ESSTCP: Enhanced Spread-Spectrum TCP

Amir R. Khakpour, Hakima Chaouchi
Institut National des Télécommunication (INT) Evry, France.
Email:{amir.khakpour, hakima.chaouchi}@int-evry.fr

Abstract

Having stealth and lightweight authentication methods is empowering network administrators to shelter critical services from adversaries. Spread-Spectrum TCP (SSTCP) [1] is one of these methods by which the client sends an authentic sequence of SYN packets to the server for authentication. Since SSTCP have some certain drawbacks and security flaws, we propose an enhanced version of SSTCP (ESSTCP) which modifies the original algorithm to reduce the computational cost and cover its vulnerabilities from denial of service and replay attacks. Some performance problems like time synchronization are also resolved. We finally try to extend the functionality of this method for different applications and numbers of users by which ESSTCP can be performed as a secure Remote Procedure Call (RPC).

1. Introduction

Hiding internet services from untrusted users would be one of the effective methods to protect not only the unpredictable attacks on local network and servers, but also to the unknown potential service and software vulnerabilities discovered gradually. Thereby, in order to distinguish between authorized users and adversaries, hidden authentication techniques should be exploited. These authentication techniques should be lightweight enough to be easily applicable on vast variety of devices and strong enough to be reliable for protecting crucial services and servers.

Barham et al. [1] proposed a few techniques in which the client authenticates to firewall and asks for access to a specific port number for connection. Since this authentication has to be done stealthily, the general idea is to send some specific packets to closed ports of the firewall and trigger daemons on the firewall by authentic packets to open the desired ports for the authenticated user. These authentication techniques are classified into three groups based on the way the authentic packets are sent. In *Spread-Spectrum TCP (SSTCP)*, the client sends a sequence of SYN packets to particular port numbers and prompts the firewall to execute the corresponding instructions for the client. In second method called *Tailgate TCP (TGTCP)*, instead of sending a sequence of SYN packets, it sends a packet with some data including

the secret and other parameters for authentication. In the third approach, *Option-Keyed TCP (OKTCP)*, firewall only allows the SYN packets which contain a key encoded in some IP and TCP header fields to pass through.

Subsequently, Port-knocking is introduced by Martin Krzywinski in [2] as a method of connection through a closed port. It is mostly focused on the server's personal firewall and protecting UNIX-based services with iptables. Since using this method protects important administrative services like SSH, SNMP, and etc, from denial of service and/or any possible attacks, and moreover it is flexible for developers to have their own implementation, it is widely embraced by the industry and academic communities and several open source implementations have been released.

The structure of this paper is organized as follows. In section 2, we examine the SSTCP method and review its advantages and drawbacks. In section 3, we will have a survey on existing implementations and proposed methods and study their benefits and flaws. We then present in detail our new proposal based on ESSTCP in section 4, and its method analysis in section 5. Finally, in section 6 we summarize and conclude the work.

2. Spread-Spectrum TCP (SSTCP)

2.1. Methodology

In this method, the client calculates the Authentic Port-Knock Sequence (APKS) and sends N TCP SYN packets to calculated port numbers. In the other side, there is a Silent Authentication Service (SAS) module on the firewall that listens for incoming knock sequence and if $M \leq N$ of them is received in correct order, it authorizes the client to send packets through the firewall on the desired port. In SSTCP, an infinite array of ports is generated by a one-way function based on a key (K) and time window value. Each new element of APKS in each time interval of "*T*" generated as

