization”, MSc Thesis, University of London, London, UK, September 2006.

[20] Michael Rash, “Protecting SSH Servers with Single Packet Authorization”, *The Linux Journal,* May 2007. [Online]. Available: http://www.linuxjournal.com/article/9621. [Accessed: Sep 30, 2007].

[21] Madhat Unspecific and Simple Nomad, “SPA: Single Packet Authorization”, *BlackHat USA,* July 2005. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-madhat.pdf. [Accessed: Nov 15, 2007].

[22] Michael Rash, “Single Packet Authorization with fwknop”, *USENIX ;login:,* February 2006. [Online]. Available: http://www.usenix.org/publications/login/2006-02/pdfs/rash.pdf. [Accessed: Nov 15, 2007]

[23] Marco de Vivo, Eddy Carrasco, Germinal Isern and Gabriela O. de Vivo**,** “A Review of Port Scanning Techniques”, *in ACM SIGCOMM Computer Communication Review,* April 1999, vol. 29, issue 2, pp. 41-48.

[24] Herbert, D. L. Devgan, S. S. Beane, “Application of network address translation in a local area network” *in Proceedings of the 33rd IEEE Southeastern Symposium on System Theory*, Athens, OH, USA, Mar 2001, pp 315-318.

[25] John Aycock, *“Computer Viruses and Malware”*, Advances in Information Security, Springer, vol. 22, July 2006.

[26] Kevin Lam, David LeBlanc, and Ben Smith, “Theft on the Web: Prevent Session Hijacking”, *Microsoft.com*, June 2005, [Online]. Available: http://www.microsoft.com/technet/technetmag/issues/2005/01/SessionHijacking/default.aspx. [Accessed: Oct 23, 2007]

[27] Shray Kapoor, “Session Hijacking: Exploiting TCP, UDP and HTTP Sessions”, July 2006. [Online]. Available: www.infosecwriters.com/text_resources/pdf/SKapoor_SessionHijacking.pdf. [Accessed: Sep 15, 2007]

[28] Kamran Ahsan, “Covert Channel Analysis and Data Hiding in TCP/IP”, M.S. Thesis, University of Toronto, Toronto, Sep 2002.

[29] Mondain, “JPortKnock”, Mar 2007. [Online]. Available: https://jportknock.dev.java.net/. [Accessed: Nov 15, 2007].

[30] Matt Doyle, “Implementing a Port Knocking System in C”, B.S. Honors Thesis, Department of Computer Science and Computer Engineering, The University of Arkansas, Arkansas, USA, May 2004.

Appendix A

SOURCE CODE

The source code for the two main classes of the project, Port Knocking Server and Port Knocking Client, is as follows.

