Techniques for Lightweight Concealment and Authentication in IP Networks

Paul Barham, Steven Hand, Rebecca Isaacs, Paul Jardetzky, Richard Mortier, and Timothy Roscoe

IRB-TR-02-009

July, 2002

Intel Research Berkeley

Techniques for Lightweight Concealment and Authentication in IP networks

Paul Barham
Microsoft Research
7 JJ Thomson Ave.
Cambridge CB3 0FB, U.K.

Steven Hand
U. Cambridge Computer Laboratory
15 JJ Thomson Ave.
Cambridge CB3 0FD, U.K.

Rebecca Isaacs
Microsoft Research
7 JJ Thomson Ave.
Cambridge CB3 0FB, U.K.

Paul Jardetzky
Sprint ATL
1 Adrian Court
Burlingame CA 94010, USA

Richard Mortier
Microsoft Research
7 JJ Thomson Ave.
Cambridge CB3 0FB, U.K.

Timothy Roscoe
Intel Research
2150 Shattuck Ave.
Berkeley CA 94704, USA

Abstract

This paper argues that there is significant benefit in providing multiple progressively stronger layers of security for hosts connecting to the Internet. It claims that this multi-layered approach allows early discard of packets associated with attacks. This reduces server vulnerability to computational denial-of-service attacks via heavyweight cryptography calculations. To this end, it presents three techniques that allow TCP/IP services to be concealed from non-authorized users of said services, while still allowing access to the services for authorized users. These techniques can be entirely implemented at the edges of the network and require no changes to the interior of the network. They work alongside, and augment, existing protocols making deployment practical.

1 Introduction

The risks associated with connection to the Internet are increasingly clear. Well-publicized incidents of systems being targeted by malicious intruders, and of large scale denial-of-service (DoS) attacks have highlighted the risks of networked systems. The use of automated attack tools [8], and vulnerability to denial of service is a real problem for high profile commercial sites, such as Amazon or Microsoft, as well as for smaller businesses, educational institutions and government organizations. A survey across a variety of industry sectors conducted by the Computer Security Institute [19] reveals that 90% of the respondents detected computer security breaches within the last 12 months and 80% of those acknowledge financial loss. Furthermore, analysis [18] has shown that a significant fraction of attacks are directed at home machines and against network infrastructure.

Established techniques to protect Internet-connected machines tend to rely either on filtering packets, or on application-level security. The first technique is implemented by firewalls, Internet-connected devices running software whose job is to filter or log unwanted network traffic. However, there are common attacks against which a firewall cannot protect. For example, firewalls do not protect against attempts to exploit bugs in application-level software. Such vulnerabilities occur because the Internet architecture assumes that services bound to a port should be accessible by any machine using the Internet protocols.

The second technique is to deploy high-strength application-level security mechanisms. However, authenticated services are built above the network layer [13, 6, 16] and are often themselves subject to attack once discovered on a host. Reliance solely on application-level security exposes the problem of the computational expense of such security mechanisms. The high computational burden of commonly deployed cryptographic schemes leaves the server open to computational DoS attacks. Furthermore, complex schemes are error prone, as exemplified by the integer overflow bug in SSH [4] which has been the target of recent attacks[1].

[1] Ironically, this bug occurred in a section of code responsible for defending against attacks which themselves exploited an

Alternative techniques, such as those used by next-generation Internet Protocols [1, 5, 11] incorporate authentication headers at the network layer. However, these are still open to computational DoS, and their key exchange mechanisms also reveal the existence of the machine and the services it is running.

Consequently, we argue that there is a need for computationally cheap and simple defense mechanisms that allow early rejection of the majority of attacks. In particular, we argue that there is significant benefit in having multiple, progressively stronger, layers of security, rather than attempting to have a single 'perfect' security layer. In more detail, when protecting a particular service:

1. The service should be *hidden*. This allows hosts to attempt to be invisible to other Internet-connected machines while still providing service to authorized parties.

2. Authorized source credentials should be easy to validate, yet difficult to forge. At the most basic level we can use identifiers from a sparse address space, but more generally we use a one-way authentication function based on some secret key. For the service to remain hidden, this stage should elicit no response if the credentials are invalid.

3. Full-strength application-specific security mechanisms are still used to provide true end-to-end authentication.

Although extremely lightweight, service hiding does deter attacks initiated by random port scanning, making it harder to exploit application-level vulnerabilities. This is not 'security by obscurity' — rather, it is akin to authenticated signaling [17, 23] where application authentication information is included in the signaling message.

Recent interest in defining the security requirements for Internet devices also includes the ability to 'stealth devices' [10]; we envisage using the schemes presented in this paper to secure network elements such as routers, firewalls and proxies. They may also be useful for subscription-based services accessed by large numbers of authorized users and for whom the cost of DoS attacks may be severe.

