On the Use of Cryptographic Cookies for Transport Layer Connection Establishment

Sebastian Kiesel
University of Stuttgart, Institute of Communication Networks and Computer Engineering (IND)
Pfaffenwaldring 47, 70569 Stuttgart, Germany
`<kiesel@ind.uni-stuttgart.de>`

Abstract

In October 2000, the specification of SCTP (Stream Control Transmission Protocol, a new transport layer protocol) was published by the Internet Engineering Task Force (IETF). SCTP uses a cryptographic cookie mechanism to protect itself against denial-of-service attacks aiming at the association startup procedure. However, the basic idea of the cookie mechanism is not new. A similar mechanism for the TCP protocol has been proposed back in 1996 and has been implemented in the TCP protocol engines of several operating systems. The TCP SYN cookie mechanism has not been published as an RFC, probably because it does not require any changes to the existing TCP specification.

This paper gives an introduction to the problem of DoS attacks against transport layer protocols and presents the basic idea of the cookie approach. The specific implementations of this idea both for TCP and SCTP are explained and compared, especially with respect to the fact that for TCP, the mechanism had to fit into the existing protocol specification, whereas for SCTP, the protocol has been designed from scratch with the cookie mechanism in mind.

1. Introduction

Virtually any communication network to which untrusted nodes are connected to may become the victim of a denial-of-service (DoS) attack. The objective of a DoS attack is to degrade or interrupt a service offered by the network or one of the end systems connected to it, making it impossible or less performant for legitimate clients to use that service. Sometimes, DoS attacks aim at causing incorrect behavior or even a system crash at the target node by totally overwhelming the system with requests.

DoS attacks may be carried out on any layer of the network protocol stack. On the network layer, for example, a simple attack would jam parts of a network with useless data packets (e.g., ICMP echo request ("ping") messages in an IP network) in order to make overloaded routers delay or drop legitimate packets. On the application layer an attacker would use application specific requests to cause exhaustion of a limited resource at the target node, such as computation power, memory, I/O bandwidth or transaction IDs, etc. Especially useful for this kind of attack are request types with the following criteria: They should make it hard or impossible to distinguish forged from legitimate requests. They should be harmful even if the attacker uses fake source addresses in the packets (in order to make it harder to track him down), and they should have a high ratio of wasted resources at the victim to the effort of generating them by the attacker.

For the rest of this paper, transport layer protocols such as TCP, which implement a connection oriented service on top of a packet network, are considered. For most of these protocols the first message used for establishment of a new connection meets all of the criteria mentioned above. Virtually all protocols use a handshake procedure for connection establishment that resembles the generalized handshake depicted in figure 1.

Node "A", which wants to establish a connection to node "Z", sends some kind of "Connection Request" message to "Z", which acknowledges this message before the data transfer can begin. Some protocols may use more protocol legs (e.g., for the negotiation of protocol parameters) before the transmission of actual user data may begin, but this does not change the principle. As long as "A" does not acknowledge the receipt of the "Connection Acknowledge" message (either explicitly with an appropriate message type or implicitly by starting the transmission of user data – this is protocol dependent), "Z" may be either in a "half open" state or already in "established" state.

As soon as "Z" has received the first message it has to dedicate some of its memory for storing the state information of the new connection. Of course, "A" has to allocate memory for this so-called Transmission Control Block (TCB) even before sending the initial message.

The problem of DoS attacks arises from the fact that "Z" has to allocate memory (i.e., commit resources) upon receiving a new "Connection Request" message. A hostile node "X" might send thousands of these messages and

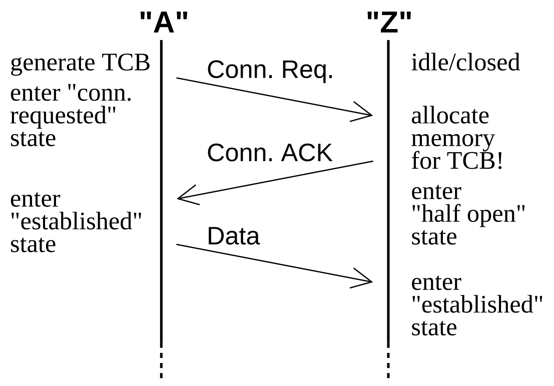


Figure 1. Connection establishment of a generalized transport layer protocol

therewith cause an overflow of the victim's buffer memory, where the state information for connections being in the "half open" state is stored (figure 2). As it does not matter for the DoS attack whether the "Connection Acknowledge" actually arrives at "X", the attacker may use random source addresses, which makes it harder to track him down and to distinguish the offending packets from legitimate ones. Depending on the protocol, the hosts that correspond to these randomly chosen addresses either discard the unexpected acknowledgements or respond with an error message. Also, "X" does not have to allocate memory for a TCB as it is not really interested in establishing a connection.