Port Knocking Server

```
package org.gregoire.portknock.server;

import java.io.*;
import java.net.*;
import java.nio.*;
import java.util.*;

import org.apache.log4j.*;
import com.ibm.io.async.*;

public class AsyncServerImpl extends Server {

    private static int counter = 0;
    private AsyncServerSocketChannel ssc;
    private AsyncServerSocketChannel finalChannel;
    protected LinkedList nextChannelList;

    static {
        //log for this class only
        logger = Logger.getLogger(AsyncServerImpl.class.getName());
    }

    public AsyncServerImpl(String type) {
        super(type);
        //list to hold all the "connectors"
        list = new LinkedList();
        //list to hold all the "next" channels
        nextChannelList = new LinkedList();
    }

    public void init() throws IOException {
        // Create a new server socket and set the blocking mode
        ssc = AsyncServerSocketChannel.open();
    }

    // Accept connections for current server.
    public void listen() {
        try {
            ServerSocket serverSocket = ssc.socket();
            // Set socket options.
            serverSocket.setReuseAddress(true);
            // Bind the server socket to the local host and port or the
            incomming host and port
            InetSocketAddress isa = new InetSocketAddress(host,
                  ports[0]);
            serverSocket.bind(isa);
```

```
            logger.debug("In listen, so timeout: " +
                  serverSocket.getSoTimeout());

            // Listen for client connections.
            AsyncSocketChannel clientChannel = ssc.accept();
            //set the timeout
            //ssc.socket().setSoTimeout(timeout);
            if (clientChannel.isConnected()) {
                // The key indexes into the selector so you
                // can retrieve the socket that's ready for I/O
                SocketHandler handler = new SocketHandler();
                Socket socket = clientChannel.socket();
                handler.setSocket(socket);
                new Thread(handler, "Socket handler " +
                      counter++).start();
                socket = null;
            }
        } catch (Exception ex) {
            logger.log(Level.ERROR, "Exception in listen: " +
            ex.getMessage(), ex);
        }
    }

    public class SocketHandler implements Runnable {

        Socket socket = null;

        public void setSocket(Socket socket) {
            this.socket = socket;
        }

        public void run() {
            try {
                handle();
                socket.close();
            } catch (IOException ex) {
            }
        }

        private void handle() throws IOException {
            InetAddress address = socket.getInetAddress();
            Client client = new Client(address);
            if (!list.contains(client)) {
                logger.log(Level.DEBUG, "Client not in list");
                list.add(client);
            } else {
                Client tmp = null;
                int listSize = list.size();
                for (int c = 0; c < listSize; c++) {
                    tmp = (Client) list.get(c);
                    if (client.equals(tmp)) {
                        //set the reference to the current client
                        client = tmp;
                        //increment the connection count
                        client.incrementSequence();
                        //if we have reached the end of the sequence
                              then open their tunnel
                        if (client.getSequence() == ports.length) {
                            //remove the record
                            list.remove(c);
```

```
                    //open tunnel and allow only current
                          requester in
                    logger.log(Level.DEBUG, "Got a good
                          sequence");
                    listenFinal(client);
                    //return control
                    return;
                }
                break;
            }
            tmp = null;
        }
    }

    logger.log(Level.DEBUG,socket.getInetAddress().toString());
    logger.log(Level.DEBUG, "List size: " + list.size());
    //listen on the next port in the sequence
    listenNext(client);
}

// Accept connections on a follow on sequence port
public void listenNext(Client client) {
    int sequence = client.getSequence();
    logger.log(Level.DEBUG, "Current port: " +
          ports[sequence]);
    ServerSocket serverSocket = null;
    AsyncServerSocketChannel nextChannel = null;
    AsyncSocketChannel clientChannel = null;
    try {
        if (nextChannelList.size() >= sequence) {
            //look for a matching channel in the channel list
            Object tmp = nextChannelList.get(sequence - 1);
            if (null != tmp) {
                nextChannel = (AsyncServerSocketChannel) tmp;
            }
            //bind the socket channel to the inet address
            serverSocket = nextChannel.socket();
        } else {
            //create channel for "next" sockets
            nextChannel = AsyncServerSocketChannel.open();
            //add it to the list
            nextChannelList.add( (sequence - 1), nextChannel);
            //bind the socket channel to the inet address
            serverSocket = nextChannel.socket();
            // Set socket options.
            serverSocket.setReuseAddress(true);
            //set the timeout
            serverSocket.setSoTimeout(portDelay);
            // Bind the server socket to the local host and
                  port or the incomming host and port
            InetSocketAddress isa = new InetSocketAddress(host,
            ports[sequence]);
