Improved Port Knocking with Strong Authentication

Rennie deGraaf, John Aycock, and Michael Jacobson, Jr.*
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{*degraaf,aycock,jacobs*}*@cpsc.ucalgary.ca*

Abstract

It is sometimes desirable to allow access to open ports on a firewall only to authorized external users and present closed ports to all others. We examine ways to construct an authentication service to achieve this goal, and then examine one such method, "port knocking", and its existing implementations, in detail. We improve upon these existing implementations by presenting a novel port knocking architecture that provides strong authentication while addressing the weaknesses of existing port knocking systems.

1. Introduction

Thirty-five years after the birth of the Internet, it has become well established that the Internet is a hostile place. Any host connected to the Internet needs to be secured against unauthorized intrusion and other attacks. Unfortunately, the only secure system is one that is completely inaccessible, but, to be useful, many hosts need to make services accessible to other hosts. While some services need to be accessible to anyone from any location, others should only be accessed by a limited number of people, or from a limited set of locations.

The most obvious way to limit access is to require users to authenticate themselves before granting them access. Traditionally, this is left up to the services themselves: before granting users access to anything important, they must first prove their identity, using any one of a number of methods. While this is effective, it is not a perfect solution.

Many network services are large, complicated systems. It is not unheard of for flaws to be found in the authentication mechanisms of some of these services which could allow attackers to gain unauthorized access. Also, some services are insecure by design: there is no way for users to authenticate themselves. A common solution to this is to limit the hosts from which users can connect.

The usual method of limiting the sources of network connections is to use a firewall. A firewall works by selectively accepting or rejecting network packets based on their source addresses or other characteristics. Unfortunately, source addresses on packets tell little about the person who sent those packets; determined attackers are quite capable of disguising the source of packets that they send. Once a port on a firewall is open to one host, any attacker could potentially (ab)use that opening. Also, not all authorized users have predictable IP addresses, so granting them access through a firewall requires opening the firewall to much or all of the Internet. Thus, while firewalls are useful and effective defenses against many attacks, they are not complete solutions.

What is needed is a mechanism to open ports on a firewall to authenticated users, without allowing other traffic to pass. The obvious way to construct such a mechanism is to run an authentication service on firewalls, which validates the identity of remote users and modifies firewall rules according to per-user access policies. Such a service could be used for a number of purposes, including:

- making services invisible to standard port scans;
- providing an extra layer of security that attackers must penetrate before accessing or breaking anything important;
- acting as a stop-gap security measure for services with known unpatched vulnerabilities; or
- providing a wrapper for legacy or proprietary services with insufficient integrated security.

There are a number of ways to create such an authentication service; one is to use "port knocking". In this paper, we examine port knocking and its existing implementations in detail. Our survey found a number of flaws common to most or all existing port knocking systems, including failure if a client is NATed, failure

*The authors' research is supported in part by grants from the Natural Sciences and Engineering Research Council of Canada.

if packets are delivered out of order, and a lack of association between authentication and connections being opened. We then present a novel port knocking architecture that provides strong authentication while addressing the problems associated with NATs and out-of-order delivery.

2. Threats against firewall authentication services

Authenticating users before allowing them through a firewall is beneficial in two situations, each of which assumes attackers with differing capabilities:

1. as an extra layer of defense for critical services
2. as a light-weight authentication service for insecure legacy services

In the first case, such a service would be used to protect critical systems against attacks for which corrective patches or upgrades have not yet been applied or are not yet available. Assuming that patches and upgrades are supplied soon after flaws are announced and deployed immediately, attackers must be assumed in this scenario to be dedicated and have access to significant resources, including novel ("0-day") attacks. In particular, such attackers may be capable of subverting routers and inserting themselves between users and the authentication service.

The second case defends against a much less capable form of attacker. For various reasons, network administrators are occasionally required to allow remote access to legacy services over the Internet. Such services are frequently not securable by design, and may be proprietary. If privacy, integrity *and* authentication are required, then a system that supplies all three services is needed, like a VPN. However, if privacy and integrity are not required, then a simple authentication system would suffice, which would need to prevent attackers from masquerading as authorized users. Traffic interception or modification is not a threat in this case, since if attackers were capable of this, then they could attack the protected service directly by modifying authorized users' unprotected data streams.

We take the conservative approach and assume in our threat model that an adversary has the more powerful capabilities relevant to the first model. In summary, we assume that an adversary can:

- monitor and intercept all network traffic
- send packets with arbitrary source addresses
- replay captured network traffic

Further, we are only interested in attacks that could cause the authentication system to incorrectly grant access to an attacker at an arbitrary address.