$$A[t] = SHA(K \| t) \& 0xFFFF \qquad (1)$$

where "*SHA*" denoted as the hash function of SHA-1, ∥ denotes concatenation, and "*t*" is the current time divided by *T*. They also propose a mechanism for synchronization on the first knock. In this mechanism, the received knock
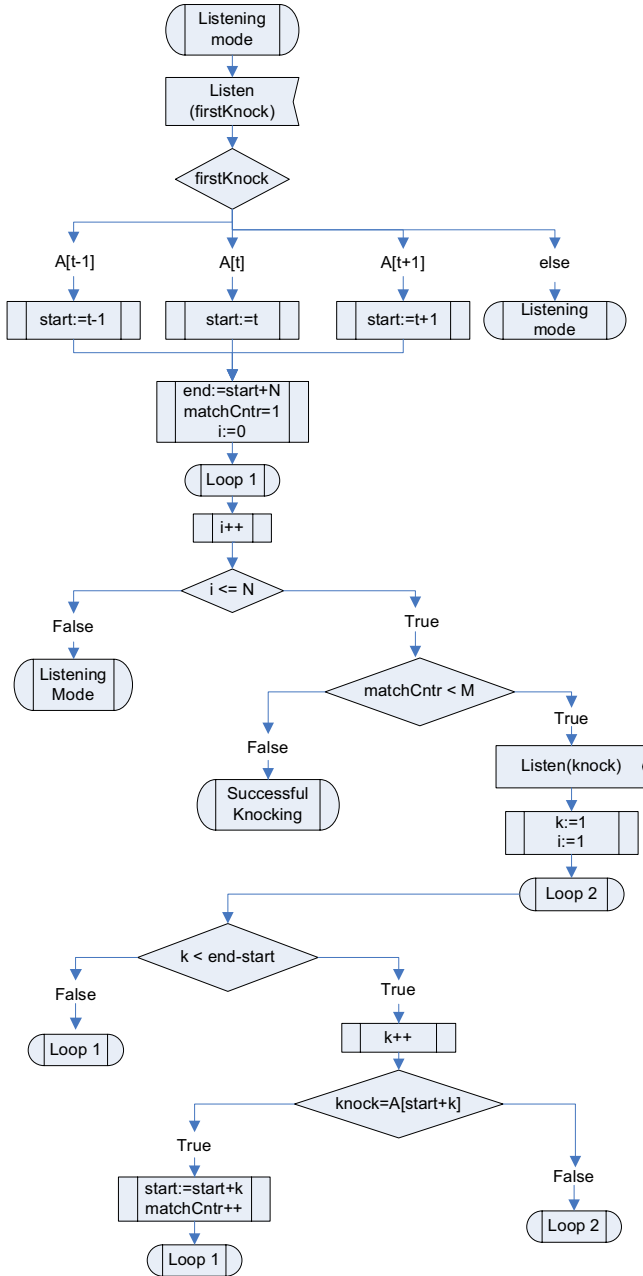
Figure 1. SDL representation of how the firewall listens to the received knocks from each knocker. (Each knock here is a destination port number for specific source IP address.)

2.2.1. Probability of guessing APKS. Since this protocol is completely passive in the network and generates no traffic, the SAS module can be implemented on layer 2 (L2) devices such as hidden security boxes which have no IP address, example of which include layer 2 firewalls and IPS/IDSs. Furthermore, knocks have no specific signature to which may reveal the existence of this service on the firewall to the attacker. SSTCP is also resilient to the packet loss and the packet reordering. Its computation complexity is relatively low, and it is invincible to brute force attacks, since the probability of guessing is

$$P_{\text{Guessing APKS in SSTCP}} = \frac{3}{2^{16}} \cdot \frac{\binom{N-1}{M-1}}{2^{16(M-1)}} = 3 \cdot \frac{\binom{N-1}{M-1}}{2^{16M}} \quad (2)$$

in which first term is for the first knock and second term is the probability of guessing other knocks. For example for the arbitrary values of *N=10* and *M=8* the probability would be 2.1738×10^{-37}, which is small enough to be safe against brute force attacks.

Barham et al. also addressed certain problems in SSTCP and discussed alternative approaches to mitigate these issues. One approach is using ISN (Initial Sequence Number) instead of the destination port, first to diminish the probability of guessing port knocks. Because the destination port field in TCP header is just 16 bits, while the sequence number field is 32 bits. Secondly, as some packets with specific destination ports might be dropped by the firewalls in between, ISN would be a good idea to be exploited. However, sending packets to a specific destination ports with different ISN would jeopardize the stealthiness of SSTCP technique. Moreover, ISN is likely to be changed by the firewalls, proxies, and NAT boxes in between which will be more explained in section 5.