            // Bind the server socket to the local host and
                  port or the incomming host and port
            serverSocket.bind(isa);
        }
        while (true) {
            // Listen for client connections.
            clientChannel = nextChannel.accept();
            //check to see if this new "connector" is the one
                  we are expecting
```

```
                if(!client.getAddress().equals
                     (clientChannel.socket().getInetAddress())) {
                    logger.log(Level.DEBUG, "Got a bad connect");
                }
                else {
                    logger.log(Level.DEBUG, "Got a follow on
                          connect");
                    handle();
                    break;
                }
            }
        } catch (Exception ex) {
            logger.log(Level.ERROR, "Exception in listen next: " +
                  ex.getMessage(), ex);
        } finally {
            try {
                clientChannel.close();
            } catch (IOException ex1) {
                logger.log(Level.ERROR, "Error closing server
                      socket", ex1);
            }
        }
    }

    // Accept connections on a follow on sequence port
    public void listenFinal(Client client) {
        logger.log(Level.DEBUG, "Current port: " + port);
        ServerSocket serverSocket = null;
        AsyncSocketChannel clientChannel = null;
        try {
            //if final channel is null then open it
            if (null == finalChannel) {
                //create channel for "final" sockets
                finalChannel = AsyncServerSocketChannel.open();
            }
            //get the socket channel
            serverSocket = finalChannel.socket();
            // Set socket options.
            serverSocket.setReuseAddress(true);
            //set the timeout
            serverSocket.setSoTimeout(portDelay);
            // Bind the server socket to the local host and port or
            the incomming host and port
            InetSocketAddress isa = new InetSocketAddress(host,
                  port);
            // Bind the server socket to the local host and port or
            the incomming host and port
            serverSocket.bind(isa);
            while (true) {
                // Listen for client connections.
                clientChannel = finalChannel.accept();

                Socket c = clientChannel.socket();
                DataInputStream din = new
                DataInputStream(c.getInputStream());
                long nonce = din.readLong();
                System.out.println(nonce);
                FtpServer ftp = new FtpServer(nonce);
                ftp.start();

                System.out.println("after ftp start");
```

```
            //check to see if this new "connector" is the one
                  we are expecting
            if(!client.getAddress().equals(clientChannel.
                  socket().getInetAddress())) {
                logger.log(Level.DEBUG, "Got a bad final
                      connect");
            } else {
                logger.log(Level.DEBUG, "Got the final
                      connect");

                if (type.equals("sequence")) {
                    //do stuff with socket
                    ByteBuffer buffer =
                          ByteBuffer.allocateDirect(1024);
                    buffer.put("ok ok ok ok ok
                          ok\n".getBytes());
                    buffer.flip();
                    IAsyncFuture writeFuture =
                          clientChannel.write(buffer);
                    try {
                        logger.debug("Bytes written: " +
                          writeFuture.getByteCount());
                    } catch (AsyncException ex) {
                        logger.error(ex);
                    }

                } else if (type.equals("tunnel")) {
                    String path =
                          client.getRequestedApplication();
                    String env[] = {
                        "JAVA_HOME=E:/j2sdk1.4.2_05"};
                    Process p = Runtime.getRuntime().exec(path,
                    env);
                    // hook processes input to browser's output
                    (async)
                    final InputStream inFromReq =
                          socket.getInputStream();
                    final OutputStream outToCgi =
                          socket.getOutputStream();
                    final byte[] txBuffer = new byte[2048];

                    new Thread(new Runnable() {
                        public void run() {
                            try {
                                while (inFromReq.read(txBuffer)
                                  > -1) {
                                    outToCgi.write(txBuffer);
                                    outToCgi.flush();
                                }
                                outToCgi.close();
                            } catch (IOException e) {
                                logger.warn(e);
                            }
                        }
                    }).start();
                }
                break;
            }
        }
    } catch (Exception ex) {
```

```
                    logger.log(Level.ERROR, "Exception in listen next: " +
                           ex.getMessage(), ex);
                } finally {
                    try {
                        clientChannel.close();
                    } catch (IOException ex1) {
                        logger.log(Level.ERROR, "Error closing socket",
                        ex1);
                    }
                }
            }
        }
    }
```