2.1. Defenses required to address threats

In order to defend against the most dangerous type of attacker, while remaining practical, an authentication service should meet the following design goals:

1. strong authentication
2. resistance to traffic interception and modification
3. interoperability with existing network protocols and devices
4. low demands on network and processor resources
5. simplicity

In order to prevent an attacker from trivially bypassing authentication by guessing or replaying a captured authentication token, authentication information must be transmitted in a cryptographically secure manner. Depending on the application, protection against traffic interception and modification may also be required. For reasons of practicality, the authentication exchange should not require large amounts of network traffic or processor time; furthermore, the protocol used should not conflict with existing standards on IP, TCP, UDP, or other network protocols and, where possible, should not be affected by or interfere with existing firewalls, gateways, or proxies. Finally, in order to lower the likelihood of the authentication service itself being compromised due to implementation flaws, the service should be simple and easy to audit.

Additionally, in order to prevent attackers with novel attacks against the authentication service from targeting it, it may be advantageous to hide the existence of the authentication service from casual port scans.

3. Mechanisms for communicating with firewalls

There are three ways that messages could be sent from an authentication client to a server:

1. through an open port
2. through a closed port
3. through a side-band or covert channel

A conventional network service that communicates over open TCP or UDP ports is obviously the easiest to build. It could be implemented quite easily using SSH or SSL port forwarding, or with IPSec. Alternately, users could log into the server using SSH and run a script to open the firewall to their machines. However, each of these methods have some disadvantages. The services involved are highly complex and difficult to audit, thus replacing the threat of novel attacks against some other service with the threat of novel

attacks against the authentication service. The added computational overhead of the encryption provided by SSH, SSL, or IPSec tunneling may not be required and may be unacceptable under some circumstances. Also, the solution using scripts launched over SSH connections requires that users have login accounts on the firewall, which may not be feasible. The alternative is to build a simple, dedicated authentication service.

A second communication mechanism is to build an authentication server that listens on a closed port. By using firewalling or packet logging software to redirect all traffic sent to some port to the server, information may be passed across closed network ports. If the transport protocol is stateless and requires no connection-establishing handshake phase (such as UDP), then packet payloads may be delivered through this mechanism. If the authentication mechanism requires the server to respond (to issue a challenge, for instance), then this may be sent from a port that is closed immediately after sending and does not receive any incoming messages. Such a system has the advantage of being invisible to port scans, although not to traffic analysis. Obviously, if the authentication system requires a response, then it should only respond to valid requests.

A further option is to use a covert channel as a communication mechanism [1]. Without an existing connection between client and server, authentication data cannot be embedded in existing traffic. However, data can be embedded in protocol headers on packets sent in what could be made to look like a port scan. A technique which is in current use is *port knocking* [2, 3], where data is encoded in TCP or UDP port numbers. Other variations, including one that encodes data in TCP option fields [4], have been tried, but using port numbers has a number of advantages in that port numbers may be set by unprivileged user-space applications, while TCP options usually cannot, and port numbers are always logged by firewalling software, while option fields are not always recorded. Thus, a port knocking client can be a simple unprivileged user application, and servers need no special packet capturing capabilities beyond basic logging support in the firewalling software. Port knocking servers have the advantage of being invisible to port scans, and port knocking exchanges appear to be port scans to casual traffic analysis (although more detailed analysis can detect that connections are opened immediately afterward). Port knocking presents a number of unique issues, which will be discussed in detail in the following sections.

4. Port knocking

Port knocking [2, 3] (also known as "Spread-Spectrum TCP" [4]) is a technique whereby authentication information is transmitted across closed network ports. A machine using port knocking closes all network ports to all hosts but logs incoming packets. A program watches the firewall logs for certain sequences of packets, which encode authentication information and requests to open or close ports. Based on this information, the port knocking system can choose to open network ports to the originating host.

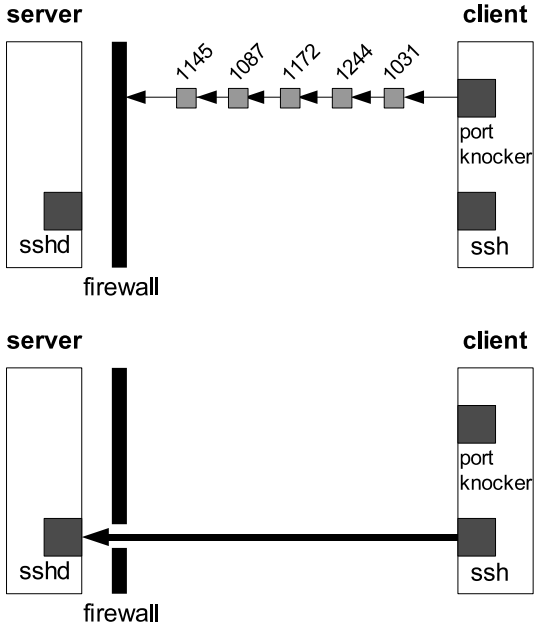

Figure 1. Port knocking example. The firewall is opened in response to a specific port sequence used for authentication.

