sKnock: Port-Knocking for Masses

Daniel Sel, Sree Harsha Totakura, Georg Carle
Chair of Network Architectures and Services
Technical University of Munich
Email: {sel, totakura, carle}@in.tum.de

***Abstract*—Port-knocking is the concept of hiding remote services behind a firewall which allows access to the services' listening ports only after the client has successfully authenticated to the firewall. This helps in preventing scanners from learning what services are currently available on a host and also serves as a defense against zero-day attacks. Existing port-knocking implementations are not scalable in service provider deployments due to their usage of shared secrets. In this paper we introduce an implementation of port-knocking based on x509 certificates aimed towards being highly scalable.**

***Index Terms*—Port-knocking, dynamic firewall, x509**

I. Introduction

PORT-KNOCKING is the concept of hiding remote services behind a firewall which drops all incoming connections to the services by default, but allows them only after the client has authenticated to the firewall. This can be seen as an additional layer of security which provides protection from port scans and exploits on the services from unauthorized clients.

There exists many implementations of port-knocking as of today and most of them employ authentication based on shared secrets. In these implementations the firewall is configured to authenticate a client based on its corresponding secret. While this approach is simple and efficient for a small number of clients, it quickly becomes unmanageable for a service provider with a large and dynamically changing client base.

Another problem with authentication based on secrets is that it is common for service providers to offer multiple servers to a common pool of clients for reasons of providing redundancy and load-balancing. In such a setup, the servers or the firewalls protecting the services have to synchronise the port-knocking secrets shared with each client. Furthermore, when a service provider uses a cloud provider for service delivery, the secrets have to be configured in the cloud provider's infrastructure which may give away the size of the client base of the service provider to the cloud provider.

To our knowledge there exists no implementation of port-knocking which demonstrates scalability on-par with what is required for a service provider using cloud computing infrastructure. To overcome the scalability barrier, we propose *sKnock* (named after 'scalable Knock'), our approach towards port-knocking using public-key cryptography to address scalability. Here, we choose to use the X509 [1] certificate standard due to its popularity. The motivation behind using certificates is that a certificate's validity can be checked by having the certificate of the signer; while adhering to x509 allows us to encode authentication information as x509 extensions and be able to use renowned libraries such as OpenSSL to parse the certificates. This, albeit a lookup in the revocation database which is usually small, aids in improving the scalability of authentication. Moreover, this approach is not subjected to the problem of synchronising the secrets among different servers and mitigates the privacy problem of knowing the size of client base as the firewall now only requires the certificate of the certification authority (CA) to authenticate the clients.

This is a pre-print version of the paper submitted to WMCSP 2016.

Our contributions to this paper are a one-way communication protocol to authenticate clients to the port-knocked firewall, an implementation of this protocol in a client and a server component. The client component provides a C library which allows applications to integrate port-knocking functionality. The server component integrates with the firewall and dynamically configures it to allow authenticated clients to communicate with the services behind the firewall.

The text in this paper is organised as following: the next section introduces an attacker model. We use it to evaluate some of the existing port-knocking approaches and their authentication process in detail to highlight the differences with our approach in Section III. Section IV introduces sKnock describing the authentication protocol and the design decisions we took to overcome common pitfalls. Section V describes our evaluations of an implementation of sKnock in Python. Its limitations are then presented in Section VI. Finally, as part of conclusion we describe the pros and cons of sKnock compared to others and suggest some future work in Section VII.

II. Attacker Model

For an attacker interested in attacking the services protected by port-knocked firewall, he has to either pose as a valid client of the service or defeat the port-knocking protection. For public services, an attacker can easily become a valid client. Therefore, we consider an attacker model where the attacker is interested in attacking the port-knocked firewall.

In this model, we consider the following capabilities to model different types of attackers:

- A1 Record any number of IP packets between a given source and destination and replay them from own IP address.
- A2 Modify, and suppress any number of IP packets from a given source and destination.
- A3 Send IP packets from any IP address and receive packets destined to any IP address.

Capability A1 can be acquired by an attacker by snooping anywhere on the network route between source and destination. Additionally, if the attacker can position himself as a hop anywhere in the route, he gains capability A2. A3 can be acquired by positioning himself in the link connecting the firewall to the Internet.

