SILENTKNOCK: Practical, Provably Undetectable Authentication

Eugene Y. Vasserman[1], Nicholas Hopper[1], John Laxson[2], and James Tyra[1]

[1] Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455 USA
[2] Stanford University, Box 15255, Stanford, CA 94309 USA

Abstract. Port knocking is a technique first introduced in the blackhat and trade literature to prevent attackers from discovering and exploiting potentially vulnerable services on a network host, while allowing authenticated users to access these services. Despite being based on some sound principles and being a potentially useful tool, most work in this area suffers from a lack of a clear threat model or motivation. We introduce a formal security model for port knocking that addresses these issues, show how previous schemes fail to meet our definition, and give a provably secure scheme that uses steganographic embedding of pseudorandom message authentication codes. We also describe the design and analysis of SILENTKNOCK, an implementation of this protocol for the Linux 2.6 operating system, that is provably secure, under the assumption that AES and a modified version of MD4 are pseudorandom functions, and integrates seamlessly with any existing application, with no need to recompile. Experiments indicate that the overhead due to running SILENTKNOCK on a server is minimal – on the order of 150 μs per TCP connection initiation.

1 Introduction

A *port scan* is a kind of network attack (or attack precursor) in which an adversary attempts to connect to all, or some subset of, TCP and UDP ports at a given IP address. Port scans are useful to attackers because the results often indicate the operating system, architecture, and even a set of specific binaries that a host is running. This information can then be used to determine what software exploits should be used to attack the host, or what level of compromise might be likely.

Of course, if a server runs no vulnerable software, a port scan is not a serious threat, but software security is a sufficiently hard problem that this cannot be seen as an immediate solution. A popular method of protecting against such network attacks is the firewall, which simply blocks all connection attempts to "internal" network hosts from "external" ones. Since there are many reasons why it might be desirable for a given service to be externally accessible — for instance, users may access a network service from a priori unknown network addresses depending on their physical location — this solution is not always satisfactory.

One class of proposed solutions to this problem is "port knocking": a firewall is deployed to protect a server, and before allowing a client connection to a particular port, that client must transmit a special "knock" that authenticates it. This knock may be either common to all authorized users of the system, or may be unique to a given user.

J. Biskup and J. Lopez (Eds.): ESORICS 2007, LNCS 4734, pp. 122–138, 2007.

Any attempts to connect that are not associated with the correct knock will be dropped; thus to an unauthorized user it should appear as if no network services are running on the server. A variety of knocking methods have been proposed, such as a sequence of (dropped) connection attempts [1], inclusion of a cryptographic authenticator in the initial connection request packet [2], "funny-looking" DNS lookups [3], and IPsec tunneling [4].

Many previous proposals for port knocking schemes have been accused of offering "security through obscurity", since it is trivially easy to detect and steal knocks in non-cryptographic systems. But making the distinction between flawed *implementations* which are only secure if the details of the system are unknown, and the *concept* of port knocking (that *even given the details of a port knocking scheme*, one cannot tell if it is being employed), we argue that this *concept* is not fundamentally flawed. Since revealing the presence of a service can only help an adversary — for example, by revealing which of a list of hundreds of exploits is the most likely to succeed, thereby decreasing the cost of an attack — the notion of concealing services from unauthenticated users (in addition to regular network and software security measures) is a potentially useful one. Separating authentication from applications is also a sound choice, since it enforces least privilege and economy of mechanism, in addition to easing deployment.

Given that the goal of a port knocking scheme should be to conceal the set of services running on a network host, all existing implementations have a serious flaw. Under relatively weak attack models, these schemes fail to conceal that a port knocking service itself is running. Since this service mediates access to all the other services exported by a host, exposing information about the presence and type of port knocking service a host is running is highly undesirable: under fail-closed semantics, crashing the port knocking service denies access to all services on the host, while under fail-open semantics detecting and crashing the port knocking service allows an ordinary port-scan to succeed. Of course, on most currently deployed operating systems, exploiting a code injection attack in a port knocking service would lead to a total compromise of the host. Since the port knocking service is such a high-value target, we argue that *the presence of port knocking itself should not be detectable*.

In this paper, we develop a formal security model which captures this notion. A formal security model is critically important in order to be certain that a given protocol, even one that seems secure at a glance, is *actually* secure. Examples of such "apparently secure" protocols, developed without formally stated security goals, are numerous [5, 6, 7], and some of them have been in operation for years (and have even become industry standards), before attacks were found. Note that all those protocols were originally designed for security, and even used well-known cryptographic primitives, but the protocols were not secure.

Essentially, our notion states that even though a (computationally bounded) adversary may observe many authenticated sessions and arbitrarily inject, delete, and reorder messages between the client and server, he cannot distinguish a port knocking client and server from a pair using ordinary TCP/IP, plus some out-of-band authentication mechanism that prevents other clients from connecting. That is, our definition allows the adversary to observe authenticated sessions and necessarily allows the adversary to observe that *somehow* sessions are being authenticated, but insists that no *additional*

information about the authenticating mechanism is leaked. This leaves many plausible explanations for the behavior, such as dynamic firewall rules[1].

We prove that a scheme which is secure in our model also resists forgery and provides replay attack protection against a global active adversary. We give a protocol for a generic networking scheme, which makes rudimentary use of provably secure steganography [8], and prove that this protocol satisfies our strong notion of security. Furthermore, we describe and analyze the security of SILENTKNOCK, an implementation of our generic protocol for the Linux 2.6 TCP/IP stack. SILENTKNOCK combines simple TCP steganography [9] with a very fast cryptographic message authentication code (MAC) [10] to provide efficient, provably secure port knocking that integrates seamlessly with existing applications (with no need to recompile) by hooking directly into the operating system kernel. SILENTKNOCK produces packets that are provably indistinguishable from TCP packets generated by the Linux 2.6 implementation of TCP, under the assumption that AES and a variant of MD4 are pseudorandom functions[2]. No applications need to be altered, no shared libraries need to be replaced, and no potentially-conflicting protocols emerge. SILENTKNOCK is lightweight, has minimal computational overhead, and is freely available for download [12].