In summary, we propose a lightweight *authenticated one-way signaling* mechanism that allows a discard decision to be made as early as possible. Such discard decisions should ensure that, with high probability, unauthorized clients not only receive no service, but no response at all. The following section details our assumptions and outlines problems that occur when current pieces of Internet infrastructure interfere with these assumptions. We continue with the development of three variations on the basic idea presented in Section 3, and discuss the relative trade-offs in Section 5.

2 Architectural Assumptions

Before detailing the assumptions underlying the techniques presented later in this paper, it is important to emphasize that we intend to augment, not to replace, the function of the stronger authentication mechanisms in IPsec, SSL and the like. We then make the following assumptions:

- We assume that keys can be shared and distributed out-of-band. In particular, we deem the problem of key exchange to be outside the scope of this paper.

- We assume that modifications to the kernel on server machines are possible. Existing client applications may remain unmodified as authentication packets that enable connections to the server from existing client applications can be generated by a separate application.

- We assume an attacker that has difficulty snooping packets in the middle of the Internet, but is able to forge or spoof packets from an endpoint. An attack scenario comprises one or more Internet end-nodes that probe and launch attacks on hosts with visible services.

- We assume that a reasonable proportion of packets introduced by a given source and addressed to a particular destination are delivered to that destination without modification.

earlier vulnerability caused by CRC weaknesses.

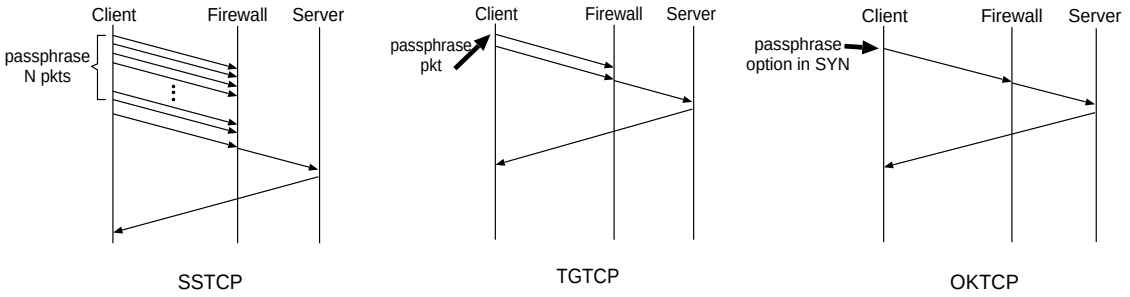

Figure 1: A high-level depiction of the three techniques.

Perhaps the most contentious of the above assumptions is the latter. Client machines are often themselves behind a firewall, NAT, or proxy device, each of which impacts the latter assumption. Note that only the outbound behavior of such devices is an issue since the passphrase mechanism is unidirectional.

Where a router is used as a firewall, it is typically configured as a simple filter, and passes or discards outbound packets based on access control lists enforced using pattern matching. Allowing arbitrary *outbound* packets is often not regarded as a significant security risk: it allows information leakage, already possible for malicious programs using, for example, outbound HTTP connections. Incoming filters are typically tighter, often discarding ICMP messages and packets containing IP options. These devices are typically configured to block the majority of TCP and UDP ports.

Network address translation devices pose more of a problem for connection authentication mechanisms. By definition they change the source IP address and port number, making end-to-end authentication of the source more difficult. Ideally the source address of the connection should be validated as part of the authentication mechanism to prevent replays and spoofing attacks.

Finally, firewall or application-level proxies based on variants of the SOCKS protocol [15] have the greatest impact on our assumption that a reasonable proportion of transmitted packets reach their destination unmodified. SOCKS uses a simple RPC-like control protocol to open holes in the firewall dynamically and terminates outbound TCP connections internally. A `CONNECT(externalIP, destport)` call returns an internal firewall `(IPaddress, destport)` and the client then makes a standard TCP connection to the specified internal address. The proxy server then initiates an external TCP connection. Similar mechanisms exist for negotiating externally visible `(IPaddress, port)` pairs for incoming TCP connections and incoming UDP datagrams. The server is thus an application-level bidirectional relay between two separate (internal and external) TCP connections.

Since there is no 1-to-1 packet correspondence between the two TCP connections, any scheme relying on out-of-band information encoded in packet headers will not pass through such a proxy. Both SOCKS and the Microsoft Proxy Protocol function in this way and support only UDP and TCP. A SOCKS-like proxy where the control protocol merely establishes an appropriate entry in a NAT forwarding table is conceivable, but not presented here. Section 4 discusses how to work around the specific problems posed by the presence of middle-boxes between the connection principals. It should be noted that such middle-boxes do not invalidate either the argument for multi-layered security or that for service hiding. Rather, they make robust implementation and deployment of the particular techniques presented in this paper more difficult.