2.2.2. Choosing M, N, and T values. Finding optimum values for the *T*, *M*, and *N* to avoid replay attack is another serious issue in SSTCP. The required time for an attacker to perform a replay attack, is $(N-M+0.5)*T$ [1]. So with a large value of *T* the server would be vulnerable to replay attacks. On the other hand, with a small value of *T*, the time synchronization between server and client and subsequently the functionality of the service might be disrupted. The proper values for M and N is also defined based on the desired security level parameters and link packet loss rate, to avoid brute force attack, and to choose a proper value for *M / N* ratio. So since it is not feasible to utilize a dynamic mechanism to calculate the link packet loss or time difference between server and client, some maximum assumed values should be chosen by the network administrator which make a suitably large window for replay attack.

2.2.3. SSTCP vs. DoS attack. As it is shown in Figure 1, after the first knock is received, an array of knock sequence from $A[start..start+N]$ is filled with

is processed as shown in Figure 1. The first knock is compared with three generated knocks at time *t-1, t,* and *t+1*. if it was equal to one of them, the SSTCP considers an array of knock from *A[start..start+N]*, and if the knocker is able to knock *M* ports out of *N* available ports, it is eligible to open its desired port. The detailed description of SSTCP is available in [1].

2.2. SSTCP Analysis

calculated authentic knocks. Then, the following received knocks will be compared to each element of array and update the *start* value with the index of the received knock in the array whereas the total authentication duration is at least $M \cdot T$ seconds. Suppose an attacker suspects a port-knock process is started by noticing that some packets from specific source IP and source port is sent to a specific destination with different destination port, periodically, almost equal to *T*. It starts DoS attack with the identical spoofed source IP and port with different destination ports sweep from 0 to 2^{16}. If these forged knock matches to one of the authentic knocks with

$$forgedKnock_{ind} > N - (M - realKnock_{ind}) + 1 \qquad (3)$$

it can interrupt the legitimate port knocking process and make client to knock the firewall again.

We will address the analysis of the SSTCP in more details in Section 5.

3. Other port-knocking implementations and related works

There are many port-knocking implementations listed in [4] which are mostly lightweight scripts installed as a server daemon manipulating the server's personal firewall when they are triggered by receiving authentic port-knock sequence. However, a few of them are based on the SSTCP concept. The basic implementations following SSTCP idea are utilizing a limited number of fixed hard-coded knocks which would be convenient for local port-knocking with trusted network environments. But as soon as the attacker eavesdrops on the knock sequence, the server is vulnerable to replay attack. These kinds of scripts are interesting for backdoors as well, because they are simple and lightweight to implement and are able to make a covert channel for intruder through the victim. But, on the other hand, this fixed APKS is a signature exploited by anti-viruses to find these kinds of backdoors on the system.

In some implementations the knocker and the server agree on a one-time APKS by negotiating that in an encrypted way. For instance, Sig2Knock [4] knocker generates the APKS and encrypts it by using the knocker's password hash and sends it (P1) to a specific port on server. After a short time interval, it starts sending knock packets and queries the server for the knock status (P2). On the other side, the server sends P3 to the client indicating whether the knock process proceeded successfully. P3 contains the encrypted port number which knocker should connect to for communicating to the real server application. To avoid replay attacks, they propose to use time stamps for P1, P2, and P3, to enable the server to reject packets with same or older time stamps. They also confined the number of knocks to three in which they use both destination port number and ISN to send a random 144 bits number. However, the

destination port range is not limited which causes management problems. Furthermore, it is susceptible to DoS attack in which the attacker generates forged knocks when the knocking process is started and suspends the real knock. The attacker is also able to send fake UDP P1 packets and compel server to try to decrypt its data with its list of passwords. Thus, forged P1 packets would cause resource allocations on the firewall which is dangerous even for regular firewall performance, although the server performs several checks to remove invalid packets. In addition, because the server sends P3, it is not passive in the network and it compromises the stealthiness of the port-knocking service and it would not be effective on L2 devices.

