EXTENSION OF A PORT KNOCKING CLIENT-SERVER ARCHITECTURE WITH NTP SYNCHRONIZATION

Traian POPEEEA[1], Vladimir OLTEANU[2]

Port knocking is a form of host-to-host communication which relies on deliberately failed TCP connection attempts. The information is encoded into a port sequence. The client attempts to initiate several three-way-handshakes and receives no reply. These connection attempts are monitored by a daemon which interprets their destination port numbers as data. This mechanism has vulnerabilities that can be exploited by hackers with the help of data sniffed off the network. Through synchronization, these vulnerabilities can be minimized. We present the enhanced security that can be offered by synchronization, the means to achieve this and describe an application that implements this.

Key words: network security, cryptography, port knocking, one-way functions

1. Introduction

Information security has been a concern for many years, even before computer networks. One of the first people said to have used cryptography is Julius Caesar who applied a circular shift to each letter, moving it three places further in the alphabet. Today, with the growth of information systems and of the Internet, increasingly sophisticated methods of protecting information are required, thus raising the issue of keeping our data protected from unauthorized access while keeping it accessible online to us and our trusted peers.

The first line of defense against any kind of attacks can be a conservatively configured firewall and all network applications updated. Also the use of Intrusion Prevention/Detection Systems (IPS/IDS) can help, as these are designed to spot known attacks and use many heuristics to detect anticipated attacks. However, packet-filtering firewalls filter on the basis of IP address, protocols, ports, or flags, thus leaving applications behind open ports vulnerable to attacks.

Port knocking is a firewall-based user authentication system that uses closed ports for authentication. Communication across closed ports is possible through the firewall log, which records all connection attempts. The communication initiator is considered the client, while the host using this security mechanism is considered the client. Information is encoded, and possibly encrypted, by the client into a sequence of port numbers. This sequence is termed the knock. Initially, the server presents no open ports to the public and is monitoring all connection attempts. The client initiates connection attempts to the server by sending SYN packets to the ports specified in the knock. This process of knocking is what gives port knocking its name. The server offers no response to the client during the knocking

[1]Faculty of Automatic Control and Computers, University "Politehnica" of Bucharest, Romania, e-mail: traian.popeea@gmail.com

[2] Faculty of Automatic Control and Computers, University "Politehnica" of Bucharest, Romania, e-mail: vl.olteanu@gmail.com

phase, as it "silently" processes the port sequence. When the server decodes a valid knock it triggers a server-side process. However, there are many attacks to which this security mechanism is vulnerable, such as brute force, eavesdropping and replay or man in the middle attacks. [1]

Using synchronization and cryptography to generate unique knock sequences with a limited life span, based on the client's IP address and the current date and time can limit some of these attacks.

The remainder of this paper is organized as follows. The paper presents a critique of port knocking in Section 2, followed by the architecture, components, functionality of the application in Section 3. Section 4 presents the technologies used to implement the project, and details about the actual implementation. Section 5 presents the test scenarios and results for the functionality tests, while Section 6 contains related work. Finally, Section 7 presents concluding remarks and outlines a path forward.

2. Port knocking critique

2.1 Benefits of port knocking

The main feature of port knocking is that it allows for stealthy authentication into a host without open ports. The method is stealthy because it is not possible to determine if the host is listening for knocks. Since information is flowing as connection attempts, rather than packet data payload, it is unlikely that this method would be easily detected. The system is flexible, because existing applications such as ssh, which perform their own authentication, do not need to be changed, as port knocking is an outer layer of security for the machine. [2]

2.2 Disadvantages of port knocking

As with any security system, the disadvantaged begin with the small inconveniences that must be endured while that system is in use. The port knocking client requires a script to perform the knock. This client and any associated data should be considered a secret and kept on removable media. The use of the client imposes an overhead for each connection and users require instruction. Port knocking cannot be used to protect public services such as mail or web, as this would require everyone to know the knock. Thus, the public services should be relocated to a demilitarized zone (DMZ) and isolated from the hosts with sensitive information.