Under these assumptions and interactions with various intermediary network devices, we investigated a number of alternative security mechanisms. These are presented in the following section. Section 4 discusses the design trade-offs of the respective schemes and Section 5 presents some experimental evaluation of one of the schemes.

3 Implementation Alternatives

This section presents a selection of implementation techniques that satisfy the goals outlined in Section 1. All effectively implement the same basic scheme: viz. require a client desiring access to a service to authenticate themselves via presentation of a secure hash in such a way that it is cheap to discard invalid attempts. In all cases, the hash is time-dependent to reduce the opportunity for replay attacks. It is 128 bits in length and generated using the SHA1 algorithm, a shared secret, and in some cases, additional information.

A Silent Authentication Service (SAS) logically residing within the destination firewall[2] observes incoming authentication attempts, discarding most packets without response. When the correct hash is presented, a 'hole' in the firewall is opened for a particular client (i.e. a particular source IP address and source port) for a short period of time.

Note that only TCP SYNs are intercepted by the firewall; TCP data segments are always allowed through as these will be discarded cheaply by the server. TCP RSTs and ICMP unreachables prompted by unwanted traffic are dropped by the firewall in order to maintain the stealth goal.

Figure 1 depicts the 3 schemes we have developed: Spread-Spectrum TCP (SSTCP), Tailgate TCP (TGTCP) and Option-Keyed TCP (OKTCP). In essence, these span the spectrum between sending several unmodified additional packets for connection setup, and sending just the traditional number in a modified (although RFC-compliant) fashion. We now describe each variant in more detail.

3.1 Spread-Spectrum TCP

In Spread-Spectrum TCP (SSTCP), the client presents the key to the SAS module by modulating a TCP header field in a sequence of SYN packets. Once a correct sequence is observed, the firewall allows the normal TCP three-way handshake to proceed normally. The field chosen must not be modified end-to-end, giving two viable options: the destination port number (DPN) and the initial sequence number (ISN).

The DPN provides 16 bits that can be modulated fairly arbitrarily, subject to constraints that may be imposed by firewalls on the client side. The ISN provides 32 bits and is less likely to interact badly with middle-boxes, but is unlikely to be preserved by proxies. The DPN was chosen for the implementation evaluated in Section 5 as it allowed the client software to be implemented using standard library calls.

The prototype SSTCP implementation calculates a sequence of destination port values using a secret key K shared between the server and all clients and a monotonically increasing time value. It sends the sequence of N SYN packets to the relevant ports on the server and if at least $M \leq N$ of these are received by the SAS module in the correct order, a SYN from that client is allowed to pass through the firewall to the appropriate 'real' port. Both N and M are software configurable but N must be pre-agreed by client and server along with the secret key.

Our prototype is implemented over Linux v2.4.18 and generates a logically infinite array $\mathcal{A}$ of port values using a one way function. Each generated element corresponds to an interval of time, T: every T a new element in this infinite sequence is generated as $\mathcal{A}[t] = SHA(K.t)\&\texttt{0xFFFF}$ (where '.' denotes concatenation and t is the current time divided by T). The amount of work calculating the sequence and authenticating each packet is thus minimized and independent of the network traffic directed at the server, reducing vulnerability of the scheme itself to DoS attack. In practice $\mathcal{A}$ is a circular buffer of 256 16-bit elements as depicted in Figure 2. Hence, the space overhead is also small and constant.

A test client sends a sequence of setup attempts to port numbers $\mathcal{A}[t], \ldots, \mathcal{A}[t+N-1]$ preceding the real connection setup. When the SAS module sees a SYN packet from a new client (identified by a <source IP address, source port> pair), it checks the presented destination port number d against $\mathcal{A}[t-1]$, $\mathcal{A}[t]$ and $\mathcal{A}[t+1]$. If there is no match, the packet is dropped. Otherwise the module records the follow-

[2] In home environments, firewall functionality is often provided in software on the server machine, e.g. Windows XP Internet Connection Firewalling.