DeGaaf et al. [5] proposed another challenge-response method similar to Sig2Knock which uses a NAT-aware authentication algorithm and a few techniques to overcome the packet-reordering phenomenon [6]. However, in addition to the weaknesses they mentioned, like other challenge-response algorithms, it is vulnerable to DDoS attack; the attacker can flood the server by requests or inundate the knocker by challenges. It can be effectively done because the three-way authentication over UDP and techniques used to avoid packet-reordering (monotonically increased knocks) is useful signature of this method which gives the idea to an attacker when the knocking process is started and when it is finished. Furthermore, the authors did not offer a clear solution for packet loss, whereas the packet loss rate on Internet is estimated from 2.7% to 5.2% based on packet window size [6].

4. Proposed ESSTCP

By considering current implementations and their respective advantages and drawbacks, we try to design a new port-knocking system which addresses the existing weaknesses and propose a comprehensive and reliable system which can be widely used for different security purposes. Thus, we extend the concept of the port-knocking where port-knock daemon is not confined to be installed on the server or a firewall to open ports. In this approach, we call the port-knock daemon as Port-Knock Detector (PKD) which can be installed completely distributed in the network and listen to the traffic for authentic knock sequence. The ESSTCP package including PKD and the knocker entities called *"JESSTCP"* is implemented using Java and JPCAP [9].

In this method, both client and PKD try to calculate the APKS by themselves based on a secret key and a shared *profile* in which all required information exists. Each profile consists of following fields:
[PKD-side and client-side]
port_range (portRange): The port range of the APKS

validity_timeout (T): The time interval in which the calculated APKS is valid, after that the APKS should be recalculated.

time_source: {LocalTime, <Time_ser_ip>} the time source from which both client and PKD would retrieve time *[Default: LocalTime]*

knock_no (N): {9_SHA-1, 31_SHA-256, 63_SHA-512} number of knocks including three choices for three levels of security. *[Default: 9_SHA-1]*

knock_wating: the waiting time between each knock (note: (maximum RTT)/2 value is recommended)

knock_type: {UDP, syn, ack, rst, urg. psh, fin} are knock segment type which can be UDP or TCP with one of the mentioned flags. Client will randomly choose one the values listed in this attribute in order to send the knock. The order of the elements in this attribute should be same in client and detector sides. *[Default: syn]*

server_ip: knocks destination IP address, which is a list of IP addresses or a network range (note: the list order in both sides should be same)

source_ip: different source IP addresses for knocks, which is a list of IP addresses or a network range (note: the list order in both sides should be same) *[Default: host_ip_address]*

[PKD-side] *(Complementary fields dedicated to PKD)*
Password (secret): Secret string which identifies each profile

accepting_knocks (M): authentic number of knocks (note: $\leq N$)

knock_timeout (T_{wait}): the listening timeout for each knock (Note: *knock_waiting < knock_timeout < 2 * knock_waiting*)

commands: the commands should be executed in case of successive port knocks.

commands_time_gap: the time gap between commands execution.

The port-knocking authentication system starts working after the user enters the password *(secret)*. The client entity requests for time either from the time server or local machine *(cur_time)*, divides (integer division) it to T and concatenates the result to the password string *(secret)*. It applies the hash function indicated in the *knock_no* part and finds random string of *(H)*. Subsequently, the knock_extract function which is explained later in detail is employed on H in order to convert the string to a desired $APKS[j]$ (j denotes the knock number). This array of the numbers is in the port range that specified in the profile *"port_range"*. The APKS calculation is as follows:

$$H = Hash\_Func(\sec ret \parallel (curr\_time \setminus T))*$$
$$APKS[j] = knock\_extract(H) \qquad (j \in [1, MAX\_knock\_no])$$

$* a \setminus b \equiv \lfloor a/b \rfloor$

The knock_extract function can be different based on developer considerations. The algorithm here is dividing the hash string into bytes and multiplying each byte value by its following one and the last byte by the first byte to generate numbers in [0, 65535] interval. However, we need an APKS in the specific port range (port_range). Thus, the remainder of division of each calculated port to port_range is added to port range minimum value to obtain the desired result for APKS. The algorithm code is provided as following.

```
for (i=0;i<hashByArr.length; i++)
{
    j= (j<hashByArr.length-1 ? i+1 : 0);
    APKS[i]=portRangeMin +
(hashByArr[i]*hashByArr[j]) % (portRangeMax-
portRangeMin); //APKS_calc function
}
```