As a simple example of port knocking, a server would close all ports and log requests to a specific port range; either TCP or UDP ports can be used. If a client transmits packets to a specific sequence of server ports (for instance, 1145, 1087, 1172, 1244, and 1031, in that order), then the server would perform some action (such as opening the SSH port to the client host). Here, the port sequence is a shared secret between the user and the server; knowledge of the secret implies that the user is authorized to access the protected service.

This particular application of port knocking is insecure, because an attacker could sniff the secret sequence from the network and replay it to get access to the protected service. However, other, more complex authentication procedures using port knocking exist, which we will discuss in the following sections.

Based on how authentication is conducted, existing port knocking applications fit into three categories:

- those that transmit a plain-text authentication token

- those that transmit a cryptographic proof of knowledge of an authentication token
- those that transmit a one-time authentication token

Examples of such systems include Krzywinski's *portknocking* system [2] (which can be configured to use either plain-text or encrypted authentication tokens), Doyle's *knockd/knockd* [5] (which sends an encrypted token), and *Spread-spectrum TCP*, by Barham et al. [4] (which uses a one-time token stream).[1]

All three types authenticate based on the knowledge of some sort of secret key and assume that key exchange is conducted by some out-of-band mechanism. Accordingly, key exchange is beyond the scope of this paper.

Port knocking has often been accused of being a form of "security through obscurity". While this is true in some cases (such as the simple example above), a well-implemented port knocking system using strong authentication is a secure system. The existence of the service and the data transmitted to it are obscured in order to raise the level of effort needed for a successful attack, but the security of the system does not rely on either of these properties.

5. Problems with existing systems

Systems that use plain-text authentication fail the requirement for strong authentication, as captured tokens can be trivially replayed. Port knocking systems using either cryptographic or one-time tokens could provide sufficiently strong authentication, provided that they were implemented properly.

In our survey of existing port knocking authentication systems, we found the following three major problems present in most or all of them:

1. **Out-of-order packet delivery.** Port-knock sequences typically contain 64 to 160 bits, and are usually sent at 8 bits per packet. Proper decoding of port-knock sequences by most servers is dependent on the order of arrival. According to Bennett et al. [6], on certain busy Internet backbone routers, the probability of out-of-order delivery of at least one packet out of a burst of 20 can be greater than 90%.[2] Of the port knocking systems we examined, only Barham's [4] made any attempt to deal with this problem.

2. **Network Address Translators (NATs).** If traffic from a client passes through a NAT [8] *en route* 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. No systems that we found fully addressed this issue.

3. **Lack of association between authentication and connection.** In no system that we found is there a logical association between the authentication sequence and the connection that is created after a port is opened. 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.

In addition, many existing port knocking applications suffer from serious flaws in how cryptography is applied to provide authentication. For instance, in Doyle's system [5], the IP address of the client, the requested TCP port, and an open/close flag are concatenated and padded into a 64-bit message, which is encrypted using the Blowfish algorithm and a shared secret key, and encoded into port numbers. The server decrypts the message and opens or closes the requested port to the client host, depending on the flag value. The claim is that this construction prevents successful replay attacks by preventing attackers from knowing what action was taken; however, an attacker capable of intercepting the authentication sequence would also be able to infer the contents of the message by analyzing network traffic for connections opened immediately after an authentication exchange, or closed immediately before. Thus, the attacker could authenticate and gain access by assuming the valid client's IP address and replaying the authentication sequence. Another system, *Cryptknock* [9], uses an unauthenticated Diffie-Hellman exchange to generate a session key, which is then used to encrypt a shared secret. The standard man-in-the-middle attack against Diffie-Hellman would allow an attacker to recover the shared secret and authenticate at will. Krzywinski's system [2] allows a wide variety of possible configurations, most of them insecure, due to plain-text password transmission, inappropriate application of encryption (in the same manner as Doyle), and insecure checksum algorithms.