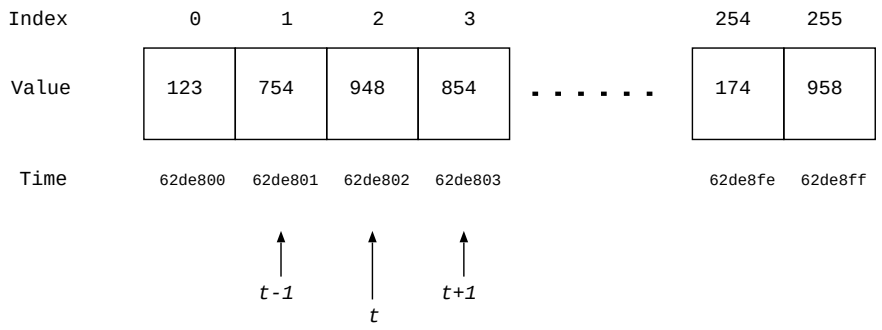

Figure 2: The SSTCP Authentication Table.

ing 3 bytes of information for this client: `start` := i for which $\mathcal{A}[i] = d$; `end` := `start` $+N$; and `match` := 1.

When the module sees a SYN packet from client in progress, it checks the presented destination port number d against $\mathcal{A}[\texttt{start}+1], \ldots, \mathcal{A}[\texttt{end}]$. If there is no match, it destroys the client state. If there is a match then `start` := i for which $\mathcal{A}[i] = d$, and `match` is incremented. Only if $\geq M$ ports `match` (in order) is the 'real' SYN is allowed through.

3.2 Tailgate TCP

In Tailgate TCP (TGTCP), the key is presented to the SAS module in a single packet, followed closely by the SYN to the server. If the key is correct, then the SAS module opens a window in the firewall for a short period of time, allowing the 'tail-gating' SYN through to reach the server. Note that the window in the firewall need only be opened for a short period of time as the firewall is assumed to pass TCP data segments by default, as they can be cheaply and silently discarded at the server.

In more detail, the client sends the key as a packet containing: (T, C, S, D, H), where: T is the current time; C is a client identifier, taken from a sparse name space; S is the source (client) IP address; D is the destination address and port number; K_c is a shared secret key between client C and the firewall; and H is $SHA1(T, S, D, K_c))$.

The SAS module performs an initial check of T against the current time, allowing rapid discard of clumsy replay attacks. C is simply checked against a short list of known client IDs, allowing rapid discard of the majority of 'random' attacks having no snooping ability.

If these first two tests are passed, implying an attacker has snooped or guessed C from a 64 bit address space, the module computes H and compares it to the value in the packet. Should these values match, then a hole is opened in the firewall for SYNs from S destined for D by adding an entry (T, S, D, H) to a list ordered by T. Entries are removed from the list at time $T + T_c$, where T_c is, e.g., 2 seconds[3].

When a SYN arrives at the firewall, the list of open ports is scanned for a matching entry and the current time compared with $T + T_c$. The SYN is forwarded if there is a match, and dropped if not. Under normal operation, client-side software will intercept calls to the `connect()` socket API and send a key packet to the SAS module of the destination firewall. The subsequent SYN will then tailgate the key packet through the firewall.

The timing of the transmission of the tailgating SYN is a potential issue: if the SYN tailgates too closely, then packet re-ordering may mean that it arrives before the key and will thus be discarded; if it tailgates too late, then the firewall hole may close before the SYN arrives. With the firewall hole open for 2 seconds, a value for inter-packet transmission of 500 ms seems appropriate.

Choice of encapsulation for the key packet is influenced by presence and type of client-side firewalls.

[3]This value is configurable by the server and does not need to be known by the client.

The simplest choice is a UDP datagram sent to the firewall IP address at a well-known port. However, such a packet may well be blocked on the outbound path by any firewall between the client and the Internet. Other options include the generation of a TCP SYN with a non-zero data portion carrying the passphrase. Since the passphrase is intended only for the SAS, the SAS module could even strip out this 'bogus' data portion before allowing the (now valid) SYN to continue to the server.

Finally, although replay of the above packet is an idempotent operation, replay attacks may be detected by comparing the hashes of incoming packets against the stored hashes in the open port list.

3.3 Option-Keyed TCP

In Option-Keyed TCP (OKTCP), the key is encoded in a suitable IP or TCP option field in the SYN from the client to the server. Suitable options include the IP Timestamp option and the TCP Echo option, though both are effectively limited in size to 36 bytes. If the SAS module in the firewall detects a correct key, it immediately allows the SYN to proceed obviating the need for any state allocation. It may also choose to remove the option to prevent confusion of the destination IP stack.

The prototype implementation uses the IP Timestamp option, which may contain multiple 32-bit timestamp values. The key is inserted as an appropriate number of timestamps, and the timestamp buffer flagged as being 'full' so that routers en-route do not try to add any further timestamps. On Linux, this 'bogus' option can be added explicitly from user-space. Windows XP only allows 'empty' timestamp options to be added, requiring the option to be inserted using an IP Firewall Hook device driver.