In addition to this, we also consider a valid client to be an attacker if the client's validity is revoked due to some reason and the client then tries to exploit the port-knocked firewall. Therefore, we assign the following properties:

B1 Attacker knows that the firewall uses port-knocking.

B2 Attacker may have previously port-knocked successfully as a valid client.

Notice that capabilities B1 and B2 are easier to acquire than A1, A2, and A3. B1 is even easier as it is possible to learn the existence of port-knocked firewall if that information is public (if it is advertised by the service provider).

In the rest of the text, we refer to attackers with a combination of these capabilities using the set notation: an attacker with capabilities A1 and B1 is termed as {A1, B1} attacker. If an attacker has a single capability, we will ignore the braces, *i.e.* A1, B1 attackers mean two type of attackers each with a capability, but not both capabilities.

Finally we assume that attackers cannot decrypt encrypted content and cannot forge signatures for arbitrary parties without the knowledge of their keys. This can be ensured in practice by proper usage of cryptography.

III. Related Work

Port-knocking is historically done by sending packets to different ports in a predefined static sequence. The sequence is kept secret and is shared with authorised clients. The firewall monitors the incoming packets and opens the corresponding port for a remote service for a client if the destination port numbers of a sequence of packets coming from that client match the its corresponding predefined sequence. An A1 attacker can observe this sequence and later replay it to defeat port-knocking.

Variants of port-knocking which use multiple packets to convey authenticating information to the firewall are termed under *Hybrid Port-Knocking*. To defend against A1 attackers, the sequence can be made dynamic with the usage of cryptography, *e.g.* by deriving the sequence from a time based one-time password (TOTP) [2]. However it is vulnerable to {A1, A2} attackers because the attacker can suppress a suspected sequence of packets from reaching the firewall and replay it from his host.

To defend against {A1, A2} attackers the IP address of the client needs to be encoded into the authenticating information in a way that that the firewall can retrieve it to open the port in the firewall for that client. If the client's IP address is in cleartext, then the authenticating information should contain a message authentication code (MAC), *e.g.* through HMAC [3], so that the attacker cannot rewrite it with his IP address.

However, encoding the client IP address in authentication information causes problems for clients behind NAT. Such clients are required to know their NAT's public IP address to be able to successfully knock the firewall. This may, however, lead to *NAT-Knocking* attacks [4] as the NAT's IP address is shared by other clients in the network and one of them could be an attacker. Aycock et. al. [5] proposed an approach based on challenge-response to solve this problem. This approach requires a three-way handshake where the client initiates by sending a request to the firewall. The request contains an identifier and the client's IP address. The firewall then sends a challenge containing the IP address it observed as the request's source IP, the IP address present in the request, and a random nonce, together with a MAC over these fields. The client then responds to this challenge by presenting a MAC over these fields with its pre-shared secret with the firewall. Since the handshake messages are authenticated with MAC, this approach is immune to {A1, A2} attackers.

A practical problem with hybrid port-knocking variants is that they fail when packets are delivered out-of-order to the firewall. To address this, the authenticating information could be sent as a single packet. This variant of port-knocking is termed as *Single Packet Authorisation* and is first documented to be used in *Doorman* [6]. Doorman authenticates clients based on a HMAC derived from the shared secret, port number to open for the client, username, and a random number. The random number is used to provide protection against A1 attackers as the firewall rejects a request with a number already seen. Since it does not include the client IP address, it is susceptible to {A1, A2} attackers.

Furthermore, there are variants which perform stealthy port-knocking by encoding the authenticating information into seemingly random looking fields of known protocols, *e.g.* the initial sequence number field of TCP, and the source port number. The advantage of these variants is that they make it difficult for an attacker to suspect port-knocking mechanisms just by observing the traffic. Among these are *SilentKnock* from Vasserman et. al. [7] and *Knock* from Kirsch et. al. [8].

SilentKnock encodes authentication token into the TCP header fields of the TCP SYN packet sent by the client. The firewall intercepts this packet from the kernel and extracts the token. The server then verifies the token and opens the corresponding port if the token is valid. The token is generated with keyed MAC with counters to prevent replay attacks from A1 attackers.