Also, many existing systems (especially ones employing encryption) make no attempt to determine if messages that they receive are authentic. For instance,

[1] A list of other port knocking implementations is available at http://www.portknocking.org/view/implementations.

[2] Paxson's results [7] on older data indicated a much lower probability of out-of-order delivery, but still suggest that out-of-order delivery will occur often enough to be an issue.

when Doyle's system receives a message, it will decrypt it and execute its contents. No attempt is made to ensure that the message was valid in the first place, so a denial-of-service attack could be launched by sending random data to it, with the result of polluting firewall tables with random rules. In most configurations, Krzywinski's system suffers from the same problems.

In some cases, the authentication token sent is not bound to the client or the server. This could permit a variety of masquerading, reflection, or man-in-the-middle attacks under certain circumstances.

Another common flaw is the use of global shared secrets for authentication, known by all users of a port knocking server. This leads to administrative difficulty in changing keys. In conjunction with systems that allow clients to request the opening of arbitrary ports, a key compromise would allow an attacker to open any port on the system.

Some systems, such as Barham's [4], use time-based pseudorandom key streams in which keys must be valid during a time window corresponding to the maximum acceptable clock drift and the maximum transmission time between client and server. This allows a window of opportunity for attackers to replay captured authentication tokens. A window size of a few tenths of seconds should be adequate for good security, but this is only practical on low-delay networks with time synchronization mechanisms in place. Between distant hosts on the Internet under different administrative domains, transmission delay alone may be up to half a second, while poor deployment of time synchronization systems makes clock drifts of several minutes common. This leaves ample time for attackers to replay captured authentication sequences.

Finally, all but one of the port knocking systems that we examined will fail if any packets are lost, Barham's system being the exception. In many cases, neither client nor server is capable of detecting packet loss.

6. A new port knocking design

Taking into consideration the strengths and weaknesses of existing port knocking designs (discussed in Section 5), we have designed our own port knocking system using a cryptographically-secure challenge-response authentication system that accounts for out-of-order packet delivery and partially addresses the complications caused by NATs. An implementation is under development.

6.1. Operation

Two common weakness in existing port knocking systems are the use of global or per-port authentication keys and the transmission of un-verified commands in the authentication exchange. In order to avoid this, our system allows passwords to be associated with actions and unique operation IDs in the server's configuration. Each action will cause a particular port to open, a port to open for a fixed period of time, or a port to close. This will allow operators to configure detailed rules that limit the possible actions taken. By setting passwords appropriately on these rules, global, per-user, or per-port passwords can be created.

During the authentication exchange, clients transmit an operation ID, which will be used by the server to identify the password to verify against and the action to be taken upon successful authentication.

6.2. Authentication protocol

As indicated above, the choices for a strong authentication mechanism are limited to challenge-response systems and one-time password (OTP) systems. We chose to use a challenge-response authentication system, rather than OTP, for the following reasons:

- While time-based OTP authentication is practical on low-delay networks with time synchronization mechanisms in place, it is not sufficiently reliable for use over the variable delays and unsynchronized clocks of the open Internet.

- OTP schemes based on sequential password updates [10] require a secure channel to transmit the next password upon successful authentication. Port knocking provides authentication only; no such channel necessarily exists. While such a channel could be created, it would be an unnecessary complication and would compromise the design requirement for simplicity.

- Other OTP schemes, using fixed password lists or streams derived using one-time functions [10], require that an index into the password sequence be stored. If the client and server keep track of this index independently, then software crashes could result in the client and server not agreeing on the index, preventing further successful authentication. Such systems would also require that password streams not be shared between users, unless the users have some out-of-band mechanism for sharing indices. If the server prompts the user with a particular index (such as with S/Key [11]), then we essentially have a challenge-response system that protects against keystroke sniffing. While this would work, we have chosen to use a simpler challenge-response system that does not use one-time passwords.

A challenge-response system necessarily requires that the server return information to the client before the exchange is complete, thus potentially exposing its existence to attackers and compromising the design consideration for stealth. However, the loss of stealth can be minimized if the client sends a fairly long request sequence to trigger the issuance of a challenge, and if unrecognized request sequences go unanswered by the server. If the length of this request sequence is approximately equal to the length of a plain-text or one-time authentication token,[3] then an observer will notice the challenge being issued after a similar volume of communication as it would observe the server responding due to a connection being opened if a plain-text or one-time system was in use. Obviously, an attacker who intercepts a request sequence could replay it to receive a challenge, but an attacker who observed the request sequence already knows of the existence of the port knocking service anyway. And as long as the authentication mechanism is secure, there is no disadvantage in sending challenges to attackers.