Port Knocking Client

```
package org.gregoire.portknock.client;

import java.util.*;
import java.io.*;
import java.net.*;
import java.util.*;

public class Main {

    private String host;
    private int socketTimeout;
    private int delay;
    private int port;
    private int[] sequencePorts;
    private static long nonce = 130;

    public void init(Properties props) {
        //get number of ports used for sequence
        int sequenceLength =

Integer.parseInt(props.getProperty("good.ports.sequence"));
        System.out.println("Loading good ports sequence...");
        //get ports used for sequence
        String tmp = props.getProperty("good.ports");
        StringTokenizer st = new StringTokenizer(tmp, ",");
        if (st.countTokens() < sequenceLength) {
            sequenceLength = st.countTokens();
        }
        //setup an array to hold the ports
        System.out.println("setup an array to hold the ports...");
        sequencePorts = new int[sequenceLength];
        for (int i = 0; i < sequenceLength; i++) {
            sequencePorts[i] = Integer.parseInt(st.nextToken());
        }

        //the time to wait in between connection attempts
        Delay =   Integer.parseInt
      (props.getProperty("connection.delay"));

        //get host name or ip address used for sequence
        System.out.println("get host name or ip address used for
      sequence...");
        host = props.getProperty("host.address");

        //the last port that will be opened for connection if sequence
      is ok
        port = Integer.parseInt(props.getProperty("good.ports.final"));

        //get the default socket timeout
        socketTimeout =
      Integer.parseInt(props.getProperty("socket.timeout"));

    }

    /**
     * Returns the socket to be used for the tunneling application. The
      socket is only returned if the sequence is successful.
     * @return socket available if sequence is successful.
     * @throws IOException
```

```
 */
public Socket connect() throws IOException {
    //try all the sequence ports in order
    for (int i = 0; i < sequencePorts.length; i++) {
        Socket sock = new Socket(host, sequencePorts[i]);
        sock.setKeepAlive(false);
        sock.setReuseAddress(true);
        sock.setSoTimeout(socketTimeout);
        //close it down
        sock.close();
        try {
            Thread.currentThread().sleep(delay);
        } catch (InterruptedException ex) {
        }
        System.out.println("Done with: " + sequencePorts[i]);
    }
    System.out.println("Sequence completed");
    //try to open the final port..
    Socket sockFinal = new Socket(host, port);
    sockFinal.setKeepAlive(true);
    sockFinal.setReuseAddress(true);
    sockFinal.setSoTimeout(socketTimeout);
    //return the socket if we dont throw an exception
    return sockFinal;
}

public static void main(String args[]) {


    long time = System.currentTimeMillis();
    Socket socket = null;
    long nonce = 0;
    Properties props = new Properties();
    try {

        props.load(new FileInputStream("client.properties"));

        try {

            Main main = new Main();
            main.init(props);
            socket = main.connect();
            if (socket.isConnected()) {
                System.out.println("Sending nonce");
                System.out.println("Socket returned and
                      connected...");
                byte[] buff = new byte[1024];
                InputStream is = socket.getInputStream();
                DataOutputStream dout = new
                      DataOutputStream(socket.getOutputStream());
                Random r = new Random();
                nonce  = r.nextLong();
                System.out.println(nonce);
                dout.writeLong(nonce);
                while (is.read(buff) != -1) {
                    System.out.print(new String(buff));
                }
                System.out.println();
                is.close();
            }
```

```
            } catch (IOException ex) {
                System.err.println("Error with connection.");
            } finally {
                try {
                    if (null != socket) {
                        socket.close();
                    }
                } catch (Exception ex) {
                }
            }

        } catch (IOException ex) {
            System.err.println("Error loading properties file, server
                  will exit.");
        }
        //normal exit
        long currenttime = System.currentTimeMillis() - time+6;
        System.out.println("Total Execution time = "+currenttime);

        FtpClient ftpclient = new FtpClient(nonce,"127.0.0.1",2100);
        ftpclient.start();

        System.exit(0);
    }
}
```