Once the buffer is full Z's protocol engine either has to discard new arriving "Connection Request" messages or to remove older entries from the buffer. Both strategies can hinder or prevent the establishment of legitimate connections. Increasing the buffer capacity makes flooding attacks more difficult for the attacker (i.e., it has to send more packets to disturb service) but cannot solve the problem in principle. Adding a timer that removes TCBs which are in "half open" state over a longer period[1] does not help either, as this timeout has to be at least as long as the maximum round trip time (RTT) that can occur in the network, for making sure that legitimate users can connect. In many wide area networks with high RTTs and high bit rates an attacker can send more "Connection Request" message within one RTT than a reasonably sized buffer can hold.

The rest of this paper is organized as follows: Section 2 introduces the basic idea of the cookie mechanism. Sections 3 and 4 explain the cookie implementations for TCP and SCTP, respectively. The paper is concluded with Section 5.

[1]actually, most protocols implement a timer, in case the network connection between "A" and "Z" gets interrupted right after the "Connection Request" message has arrived at "Z"

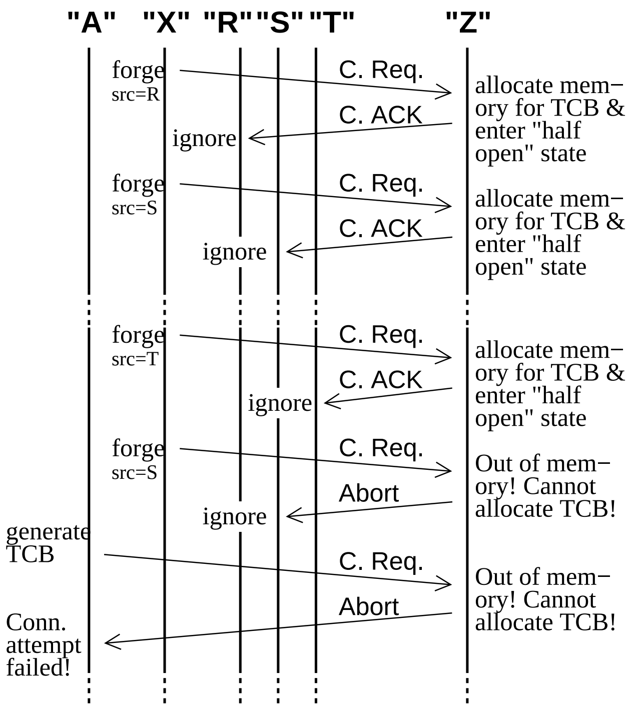

"A"
"X"
"R"
"S"
"T"
"Z"
forge src=R
C. Req.
allocate memory for TCB & enter "half open" state
C. ACK
ignore
forge src=S
C. Req.
allocate memory for TCB & enter "half open" state
C. ACK
ignore
forge src=T
C. Req.
allocate memory for TCB & enter "half open" state
C. ACK
ignore
forge src=S
C. Req.
Out of memory! Cannot allocate TCB!
Abort
ignore
generate TCB
C. Req.
Out of memory! Cannot allocate TCB!
Abort
Conn. attempt failed!

Figure 2. DoS attack against the generalized protocol

2. Cookie mechanisms for connection establishment

In the previous section it was reasoned about why it is neither possible to protect a buffer based mechanism against this type of flooding attack without possibly blocking legitimate peers, nor feasible to increase the buffer capacity to a size large enough to make overflows impossible. So, the only solution to this problem is to make the buffer unnecessary.