Similar to SilentKnock, the stealth property in Knock is achieved encoding the authentication token into the TCP header fields. Additionally, Knock allows for the client and the server to derive a session key which is then used to authenticate application data, thus preventing an {A1, A2, A3} attacker to take over the connection after successful port-knocking.

While the concept of stealthy port-knocking can be applied to any operating system, the current state of implementations for SilentKnock and Knock are limited to the Linux kernel. This poses a deployment barrier for service providers as they have to require their consumers to run complying software setup on their hosts.

IV. sKnock

In all of the approaches presented earlier, the client and the firewall depend on a shared secret to authenticate and gain defence against A1, A2 attackers. In the case of B2 attackers, these approaches require the shared secret to be invalidated at the firewall. This brings in the inconvenience and scalability problems discussed in Section I.

sKnock addresses the scalability problem by using certificates: each client gets a certificate which it uses to encrypt and sign the authentication information; the firewall requires the CA certificate to authenticate the client. B2 attackers are defended by limiting the certificate validity to an expiry date and having a certificate revocation list to invalidate certificates before their expiry.

In the reminder of this section we describe sKnock's authentication protocol and give a brief description of its prototype implementation in Python.

A. Protocol

The one-way authentication protocol of sKnock requires the client to send an authentication packet before opening a connection to the remote services behind the firewall. The authentication packet is a UDP packet containing the client's certificate and the port number of the remote service it wants to communicate with on the server. Additionally, it contains the client's IP and timestamp to provide protection against A1, A2 attackers. The format of the packet is shown in Fig. 1. To protect the privacy of the client, this information in the packet is encrypted with an ephemeral key which is derived from the server's public key using Elliptic Curve Diffie-Hellman (ECDH). The ephemeral key is freshly chosen for every authentication packet sent by the client. The client's Diffie-Hellman share required for generating the ephemeral key at the server is also included in the packet.

Since we want to keep the overhead of port-knocking low and also reduce the number of packets involved in the authentication to perform well under packet loss, it is important to fit this payload in one UDP packet. The limiting factors here are the network MTU sizes and the size of the client certificate. A common network MTU of 1500 bytes has eliminated the use of RSA and DSA public keys of lengths 2048 bits and above in the certificates. Fortunately, we were able to use Elliptic Curve Cryptography (ECC) public keys of lengths up to 256 bits offering security equivalent to that of 128 bit AES or 3072 bit RSA keys [9] resulting in a packet size of about 800 bytes. While ECC certificates are not as common as RSA or DSA certificates, this was the only option which gave us the possibility to keep the payload size lower while using x509 certificates.

Reliability against packet loss is achieved by retrying the authentication protocol. For connections to TCP services, the server could be configured to reject connections to closed ports by sending a TCP RST such that a failed authentication protocol will immediately result in TCP connection failure at the client which can then immediately retry. Whereas for UDP, the application requires its own protocol for determining the failure or, alternatively a timeout. The optimal value for the timeout would the sum of round-trip time (RTT) to the firewall, delay for processing the authentication packet and, delay for opening the corresponding port in the firewall.

B. sKnock Certificates

sKnock uses x509v3 certificates with the requirement that the certificates' public and private keys should be 256 bits long and generated using Elliptic Curve Cryptography. The certificates are used only by the clients of a service provider to port-knock the provider's servers. The service provider could either act as a Certificate Authority (CA) for signing these certificates or rely on another party; it is only important to have the same CA certificate configured on the servers.

In addition to authenticating the client, the client certificates also carry authorisation information specifying the protocol, port pairs the client is authorised to connect to. This information is encoded in the certificates using x509v3 extensions under object identifier for Technical University of Munich, 1.3.6.1.4.1.19518 as *Other Name* in the *Subject Alternative Name* (SAN) extension.