Related work. The first published description of a port knocking scheme seems to be the work of Barham *et al.* [2], who describe a scheme whereby a pass-phrase is transmitted (in cleartext) to a firewall either through a series of SYN packets, in a single "knock" packet, or as an option in the SYN packet. Krzywinski [1] describes a similar scheme where a client opens a port by attempting connections to a secret sequence of port numbers[3]; a number of similar systems are described at [13]. Several authors [3, 14, 4] have proposed that knocks should be cryptographically protected to prevent replay attacks, but still fundamentally involve the use of extra packets or nonstandard TCP options that allow the detection of a knock (these systems provide authentication only, i.e. they make no attempt to hide the use of authentication mechanisms). deGraaf *et al.* [4] and Manzanares *et al.* [15] describe some other attacks and weaknesses of previous port knocking schemes, which our notion of security precludes — that is, any scheme that satisfies our security notion necessarily is also secure against the attacks mentioned in these papers.

There is an extensive literature on TCP/IP steganography and covert channels [9, 16, 17, 18, 19], although Murdoch and Lewis [9] show that many of these proposals are easily detected. We introduce a cryptographic formulation of security similar to that in [8], and our notion of a secure port knocking scheme can be seen as a simple instance of a covert computation [20] or the dining Freemasons problem [21]. We are, however, unaware of previous work relating steganographic computation and port knocking, or

[1] e.g. a service that is only available at preset times, or a software firewall that allows the user to approve connection requests.

[2] Linux 2.6 chooses TCP sequence numbers using 24 rounds of MD4 applied to the source and destination IP address, destination port, and 32 secret random bits, using a randomly generated, secret initial chaining value that changes every 5 minutes. See the functions `secure_tcp_sequence_number` and `half_md4_transform` at [11].

[3] The server will monitor connection attempts on all closed ports and opens a port if a specific sequence of connection attempts is detected.

any previous work implementing the schemes of [20, 21]. We note that our system, like those in [20, 21] differs from covert channels alone because we provide covert one-way *authentication*, handle synchronization issues, and formally reason about what it should mean to hide an authentication service.

2 Formal Definition of Port knocking

In this section we provide our formal model of a secure TCP port knocking scheme, and prove several relationships between our definition and formalized versions of earlier security properties. We begin by stating our formal model of the TCP protocol:

Definition 1. *A* TCP implementation *is a triple $\mathcal{P}$ of efficient, probabilistic programs* Client*,* Server*, and* Init*.* Client *has three arguments: a* state s*, a* command c*, and a* packet r*;* $\mathsf{Client}(s, c, r)$ *outputs a new* state s'*, and a* packet p*. Similarly,* Server *takes as input* state s*,* command c *and* packet p *and outputs a* state s'*, a* packet r*, and a* message m*.* Init *takes as input either* `client` *or* `server` *and outputs a* state s*.*

Standard TCP Client *commands* are of the form "connect to port 80 from port 1234," "send M from port 1234 to port 80," or "close the connection to port 80 from port 1234." Server *commands* are of the form "listen on port 80" or "close the connection from C:1234 to S:80." With a null command, Client simply outputs the next packet to be sent to the server and Server outputs a packet acknowledging the input packet and a message consisting of the data received in the last packet. Client and server *state* includes the TCP states of all connections, and buffered messages. Standard TCP *packets* p have two fields we will make use of. The syn flag, $p.syn$, is always set on the first packet sent on a new connection. Packets with this flag set are the standard way of "knocking" at a port to establish a connection. TCP/IP connections are uniquely identified by the tuple (client IP, client port, server IP, server port) which we refer to as $p.id$.

For a given *command sequence* $\mathcal{C} \in (\textit{command} \times \textit{command})^*$ we define the *standard interaction* of a TCP implementation $\mathcal{P}$ as the following process. First, we initialize $s_0 = \mathsf{Init}(\texttt{server})$, $q_0 = \mathsf{Init}(\texttt{client})$, and set (p_0, r_0) to null. Then for each pair $(\kappa_i, \sigma_i) \in \mathcal{C}$, we let $(q_i, p_i) = \mathsf{Client}(q_{i-1}, \kappa_i, r_{i-1})$ and $(s_i, r_i, m_i) = \mathsf{Server}(s_{i-1}, \sigma_i, p_{i-1})$. We define the *output* of the standard interaction on $\mathcal{C}$, $\mathcal{P}(\mathcal{C})$, to be the concatenation $m_1||m_2||\cdots||m_\ell$.

Definition 2. *A* Port knocking protocol *is a TCP implementation $\mathcal{H}$ in which both* Client *and* Server *take as additional input a secret key. We let $\mathcal{H}_K(\mathcal{C})$ denote the result of the standard interaction between* Client *and* Server *where the key input to both is K.*

We say that $\mathcal{H}$ *extends* TCP implementation $\mathcal{P}$ if for every command sequence C, there is an efficiently computable command sequence C' such that $\mathcal{H}_K(C')$ and $\mathcal{P}(C)$ are computationally indistinguishable, for uniformly chosen K. This requirement states that a port knocking protocol, in which the client and server share a secret key, should allow any communication that is allowed by the TCP implementation it extends. We note that a TCP implementation is trivially an (insecure) port knocking protocol with null keyspace: every TCP session is initiated when the client "knocks" at the server port he wishes to connect to.

Security Condition. The informal idea behind our definition and construction is that a port knocking scheme should hide not only a set of network services, but the very fact that port knocking is in use, to the extent possible; and this condition should hold even against an adversary who is allowed to make connection attempts to the server and see authenticated connections by the client. To that end, we define security of a port knocking scheme $\mathcal{H}$ in terms of an adversary's inability to distinguish between two experiments, corresponding to two different "worlds" in which he might find himself. In both experiments, the adversary is given black-box access to Client and Server subroutines (i.e., oracles) that output only packets and maintain state internally, so the attacker may issue commands and deliver packets to the client and server. We stress that these oracles are "black boxes" only insofar as the adversary cannot *a priori* infer which of the two possible sets of oracles he is interacting with; adversaries are assumed to know the implementation details of each of the two possible oracle pairs.