6.2.1. Basic unilateral authentication. A slight modification to the ISO two-pass unilateral authentication [10, 12] exchange between a client A and a server B is shown in Algorithm 1. In the following discussion, we refer to message 1 as the *request*, message 2 as the *challenge*, and message 3 as the *response*.

Algorithm 1 Two-pass unilateral authentication

1: $A \rightarrow B$: req
2: $B \rightarrow A$: N_A
3: $A \rightarrow B$: $MAC_{K_{req}}(N_A, ID_A, ID_B)$

where A is the client
B is the server
req is a request for authentication
N_A is a nonce sent to A
K_{req} is a secret key shared by A and B
ID_X is the identity of a host X
MAC is a cryptographic message authentication function
, (a comma) represents concatenation

A begins the sequence by sending a request, which is a public string that will serve as the operation ID (see Section 6.1). B responds to a recognized request by issuing a unique nonce as a challenge, and A responds with a MAC covering the nonce and the IP addresses of A and B. Upon receipt of the response, B re-computes the MAC using the the nonce, A's IP address as taken from the packet headers, and B's own IP address. If the MAC is valid, then B will perform the action associated with the operation ID.

We have added A's identity to the response in the original ISO algorithm in order to prevent a man-in-the-middle attack related to the *grandmaster postal-chess problem* [10] in which an attacker C initiates the protocol and receives a challenge, intercepts (and blocks) a challenge issued to A in another run of the protocol and forwards its own challenge to A in order to get A to generate a valid response for C. By covering both IDs with the MAC in the third message, Algorithm 1 prevents C from subverting the protocol to authenticate to B as itself, but does not prevent C from subverting the protocol by masquerading as A. However, this attack has the end result of a port being opened to A's address, which is what A was originally trying to do in the first place. To actually accomplish anything, C would then have to communicate with the newly opened port while masquerading as A. This attack then becomes equivalent to hijacking a successful authentication; see Section 7.1 for possible defenses.

6.2.2. Protocol considerations for NATs. One flaw in Algorithm 1 is that it requires the client, A, to know its identity as seen by the server, B. Unfortunately, if the client is behind a NAT, then it may not know its public address, and it may not even know that the NAT exists. If A's address is re-written, then A will use its private address ID_A to compute the challenge, but B will use A's public address PID_A to verify it, and authentication will fail. As a partial solution to this problem, we propose Algorithm 2, a NAT-aware variation on Algorithm 1.

Algorithm 2 NAT-aware unilateral authentication

1: $A \rightarrow B$: req, ID_A
2: $B \rightarrow A$: $PID_A, MAC_{K_{req}}(PID_A, ID_A), N_A$
3: $A \rightarrow B$: $MAC_{K_{req}}(N_A, PID_A, PID_B)$

where A is the client
B is the server
req is a request for authentication
ID_X is the private IP address of host X
PID_X is the public IP address of host X
N_A is a nonce sent to A
K_{req} is a secret key shared by A and B
MAC is a cryptographic message authentication function
, (a comma) represents concatenation

In order to avoid needing to use third-party identity oracles, B tells A its public identity in the challenge. However, if this information is not somehow associated

[3] We doubt that any practical benefit would be obtained from using request sequences longer than 5 to 10 bytes.

with something known to A (such as A's private identity), an attacker C could block the challenge and substitute its own. A would then generate a response that is invalid for itself, but valid for C. So, we add a MAC to the challenge covering A's public and private identities. This requires that B know A's private identity, so it will be sent along with the request.

This algorithm should also work even if A is not NATed; this is important in the case that A does not know if it is NATed or not.

6.3. Communication protocol

Since we are using port knocking as a data transport mechanism, all messages sent from the client to server (i.e., the request and response) must be encoded as sequences of TCP or UDP port numbers. We chose UDP over TCP because, UDP being stateless, it is less likely to impose processing overhead on any gateways or packet filters present between client and server. Also, UDP headers are smaller than TCP headers, so the null-payload packets used are significantly smaller. Information is encoded only into destination port numbers, because this is the only field in the UDP/IP header that is not required for network-layer delivery, is not likely to be modified by gateways or proxies, and can be specified by unprivileged user programs. This limits the amount of data that can be sent per packet to 16 bits, assuming that all 65536 UDP ports are monitored for port knocking traffic. In order to avoid monitoring ports that are already in use (whether opened by port knocking or not) and ports used by outgoing communication, it may be necessary to limit the port range being used.