The basic idea (figure 3) is that "Z" must not commit any resources upon receipt of the "Connection Request" message. Instead, "Z" could answer with a message containing a cryptographically signed token ("cookie"), which should be hard to guess for an attacker but easy to recognize by "Z" as actually issued by itself. Furthermore, this message should contain all state information needed to build the TCB of "Z" for the respective connection. After sending this message, all state information should be discarded, making the "Z" invulnerable to queue overflows. An end system that really wants to connect has to return the cookie with its second message (the third message in the protocol run). After getting back the cookie from the connection initiator, "Z" verifies the authenticity of the cookie and then sets up the TCB using the information contained in the cookie itself

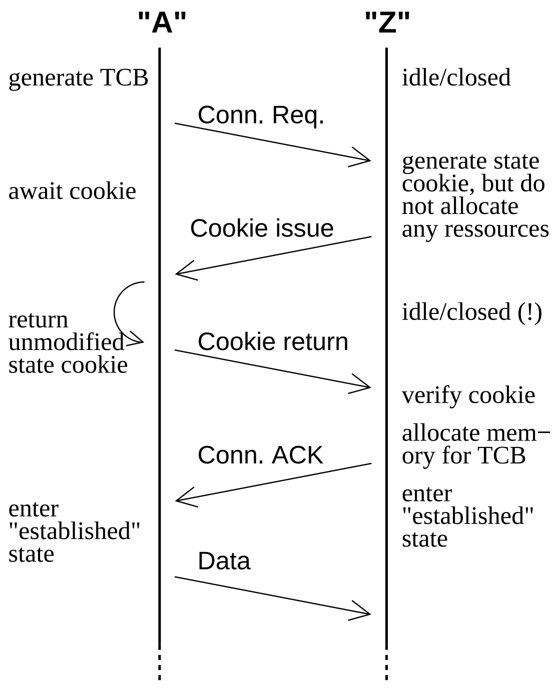
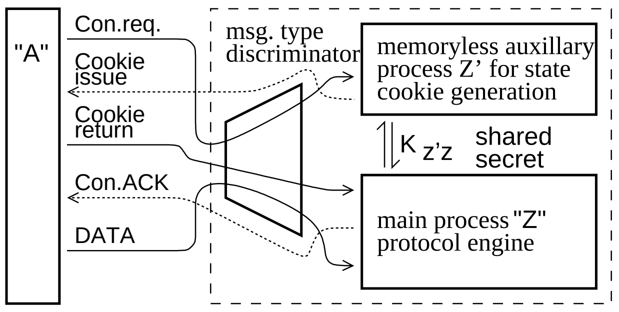


Figure 3. State cookie mechanism for protecting connection setup



"A"
Con.req.
Cookie issue
Cookie return
Con.ACK
DATA
msg. type discriminator
memoryless auxillary process Z' for state cookie generation
K z'z
shared secret
main process "Z" protocol engine

Figure 4. State cookie mechanism modeled with two processes at "Z"

and in the message that bears the cookie.

From a theoretical viewpoint both the protocol and the protocol engine running at the receiving node "Z" can be decomposed into two distinct phases or subprocesses, respectively. The first two legs of the protocol are used by "A" to acquire a ticket (the cookie), which allows it to begin the second protocol stage during which the actual user data is transmitted, beginning with the third message. The function of "Z" can be separated into one auxiliary process which is started upon receipt of a new "Connection Request" message and terminates immediately after generating the cookie, and the main process that runs the rest of the protocol run (see figure 4). The author has used such a decomposition for a formal analysis of the cookie mechanism used by the SCTP protocol (see below) with the BAN logic [1].

3. TCP

The "Transmission Control Protocol" (TCP) [6, 9] used in the Internet is a protocol with the properties described above. It is vulnerable to a DoS attack called "swamping" or "SYN flooding" (the latter name is due to the fact that the first packet of a TCP connection has the so-called "SYN bit" set). In September 1996, after it had been found that an attack against a server actually had exploited this weakness, which was known in theory for a long time, Dan Bernstein and Eric Schenk proposed to "eliminate the queue. Don't create the TCB until the ACK comes back." [2].

The most interesting detail about TCP SYN cookies is that these were added to the existing protocol, without changing the original TCP protocol specification. Therefore, this section is organized as follows: in the first subsection, the basic idea is given and the relevant requirements of the TCP standard are summarized. Then, details of the TCP SYN cookie implementation are explained. In the last subsection, some implications of adding cookies to the existing protocol are presented.

In practice, SYN cookie implementations are available for Linux (as an optional feature in the standard kernel source package) and other operating systems.

3.1. TCP SYN cookies: The basic idea

When designing TCP SYN cookies, one of the requirements was that the implementation should be interoperable with all existing TCP implementations conforming to the standards. Therefore, no new cookie data field could be added to the TCP header. This would have required changes to all existing end systems in the Internet: They would have to read the cookie from the incoming (second) message and copy it into the outgoing (third) message of the protocol run.