In the "hidden world", these subroutines implement the Client and Server routines from $\mathcal{H}$, with a shared secret key K. The adversary is allowed to interact arbitrarily with these subroutines, and in particular may make as many queries to both as he desires. This models what an adversary who is attacking a port knocking implementation will see. In what we call the "plausible world," the client and server subroutines are essentially those of the TCP implementation $\mathcal{P}$, *except* that they are slightly modified. The "plausible" client and server are modified to share a queue of packets Q. Whenever the client generates a packet p, Q is scanned for a packet q with $q.id = p.id$; if none is found, p is added to the end of Q. The server also maintains a list Open of ids. Whenever it is called with a packet p, the server checks to see if $p.id \in$ Open, and if it is, calls $\mathcal{P}$.Server on p; if p is at the front of Q, the server adds $p.id$ to Open, removes p from Q, and calls $\mathcal{P}$.Server on p; otherwise, the server does not respond to p. In essence, client and server share an out-of-band signaling mechanism such that only recent connections initiated by the client are processed by the server. Notice that the packets output by the "plausible world" client are identical to the packets output by $\mathcal{P}$, and if the adversary simply relays the packets between Client and Server, he will see a perfectly normal TCP session. However, if the adversary interacts only with the Server oracle, his connection attempts will be ignored, because his packets are not on the shared queue. Thus this "plausible world" formalizes the idea of revealing that there is authentication going on, but not revealing any additional information about the authentication.

We say that a port knocking scheme is secure if an adversary who can see many authenticated sessions and attempt to make many connections cannot tell if he is in the "hidden world" or the "plausible world", that is, he cannot tell from the results of his attack whether port knocking or some other plausible form of authentication is being employed. Formally, we define the experiments $\mathsf{Exp}^{\mathsf{hw}}_{\mathcal{H},A}$ and $\mathsf{Exp}^{\mathsf{pw}}_{\mathcal{P},A}$ as in figure 1, and we define the *port knocking advantage of A against $\mathcal{H}$ with respect to $\mathcal{P}$* to be

$$\mathsf{Adv}^{\mathsf{pk}}_{A,\mathcal{H},\mathcal{P}}(k) = \Pr[\mathsf{Exp}^{\mathsf{hw}}_{\mathcal{H},A}(1^k) = 1] - \Pr[\mathsf{Exp}^{\mathsf{pw}}_{\mathcal{P},A}(1^k) = 1] \,.$$

We say that $\mathcal{H}$ is a (t, q_C, q_S, ϵ)*-secure port knocking scheme with respect to* $\mathcal{P}$ if for every time-t adversary A that makes at most q_C Client queries and q_S Server queries, $\mathsf{Adv}^{\mathsf{pk}}_{A,\mathcal{H},\mathcal{P}}(k) \leq \epsilon$. We call such an adversary a (t, q_S, q_C) adversary.

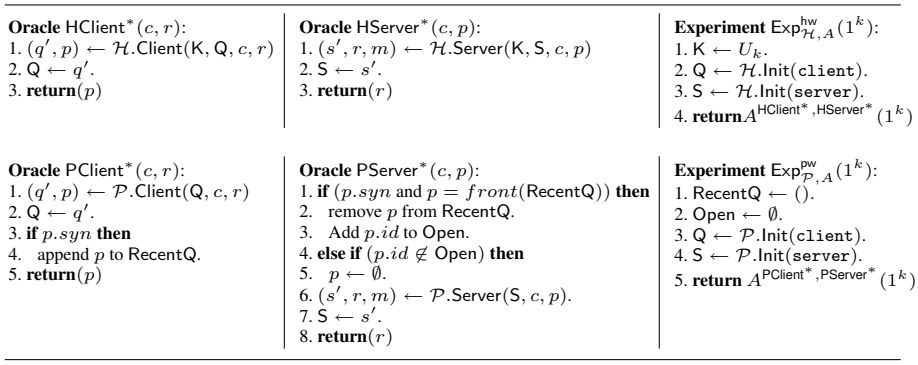

Fig. 1. Definition of hidden world (top row) and plausible world (bottom row) experiments

Related notions. Given a new notion of security, it is natural to ask whether it is the *right* notion. In the full version, we give some evidence for the strength of our notion, by considering several security conditions which have been implicitly or explicitly used as the security goals of earlier port knocking schemes, and showing that our security notion is stronger.

3 System Design

In this section we introduce SILENTKNOCK, our implementation of a secure port knocking scheme, and discuss how this implementation embodies the security model defined above. We first discuss several adaptations necessary for secure and reliable interaction with TCP/IP, such as replay attack protection, client/server synchronization, and indistinguishability. Next, we analyze a number of possible attacks on our implementation. Finally, we present results showing our system in action. A generic presentation of the scheme, along with security proof, appears in the appendix.

SILENTKNOCK is designed to be an application-agnostic transport-level authentication layer. It resists forgery and replay attacks while leaking no further information about the authentication method employed. We use kernel hooks to ensure that applications do not need to explicitly support our system in order to benefit from it. We use keyed MACs as secure authenticators to resist forgery attacks and a two-part counter to counteract replay attacks while ensuring that client and server counters stay synchronized even in the presence of moderate packet loss. We provide an implementation of a previously proposed operating system-specific steganographic embedding scheme for TCP/IP [9] and use it to embed authentication information into TCP headers.

Universal Compatibility. We provide ease-of-use (for end-users, system administrators, and programmers) by choosing an application-agnostic design. By using hooks directly into the operating system kernel, we avoid modifying any of the network kernel or library calls made by application software or requiring supports for SOCKS-type proxies. This allows any application to transparently use SILENTKNOCK (without application awareness or modification), provided that the network protocol used by the

application has a steganographic embedding/extraction method supported by SILENTKNOCK . We note that for certain protocols, such as TCP, with many implementations that may have subtle differences, each implementation may require a different steganographic embedding routine to preserve indistinguishability. Our goal is to seamlessly support as many transport protocol implementations as possible, although currently only TCP under Linux 2.6 is supported.

Design Choices. Our implementation is designed to run on the Linux operating system with a 2.6 kernel. We chose Linux 2.6 due to our familiarity with the system and the availability of the netfilter/libIPQ API [25], which allowed us to implement our system entirely in user space instead of modifying the operating system. We use Poly1305-AES [10] as our MAC function since it is optimized specifically for network packets and has very fast implementations available for most processor types. We implement Murdoch and Lewis' system for embedding steganographic information into TCP initial sequence numbers (ISNs) [9] and use the TCP timestamp option (enabled by default in Linux 2.6) to embed an additional byte of information into the timestamp, delaying packets when needed. For additional details on the adjustments necessary to make random ISNs consistent with the Linux 2.6 network stack, see [9].