Contrary to most port-knocking implementations which use APKS as destination port number we define the knock packets PK[i] as:

$$PK[i] \to sport = APKS[2i-1] \qquad 1 \leq i \leq \frac{APKS.length}{2} - 1$$
$$PK[i] \to dport = APKS[2i]$$
$$PK[i] \to dest\_IP = IP\_poll(server\_ip)$$
$$PK[i] \to type = Type\_poll(knock\_type)$$

Figure 2 is an example of APKS calculation in which 9 knocks are calculated and the last pair is reserved. We use this pair for source/destination IP and type polling. It would be also helpful to calculate the After-Knock Key (AKK) which will be explained in Section V.A.8.

The polling functions algorithms also can vary based on the developer's consideration, for example we propose a simple algorithm which is $(APKS[n-1] \setminus j) \bmod ip\_pool\_len$ for the IP_poll and $(APKS[n] \setminus j) \bmod ip\_pool\_len$ for Type_poll. The knocker may use AKK as the source port of the connection that will request after knocking process. In this case, PKD-side will be aware of the knocker source port for the connection, so it can modify the accepting rule more efficiently. To calculate AKK, we suggest a multiplication of $APKS[n]$ and $APKS[n-1]$ and calculate the remainder of the division to *max_port_no* (65535), and if the result is less that 1024, it will be supplemented with 1024 to make sure that the source port is not in well-known port range.

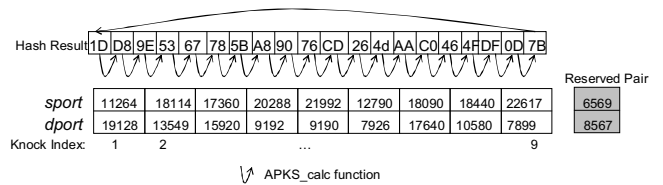| Hash Result	1D	D8	9E	53	67	78	5B	A8	90	76	CD	26	4d	AA	C0	46	4F	DF	0D	7B	Reserved Pair
sport	11264		18114		17360		20288		21992		12790		18090		18440		22617				6569
dport	19128		13549		15920		9192		9190		7926		17640		10580		7899				8567
Knock Index	1		2				...										9				

$\bigvee\nearrow$ APKS_calc function

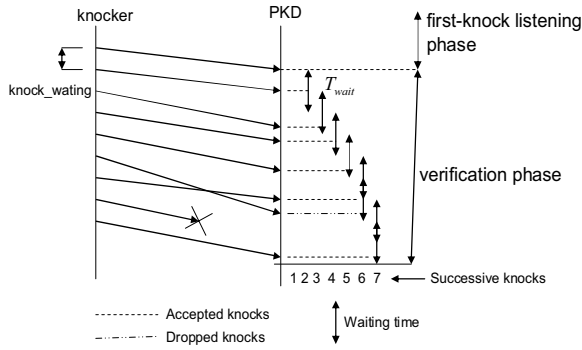Figure 2. APKS calculation example (N=9, and port_range= [5000, 25000])

Figure 3. ESSTCP timing diagram

On PKD side, we perform same operation. Since there is more than one profile in PKD, this calculation should be done for each profile. Accordingly, we assign a thread for each profile in which the APKS calculation is taking place periodically with period of T. When the APKS is computed by each thread, it starts filtering the captured traffic for its determined first knock with particular characteristic already declared in profile and corresponding calculation (first-knock listening phase). As it is shown in Figure 3, while the first knock is detected, a counter initiates and adds up and the detector listens for the next knock (verification phase). As soon as the counter value is equal to the M in the corresponding profile, it executes the commands in the profile. As matter of fact, the PKD only accepts one connection in each time slot T_i. In other words, computed knocks are valid only for one knocking trial. Consequently, when the first knock is detected, another thread will be triggered. In this thread, PKD listens for each knock for T_{wait} and if it does not detect any knock in this period, it considers that the knock packet is lost. If the knocker is not successful in accomplishing M successive knocks, it has to wait for next time slot T_i and knock the detector again. Furthermore, the first-knock listening phase is completely independent to the verification phase. It is important in case suppose the detector is in verification phase and waiting for residual knocks, but T_i is finished, in this case for new time slot, new knocks will be calculated and detector will be in first-knock listening phase.