The solution was to use the sequence number fields for carrying the cookie. In every TCP implementation the sequence numbers usually do not start with zero. Instead, each peer announces its "Initial Sequence Number" (ISN) in its respective first packet (Values x and y in figure 5). These packets are the only two packets that have the SYN (synchronize) flag set. The ISN values are – increased by one – acknowledged by the peer in the following message. That is, by subtracting one from the ACK number of the third packet, peer "Z" can yield the value it had issued itself

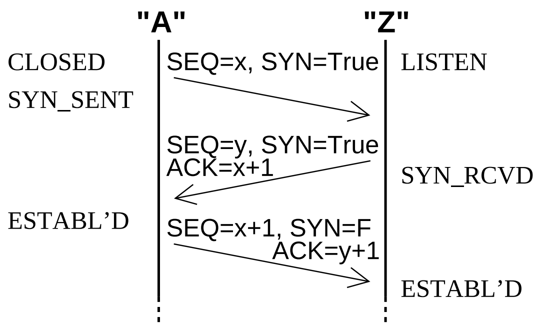


Figure 5. TCP three-way-handshake

as its ISN in the second packet. A method for transparently carrying a 32 bit value from "Z" to "A" and back to "Z" within the second and third leg of the protocol run was found. However, "Z" cannot choose this value (its ISN) totally unconstrained:

RFC 793 and RFC 1323 [6, 9] require that when establishing a TCP connection, the two peers announce their respective ISN (a 32 bit unsigned integer value) in the first 2 packets (SYN bit set) of the connection. Additionaly, RFC 1323 [9] requires the ISN to be initialized with a 32 bit counter, incremented "approximately" 2^{18} times per second (equals 262.144 kHz[2]). This results in a ISN wrap around every $2^{32}/(2^{18}\text{Hz}) = 2^{14}\text{s} = 16384\text{s} \approx 4\frac{1}{2}\text{h}$.

3.2. TCP SYN cookies implementation details

The TCP SYN cookie approach has to store a cryptographic cookie (long enough to be sufficiently secure) and all needed state information in Z's ISN field which is only 32 bits wide and cannot be chosen arbitrarily. This implies several restrictions on the efficiency of the protocol (MSS value, see below) and on the availability of additional TCP options such as window scaling [7]. Therefore, a host will always generate TCP SYN cookies, but it will use them (i.e., not add entries into the SYN queue and process cookies coming back from the peers) *only* if it is actually under attack, otherwise it will use the normal mechanism of a SYN queue. Using the TCP SYN cookie mechanism when the host is not under attack could – at least in theory – open a security hole in some environments, see section 3.3.

Fortunately, only one value is required for creating the TCB, which cannot be deduced from the normal contents of the third packet (i.e., the packet that returns the cookie): the maximum segment size (MSS). The initiator of a connection sends its MSS value, i.e., the maximum size of a single TCP segment ("packet") it can receive, in the first packet. The receiver of the SYN packet is expected to answer with its own MSS or the value received from the initiator, whichever is smaller, and to store this value in its TCB. However, it is possibly inefficient but RFC compliant to answer with an MSS which is smaller than both the own MSS and the value received from the originator. Therefore it is feasible to respond with one value out of a short (8 values) list of common MSS values, as long as the chosen value is smaller or equal than both peer's MSS values. The Linux implementation of TCP SYN cookies uses a table with eight entries: 64, 256, 512, 536, 1024, 1440, 1460, 4312 bytes[3]. Only 3 bits are needed to store the index into this table, compared to 16 bits for a MSS up to 65535 bytes.

Other features like window scaling [7] are optional, the connection initiator must be prepared that the peer might not be capable of handling them. They will be turned off if SYN cookies are actually in use.

This formula was proposed to compute the ISN:

- Shall `sec1` and `sec2` be two (constant) secret values.
- Shall `cnt` be a counter that increases about once a minute. For practical reasons, on a POSIX compliant system this could be the return value of the `time` (2) system call[4], divided by 64, i.e., shifted right by 6 bits: `cnt=time()>>6;`
- Shall `MT[0..7]` be an array of 8 sorted MSS values.
- Upon arrival of a packet with the SYN bit set, which requests a new connection with the parameters `(saddr, sport, daddr, dport, sMSS, sISN)`, compute the following values:

$$\texttt{MSSi} = \begin{cases} 7 & : \texttt{sMSS} \geq \texttt{MT[7]} \\ 6 & : \texttt{MT[7]} > \texttt{sMSS} \geq \texttt{MT[6]} \\ & \dots \\ 1 & : \texttt{MT[2]} > \texttt{sMSS} \geq \texttt{MT[1]} \\ 0 & : \texttt{MT[1]} > \texttt{sMSS} \geq \texttt{MT[0]}^{5} \end{cases}$$