3.1 Protocol

The SILENTKNOCK algorithm is outlined in Figure 2. A SILENTKNOCK client initiates a connection (composes a TCP SYN packet) to a SILENTKNOCK-enabled server and steganographically embeds an authentication token into the packet. The embedding algorithm and resulting packet header structure are described in Figures 3 and 4, respectively. The server receives a SYN packet and extracts the authenticator. If verification is successful, the server allows the connection to continue, otherwise the packet is dropped. The client and server share a key, as well as a counter which is incremented for every client connection attempt (we discuss counter synchronization later). The counter prevents replay attacks by ensuring that every SYN packet sent by the client is different from any packets sent previously, and is also used as the nonce required by our MAC function. The key, initial counter, and resynchronization interval are exchanged out of band, since negotiation is impossible in case of one-way communication.

MAC. Instead of an additional sequence of knocks, we use a keyed MAC for client authentication, applying it to the source and destination (IP, port) tuples as well as the counter, so every connection attempt is guaranteed to contain a unique MAC. We employ Poly1305-AES [10] for our MAC function since it is designed specifically to work on small bits of data such as network packets and is implemented in optimized assembly for a number of popular platforms. The connection counter serves as the nonce required by Poly1305-AES. Assuming that AES is a pseudorandom permutation, an adversary should not be able to compose a valid MAC, or even distinguish one from random bits, for the next SYN packet without knowing the key (even if we assume that all other factors are public information).

Steganography and Indistinguishability. We use the TCP sequence number and timestamp fields of the TCP SYN packet to embed our MAC information [9]. Unfortunately, we are not able to include the complete MAC, as our current implementation

```
1. B → A: MAC_{k,ctr_B}(m); encoded in TCP/IP headers of SYN packet
2. A: Set ctr_A ← ctr_A + 1
   for i = 0 to ft:
      if (MAC_{k,ctr_A−1+i}(m) = MAC_{k,ctr_B}(m))
        Set ctr_A ← ctr_A + i + 1; resynchronize counter if client is ahead
        A → B: SYN-ACK
        goto 5
3. B: if (SYN-ACK received) then
        Set ctr_B ← ctr_B + 1, goto 5; connection was successful
4. B: if (SYN-ACK not received) then
        Set ctr_B ← ctr_B + 1; assume server got SYN, but SYN-ACK was lost
        goto 3
5. A, B: proceed with TCP connection
   if (FIN or RST received) then
     goto 1
```

Fig. 2. The pseudocode for SILENTKNOCK. A is the server, B is the client, ctr_P is a per-IP-address counter maintained by principal P, k is a value derived from B's IP address and a symmetric key shared between A and B, m is a TCP flow identifier, and ft is a failure-tolerance parameter.

only allows a total of 32 bits to be embedded (24 bits in the sequence number and 8 bits — the least significant byte — in the timestamp), assuming Linux sequence numbers[4] (see Figure 4). Since we must not allow distinguishability based on discrepancy between the observed packet dispatch time and the packet timestamp, we delay packet transmission, but only use the last timestamp byte to minimize delay times. Although 32 bits is a relatively short MAC, recall that even at this length, an adversary would still have to compose, on average, 2^{32} packets to break the authentication (requiring, for example, 6 weeks to transmit over a T1 link). We remark that standard methods to deal with online guessing attacks can also be applied here, such as account freezing or processing delays.

One issue that arises when using the TCP timestamp field (rather than just the ISN) to encode MAC data is the possibility of lost SYN packets. For instance, if a client generates a SYN packet but a SYN-ACK from the server does not arrive, the client must re-transmit the SYN packet. However, TCP requires that re-transmitted SYN packets have the same sequence number but different timestamp [26], so we can no longer encode stegotext in the timestamp: if the SYN packet was lost due to a malicious host, or if an adversary is observing all SYN packets, that adversary would detect that the least significant byte of the timestamp in the original and re-transmitted SYN packets are identical. The probability of this is only $1/256$, so the adversary could conclude that SILENTKNOCK was in use.

To solve this problem, we ensure that the last byte of the timestamp looks random to our adversary, even when we are trying to re-transmit the same MAC. We can use two existing properties of our system to help us, the first having originally caused this problem: the higher order bytes of the new timestamp must be different from the one in the original SYN packet[5]. Secondly, we do not transmit the entire MAC (only the

[4] OpenBSD has 30 bits of entropy available in the sequence number, while Linux 2.6 only has 24 bits.

[5] In reality, we only use the middle two bytes of the timestamp, since the upper byte is extremely unlikely to change, and the bottom byte will be replaced by stegotext.

P **: TCP SYN packet**
$P_{seq} = \{S_1, S_2, S_3, S_4\}$ **: Sequence number of packet P (4 bytes)**
$P_{ts} = \{T_1, T_2, T_3, T_4\}$ **: Timestamp of packet P (4 bytes)**
$m = (IP_B, source\ port, IP_A, destination\ port)$ **: Authentication information**
$MAC_{K,ctr}(m) = \{M_1, M_2, \ldots, M_{16}\}$ **: 16 byte MAC**
$S_2 = M_1, S_3 = M_2, S_4 = M_3$
$T_4 = h_M(\{T_2||T_3\})$ **: n-Universal hash function**

Fig. 3. The steganographic encoding protocol. Decoding is performed by reversing the operations in this protocol.

first 32 bits), so the adversary has no knowledge of the rest. We use these undisclosed MAC bytes to key an n-universal hash function (e.g. $h_{\boldsymbol{a}}(x) = a_1x^{n-1} + a_2x^{n-2} + \cdots + a_{n-1}x + a_n$) [27], which is applied to the middle bytes of the (changed) timestamp to determine the last byte of the timestamp, ensuring that any n or fewer distinct timestamps have last bytes that are indistinguishable from random. [6]. Since the server computes the same MAC, the server can reverse this process and extract the stegotext. Therefore we preserve the integrity and indistinguishability of stegotext in our timestamp even for re-transmitted packets (note that a packet will again need to be delayed so transmission time is consistent with the new timestamp).