Messages sent from the server to client need not follow these restrictions. Therefore, we will send the challenge as a single UDP packet, directed to the source port used in the knock sequence.

6.3.1. Considerations for packet re-ordering. As described in Section 5, if packets are delivered out of order by the network layer, the message received will be undecipherable, and authentication will fail. We have devised two methods to overcome this problem.

The simplest way is to split each port number into data and sequence number fields. Upon arrival, the packets can be sorted by sequence number before the data fields are decoded. Note that this limits the amount of data that can be carried per packet, and that the sequence number field must be wide enough to uniquely identify each packet.

We present an analysis comparing the minimum protocol execution time under the optimal number of data and sequence number bits per packet to the size of the port range required in Table 1. For the purposes of this analysis, we assume that request sequences are 10 bytes long, challenges and MACs are 20 bytes, and Algorithm 2 is used for authentication, thus making the request message 112 bits long, the challenge 352 bits, and the response 160 bits. We further assume that network bandwidth is 64Kbps with a round-trip time (RTT) of 100ms, and disregard all processing time.

Table 1 Port knocking with sequence numbers: size of port range vs. minimum protocol execution time

Data bits per packet	Sequence number bits per packet	Size of port range	Execution time (seconds)
12	4	65536	0.243
10	5	32768	0.257
9	5	16384	0.268
8	5	8192	0.278
7	5	4096	0.300
6	5	2048	0.320
5	5	1024	0.352
3	6	512	0.481

This analysis shows that even under the fairly pessimistic network conditions that we assumed, and the rather inefficient transport mechanism of port knocking for the request and response messages, authentication is fairly fast. In all but the most inefficient configurations, execution time is dominated by the RTT. There is little speed to be gained by encoding more than 8 bits of data per packet (over a range of 8192 ports); allocating a contiguous range of 8192 ports should not be difficult on most systems. A slight further speed improvement could be obtained by using different numbering schemes for the request and response messages. Note that it is not possible to send 160 bits of data to a range of fewer than 512 ports with each packet carrying a distinct sequence number. Also, this encoding limits the maximum request sequence length to about 16 bytes.

A second way of encoding sequencing information into a port knocking sequence is to encode data as the differences between a sequence of monotonically increasing port numbers. For example, while sending 8 bits per packet to a range starting at port 1024, the sequence "121, 63, 148, 220, 7" could be encoded as "1145, 1208, 1356, 1576, 1583". An analysis of this technique is presented in Table 2, under the the same assumptions as above. This technique makes it impossible to accurately detect packet loss, but a similar technique allowing loss detection would be to send each packet containing b bits of data to a unique range of 2^b ports, where the range for packet n begins above the top of the range for packet $n-1$. Note that the port range and time

requirements of both of these techniques are identical.

Table 2 Port knocking with monotonically-increasing port numbers: size of port range vs. minimum protocol execution time

Data bits per packet	Size of port range	Execution time (seconds)
12	57344	0.243
11	30720	0.250
10	16384	0.257
9	9216	0.268
8	5120	0.278
7	2944	0.300
6	1728	0.320
5	1024	0.352
4	640	0.397
3	432	0.481
2	320	0.635
1	320	1.111

This technique makes complete use of its "sequence numbers", so it can encode the same amount of information into smaller port ranges. Once again, there is little to be gained by encoding more than 8 bits of data per packet, but this time, only 5120 ports are required – this method encodes the same amount of information over a smaller port range. Note that the 16-bit port space provided by UDP does not allow sufficient ports to be allocated to send more than 12 data bits per packet.

An alternative would be to eschew sequence numbers altogether and enforce a brief delay between sending packets; this would significantly reduce the chance of out of order delivery. Unfortunately, as shown in Table 3, the delay must be very short before this method begins to out-perform the two presented above, and the probability of failure increases significantly as the delay decreases. Also, it would not be possible to detect packet loss while using this method.

6.4. Weaknesses of our design

Despite the improvements that we have made over other port knocking designs, our system still has its flaws.

Our system does not address the lack of a logical connection between an authentication sequence and any connections that open as a result. The architecture of current Internet protocols, firewalls, and application software make a complete solution difficult. Some partial solutions are described in Section 7.1.

While our system will allow NATed clients to authenticate, successful authentication will cause the target firewall to open the requested port to all other hosts

Table 3 Port knocking with enforced delays between packets: size of port range vs. minimum protocol execution time

Data bits per packet	Size of port range	Execution time (delay = 5 ms)	Execution time (delay = 2 ms)
15	32768	0.311	0.260
14	16384	0.319	0.265
13	8192	0.336	0.276
12	4096	0.353	0.287
11	2048	0.370	0.298
10	1024	0.387	0.309
9	512	0.413	0.323
8	256	0.438	0.342

on the same NATed network having the same public IP address as the client. In most cases, this should not be a serious exposure. It may not be possible to completely solve this problem without installing protocol interpreters on the NATs.