$$\texttt{h1} = hash^{6}(\texttt{sec1,saddr,sport,daddr,dport,sec1})$$

$$\texttt{h2} = hash(\texttt{sec2,cnt,saddr,sport,daddr,dport,sec2})$$

$$\texttt{H2} = \texttt{h2 \& ((1<<24)-1)}$$

$$\texttt{C} = \texttt{cnt<<24}$$

[2]Note: the Linux kernel uses 1 MHz

[3]Note that there is no very large value (e.g., 64kB) in the list. This makes networks capable of handling large segment sizes (like HIPPI) rather inefficient (but only if the destination host is under attack during connection establishment)

[4]number of seconds since 01/01/1970 00:00:00 UTC

[5]`MT[0]` must be equal to the smallest MSS which is allowed by the TCP/IP standards. Therefore, the case `sMSS<MT[0]` cannot occur.

[6]The MD5 Message-Digest Algorithm[8] has been proposed and used in the Linux implementation as hash algorithm.

- Compute Z's ISN from these values and other parameters. This is the TCP SYN cookie:

```
dISN  =  (MSSi << 29)
      +  (C+H2+h1+sISN)&((1<<29)-1)
```

- If the node is *not* under attack, i.e., if there is room in the SYN queue left, create a TCB and send the SYN ACK packet with the dISN as calculated above.

- If the queue is full send the SYN ACK packet anyway, without creating a TCB. Indicate in the packet that the node is not able to accept any of the extra options such as window scaling. Store the current time as the "last queue overflow time".

- When the third packet of the connection comes back from the peer, find or generate the corresponding TCB:

1. Look for a TCB with `(saddr, sport, daddr, dport)` in the queue. If found, use this TCB.

2. If the "last queue overflow time" is longer than a few minutes ago the node should not accept a cookie. Discard the packet and give up.

3. If the last queue overflow was recently check whether the packet bears a valid SYN cookie. With the third packet the peer should acknowledge the dISN. Therefore, recalculate the expected cookie value from the parameters of the packet using the algorithm given above. For `cnt`, use the 4 most recently used values. Compare these calculated values with the bottom 29 bits of the assumed cookie in the packet. If there is a match generate a TCB using the parameters in the packet. Use the top 3 bits of the cookie as an index into the MSS table `MT[]` to figure out the appropriate MSS.

A note on the addends of `dISN`: `h1` is a constant offset to the ISN that does never change for a given IP addresses / TCP ports combination. This value should be hard to guess for any attacker. `cnt` and `h2` simulate the behavior of the ISN counter as required by the TCP standard, with two restrictions: the counter is only 29 bits wide and only the top 5 bits increase steadily (once every 64 seconds) whereas the lower 24 bits jitter unpredictably because of the second hash. The second hash is used to prevent that an attacker who has access to a list of old, outdated cookies sent to another host could guess the cookies that the target host would issue now to that host. Adding `sISN` ensures that the issued ISN increase steadily even within short time periods, assuming that the connection initiator uses a standards conforming ISN itself.

3.3. Security implications when using TCP SYN cookies

The SYN cookies approach does not violate any requirement of the TCP specification. However, it changes the semantics of the third packet of a connection. In certain environments, this may impose a security vulnerability which is extremely unlikely to be exploitable in practice, but given at least in theory.

The first and second packet of a TCP connection can be distinguished easily from each others and the subsequent packets: The first packet has the SYN flag set, but not the ACK flag. The second packet has both flags set. The third and all subsequent (data) packets – until the beginning of the connection release phase – have only the ACK flag set.

Many sites use simple stateless packet filters to protect their hosts from unwanted TCP connections from the Internet. These are basically routers that drop (discard) specific IP packets instead of forwarding them, based on various criteria, such as IP addresses and protocol types and parameters. A frequently used setup is to allow outbound TCP connections (initiated from a local host to a host in the Internet) but to disallow inbound connections by simply dropping all inbound packets which have the SYN but not the ACK bit set, and by allowing all other packets to pass. As all the "first" packets are filtered in inbound direction the remote hosts cannot initiate TCP connections to the local hosts.