Counter management. To protect against replay attacks, we employ a per-user counter, incremented after every connection attempt. If a given user has never before accessed a SILENTKNOCK-protected server, the counter is initialized to 0 by both the client and the server. The counter poses additional challenges, such as what happens when the client and server counters become desynchronized. Desynchronization can occur in two ways: either the client's SYN packet never arrives at the server, leading to the client having a counter higher than the server's, or the server's SYN-ACK can be lost, meaning the client and server are actually in sync, but the client does not know this. A client would have a hard time attempting to resynchronize after a failed connection, since the client does not know whether the server received the SYN packet and verification failed, or whether the server received and verified the SYN packet but the SYN-ACK was lost, or whether the SYN never arrived at the server. We allow for automatic in-protocol resynchronization after a certain time period.

For this purpose, we enforce the equation $ctr_{server} \leq ctr_{client}$ by having the client always increment its counter when sending a SYN packet. The server, however, will only increment its counter upon successful MAC validation, to prevent malicious desynchronization by sending bogus packets to the server. In the naïve scheme of insisting the counter be exactly right, the server and client may never again get into sync once desynchronized, since the client will increment its counter on each connection attempt, but the server's counter remains the same.

To counteract permanent desynchronization, we adopt a two-part counter design. Using a 64-bit counter, the first 32 bits (called the RESYNC field) are initialized to 0 (at the

[6] By default, Linux 2.6 TCP only attempts to re-transmit a failed SYN packet five times, so 5-universal hashes are sufficient. If this number were to change, both the client and the server would need to modify their hash function (for n retransmissions, an m-universal hash function, where $m \geq n$ must be used).

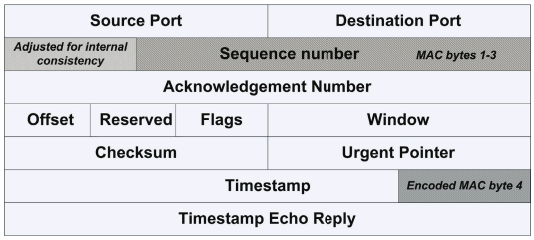

Fig. 4. The TCP SYN packet after steganographic embedding. The "internal consistency" adjustment in the sequence number is performed to keep the modified sequence number consistent with what Linux is expected to produce.

time of first connection) and are incremented once every configured unit of time (such as every hour, day, month, leap-year, etc.). The time period must be agreed upon by the client and the server as part of out-of-band setup. The latter 32 bits (called the CTR field) are always reset to 0 when RESYNC is incremented. Using this two-part counter, we allow resynchronization to occur automatically once the RESYNC increment time elapses. If there is substantial relative clock drift between the client and server, it is possible that client connections will fail (or even become desynchronized) when the client initiates a connection at a time when one entity has incremented RESYNC and reset CTR but the other has not. However this is extremely unlikely and would repair itself during the next RESYNC increment. Checking more than one consecutive value of the counter as part of the MAC would make desynchronization unlikely for most (transient) network-level failures, but would also degrade security linearly, since it allows multiple MACs to be valid at any given time. If multiple counters are checked, the server should save the counter that matches whichever MAC successfully verified, and increment that counter for use next time. This way, the server and client should be in sync for the next connection attempt. (The number of alternate CTR values checked by the server is specified by the ft parameter in Figure 2.)

3.2 System Architecture

The SILENTKNOCK system is composed of two separate programs - "`sknockd`" (running on the server), and "`knockproxy`" (running on the client). Connections are authenticated on a per-flow instead of per-source (IP address) basis. While `knockproxy` actively modifies packets as they leave and enter the client machine, `sknockd` (on the server side) does not do any packet modification. Combined with the very low verification overhead of our chosen MAC function, this should minimize the load on the server. We use the libIPQ API to register interest in packets with certain flags and (IP, port) tuples with the kernel, and those packets are rerouted by the netfilter system to user-space[7]. On both client and server side, we only send packets we are potentially interested in to user space, to avoid excess context switching between user-space and kernel-space. Both `sknockd` and `knockproxy` currently detect closed connections

[7] This re-routing happens after processing by the network stack for outgoing packets, but before processing for incoming packets.

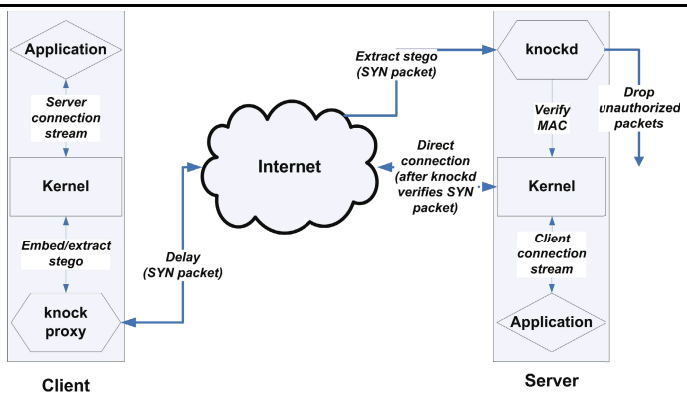

Fig. 5. The architecture of SILENTKNOCK. The client-side application initiates a connection to a server in the usual manner. The kernel composes a SYN packet, but `knockproxy` intercepts the packet before it is sent, and embeds a MAC into the ISN and timestamp fields. The server receives the packet, and `sknockd` examines it before passing it to the kernel. If `sknockd` successfully extracts and verifies the MAC, the packet is accepted by the kernel and passed to the application; otherwise it is dropped. Once the SYN packet is accepted, `sknockd` no longer examines other packets for that connection (except for terminating packets FIN and RST). `knockproxy`, however, is forced to rewrite every incoming and outgoing packet for the connection to prevent the client TCP stack from getting confused due to a sequence number mismatch.

by listening to FIN and RST packets, and timeout support (in case the FIN or RST packets are never received due to packet loss) will be added in the future.