5. ESSTCP method analysis

Here in ESSTCP, we tried to modify and add some complementary techniques to mitigate the drawbacks of SSTCP and add some features to broaden its functionalities and applications. In design, we consider the scalable number of users which would be interesting for enterprises and large scale applications. However, this method has its own disadvantages.

5.1. Advantages

5.1.1. Time Synchronization. In view of the fact that the time difference between client and PKD may prevent a successful authentication, we propose one large time window instead of N small time windows ($T_{ESSTCP} \geq N \cdot T_{SSTCP}$) for N authentic knocks. Although ESSTCP empowers administrator to use time server, the T_{ESSTCP} can overcome trivial time differences ($\Delta T \leq T_{ESSTCP}$).

5.1.2. CPU utilization. Because PKD is installed as a module on firewall certain resource constraints are considered. The implementations of both SSTCP and ESSTCP are divided into two main procedures: APKS Calculation and Knock Detection.

In ESSTCP, A[t] is filled by one iteration of retrieving the random hash string and calculating the knocks compare to SSTCP in which A[t] is filled by N iteration of almost same procedure. Figure 5 shows how ESSTCP removes the $N-1$ spikes of the SSTCP APKS calculation.

In Knock Detection, the ESSTCP is more efficient because it only monitors a short range of ports whereas the SSTCP needs to monitor all incoming SYN packets. Moreover, The CPU consumption would be very crucial when the PKD counters SYN DOS attacks. However, the ESSTCP is less vulnerable to this phenomenon since the small port range is secret.

5.1.3. Expandability by profiles. In ESSTCP, profiles are exploited for different users and/or different services where in each profile the number and type of knocks besides the corresponding executing commands are predefined. So ESSTCP could be called as a stealth connectionless Remote Procedure Call (RPC) working on link layer that can be triggered by the user with a secret APKS remotely.
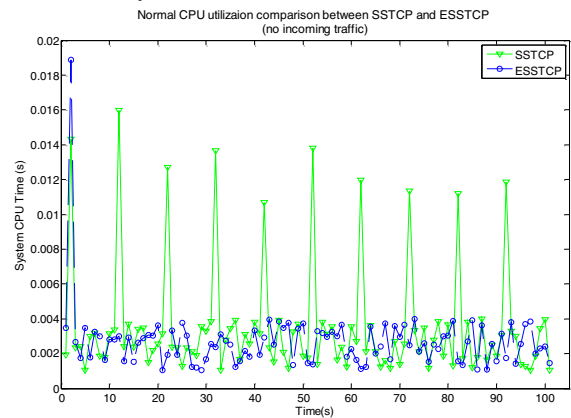

Figure 4. CPU time consumption for APKS calculation in SSTCP and ESSTCP (for a specific profile) on Pentium 4 Processor 2.8 GHz, where N=10, $T_{ESSTCP} = 110s, T_{SSTCP} = 10s$

5.1.4. Improving the probability of guessing APKS. Probability of guessing knocks for ESSTCP can be calculated by following formula:

$$P_{\text{Guessing APKS in ESSTCP}} = \frac{1}{2^{32}} \cdot \frac{\binom{N-1}{M-1}}{PortRange^{(2.M)}} = \frac{\binom{N-1}{M-1}}{2^{32} \cdot PortRange^{2M}} \quad (4)$$

where first term is the probability of guessing the APKS port range (PortRange = PortRangeMax – PortRangeMin) and second term is the probability of guessing APKS sequence which contains the knocks' source and destination port numbers. The difference between (2) and (4) is mainly because of the power of denominators in both equations where in equation (4) it is *2M* due to considering source and destination addresses in ESSTCP. As it can be deduced from the formulas and Figure 5, ESSTCP improves the likelihood of guessing APKS, with very short port ranges compare to SSTCP which uses all available port numbers. As it is shown, with ESSTCP *PortRange=100* the probability of guessing SSTCP for different *M* and *N* values is almost achievable. The subgraph in Figure 5 also explicitly illustrate this probability improvement in a given example with *N=35* and *M=20.*