The SAS module simply parses an option in a TCP SYN packet to extract the client's notion of time and a 256-bit hash of a shared key concatenated with other information as in TGTCP[4]. The time sent, t, is verified to be within an acceptable window, the hash function then applied to the client's notion of time and the result compared with that sent. If there is a match then the SYN is accepted.

We can allow client differentiation and lightweight early discard by including a per-client or per-group identifier C in the option. This also makes it easier to revoke keys. If the space limitation in the TCP header was too stringent, then the same technique of appending a data portion to the SYN as in TGTCP could be used.

4 Design Trade-Offs

The three schemes presented above are intended to illustrate certain design trade-offs. The schemes are discussed in general terms below, while the succeeding section presents some performance figures from one of the schemes (SSTCP). It should be noted that simple prototypes of the other two schemes have also been constructed.

In SSTCP the client sends a particular sequence of packets culminating with a TCP SYN packet. If enough of the correct sequence is received by the host, the 3-way handshake proceeds as normal. If not, the port appears dark. The sequence represents a 'passphrase' which is lightweight (to minimize DoS vulnerability) key-dependent (to prevent correct port generation by unauthorized hosts) and time-dependent (to minimize the opportunity for replay attacks).

The server implementation requires only a small fixed overhead (memory and time) in the idle case. For 'bad' SYN packets the cost is a table lookup and 3 comparisons and even for 'good' packets the space and time overhead is not large. The scheme is also resilient to packet loss of up to $(M - N)$ packets, handles clock skew of at least T between client and server, and operates even when the client is behind a NAT box.

However it does have certain problems including:

- it may not pass firewalls because of its use of random destination ports;
- its inability to dynamically choose M, N based on client preference, network loss probability,

[4] Note that the 36 byte option limit requires a more compact encoding than with TGTCP.

etc.;

- there is a small probability ($3/2^{16}$) that the correct initial port value is guessed by an attacker, thus generating state which requires space and management;
- the possibility of many source port allocations when using PNAT;
- vulnerability to a trivial replay attack within time $(N - M + 0.5) * T)$.

Some of these issues are mitigated by modulating the ISN instead of destination port, but the last is not. To address the last issue, we could use nonces to "remember" recent connection set ups, but this adds more complexity and more state. Another approach is to use a nonce (a well chosen ISN) in the first matching packet of the sequence to perturb later sequence values. Again, this is at a cost of increased computational complexity after the inexpensive early recognition mechanism.

SSTCP sacrifices some robustness by requiring that all passphrase data be conveyed in existing and small fields in the TCP header for "stealth" purposes. TGTCP takes an alternative approach by concatenating all of the information carried in these fields into the data payload of a conventional IP packet, which when received and verified by the firewall allows a following SYN packet to pass through. Although this scheme is more resilient to packet loss than the SSTCP mechanism and is much more efficient in terms of packets transmitted, it is sensitive to the timing of the transmission of the tailgating SYN.

A further concern for TGTCP is that the passphrase packet is potentially blocked by aggressive firewalls at the client end. This problem can be avoided by choosing an encapsulation which is very likely to be 'allowed out' – for example an ICMP echo packet, or a TCP data segment destined for port 80. However, the use of TCP encapsulation introduces two problems when the client is behind a NAT device:

a) It requires the passphrase to be encapsulated in (or be preceded by) a SYN packet and will therefore create an entry in the NAT table.

b) The passphrase packet is tied to a source IP address to prevent a snooping attacker from replaying the passphrase from his own machine. This IP address would need to be the public address of the NAT device.

OKTCP is an extension to the existing TCP connection state machine to include an IP or a TCP option on SYNs which contains the result of a hash function applied to a time varying and keyed quantity, similar to that used in SSTCP. The major advantage of this scheme over SSTCP and TGTCP is the atomicity with which one gains access to the service. That is, there is no added window of vulnerability between the authentication and the acceptance of a connection. Of course, replay attacks within time X, or thereabouts, are still possible (although can be bypassed with a nonce scheme). In addition the race conditions and potential NAT issues with TGTCP are avoided.

It is not clear whether is it preferable to use an IP or TCP option to implement OKTCP. Whilst it is trivial to invent a new TCP option number with an opaque payload, the *data offset* field in the TCP header effectively limits the maximum option size to 40 bytes, of which a large number may already be consumed by MSS and SACK negotiation options. None of the valid TCP options have the ability to carry a large opaque payload. The closest is the TCP Echo option described in [9] which allows only a 4 byte payload.

Perhaps the biggest problem with this scheme is that many firewalls discard incoming packets with IP options by default, since they potentially include dangerous IP source routing options. It is also unclear how an opaque TCP option will be treated by the network. If the packet successfully makes it through the Internet to the destination firewall then it may be stripped out before forwarding the packet onto the internal network. More experiments are necessary to determine how well this may work.