Knock Daemon. `sknockd`, the server side of the SILENTKNOCK system, listens for connections on a port it reads from its configuration file (the port offering the protected service, i.e. SSH on port 22), and examines incoming SYN packets on those ports before the TCP/IP stack sees them. When a packet is received, `sknockd` checks the source IP address of the packet and retrieves the secret key as well as the counter for that IP address from its configuration file (per-user shared keys are also supported). Using the TCP steganographic algorithm, `sknockd` extracts stegotext from the packet, treats it as a MAC, and attempts to verify it. If verification succeeds the packet is accepted, and passed on to the TCP/IP stack, otherwise the packet is dropped. `sknockd` then increments the per-IP connection counter (CTR). This is the extent of `sknockd`'s involvement with the connection — all other packets are processed directly by the network stack in the kernel, and are not seen by `sknockd` (except for detection of connection closing). Since the SYN packet is copied only once (from kernel to user space) and not modified (does not have to be copied back), and since our chosen MAC is very fast, the entire operation is very efficient. Furthermore, since only SYN packets are examined, the load on the server is minimized.

There is a small trick to preserving indistinguishability when we in fact are intercepting certain packets — we must prevent the adversary from being able to set the SYN flag on a packet that is part of an existing (previously authenticated) stream, because if `sknockd` drops that packet (due to incorrect MAC), the adversary will be able to conclude that SILENTKNOCK is in use. Therefore, when `sknockd` tells the

netfilter to allow a certain connection (after verifying the MAC), we insert the ALLOW rule into netfilter *before* the rule that forwards all SYN packets to `sknockd`. Thus, authenticated streams (having a known source (IP, port) tuple) are never again processed by `sknockd`, even if they (incorrectly) contain SYN packets, preserving default TCP stack behavior. This solution (inserting the ALLOW rule for a flow before the `sknockd` rule for SYNs) frees `sknockd` from storing any per-flow state outside of netfilter. The number of initial netfilter rules is linear in the number of SILENTKNOCK-protected services, and future rules scale linearly with the number of active connections to protected services. While the number of rules may become large with many active connections, this can not be avoided, and we must rely on the efficiency of the underlying packet filter implementation to scale gracefully under load. Memory requirements for per-user keys (and pre-computed MACs) are linear in the number of users configured, and per-IP counter storage is linear in the number of client IP addresses.

Knockproxy. `knockproxy` reads a configuration file to find out which servers support SILENTKNOCK, and for which services (listed by destination (IP, port) pairs). The configuration file also includes the key shared with the server, and the last value of the connection counter (if this is the first time connecting to that server, the counter is initialized to 0). `knockproxy` registers interest for all SYN packets going from localhost to that (IP, port) pair. When it receives such a SYN packet (generated by the local TCP/IP stack), it computes a MAC using the server shared key and steganographically encodes the information in the TCP initial sequence number and timestamp. It then registers interest for all incoming and outgoing packets for that (IP, port) tuple, increments the associated connection counter, and sends the packet over the wire[8]. Since we have modified the sequence number from what the local TCP stack expects it to be, we must modify it again in the return packets before the TCP stack sees them, otherwise we will confuse the stack and reset the connection. Likewise, we must continue to modify all future outgoing packets for that connection, otherwise the remote host will reset the connection when it detects a sequence number mismatch. Once the connection is closed, `knockproxy` de-registers interest in that tuple (connection closure is detected the same way for both `sknockd` and `knockproxy`). The number of initial netfilter rules is linear in the number of SILENTKNOCK-protected services that might be contacted, Future rules scale linearly in the number of active portknocked connections.

3.3 Timing Analysis

The indistinguishability of the SILENTKNOCK implementation relies on the adversary gaining no information through timing attacks — if `sknockd` takes an overly long time to process packets, a smart attacker with knowledge of traffic timing before SILENTKNOCK was installed on a server would realize that some kind of additional processing is occurring (but not necessarily that SILENTKNOCK is in use). If the difference in timing is large enough, it makes for a good distinguisher for SILENTKNOCK in practice, even though timing information is not included in our formal model. On the other hand, if the timing difference is small (compared to timing noise between the adversary and the server — delays imposed by slower or overloaded routers, etc.) or the adversary

[8] The packet may be delayed, depending on the modification made to the timestamp field.

Table 1. Average time difference between receiving a SYN packet and emitting a SYN-ACK packet. The third experiment avoids the context switch incurred by user-space iptables manipulation, and gives a performance estimate for a `sknockd` kernel module.

Experiment	SSH only	`sknockd`	without commit
Average response time (μs)	242.86	389.33	295.44
St. Dev. (μs)	8.59	13.36	8.64
Slowdown factor	1	1.60	1.22

lacks precise knowledge of the timing characteristics of the server, this "side channel" will not lead to a good distinguisher in practice. Therefore, we have attempted to minimize this information leakage, and can minimize it further by implementing a number of optimizations, such as more aggressive pre-computation during idle time.

Results of our timings tests are shown in Table 1. We measure the time an SSH server running `sknockd` takes to process SYN packets and compare to an ordinary SSH server. We record the time between when the server receives a SYN packet (containing a valid MAC) and the time it sends a response (SYN-ACK) packet. The first column shows the baseline (standalone SSH server) time; the second column shows time with SSH and `sknockd` running together; the third column is similar to the second, except that `sknockd` has been modified to *not* make the iptables commit kernel call (`iptc_commit`), which inserts the iptables connection rule constructed by `sknockd` into the kernel packet filter table. We made this modification to simulate the amount of time the server would take to emit a SYN-ACK packet with `sknockd` running in kernel space, enabling direct manipulation of the packet filter table, without incurring the overhead of a user space to kernel context switch[9]. While servers running `sknockd` are always slower than servers running SSH alone, modifying `sknockd` to remove the iptables commit call reduces the time difference significantly.

Although information leakage (thought timing information) occurs in practice, the amount of information revealed is minor. Even using user-space `sknockd`, an adversary located a few hops away, and with perfect knowledge of the server timing distribution without `sknockd`, would need to witness several hundred accepted packets to gain a significant advantage in distinguishing `sknockd` from a dynamic firewall[10]; with the simulated kernel-space `sknockd`, the adversary is unlikely to detect the processing time difference unless he is located on the same LAN as the server. To further minimize this difference, we implemented AES pre-computation for Poly1305-AES nonces. At the moment we precompute only the initial counter value, but we can precompute and store values of the next several counters, allowing for verification to be performed without any online cryptographic computation.