5.1.5. Vulnerability to sweeping attack. Sweeping attack happens when the attacker floods the PKD by SYN packets with different destination ports sweeping from 1 to 65536. The attacker starts sweeping attack immediately after it detects the client knocking process is commenced. As it is mentioned in section 2 and based on (3), for SSTCP, it is highly probable that one of these forged knocks matches one of the APKS elements and interrupts the knocking process. For instance, assume a practical situation in which *M=10, N=8, T=5s, link_BW=2Mbps,* and *SYN_packet_size=512bits*, in SSTCP the estimated time for sweeping would be 16s, while the total available
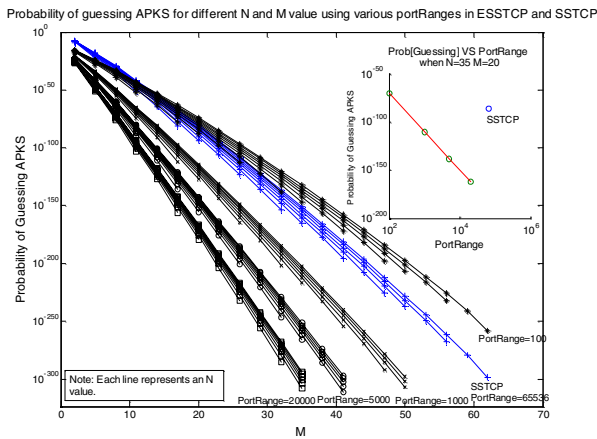

Figure 5. The comparison between probability of guessing of SSTCP and ESSTCP with different port ranges, the subgraph illustrates this comparison for an example where N=35 and M=20 more precisely.

knocking time is 40s. So the attacker can generate all possible knocks to satisfy the condition given in (3) to hinder the legitimate client to have a successful knocks. On the other hand, in ESSTCP, due to the fact that port range is secret, the attacker needs to sweep both destination and source addresses that would take 291 hours 16 minutes and 16 seconds for the given example. This long period is substantially more than T_{ESSTCP}.

Barham et al. address to use ISN to mitigate the concerns on the low number of the knocks in SSTCP, however, in ESSTCP source addresses next to destination addresses are preferred. It is because of the fact that since SSTCP needs to be installed and implemented either as executable script or as a hardware module, for existing firewalls and network tools technically it is more practical to have control over source ports rather than the TCP sequence number. In addition, ISN is more prone to be changed by middle boxes, because some firewalls and security boxes change the ISN to ensure that it is completely random and the connection is not vulnerable to ISN prediction attacks [7, 8].

Sweeping attacks can be also done in order to perform a successive knock. ESSTCP lessen significantly the probability of prosperity of attacker, although choosing a big M value is also very important to impede the attacker to attain the successive knock.

5.1.6. ESSTCP and replay attack. As it is mentioned in section 5, the calculated APKS is valid just for one knock trial, therefore the attacker can not perform the replay attack on the server*, while on the contrary* SSTCP has a considerably large replay attack time windows which was *mentioned* in section 2.2.2.

5.1.6. Port knocking stealthiness. The Spread Spectrum methods are completely hidden, and as long as we ensure of secrecy of the service and the existence of the PKD, most of the security problems and possible attacks would be resolved. SAS module in SSTCP can be easily detected by attacker, because of its specific signature which is nothing but some SYN packets coming from a host to specific server with different destination ports and after that the host connects to a closed port. However, in ESSTCP, PKD is installed dissipatedly and a pool of IP addresses for destination addresses and source ports are predicted. Thus, ESSTCP causes some difficulties for attacker to detect the PK service on the server.