Notwithstanding these issues, each of the three alternative schemes does achieve the principle goal of supporting progressively stronger layers of security for end systems, thus reducing the costs experienced under a DoS attack. However the widespread violation of the 'end-to-end' principle in the Internet does mean that the robust implementation and deployment of any such scheme involves careful consideration of the relevant network configuration and

the performance and security trade-offs involved.

5 Experimental Evaluation

This section evaluates the trade-offs made by the different schemes in terms of three properties of note:

- *stealthiness*: how easily may an attacker discover the existence of a service hidden by the SAS;
- *robustness*: we identify two types of robustness, namely how robust is the scheme to the interposing of middle-boxes of various types, and how robust the scheme is to attacks of various types; and
- *effectiveness at DoS mitigation*: how lightweight is the scheme, and thus how does it improve the security of the service with respect to server DoS attack.

5.1 Stealthiness

Stealthiness is reckoned by the ease by which an attacker may discover the existence of a service hidden by the firewall. In all the schemes no response is elicited from the firewall or server until a valid key has been presented. However, if an attacker can eavesdrop on legitimate clients (i.e. a man-in-the-middle attack), they can deduce the service port after the scheme succeeds. Use of hashing in all schemes ensures that an attacker should still find it difficult to actually gain access to a service whose existence is inferred. We claim that this can dramatically reduce the rate of attack attempts against a given service instance. In consequence, all three varieties are deemed to have good stealth properties.

5.2 Robustness

Robustness is a general term used here to cover a variety of potential issues, such as interaction with NATs, firewalls, and proxies, and the effects of packet loss and variation in network latency. Both SSTCP and TGTCP are vulnerable to packet loss and reordering due to variation in path and latency: in SSTCP this can slow down the speed of authentication, and in TGTCP it requires retransmission of both key and SYN packets. OKTCP does not require any extra mechanisms to deal with packet loss over existing TCP retransmission behavior.

SSTCP may have poor interactions with NAT boxes as the many SYNs carrying the key are likely to cause allocation of many entries in the forwarding tables of the NAT.

5.3 Effectiveness at DoS Mitigation

Resource usage during a DoS attack is dominated by the cost of rejecting a bad connection setup. To illustrate the relative resource utilization, we instrumented the linux kernel on a 450Mhz Pentium II processor with calls placed to record the cycle timer in the code responsible for checking destination port numbers against the calculated secure hash values kept in the kernel. A user-space openSSH SSHD server was also instrumented to record timings for both RSA and DSA cryptographic authentication. In particular, cycle values were extraced around the calls to `RSA_verify` and `DSA_do_verify` in libcrypto.so.

	CPU cycles	Std. dev.
SSTCP Packet Check	212	36
SSTCP Port Calculation	6,676	1,470
RSA Verification	757,590	30,546
DSA Verification	11,932,745	174,243

Table 1: Time for cryptographic operations.

Table 1 shows the relative number of average processor cycles necessary to perform authentication steps. For SSTCP, this is the time to perform three 16 bit lookups in a table of pre-calculated hash values. In contrast, we show the time taken in the Secure Shell Daemon (SSHD) RSA and DSA encryption steps after key exchange has occurred. For illustrative purposes, we show how long it takes to calculate one 16 bit table hash value and a 128 bit key (this is performed once every 10 seconds to keep the lookup table up-to-date). We include the standard deviation for the results to show variation that may occur from

operating system scheduling and processor context switches.

We see between three to four orders of magnitude difference in the amount of CPU resource necessary to discard an unwanted connection.

To look at resource usage during an emulated TCP SYN flood, the target machine was attacked using a single machine generating TCP traffic to port 22 as fast as possible. This traffic was then rate limited to 1.5Mbps using a router between the assailant and victim.

The receiver must service the network interrupts, process the packets and make appropriate responses. In particular, under normal circumstances, TCP state is allocated on the receipt of a valid TCP SYN, timers are allocated, and a SYN/ACK is sent to the originator of the connection. If an ACK is then received, the kernel returns from an accept() call and allows the application the send and receive data where user level authentication may take place.

Using SSTCP, TGTCP and OKTCP, we, with high probability, reject unauthenticated clients early without the need to allocate a Transmission Control Block or a packet buffer to send a SYN or an ACK. Instead a good hash function is used to generate the port value to check against.

	KB Memory Used
SSTCP Enabled	32968
SSTCP Disabled	39068

Table 2: Memory usage under TCP/SYN attack.