The TCP SYN cookies approach changes the meaning of the packets. The first two legs of the protocol run are used by the connection initiator to acquire a cookie, which allows it to initiate the real connection starting with the third packet. That is, if a host in the Internet was able to acquire a valid cookie by some other means it could initiate a TCP connection starting with the third packet. A simple packet filter in a setup as described above would forward this packet, therefore allowing the establishment of an inbound connection – despite its contrary policy. Note that there is no simple solution as it is not possible to distinguish a "third" packet from all the subsequent data packets without maintaining any state information at the packet filter.

One possibility for acquiring a cookie would be to simply guess the 32 bit value. In November 2001 a security advisory [3] described a weakness in the Linux TCP SYN cookie implementation. Because a bad random number source yielding predictable numbers was used the likelihood of guessing a valid cookie was increased dramatically.

If the target host uses a good random source it is very improbable to succeed with guessing the cookie, but – at least in theory – still not totally impossible. To protect itself against long-term brute force attacks a host must not accept SYN cookies if it is not currently under attack.

4. SCTP

The “Stream Control Transmission Protocol” (SCTP) [10] is a relatively new transport layer protocol for the IP protocol stack. It has been developed by the IETF (Internet Engineering Task Force) SIGTRAN (SIGnaling TRANsport) working group [4] to be the base of an architecture for the transport of SS7 signaling data over IP networks. The “Signaling System No. 7” (SS7) is a packet-oriented network used for controlling the operation of the (connection-oriented) Public Switched Telephone Network (PSTN). Despite this specific aim, SCTP is designed as a generic transport layer protocol, such as TCP and UDP. This general approach is emphasized by the fact that further development of SCTP has been turned over to the IETF TSVWG (Transport Area Working Group) in the meantime.

The next subsection will give a brief overview on SCTP before SCTP’s cookie mechanism is considered in the following subsection.

4.1. SCTP services and features

SCTP [10] is the core protocol of the SIGTRAN architecture. In the IP protocol stack SCTP is located above IP as transport layer protocol, in the same layer as TCP [6] and UDP [5] (see figure 6). SCTP combines properties of UDP and TCP, and adds new features as well.

SCTP uses the unreliable connectionless packet service offered by IP to provide its upper layer protocol (ULP) with a reliable datagram service. Unlike UDP, SCTP detects packet loss, duplicate packets or bit errors and retransmits or discards the respective packets. Unlike TCP, SCTP preserves the boundaries of messages: distinct byte blocks are passed from and to SCTP’s ULP instead of a continuous byte stream as with TCP. For a comparison of UDP, TCP and SCTP see table 1.