The implementation of any system that manipulates firewall rules in an automated fashion must be robust to prevent legitimate users from being locked out. Also, if the service daemon crashes, the host may become isolated. Appropriate measures should be implemented to avoid such a scenario. [3]

Also, there are certain practicality issues when using port knocking outside of the lab, in the real world. Port knocking mechanisms are more useful when used to access remote machines rather than machines in the same local network. The first significant issue

which applies to port knocking is the issue of Network Address Translation (NAT) and its effects on the authorization granted. NAT is used to share a single or few public IP addresses between a private network of machines. Thus, each computer on the local network appears with the same IP on the Internet. Due to the fact that the port knocking daemon must open the requested port for a specific IP, this raises two sub-problems. Firstly, the knocking client must know its public IP address for our solution to work. However, even once the issue of public IP discovery is put aside, we are left with the issue of who is actually authorized at the end of the port knocking process. A completed knock on the server will open a hole in the firewall for your public IP, allowing access for all machines in the same private network. [4]

Also, if we consider the subsequent connections following a successful authentication, it is important to point out that there is no formal association between the client who knocked and the client who is actually attempting to connect to the opened port. So a successfully opened port can then be hijacked by an attacker with the ability to impersonate the client. [5]

Out-of-order delivery is another issue because the Internet is a vast network of varying latencies and some routers may implement load balancing. Although basic port knocking may work flawlessly in local network tests, the delays inherent in network communications may be a problem when attempting to use the mechanism in practice. Port knocking requires that the knock sequence arrive in the correct order for the sequence to decode properly. If a single knock arrives out of order, the sequence will be broken and the server will not perform any action, ultimately resulting in a denial of service. Similarly, should an unprivileged attacker aim to perform a denial of service attack on a particular client, he can simply send repeated packets to random ports on the server, spoofing the client's IP address. The server will be unable to differentiate between the attacker's and the client's packets and so the sequence will be broken and authentication will fail.

Our port knocking extension does not intend to cancel all of these limitations but hopes to reduce their risk factor.

3. Architecture

The application developed to sustain this paper represents an extension to an existing implementation, created by M. Doyle [6]. This implementation is based on a client-server architecture, done on a *nix architecture. The original implementation was written in C and tested under Cygwin Linux and FreeBSD. The server begins by calculating the initial MD5 hash of the log file. MD5 is an iterative one-way hash function that takes an arbitrary length string and calculates a 128-bit hash value for that data. In other words, MD5 generates a unique digital "fingerprint" based on the data currently stored in the log file. Then it sets the log file pointer to the end of the file, which will tell it where to begin reading once log entries have been appended to the log file. By recalculating the MD5 hash at regular intervals, it is possible to detect when entries have been appended to the log file, and promptly search this new data for entries matching the regular expression. The server stores valid knock sequences in configuration files, the entries being statically loaded at

service startup, thus being vulnerable to attacks. The client prompts the user to manually insert the known port sequence.

Our extension is based on two main enhancements. First, the synchronization between the two entities involved in the knock process and second, the use of one-way functions to ensure a secret knock sequence.

In order to synchronize to server and client, we are using Network Time Protocol (NTP) and interaction with the operating system current time. At the knock daemon startup, a synchronization request to a NTP server is initiated, ensuring that the server is correctly set up. Also, the client will send a similar request for synchronization to a NTP server. After this step, both server and client should have the same system clock, with a minimal skew. Before the actual knock, the client will get the local system time, and so will the server.

The local system time, along with other parameters will be used to determine the knock sequence, as it shown in Figure 1.

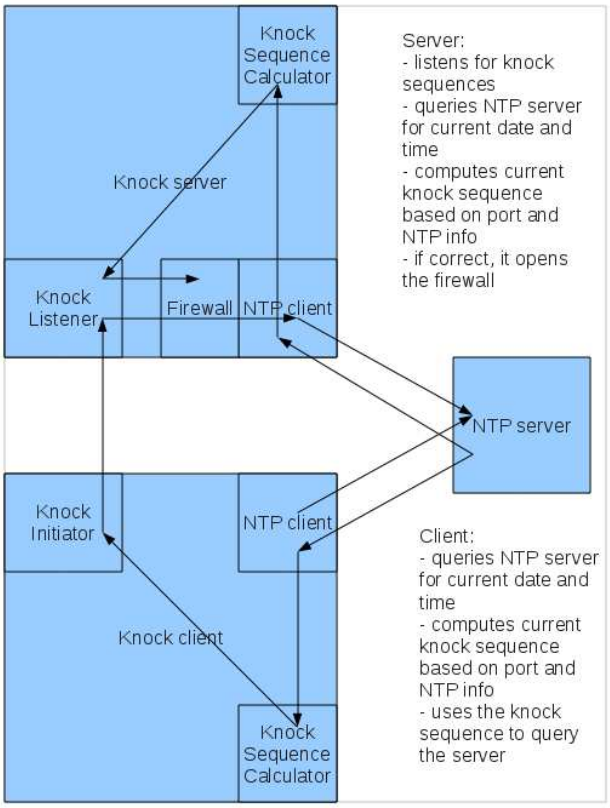

Fig. 1. Application architecture.