Table 2 shows the memory usage of a server under a sustained TCP SYN attack at 1.5 Mb/s. The numbers reported are an average over a 10 minute interval with the SSTCP mechanism enabled and with the mechanism disabled. The measurements were taken with the systat package recording values every 10 seconds at relatively low CPU utilization (20% in both cases). The results show approximately a 20% savings in allocated memory at the same CPU utilization.

Under load, we observed that CPU utilization is dominated by interrupt service overhead when we allow either the TCP stack or the SSTCP module to discard bogus SYNs. However, from Table 1 we can see that a DoS attack at the application level would quickly overwhelm the CPU.

Both TGTCP and OKTCP require no additional state in the kernel. SSTCP, however, allocates 18 bytes of state for a successful match until the sequence is complete. The probability of a guessing the correct initial port value is $3/2^{16}$ and if any subsequent packet in the sequence fails to match, the space is reclaimed immediately. We believe this is an acceptable cost for the additional benefits.

6 Related work

The use of one-way functions for message authentication is well suited to this application and has been presented previously in [7, 22, 2].

Connection-oriented networks with out-of-band signaling can hide the existence of a service access point, but do so by interposing a trusted entity [20] between the client and server: the signaling system itself (controlled by the network provider). The problems addressed above are thus simply shifted onto the service provider, who must ensure both that the signaling system is secure and that it implements the security policies desired by the customers.

Networks and protocols lacking a central signaling entity (such as TCP/IP) must handle authentication in end systems. Some work in this area is mentioned in Section 1. Unfortunately, this functionality is currently placed in the network stack above the layer responsible for establishing a transport-layer connection (TCP). IPSec [11] includes authentication mechanisms, but does not address multi-layered security, the early discard of attack packets, or the ability to stealth services.

Thus, since authenticity can only be established after the connection has been set up and well-known port numbers are used for most Internet services, the host leaks information as to the services it offers, and must commit significant system resource (a TCP control block) before it can decide if it wishes to talk to the peer host.

As mentioned earlier, the SOCKS protocol [15] pro-

vides authenticated firewall traversal by means of an RPC-like control protocol over TCP. Our approach requires no signalling from the server (or an intervening network) back to the client, and eliminates the visibility of the SOCKS control port, itself a potential vulnerability particular when strong authentication is used for SOCKS connection setup.

The Firewall Control Protocol [14] (FCP) has been proposed by the IETF IP telephony work to allow applications (the SIP signaling system) to open pinholes in a firewall. This is useful for allowing dynamic IP telephone calls through a firewall – a different objective from stealth authentication. The IETF Middle Box Communication working group is also evolving a similar framework [21] to generalize the approach to many applications.

TCP *SYN cookies* [3] are a well-known and effective technique for mitigating the effects of SYN-flood attacks on servers. The creation of the TCP Control Block is delayed until the 3-way handshake is complete; instead, the server returns an ACK with an initial sequence number which is a cryptographic hash of the incoming information in the SYN packet, plus a secret and a counter that changes every minute. SYN cookies were devised for the case of public servers where a shared secret between clients and server is neither feasible nor desirable, and concealment of the service's existence is explicitly not a goal. SYN cookies also provide no protection against more sophisticated attacks where the attacking machines set up TCP connections rather then simply ending SYNs.

In contrast to SYN cookies, the techniques in this paper are intended for cases where clients of the service are authorized in advance. Our techniques not only defend against arbitrarily sophisticated Denial-of-Service attacks by low-cost filtering most unauthorized traffic before it reaches the service itself, but also help conceal the existence of the service to port scanners and the like.

Recently, *Secure Overlay Services* [12] or SOS have been proposed as a method of securing IP communication against Denial of Service attacks. The assumptions of SOS are strikingly similar to ours: a pre-determined (and pre-authorized) set of clients who should be allowed to access a service, and attackers with considerable resources but without prior knowledge of the location of the service. The SOS technique works by having the service only accept connections from a small number of authorized nodes, which participate in a much larger (several thousand node) overlay network. A client makes a connection to some overlay node, which then routes the connection to one of the authorized nodes over $\log(N)$ unpredictable hops in the overlay, where N is the total number of nodes. Much of the SOS work is concerned with how the system performs in the presence of some fraction of compromised overlay nodes.

We observe that SSTCP, OKTCP and TGTCP all solve the same problem without the need to deploy a thousand-node overlay network, and without incurring the overhead of multi-hop overlay routing. One conceivable advantage of SOS over our techniques is that since SOS is filter-based, the filters can be pushed out from the server to an unmodified router at the other end of the access link, making it less practical to attack the system by simply saturating the link with traffic.