C. sKnock Server

Our prototype implementation of sKnock is developed in Python and works with the Linux *iptables2* firewall. The firewall is dynamically configured to allow traffic from port-knocked connections after the clients are authenticated, while the rest of the traffic is dropped by the firewall, including the sKnock authentication packets. To read the authentication packets we used a raw socket, which in Linux is not subjected to the firewall rules and hence can receive all the traffic reaching the host. As the raw socket receives all the traffic reaching the host, an efficient filtering process is required to filter out valid authentication packets from the rest. This is done by discarding packets which do not meet the following criteria in the order listed:

1) Packet's IPv4 or IPv6 header is valid
2) Packet is UDP and its header is valid
3) Decryption of the packet succeeded
4) Packet has valid sKnock header
   - Byte 32 is 0
   - Timestamp within allowed interval
   - Client IP matches packets source IP
5) Timestamp is within the allowed period
6) Client certificate is valid and not in the revocation list
7) Client certificate authorised for opening requested port
8) Client signature is valid

If the authentication packet passes all of the above checks, the requested port is opened in the firewall for the client sending it.

D. sKnock Client

sKnock client is an implementation of the sKnock protocol for client side applications. The client implementation is available as a library in Python and C. Applications can use

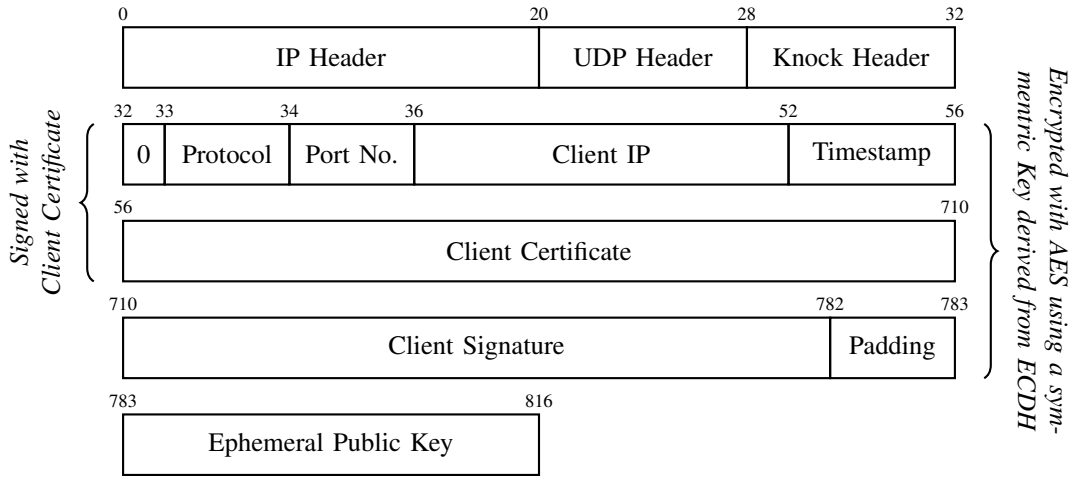

Fig. 1. Packet format of the sKnock authentication packet starting with a IPv4 header. The fields for *Protocol* and *Port No.* provide the port number to be opened in the firewall for the given protocol (TCP or UDP). The *Ephemeral Public Key* contains the public key used by the client and is required for the server to determine the AES key using Elliptic Curve Diffie-Hellman (ECDH).

the libraries to perform port-knocking at a sKnock firewall. The libraries contain the following functions:

- *knock_new (timeout, retires, verify, server certificate, client certificate, client certificate password)*: creates a new handle with the given certificates. The fields *retries* and *timeout* specify how many times sKnock should retry port-knocking and how long it should wait before determining a failure; applies to TCP connections. The *verify* flag specifies whether the library should test whether the port has been successfully opened or not; applies to TCP connections. *server certificate* and *client certificate* specify the paths to the server and the client certificates. *client_cert_passwd* is the field containing the password for the client certificate. This function returns a handle which is required by the next function.
- *knock_knock (handle, host, port, protocol)*: perform port-knocking by sending the authentication packet to the *host*. *handle* is the value returned from *knock_new()*. *port* and *protocol* specify which port should be opened in the firewall after successful port-knocking.

Applications can port-knock a sKnock firewall by using these two functions before they open a connection to a remote service behind the firewall.

In addition to this, the sKnock client implementation contains a command-line helper program to port-knock the server.

V. Evaluations

Since sKnock is intended as the scalable port-knocking solution, we evaluated its scalability and performance using a variety of tests. As part of this we evaluated the performance of the iptables2 firewall software, our filtering processes for filtering valid authentication packets, and the latency overhead incurred during connection establishment due to port-knocking.