The authentication mechanism that we use is unilateral; no attempt is made to authenticate a server to a client. If an attacker is able to masquerade as a server to a client, then the client has no way of knowing that the server to which it is connecting is not the intended server. Similarly, an attacker could gain access to a server through an interleaving attack [10], if the attacker is capable of masquerading as a valid client to a server and as a valid server to a client. Port knocking systems in general leave server authentication up to higher-level protocols, but in the case of such an interleaving attack, this is already too late. A potential solution to this problem is presented in Section 7.2.

Many potential client machines are located behind firewalls that permit outbound traffic to well-known ports, such as HTTP, but would not pass outbound UDP traffic destined to seemingly random ports. In such an environment, authentication would fail. Workarounds to this problem are presented in Section 7.3.

While the techniques presented in Section 6.3.1 allow the system to work if packets are delivered out of order, the system will fail if any packets are lost. If a packet in the request sequence is lost, then the challenge will never be issued; a client should eventually realize this and re-transmit the request. The same should hold if the challenge is lost. If a packet of the response sequence is lost, then authentication will not complete. Since the server does not notify clients of successful authentication, the error will only be noticed when the client attempts to open a connection. The only way to recover at this point is to start over.

7. Further extensions

There are a number of ways by which our system could be extended, including associating knock sequences with connections, mutual authentication, knocking at different protocol layers, virtual port knocking servers, and public key authentication.

7.1. Authentication-connection association

Under some circumstances, it may be possible to solve the problem of the lack of a logical relation between authentication and the subsequent connection. For TCP connections, for instance, a port knocking client opening a connection to a server through a knocked-open port could initially exchange a secondary authentication with a small "wrapper" server, part of the port knocking system. A successful secondary exchange would result in the client handing the open TCP connection off to the real client program; on the server side, the wrapper would hand off or tunnel the TCP connection to the real server. Such a mechanism could be used in conjunction with client and server software that support passing off connections, or with client- and server-side port knocking proxies.

Instead of opening the port associated with a request sequence, the server could open a random port and forward everything sent to it to the target port. The random port number could be encrypted with AES using the shared key and appended to the challenge message. If the server closed the random port as soon as the first connection to it was received, then the port would already be closed by the time an attacker has determined which was opened. This would work for both TCP and UDP and requires that client applications can be configured to connect to arbitrary ports.[4]

Another alternative for TCP connections would be to negotiate the initial sequence number (ISN) to be used in the subsequent connection during the authentication phase. As long as this value is hidden from any attackers, it could be used to verify the TCP connection. This solution would require either that the authentication mechanism be built into operating system TCP/IP stacks or that operating systems be given support for user-space ISN generation.

7.2. Mutual authentication

As noted in Section 6.4, Algorithm 2 is still vulnerable to attacks in which an attacker masquerades as a server. One solution is to add a nonce to the client's response (step 3), and add another step to the exchange to authenticate the server to the client, in a exchange similar to ISO three-pass mutual authentication [12]. Algorithm 3 is a variation on Algorithm 2 that supports mutual authentication.

Algorithm 3 NAT-aware mutual authentication

1: $A \rightarrow B$: req, N_B
2: $B \rightarrow A$: $PID_A, MAC_{K_{req}}(N_B, PID_B, PID_A), N_A$
3: $A \rightarrow B$: $MAC_{K_{req}}(N_A, PID_A, PID_B)$

where A is the client
B is the server
req is a request for authentication
ID_X is the private IP address of host X
PID_X is the public IP address of host X
N_X is a nonce sent to host X
K_{req} is a secret key shared by A and B
MAC is a cryptographic message authentication function
, (a comma) represents concatenation

Unfortunately, this algorithm would require a client to transmit its own nonce of about 20 bytes by port knocking as part of the request message (although it no longer needs to send its private identity). The analysis we performed in Section 6.3.1 will no longer be valid for this protocol; significantly increased time requirements and port ranges will be required to transmit the extra data. One way to shrink the port range required by this algorithm at a cost of a low chance of failure would be to assume that all packets sent with sequence numbers congruent modulo n would arrive in order and reset the packet sequence number or port range every n packets. For $n = 20$, the port ranges required by such a system are the same as those presented in Section 6.3.1, albeit with a chance of failure if re-ordering across an n-packet boundary occurs.[5]

7.3. Generalized knocking

The concept of port knocking does not necessarily have to be applied at the network/transport layer; it can also be applied to create covert channels at the application layer. In such a system, a client would sent knock messages to the same well-known port, where a listening server would reconstruct the authentication messages. For example, information could be encoded as DNS[6] or HTTP requests. To send the sequence "123, 456, 78", a client could look up "123.somedomain.zz", "456.somedomain.zz", then "78.somedomain.zz", or

[4] Thanks to an anonymous reviewer for suggesting this method.