The knock sequence calculator module is based on one way functions. In a nutshell, one-way functions are functions that that easy to compute, but hard to invert. Given an one-way function F, an input x and an output y, $y = F(x)$ can be computed in polynomial time, but $x = F^{-1}(y)$ cannot.

A function's "one-way" status actually relies on the absence of any known algorithm that is able to compute the pre-image of an output in polynomial time, rather than the proof that no such algorithm exists. Whether or not a demonstrably one-way function exists is still an open question [7].

In our application, we use hash functions to generate knock sequences based on a pre-shared key (PSK).

A PSK contains the following fields:

- the one-way function's name;
- which parameters to ignore;
- time granularity expressed in seconds; all time values are truncated to the highest multiple of this value;
- the actual key; which is a string of randomly-generated characters.

Our one-way functions take the following parameters:

- the client's IP,
- the time,
- the key

These parameters are concatenated and a hash is computed. The resulting hash represents the knock sequence (the first 16 bits represent the first port, the next 16 bits represent the second one etc.).

The knock sequence will be valid only for a limited time. Let g be the time interval defined by the granularity and t be the current time. All knock sequences calculated by the same client in the interval between $t_0 = t - (t\%g)$ and $t_1 = t_0 + g$ are identical.

Let $d < g$ be the delay between the time the knock sequence is calculated and the time the first packet in the sequence reaches the server. Assuming perfect synchronization with the NTP server, the probability that the first packet is received after t_1 is $p = \frac{d}{g}$. To avoid rejecting these late arrivals, we have extended the validity interval to $[t_0, t_0 + 2g)$.

Once the client successfully knocks, the (IP, t_0) tuple used to generate the knock sequence becomes invalid. This can be superseded by setting the knock delay to a value greater than or equal to $2g$.

4. Implementation

Being an extension to an already existing project, we maintained C as the programming language and the underlying *nix architecture.

The functionality of the application requires, at this time, the presence of the NTP client application, existent in all *nix distributions, which we use to synchronize the server and the clients. This application is called from our program using the system() function. For the actual use of the time value, we use time() function, which offers a granularity of a second, which represents the minimum granularity we can use.

We use the libssl library, consisting of cryptographical functions to generate the hashes. MD5, SHA256 and SHA512 are currently supported. Both the server and the client use the same knock sequence calculator for consistency.

The client reads the PSK, generates the knock sequence for $t_1 = t - (t\%g)$, where t is the time reported by the NTP server, and immediately attempts to knock.

The server parses its configuration file at startup, queries the NTP server to synchronize the machine's system clock and subsequently reads the PSK. Upon receiving the first packet of a knock sequence, it calculates both valid knock sequences for the client's IP (one for $t_1 = t - (t\%g)$ and one for $t_0 = t_1 - g$, where t is the time the first packet was received). If either of the two sequences matches the client's sequence, the knock attempt is successful.

As stated earlier, the key is a string of randomly-generated characters. As such, keys are generated by reading data from /dev/random.

The logical flow through the application could be summarized like this:

- at server initialization, a key is generated
- the server obtains NTP time
- the client wants to initiate a sequence
- the client obtains NTP time
- the client computes the knock sequence based on the PSK
- the client sends TCP SYN packets
- when the server detects a knock sequence, it computes the keys for all ports, based on time and source IP address
- the server compares the incoming knock sequence to the ones computed by him
- if there is a match, a port is opened in the firewall

5. Experimental results

In order to test the capabilities of our application, we have used a lab environment on a single physical host, running several virtual machines. All the experiments were performed on a Intel Core 2 Duo E7600 system with 2GB RAM, and Windows XP SP3 as host operating system. The local network containing the server and clients was simulated with VMware virtual machines running Debian 6.0 testing with Linux kernel 2.6.32.

The tests were divided in two categories: tests used in order to determine the granularity of the knock sequence and functionality tests.

The granularity was computed as follows: $g = s_c + s_s + n*l$; where s_c and s_s represent the latency induced by the computation of the knock sequence by the client and server, respectively. n represents the number of knock packets contained by a sequence and l the network latency.

We have computed the knock sequence generation delay by computing 1 million keys consecutively and measuring the time, as you can see in Table 1. The results show that

the actual computational effort is negligible in determining the granularity, being less than 1ms per key.

Table 1

Key generation for 1 million keys

Key length One way function	512 bits	2048 bits
md5	1.163s	3.744s
sha256	3.765s	13.197s
sha512	2.759s	8.823s

The network latency was tested with intercontinental ping tests, resulting in round-trip-times less than 500ms, with an average of 100ms. Considering the average, for 512 bit-keys, representing a knock sequence of 32 ports, this would determine a granularity of 3.2 seconds. For 2048 bit-keys, the granularity, should be 12.8 seconds.