7 Conclusions

We have presented Spread-Spectrum TCP, Option-Keyed TCP, and TailGate TCP as design alternatives for silent authentication of clients of an Internet service. Under these schemes, the service is invisible to port scans, and highly resistant to Denial of Service attacks which attempt to exhaust server CPU resources in connection setups (particular in end-to-end authentication checks). Experimental evaluation of SSTCP has demonstrated its feasibility and effectiveness.

The lightweight authentication provided by these schemes is intended to complement the full end-to-end security check provided by protocols like SSL and SSH to provide a measure of "defense in depth" for Internet services. The benefits brought be these schemes are:

1. *reduced opportunity for attack attempts* since exploits are less likely to be attempted on machines that do not appear to exist, or which do

not appear to be running any services;

2. *reduced vulnerability to attacks* since an attacker must circumvent our scheme before proceeding with the attack; and

3. *reduced cost of rejecting attackers* since some large fraction of packets may be discarded after ultra lightweight checks.

These benefits have been achieved by modifying only end-points, requiring changes to neither routers nor middle boxes. Although the presented schemes apply only to TCP, it is straightforward to extend these approaches to UDP-based protocols which use other signalling protocols to setup connections, such as RTP signalling using RTSP. We believe that this approach has wider applicability in evolving Internet protocols.

8 Acknowledgements

We would like to thank Bryan Lyles, John Larson, Christoph Schuba, Anthony Joseph, Gene Spafford, Richard Clayton and Ross Anderson for reading and commenting on earlier drafts of this paper. Their critical eyes helped hone the position of the ideas and contribution. Special thanks go to Travis Dawson for helping launch our DoS attacks.

References

[1] R. Atkinson. RFC 1826: IP Authentication Header, August 1995.

[2] M. Bellare, R. Canetti, and H. Krawczyk. Message authentication using hash functions: the HMAC construction. *CryptoBytes*, 2(1):12–15, Spring 1996.

[3] D. J. Bernstein. SYN Cookies, February 2002. Available as `http://cr.yp.to/syncookies.html`.

[4] CERT advisory CA-2001-35: Recent activity against secure shell daemons, December 2001. Available as `http://www.cert.org/advisories/CA-2001-35.html`.

[5] S. Deering and R. Hinden. RFC 1883: Internet Protocol, Version 6, Specification, December 1995.

[6] T. Dierks and C. Allen. RFC 2246: The TLS Protocol Version 1.0, January 1999.

[7] L. Gong. Using One-way Functions for Authentication. *Computer Communication Review*, 19(5):8–11, 1989.

[8] J. S. Havrilla. Types of Intruder Attacks, April 2002. Available as `http://www.cert.org/present/cert-overview-trends/module-4.pdf`.

[9] V. Jacobson and R. Braden. RFC 1072: TCP Extensions for Long-Delay Paths, October 1988.

[10] G. M. Jones. Internet-draft: Network security requirements for devices implementing internet protocol, June 2002. <draft-jones-netsec-reqs-00.txt>.

[11] S. Kent and R. Atkinson. RFC 2401: Security Architecture for the Internet Protocol, November 1998.

[12] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proceedings of ACM SIGCOMM'02, Pittsburgh, PA, USA*, pages 61–72, August 2002.

[13] J. Kohl and C. Neuman. RFC 1510: The Kerberos Network Authentication Service (V5), September 1993.

[14] J. Kuthan and J. Rosenberg. Internet-draft: Middlebox communication: Framework and requirements, November 2000. <draft-kuthan-fcp-02.txt>.

[15] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. RFC 1928: SOCKS Protocol Version 5, March 1996.

[16] J. Linn. RFC 2078: Generic Security Service API Version 2, January 1997.

[17] B. Lyles. Authenticated Signalling, T1S1.5/94-118, 1994.

[18] D. Moore, G. M. Voelker, and S. Savage. Inferring internet Denial-of-Service activity. In *Proceedings of the 10th USENIX Security Symposium*, pages 9–22, Washington, DC, USA, August 2001.

[19] R. Power. 2002 CSI/FBI Computer Crime and Security Survey. *Computer Security Issues and Trends*, 8(1), 2002.

[20] T. Russell. *Signaling System 7*. McGraw-Hill, 1995.

[21] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan. Internet-draft: Middlebox communication architecture and framework, February 2002. <draft-ietf-midcom-framework-07.txt>.

[22] G. Tsudik. Message authentication with one-way hash functions. In *Proceedings of IEEE Infocom 1992, Florence, Italy.*, May 1992.

[23] J. Xu and M. Singhal. Certificate path generation protocol (cpgp) for authenticated signaling in atm networks. In *Proceedings of the 6th International Conference on Network Protocols (ICNP 1998), Austin, Texas.*, pages 282–289, 1998.