All evaluations were performed on DELL OptiPlex 9020M machines equipped with a quad-core Intel(R) Core(TM) i5-4590T CPU @ 2.00GHz and 16GB of memory. The operating system running on the machine was Ubuntu Linux 12.04.5 LTS. The underlying cryptographic routines were provided by *OpenSSL-1.0.1* and the firewall was *iptables2-1.4.12*. The machines were connected with a *1 Gbps* Ethernet cable when required to form a network.

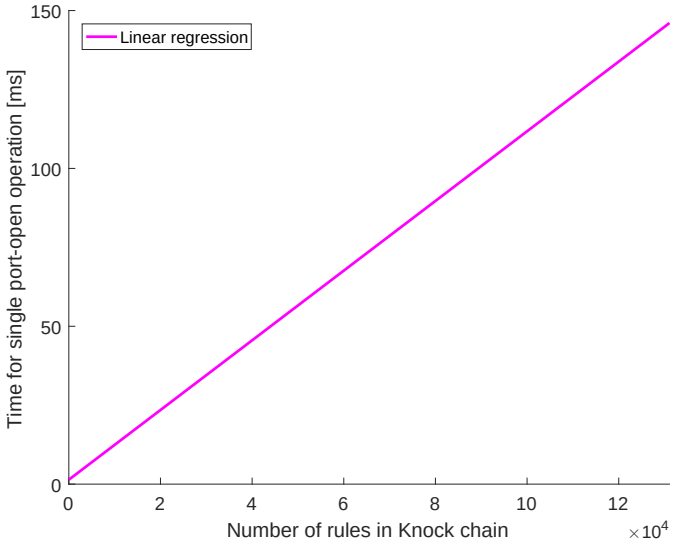

Fig. 2. Delay in adding new rules to the firewall. Rules are added sequentially to open 131070 (65535 TCP + 65535 UDP) for an IPv4 client.

A. Firewall

We evaluated the scalability limitations of using iptables2 firewall by measuring the time taken to open 131072 ports (65535 TCP + 65535 UDP) for an IPv4 client. The test adds a rule in the firewall to open a port for the given client and measures the time taken for the firewall to add this rule. This is repeated sequentially for each port until all of them are open for the given client. The evaluation data is shown in Fig. 2.

The results show that the delay in adding new rules is linearly proportional to the number of rules present in the firewall.

B. Packet Processing

Since sKnock uses a raw socket, it has to filter valid authentication packets from the rest of the traffic reaching the host. For this we defined a filtering process in Section IV-C. In this evaluation we measured the performance limitation of this filtering process with a static test by pre-generating some valid authentication packets together with some TCP and UDP packets complete with an Ethernet header and random payload. The pre-generated packets are read from a static list, therefore the delay incurred by raw sockets is not accounted.

In our test environment, we decided to run two different measurements: one synthetic worst-case scenario, which simulates that all incoming packets are valid authentication packets and a realistic scenario with 1% of port-knocking traffic while the rest of the packets are irrelevant to sKnock. Additionally the test data for the second scenario contains 5% of packets bigger than the configured minimum authentication packet size. Among these packets the TCP to UDP ratio is set at 5:1 in order to resemble the Internets' traffic patterns as close as possible [10].

The test-run performed to evaluate the worst-case processing power of our implementation yielded a result of roughly *5300 pps* (packets per second), which translates to a processing time of less than *0.19ms* per packet for valid authentication requests.

Our second test case, which is a closer approximation for real-world operation, yielded even higher processing capabilities with an average throughput of over *6100 pps*. Here our implementation achieved processing times of about *0.16ms* per packet.

C. Connection Overhead

In this evaluation we measured the latency incurred while opening connections when using sKnock's port-knocking. We used a simple protocol based on timestamps to measure this latency: the client sends its timestamp to the server as the first packet after port-knocking; the server then responds with its timestamp to the client. The server observes one-way latency while the client observes round-trip latency. The clocks of the client and the server are kept in sync using Precision Time Protocol (PTP) [11].