The functionality tests have been incremental. The first step was testing the internal correct implementation of the desired algorithm. For this, we have generated knock sequences at different time intervals, using different source IP addresses and with different initial keys. Using a controlled variable instead of the actual system time, we have generated a significant number of knock sequences, which were compared at random. The tested knock sequence pairs did not present any collisions.

The rest of the functionality tests included connecting a single client on a server on a single port, connecting a single client on a server on multiple ports consecutively, and connecting multiple clients on the same server.

6. Related work

Over the years, there have been developed several variants of client authentication and synchronization based on port knocking. Two of the most interesting variants are Single Packet Authorization (SPA) [8] and Cryptographic One-Time Knocking (COK) [9].

In Single Packet Authorization, the knock, which is called an Authorization Packet (AP), is encoded within a single packet. This can provide certain advantages not found in traditional port knocking schemes, such as eliminating the problem of out-of-order packet delivery. In traditional port knocking, if a knock sequence arrives out of order, which can easily happen as the server is not sending back any acknowledgements, then the port knocking daemon will not recognize the knock and thus access will not be allowed, as mentioned above. SPA simplifies the process by encoding all of the necessary information into a single packet, typically UDP or ICMP. The information encoded in these packets can be as simple as a timestamp (for replay protection), client IP address, and password combination. The resulting AP that is sent to the server is usually constructed by feeding the fields above into a hash function. This hash would then be packed into a UDP packet and

sent off to the server. Upon receiving the packet, the SPA daemon would recalculate the hash by hashing the password (which it knows), the current date and time (accurate in minutes), and the IP address of the client that knocked (which can be found in the IP header of the UDP packet). If the resulting hash is the same as the one received, then the required port would be opened for the client IP. If the hashes do not match, or the received hash had already been received previously (ie. The hash is a replay of an old hash), then no action is performed.

COK is an implementation that uses one-time-passwords (OTP) to generate a non-replayable knock sequence. The server stores a password, and a cryptographic hash function, used for the OTP. Initially, the knock sequence is based on the n-th iteration of the function. After the authentication, the knock sequence will be based on the (n-1)-th iteration, and so on. In this way, the system works backwards through the n passwords. The projects implements a SPA-based knock and a DNS knock, where the formed one-time-password knock is sent via a DNS lookup to a given listening host. (most likely to avoid detection if the listening host is also a DNS server)

7. Conclusions and future work

The synchronization and cryptography elements we have added to the existing application offer a security enhancement, limiting the number of attacks that can be performed on the server and the effects of those to which it is still vulnerable. The small life span of the knock sequence renders sniffing useless by itself, but combined with IP spoofing and replay attack, can become a denial of service attack on the client, as it would ruin his valid sequence. Also brute force attacks become almost useless, because finding one collision at one specific time interval does not imply finding the key as source IP and time are factored in. The use of hash function has proven to add a relatively small latency penalty, as the CPU-intensive part of the code is much smaller than the latency induced by the network.

We will continue developing this project and our short-term goals are integrating the NTP client in our code, in order to reduce the context switching penalty between our application and the NTP client, allowing also repeated NTP queries in order to limit the possible clock skew. Also, implementing and testing more one-way functions can provide a better balance between the length of the knock sequence (a factor in sequence granularity) and the “resistance” to brute-force attacks.

BIBLIOGRAPHY

[1] M. Krzywinski, Port Knocking: Network Authentication Across Closed Ports. SysAdmin 2003. Magazine **12**: 12-17

[2] P. Barham, S. Hand, R. Isaacs, P. Jardetzky, R. Mortier, T. Roscoe, Techniques for Lightweight Concealment and Authentication in IP Networks, Intel Research Berkeley, 2002

[3] S. Krivis, Port Knocking: Helpful or Harmful? – An Exploration of Modern Network Threats, GIAC Security Essentials Certification, 2004

[4] R. deGraaf, C. Aycock, M. Jacobson, Improved Port Knocking with Strong Authentication. ACSAC 2005, pp. 409-418. Available at: http://www.acsa-admin.org/2005/papers/156.pdf

[5] B. Maddock, Port Knocking: An Overview of Concepts, Issues and Implementations, GIAC Security Essentials Certification, 2004

[6] M. Doyle, Implementing a Port Knocking System In C, Honors Thesis, The University of Arkansas, 2004

[7] L. A. Levin, A Tale of One-way Functions, http://arxiv.org/abs/cs.CR/0012023

[8] S. Jeanquier, An Analysis of Port Knocking and Single Packet Authorization, MSc Thesis, Royal Holloway College, University of London, 2006

[9] D. Worth, COK: Cryptographic Port Knocking, Black Hat USA 2004