[5] The exact probability of failure using this method is beyond the scope of this paper.

[6] At least one existing port knocking system, CÖK [13], uses DNS requests as a transport.

request “123.html”, “456.html”, then “78.html”.

Such a system would work even if an egress firewall prevents a client from sending UDP port knocking sequences, as described in Section 6.4. Optionally, the next connection made after a successful authentication could be tunneled to the desired service; this would allow connections to arbitrary ports to be made past egress filters that would block a direct connection.

Obviously, there is a great deal more overhead involved in such a system, but it does demonstrate that knocking is feasible even if the client's communication ability is constrained.

7.4. Other extensions

A more surreptitious knocking scheme would involve multiple computers acting as a “virtual port knocking server.” A knock sequence need not be sent to a single computer but could be split and sent to multiple computers, which would coordinate the information (perhaps by centralizing individual computers' firewall logs) to authenticate the client. The port opened by a successful knock sequence need not even be on the same machine(s) as where the knock packets were sent, further confounding an attacker.

It should be possible to alter the authentication protocol to use digital signatures and public-key certificates rather than a private-key MAC. This would not affect the overall security of the protocol, but could make key management much simpler.

8. Conclusion

Port knocking can be used to construct authentication systems on firewalls with the goal of only allowing authorized users access to open ports.

Port knocking systems may be dismissed as simply “security through obscurity”. While we disagree with this – port knocking can be done securely – the fact of the matter is that such systems are in active use, and it is thus advisable to make them as strong as possible. They can be seen as complementary to existing defensive systems, as part of defense in depth, or as a means to secure weak legacy systems.

Existing port knocking systems have three main flaw: they do not always work reliably in the presence of NATs, they fail if packets are delivered out of order, and they do not associate authentication exchanges with connections opened afterwards. The port knocking system that we have presented here improves on current systems by using a novel authentication algorithm that is unaffected by NAT and an efficient packet-reordering system that ensures that messages can be decoded on delivery.

References

[1] B. W. Lampson, “A note on the confinement problem,” vol. 16, no. 10, pp. 613–615, 1973.

[2] M. Krzywinski, “portknocking.org,” URL: http://www.portknocking.org, Nov. 2004, accessed Nov 29th, 2004.

[3] ——, “Port Knocking,” URL: http://www.linuxjournal.com/article/6811, June 2003, accessed Nov 12th, 2004.

[4] P. Barham, S. Hand, R. Isaacs, P. Jardetzky, R. Mortier, and T. Roscoe, “Techniques for Lightweight Concealment and Authentication in IP Networks,” Intel Research, Tech. Rep. IRB-TR-02-009, July 2002.

[5] M. Doyle, “Implementing a Port Knocking System in C,” Physics Honors thesis, University of Arkansas, 2004.

[6] J. C. R. Bennett, C. Partridge, and N. Shectman, “Packet reordering is not pathological network behavior,” *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, pp. 789–798, Dec. 1999.

[7] V. Paxson, “End-to-end Internet packet dynamics,” *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, pp. 277–292, June 1999.

[8] P. Srisuresh and K. Egevang, “Traditional IP Network Address Translator (Traditional NAT),” Internet Engineering Task Force, Network Working Group, Tech. Rep. RFC 3022, Jan. 2001.

[9] J. Walko, “Cryptknock,” URL:http://cryptknock.sourceforge.net/, June 2004, accessed May 25th, 2005.

[10] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, Oct. 1996.

[11] N. M. Haller, “The S/KEY one-time password system,” in *Proceedings of the Symposium on Network and Distributed System Security*, 1994, pp. 151–157.

[12] ISO/IEC, “Information technology – Security techniques – Entity authentication – Part 4: Mechanisms using a cryptographic check function,” International Organization for Standardization, Geneva (Switzerland), Tech. Rep. ISO/IEC 9798-4, 1995.

[13] D. Worth, “CÖK: Cryptographic One-Time Knocking,” Talk slides, Black Hat USA, 2004.