Since sKnock server requires some processing time to validate a port-knocking request and to add a new rule in the firewall to open the requested port, attempts to connect to a port may fail if the connection packets immediately follow the port-knocking request. Moreover, out-of-order delivery further increases the risk of such failures. To determine the optimal wait period between the authentication packet and the subsequent connection packet, we developed a calibration script. The script tries to open connections with a wait period derived from a start value and retries successful connections with shorter wait periods until a tiny (configurable) fraction of connection open requests fail.

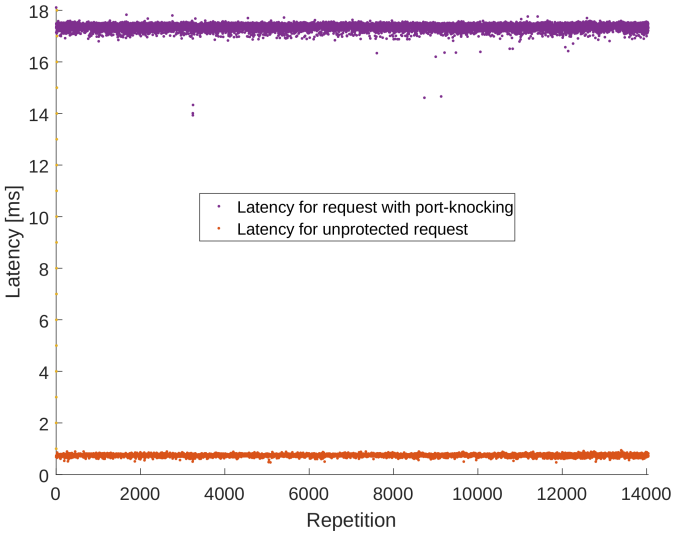

Fig. 3. Latency overhead caused by sKnock (UDP)

TABLE I
LATENCY DUE TO CONNECTION OVERHEAD

	TCP	UDP
Without port-knocking	1.97 ms	0.74 ms
With port-knocking	18.25 ms	17.37 ms
Port-knocking overhead	16.27 ms	16.63 ms

In this evaluation the calibration script yielded an optimal value of *11 ms* for the wait period allowing for a failure rate of *0.2%* with an accuracy of *99%*. To remove noise from the underlying network, we repeated the simple timestamp-based protocol number of times.

The observed latency for UDP run of the simple protocol with an without port-knocking protection can be seen in Fig. 3. The evaluation is also repeated for TCP and the results are summarized in TABLE I.

VI. LIMITATIONS

A major limitation of sKnock lies in the amount of overhead caused by the authentication packet. This is far greater than what is required by other implementations. However, if the connection is long-lived, this overhead will be tiny and since only one packet is used, the delay for port-knocking is kept to a minimum.

The other limitations of sKnock which we are aware of are as following:

A. Incompatibility with NAT

Since the current protocol requires the client to include its IP address into the authentication information, clients using NAT gateways cannot successfully port-knock the firewall.

Deployments seeking compatibility with NAT could ignore the client IP check, but they will then be susceptible to A2 attackers.

B. Vulnerability to Attacks

Any port-knocking scheme employing encryption is further subjected to the *DoS-Knocking* attack [4] where an attacker carries out a DoS attack by sending many legitimately-looking invalid port-knocking requests to keep the firewall busy while legitimate traffic is left in starvation. We agree that this will be also be the case with sKnock, but due to the usage of raw sockets, only new valid legitimate port-knocking requests are denied service as they are to be processed by sKnock which is kept busy with the DoS requests. Since sKnock's current implementation in Python cannot not occupy all of existing processor cores due to the presence of a global interpreter lock any already opened connections are not starved in this case because the firewall is not processing the port-knocking requests and it has already been configured to allow traffic from previously port-knocked clients.

sKnock is susceptible to replay attacks by {A1, A2, A3, B1} attackers. Such an attacker could wait for the port-knocking to succeed and then take over the connection by masquerading as the client. These attackers can be defended by authenticating the connection data which follows port-knocking as done by Knock [8]. This defence is not yet implemented in sKnock.

C. Performance

The current Python implementation of sKnock has limitations in terms of operational efficiency due to Python interpreter being single threaded. While this implicitly gives partial defence towards DoS-Knocking attacks when run on a system with a processor having more than a single core, the throughput could be improved by parallelising request filtering and validation.