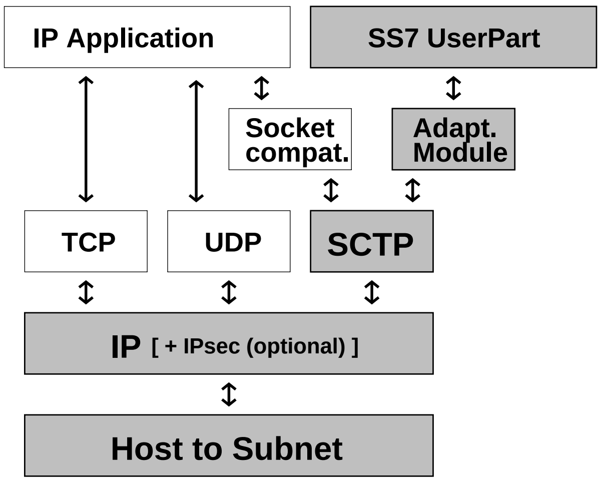


Figure 6. IP Protocol stack with SCTP

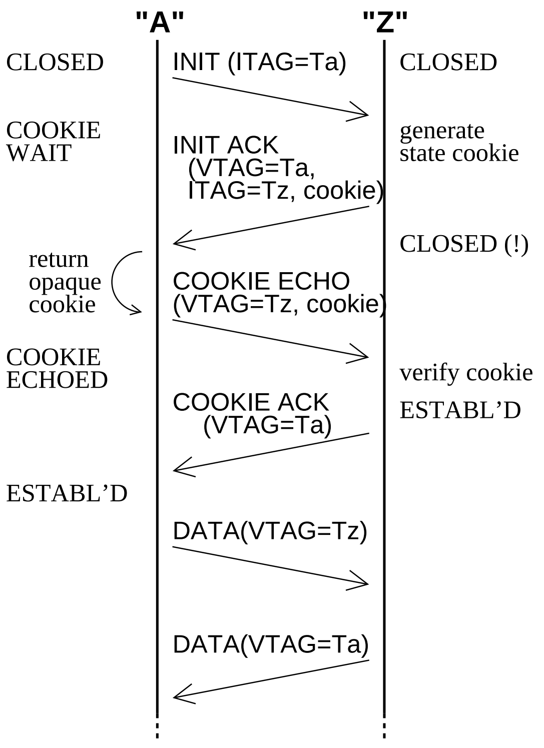


Figure 7. SCTP: init procedure signaling-time diagram

SCTP allows to split one association (SCTP term for connection) into up to 65536 logical subchannels per direction, so-called streams. Each user message is transmitted in one of these streams. Unless the “U” (unordered) flag is set for a message, SCTP ensures in-order delivery within the same stream. If one message is lost or corrupted in the network and has to be retransmitted, only the corresponding stream is subject to head-of-line blocking whereas messages for other streams can be delivered, anyway. Using one association split up into several streams – instead of using multiple associations bearing only one stream each – reduces the overhead at connection setup and improves the efficiency of the TCP-like fast retransmit algorithm by using it on the aggregate message flow.

For increased availability and reliability SCTP supports multihoming, that is, a node may be attached to the IP network with several interfaces, each having its own IP address.

SCTP uses two techniques as countermeasures against “blind spoofing” and denial-of-service attacks.

For protection of already established associations against blind spoofing attacks, SCTP uses so called “verification

	UDP	TCP	SCTP
IP protocol number	17	6	132
Connection oriented	no	yes	yes
Handshake at connection setup	N/A	3-way	4-way
Transmission of	user messages	byte stream	user messages
Multiplexing of connections	port numbers	port numbers	port numbers
Multiplexing within one connection	no	no	up to 64k streams
Sequence numbering & ACK'ment	no	yes	yes
In-order delivery of messages / bytes	no	yes	in same stream
Optional out-of-order delivery	N/A	limited (URG flag)	yes
Automatic retransmit of lost packets	no	yes	yes
Detection of duplicated packets	no	yes	yes
Bit error detection (checksum)	optional	yes	yes
Automatic retransmit on errors	no	yes	yes
Flow control, congestion avoidance	no	yes	yes
Multihoming support	no	no	yes
Connection / association restart	N/A	no	yes
Countermeasures against denial-of-service & blind spoofing attacks	no	optional, partly (SYN cookies)	yes (cookie, V-tags)
Countermeasures against man-in-the-middle attacks	no	no	no

Table 1. Comparison of UDP, TCP and SCTP

tags". In its respective first message each peer sends a 32 bit random value to the other peer. For all subsequent messages of the same SCTP association the sender of the message has to fill the "verification tag" field in the message's SCTP header with the value which it had received from the other node during the association startup. If a SCTP node receives a SCTP packet with a wrong verification tag, it has to discard the packet silently. Therefore a potential blind spoofing attacker, which is able to send packets with faked source IP addresses to one of the SCTP peers but does not know the verification tag values, cannot inject messages into an existing SCTP association However, the verification tags are transmitted in clear text, that is, they do not provide any protection against attackers which are able to eavesdrop on legitimate messages.

The second mechanism is described in the next subsection.

4.2. SCTP's 4-way handshake with cryptographic cookie

For an association establishment SCTP uses a 4-way handshake with a cryptographic cookie (figure 7), which uses the same idea as the TCP SYN cookies approach. However, unlike TCP, where the cookie mechanism has been added later without changing the original protocol specification, the SCTP protocol has been designed and specified to use the cookie mechanism.

SCTP uses a type indicator field in its messages ("chunks" in SCTP's terminology), which makes it possible to distinguish all four messages used for the handshake (INIT, INIT ACK, COOKIE ECHO and COOKIE ACK) from each others and from subsequent messages bearing user data (DATA). This also simplifies packet filtering. For faster association setup the first DATA chunks may be bundled with the COOKIE ECHO and COOKIE ACK chunks, respectively.

The SCTP specification provides for an own data field for storing the state cookie in the respective messages. The only size limit for the cookie is the requirement that the cookie together with all the other parameters has to fit into the INIT ACK or COOKIE ECHO message, respectively, which must not exceed 64 kB each. Therefore, all state information and all options may be stored in the cookie, including any auxiliary parameters such as timers, etc.

Note that the SCTP standard does not specify neither the data format of the cookie nor the cryptographic algorithm to be used for signing the cookie. Instead, these are "implementation specific" issues left to the implementor's choice. No standardization is required as the only node which has to read and interpret the cookie is "Z", the very same node that issued the cookie. This implies that the association initiator "A", which has to receive and return the (unmodified) cookie, cannot tell anything from the cookie, as it must not assume that "Z" uses the same SCTP implementation as "A" uses itself.