While we do not test the client-side `knockproxy` for timing distinguishability, mainly due to time constraints, the use of `knockproxy` would be much more difficult to detect than `sknockd`. Since the processing of SYN packets occurs before

[9] We can currently account for at least 4 user-space/kernel context switches in `sknockd`.

[10] Due to the fact that 90% of Internet flows experience a standard deviation of 1ms or more in round-trip time [28], while the magnitude of timing difference even in the case of user space `sknockd` is only about 0.15 ms.

any observable event, and processing subsequent packets in a flow requires no manipulation of kernel data structures and no cryptographic computation, observable timing differences would be very small. If a remote adversary were to test for the presence of `knockproxy`, the largest observable effect would be in the re-transmit timeout, which may be altered by the packet delay imposed by timestamp modification. However, since retransmit clocks have granularity measured in seconds [26], and our timestamp modification has millisecond granularity, detection is unlikely.

4 Discussion

4.1 Limitations of SILENTKNOCK

Here we would like to note a number of limitations of our system. First, we only attempt to authenticate the start of a connection, but provides no guarantee that connections stay authentic. In other words, our system does not protect against connection hijacking (a well-known problem in TCP security) [29]. Furthermore, due to the limited bandwidth for authentication, SILENTKNOCK can only support symmetrically-keyed authentication. We believe it is up to the application to provide connection hijacking protection and relevant user authentication (e.g., SSH [6]).

Our solution relies on embedding stegotext in TCP/IP, and we are therefore limited in the size of the MAC field we can send. Currently, we only support 32 bits out of a 16-byte MAC. Furthermore, different operating systems have different TCP initial sequence number properties, and thus the amount of data that can be embedded in the SYN packet is highly dependent on the OS composing the packet. Thus, it is necessary that the server know the OS of the client in order to correctly extract the stegotext; alternatively, the server can attempt multiple extractions, but this will increase the cost of filtering and degrade security by a factor of the number of OSes supported.

Identities, Addresses, and NAT. In any distributed authentication system it is necessary to decide what the identities in a system correspond to. Three natural choices are to let identities correspond to network addresses, to physical hosts, or to human users. Our current implementation allows two options: identities (keys) may be associated either with IP addresses or users; each has different consequences for usability and security.

When identities are bound to IP addresses, we must assume that only a single client machine will be accessing a SILENTKNOCK-protected server from a given IP address, since a single counter is used for each identity. This assumption breaks down in the presence of NAT (network address translation) and similar devices. Therefore, in this scenario, we must limit our system to only one client per NAT. We stress, however, that unlike previous implementations, where NATs presented a security problem [15], adversaries sharing a NAT with a valid `knockproxy` client gain no advantage.

We also support associating identities with users by issuing a key to each user and checking the MAC on each SYN packet against each user's key. This can be done at essentially no extra computational cost due to the design of the Poly1305 MAC, which is computed by adding a keyed non-cryptographic hash of the message to the AES encryption of a nonce mod 2^{128}. Suppose we assign different AES keys (but a shared

non-cryptographic hash) to different users, and precompute the AES encryption of different users' counters, for the next ft values. Then, given a packet p with embedded tag t, we can check whether $t = \text{MAC}_{K,r}(p, n) = \text{Poly1305}_r(p) + \text{AES}_K(n) \mod 2^{32}$ for some user's key K and counter n as follows. We first compute $t - H(p) \mod 2^{32}$, and then we search for the resulting value in our table of precomputed encrypted nonces; if the value is found, we accept the packet and remove older encrypted counters for the same user. This search can be implemented in essentially constant time (with respect to the number of users) using a number of approaches, such as hash tables or tries. After accepting the packet, we insert the next precomputed nonce for the same user into the table. While this solves the NAT problem mentioned above, it causes security loss by a factor of the number of users (and thus the number of user keys) due to the requirement that we check the MAC against all user keys. Alternatively, once IPv6 is a viable alternative to IPv4, we may be able to use unique *target* IP addresses as part of the key, such that a server running `sknockd` has one IPv6 address per user.

Denial of Service. While we have implemented some measures to prevent distinguishing or denial of service attacks due to packet dropping, our scheme is vulnerable to a selective denial of service attack. An adversary who modifies *all* packets on a network by consistently rewriting sequence numbers or timestamps can cause MAC verification to fail at `sknockd`, while not impacting the status of most standard TCP traffic. We note that this attack is both expensive, in that it requires the attacker to touch every packet in — and maintain per-flow state for — all connections on a network, and may effect other protocols that authenticate the TCP header, such as IPsec [30] or TCP-MD5 [22]. Additionally, such selective denial of service is much easier for other port knocking or general IP service authentication schemes, as in those cases it is easy to identify knock sequences or authenticated packets and drop them, while maintaining no other state. Finally, if the server logs failed connection attempts, it will be easy to notice such attacks since, for instance, altering the timestamp will still give a 24-bit MAC match in the sequence number, which is unlikely.

4.2 Conclusion

Following our formal security model for port knocking, the SILENTKNOCK implementation provides a provably indistinguishable system with reasonable overhead, and an especially light load on the server. The system is currently usable by any Linux 2.6 application using TCP/IP as its network protocol, and is completely compatible with TCP/IP as described by relevant RFCs [26, 31]: it is possible for a client running `knockproxy` to connect to a server not running `sknockd`. Furthermore, since all "knocks" are destined for ports potentially providing services, the system is compatible with all currently-deployed firewalls, including host-based software firewalls.

We provide per-flow, *not* per-source (IP address) authentication, meaning that even if host A already has an active and authenticated connection to host B, a new connection from host A to host B (presumably using different outgoing port on host A's side) would need to be uniquely authenticated. Furthermore, all of the knock "sequences" we use are one-time (not replayable) since we employ a connection counter that is unique to every IP address, and thus every client.

4.3 Future Work

For future work, there are a number of implementation-level issues to address. The most pressing issue of these is porting `sknockd` to a kernel module, to eliminate the overhead of kernel/user space switching. Along with this conversion, we plan to implement several other optimizations, including more aggressive pre-computation. We expect that these modifications will further decrease overhead for `sknockd`.