Another implementation specific problem with ECC keys is that ECC is relatively new and OpenSSL supports only a limited number of well known curves. Moreover, due to their novel state, there exists no hardware implementation as of this writing to significantly speed up their processing, thus limiting the performance of our implementation.

D. Trust in NIST Curves

sKnock relies on OpenSSL for providing PKI support. As of this writing, OpenSSL does not yet support any of the curve determined as SafeCurves [12] providing 256 bit keys. This led us to use NIST-P 256 curve whose usage may not be secure [12] and hence should be replaced when the required support is available in OpenSSL.

VII. Conclusion & Future Work

When compared to other port-knocking implementations sKnock has high overhead in terms of payload and processing requirements. Furthermore, stealthy port-knocking with sKnock is not possible. As an advantage, sKnock provides easy deployability to service providers as it can be readily integrated into their PKI infrastructure and as it does not require changes to the client's operating system.

The overhead incurred due to using x509 certificates could be reduced by implementing a custom format for encoding the certificates into the authentication information. This will however deny us the usage of well-audited PKI libraries, increase development costs and induce security issues. Alternatively, usage of other formats such as OpenSSH certificates could be explored.

During our evaluations we found that the lack of native parallelism in Python severely limited the performance. Additionally, there was also some overhead involved in converting data structures between the underlying cryptographic libraries and the firewall interface, which were available as C libraries. Therefore, we believe that on a multi-core system the performance of sKnock could be improved by implementing it in a system programming language such as C or Rust.

Another improvement could be made to improve support for UDP data streams. The current prototype requires the client to keep sending authentication packets periodically as long as the application is using the UDP stream to avoid the firewall from timing out the connection and closing the corresponding port. This could be improved by adding application level UDP connection tracking at the server, where the remote service can close the firewall when the connection is no longer required.

Acknowledgment

This work was carried out under the scope of the SafeCloud EU research project funded through EU grant agreement 653884.

References

[1] I. ITU, "Information technology–open systems interconnection–the directory: Publickey and attribute certificate frameworks," *Suíça, Genebra, ago*, 2005.

[2] D. M'Raihi, S. Machani, M. Pei, and J. Rydell, "Totp: Time-based one-time password algorithm," Internet Requests for Comments, RFC Editor, RFC 6238, May 2011, http://www.rfc-editor.org/rfc/rfc6238.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6238.txt

[3] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac: Keyed-hashing for message authentication," Internet Requests for Comments, RFC Editor, RFC 2104, February 1997, http://www.rfc-editor.org/rfc/rfc2104.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2104.txt

[4] A. I. Manzanares, J. T. Márquez, J. M. Estevez-Tapiador, and J. C. H. Castro, "Attacks on port knocking authentication mechanism," in *International Conference on Computational Science and Its Applications*. Springer, 2005, pp. 1292–1300.

[5] J. Aycock, M. Jacobson *et al.*, "Improved port knocking with strong authentication," in *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 2005, pp. 10–pp.

[6] M. Krzywinski, "Port knocking from the inside out," *SysAdmin Magazine*, vol. 12, no. 6, pp. 12–17, 2003.

[7] E. Y. Vasserman, N. Hopper, and J. Tyra, "Silentknock: practical, provably undetectable authentication," *International Journal of Information Security*, vol. 8, no. 2, pp. 121–135, 2009.

[8] J. Kirsch and C. Grothoff, "Knock: Practical and secure stealthy servers," 2014.

[9] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Nist special publication 800-57," *NIST Special Publication*, vol. 800, no. 57, pp. 1–142, 2007.

[10] M. Zhang, M. Dusi, W. John, and C. Chen, "Analysis of udp traffic usage on internet backbone links," in *Applications and the Internet, 2009. SAINT'09. Ninth Annual International Symposium on*. IEEE, 2009, pp. 280–281.

[11] H. Weibel, "High precision clock synchronization according to ieee 1588 implementation and performance issues," *Proc. Embedded World 2005*, 2005.

[12] D. J. Bernstein and T. Lange, "Safecurves: choosing safe curves for elliptic-curve cryptography," *URL: http://safecurves. cr. yp. to*, 2013.