5. Conclusions

Both the TCP SYN cookie mechanism and the SCTP association startup procedure use the same basic idea. However, it makes a big difference whether using this mechanism was planned when designing the protocol or not.

The TCP SYN cookie mechanism has to use a rather sophisticated alignment of several variables into one field of the TCP header, in order to remain compliant to the existing TCP standard. Despite this effort only 3 bits for storing parameter information and only 29 bits for the cryptographic cookie are gained. This is sufficient for the basic protocol operation, but additional options such as large windows cannot be used if the host is under attack during connection setup. Changing the semantics of protocol messages may impair the function of packet filters ("firewalls") at least in theory, even if the modified protocol is full compliant to its original specification.

The SCTP protocol in contrast, which was designed from scratch with the intent to use a cookie mechanism, may use cookies of almost arbitrary size. This allows to store the whole parameter range in the cookie (as opposed to "8 common out of 64k possible MSS values") and to include all optional parameters. Explicit timestamps may be included to ensure that the cookie was issued recently. Having no relevant size constraint to take into account implementations may use almost any cryptographic algorithm which appears to be appropriate for signing the cookie. If all state information is stored in the cookie and if a sufficiently strong cryptographic algorithm is used for signing the cookie the handshake procedure may always make use of the cookie mechanism, no matter whether the host is currently under attack or not. This makes it unnecessary to implement both a TCB buffer and the cookie mechanism. Furthermore the current state of the host (whether being under attack or not) has no influence on the protocol run and on the semantics of the messages. If the cookie mechanism was envisioned from the beginning of the protocol design all messages may be given reasonable type identifiers. This simplifies state checking and packet filtering.

The cookie mechanism solves the problem of DoS attacks targeting at exhausting the victim's memory. However, this protection is achieved at the price of having to compute a cryptographic cookie. If generating this state cookie consumes a significant amount of CPU cycles, DoS attacks targeting at exhausting the victim's CPU resources are to be feared. Note that above a certain packet rate every INIT-flooding attack will succeed by clogging the victim's network link, no matter what defense methods are deployed at the node. But then this is no longer a "real" INIT-flooding attack, but an "ordinary" brute-force flooding attack with packets that happen to bear an INIT message. This is especially true for so-called "distributed DoS attacks" in the Internet, which use hundreds of (compromised) nodes under one common control to attack one target. The CPU effort needed for generating the cookie does not only depend on the cryptographic algorithm for signing the cookie but also on the extent to which other actions such as checking access control lists or determining other connection parameters are performed prior to issuing the cookie. If CPU profiler measurements showed that this is relevant in practice protocols such as SCTP could be further optimized and made even more invulnerable to DoS attacks by modifying the protocol and delaying the negotiation of "expensive" parameters to later protocol stages.

Cookie mechanisms can improve the security of a protocol against blind spoofing attacks at reasonable costs when they are considered in an early stage of protocol design without restrictions of prior standardization steps which did not take security requirements into account.

6. Acknowledgements

The author would like to thank Christian Hauser and Matthias Kabatnik for the fertile discussions we had.

References

[1] Michael Burrows, Martín Abadi, Roger Needham: *"A logic of Authentication"* ACM Operating System Review, Vol. 23, No. 5, December 1989

[2] D. J. Bernstein's Website about TCP SYN cookies `http://cr.yp.to/syncookies.html`

[3] Roman Drahtmueller, Andi Kleen: Linux Kernel security advisory on the Bugtraq mailing list. `http://www.securityfocus.com/archive/1/224531`

[4] The IETF (Internet Engineering Task Force) SIGTRAN (Signaling Transport) working group. `http://www.ietf.org/html.charters/sigtran-charter.html`

[5] RFC 768: Postel, J.: *User Datagram Protocol.* RFC, August 1980.

[6] RFC 793: Postel, J., Editor: *Transmission Control Protocol, Protocol Specification.* RFC, September 1981.

[7] RFC 1106: R. Fox: *TCP Big Window and Nak Options.* RFC, June 1989.

[8] RFC 1321: Rivest, R.: *The MD5 Message-Digest Algorithm.* RFC, April 1992.

[9] RFC 1323: Jacobson, V., Braden, R., Borman, D.: *TCP Extensions for High Performance.* RFC, May 1992.

[10] RFC 2960: Stewart, et al.: *Stream Control Transmission Protocol.* RFC, October 2000.