5.1.7. After-knock DoS attack. It happens when the SAS module or PK server opens the port and permit SYN packets pass through the firewall from the knocker to desired service, the attacker may spoof the knocker address and flood the open port during opening time period and impede the knocker to connect to the server. This attack can be considered almost for all PK implementations except for those which use PK server or

firewall as a proxy to connect to the server like Sig2knock [4].

Avoiding after-knock DoS attack, ESSTCP provides a random and secret number AKK to be utilized by the knocker application and PKD to recognize the legitimate knocker in further connections. For example, the firewall can open the port for a connection to a specific secret source port which is AKK or deviation of that. It also may utilize to secure the services like SNMP; for instance, it might be as a kind of dynamic secret community string.

5.2. Disadvantages

The NATed clients are the most important concern in ESSTCP and other port-knocking methods. In the view of the fact that we utilize the TCP header for authentication and except destination port field, all other fields are changed by in Dynamic Address Assignment NAT [10], and on the other hand, 16 bits is not enough to secure the server from port sweeping attack, solving this problem is a big challenge. Thus, if the knocker connection provider use dynamic address assignment NAT the PKD would not receive the correct knocks, unless knocker use a tunneled connection like VPN to connect to the PKD. However, in other NAT methods like static address assignment NAT which keeps the information in TCP header and just change the source IP address, knocker has no problem to knock the PKD. Likewise, in IPv6 networks which has almost no limitation for the number of the IP addresses and only static address assignment NAT will be utilized for security purposes, ESSTCP is a comprehensive method for scalable remote procedure calls, and a reliable and secure technique for port-knocking.

6. Conclusions

SSTCP was proposed as a lightweight authentication protocols for client to send SYN packets through closed ports. Port-knocking implementations and techniques which mostly focus on servers' personal firewalls follow that basic idea. They are useful for sheltering critical services furtively from potential vulnerabilities and denial of service attacks. However, port-knocking approaches have some general limitations. They are mostly vulnerable to replay attacks, and also DDoS attack in case of disrupting the port-knocking process. Besides, those approaches in which the server is not passive and generate packets are jeopardizing the covertness of the service for attackers and third parties.

Our approach which is an extension of original SSTCP basically try to mitigate and solve certain drawbacks such as time synchronization, vulnerability to DoS attack and replay attack, and add some features to enhance its scalability and flexibility. In general, ESSTCP may

satisfy basic stealth, lightweight, and secure authentication requirements.

7. Acknowledgment

8. References

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

[2] M. Krzywinski, "Port Knocking," June 2003, *Linux Journal*, Available: http://linuxjournal.com/article/6811

[3] --,"Port Knocking Implementations," accessed on September 25th 2006, Available: http://www.portknocking.org/view/implementations

[4] Cappella, C. K. Tan, "SIG^2 Port Knocking Project: Remote Server Management using Dynamic Port Knocking and Forwarding," May 2004, Available: http://www.security.org.sg/code/sig2portknock.pdf

[5] R. deGraaf, J. Aycock, and M. J. Jacobson Jr.," Improved Port Knocking with Strong Authentication" in *Proc. of the 21st Annual Computer Security Applications Conference*, Tucson, AZ, Dec. 2005, pp 409-418.

[6] V. Paxson, "End-to-end Internet Packet Dynamics," *IEEE/ACM Trans. on Networking*, vol. 7, no. 3, Jun. 1999, pp. 277-292.

[7] --, "Hotfix for Symantec Enterprise Firewall 6.5 axtvpn.sys Module," May 3rd 2006, Available: http://service1.symantec.com/SUPPORT/ent-gate.nsf/0/49f229040579738888256c2300643901?OpenDocument

[8] *Cisco Catalyst 6500 Series Switch and Cisco 7600 Series Router Firewall Services Module Configuration Guide, 3.1,* accessed on July 5th 2006. Available: http://www.cisco.com/application/pdf/en/us/guest/products/ps708/c2001/ccmigration_09186a0080577bef.pdf

[9] *Jpcap - Java package for packet capture,* access by June 23rd 2006. Available : http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html

[10] P. Srisuresh and M. Holdrege, IP Network Address Translator (NAT) Terminology and Considerations. RFC 3626, Aug. 1999.