Other possibilities for future work include the use of additional TCP/IP fields for steganographic embedding. For instance, under Linux 2.6, ephemeral TCP ports and IP IDs are assigned pseudorandomly per destination host, and change every five minutes. Thus, in an environment that requires a longer MAC, these fields could be utilized, gaining an additional 34 bits of authentication, at the expense of disallowing more than one connection to a given IP address and port per five minute period. Using the source port number and IP ID field, and limiting connections to once per five minute period would also allow extension of SILENTKNOCK to the UDP protocol, with 34 bits of authentication; unfortunately, the UDP header format does not include any other standard, variable elements, so 34 bits per five minutes seems to be an upper bound on the authentication strength for UDP.

Another important issue to address in the future is usability. Our current implementation is fairly configurable and relatively straightforward for computer scientists or system administrators to use. However, in order to be deployed widely, (say, as widely as VPNs), we will require a more friendly interface. A related issue that we have not addressed here is key management. It will be interesting to consider these issues in depth.

Acknowledgements. We thank Luis von Ahn, Yongdae Kim, David Molnar, Stephen Murdoch, and Hal Peterson for helpful discussions and comments regarding this paper. This work was supported by NSF grant CNS-0546162.

References

[1] Krzywinski, M.: Port knocking: Network authentication across closed ports. SysAdmin Magazine 12(6), 12–17 (2003)
[2] Barham, P., Hand, S., Isaacs, R., Jardetzky, P., Mortier, R., Roscoe, T.: Techniques for lightweight concealment and authentication in IP networks. Technical Report IRB-TR-02-009, Intel Research Berkeley (July 2002)
[3] Worth, D.: CÖK: Cryptographic one-time knocking. In: Black Hat USA (2004)
[4] deGraaf, R., Aycock, J., Jacobson, M.J.: Improved port knocking with strong authentication. In: Srikanthan, T., Xue, J., Chang, C.-H. (eds.) ACSAC 2005. LNCS, vol. 3740, pp. 451–462. Springer, Heidelberg (2005)
[5] Fluhrer, S., Mantin, I., Shamir, A.: Attacks on RC4 and WEP. RSA Laboratories, Cryptobytes 5(2) (2002)
[6] Bellare, M., Kohno, T., Namprempre, C.: Authenticated encryption in SSH: provably fixing the SSH binary packet protocol. In: Proc. CCS '02, pp. 1–11 (2002)
[7] Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 1–12. Springer, Heidelberg (1998)
[8] Hopper, N.J., Langford, J., Von Ahn, L.: Provably secure steganography. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 77–92. Springer, Heidelberg (2002)

[9] Murdoch, S.J., Lewis, S.: Embedding covert channels into TCP/IP. In: Barni, M., Herrera-Joancomartí, J., Katzenbeisser, S., Pérez-González, F. (eds.) IH 2005. LNCS, vol. 3727, pp. 247–261. Springer, Heidelberg (2005)
[10] Bernstein, D.J.: The Poly1305-AES message authentication code. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, Springer, Heidelberg (2005)
[11] Linux 2.6.17.13 kernel source. drivers/char/random.c
[12] Vasserman, E.Y., Hopper, N., Laxson, J., Tyra, J.: Silentknock (April 2007), http://www.cs.umn.edu/~eyv/knock/
[13] Krzywinski, M.: Port knocking, http://www.portknocking.org/
[14] Graham-Cumming, J.: Practical secure port knocking. Dr. Dobb's Journal (November 2004)
[15] Manzanares, A.I., Marquez, J.T., Estevez-Tapiador, J.M., Castro, J.C.H.: Attacks on port knocking authentication mechanism. In: Gervasi, O., Gavrilova, M., Kumar, V., Laganà, A., Lee, H.P., Mun, Y., Taniar, D., Tan, C.J.K. (eds.) ICCSA 2005. LNCS, vol. 3483, pp. 1292–1300. Springer, Heidelberg (2005)
[16] Ahsan, D.K.: Practical data hiding in TCP/IP. In: Proc. Workshop on Multimedia Security at ACM Multimedia, ACM Press, New York (2002)
[17] Rowland, C.H.: Covert channels in the TCP/IP protocol suite. First Monday 2(5) (1997)
[18] Conehead: Stego hasho. Phrack 9(55) (1999)
[19] MacDermid, T.: Stegtunnel, http://www.synacklabs.net/OOB/stegtunnel.html
[20] Ahn, L.v., Hopper, N., Langford, J.: Covert two-party computation. In: Proc. STOC '05, pp. 513–522 (2005)
[21] Bond, M., Danezis, G.: The dining freemasons: Security protocols for secret societies. In: Proc. 13th International Workshop on Security Protocols, Cambridge, England (April 2005)
[22] Heffernan, A.: Protection of BGP sessions via the TCP MD5 signature option (1998), http://www.ietf.org/rfc/rfc2385.txt
[23] Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional, Reading (2005)
[24] Ring, S., Cole, E.: Taking a lesson from stealthy rootkits. IEEE Security and Privacy 2(4), 38–45 (2004)
[25] Welte, H., Kadlecsik, J., Josefsson, M., McHardy, P., Kozakai, Y., Morris, J., Boucher, M., Russell, R.: The netfilter.org project, http://www.netfilter.org/
[26] Postel, J. (ed.): Transmission control protocol (1981), http://www.ietf.org/rfc/rfc0793.txt
[27] Carter, J.L., Wegman, M.N.: Universal classes of hash functions (extended abstract). In: Proc. STOC '77, pp. 106–112 (1977)
[28] Aikat, J., Kaur, J., Smith, F.D., Jeffay, K.: Variability in TCP round-trip times. In: Proc. IMC '03, pp. 279–284 (2003)
[29] Bellovin, S.M.: Security problems in the TCP/IP protocol suite. SIGCOMM Comput. Commun. Rev. 19(2), 32–48 (1989)
[30] Kent, S., Atkinson, R.: IP authentication header (November 1998), http://www.ietf.org/rfc/rfc2402.txt
[31] Jacobson, V., Braden, R., Borman, D.: TCP extensions for high performance (1992), http://www.ietf.org/rfc/rfc1323.txt
[32] Shoup, V.: On fast and provably secure message authentication based on universal hashing. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 313–328. Springer, Heidelberg (1996)
[33] Boneh, D., Franklin, M.: Identity-based encryption from the weil pairing. SIAM J. Comput. 32(3), 586–615 (2003)