Global Information Assurance Certification Paper

Gerald W. Gordon
GSEC Practical Assignment version 1.3

SYN COOKIES, AN EXPLORATION

Abstract

SYN cookies are a method for defeating a SYN flood attack. A SYN flood attack is a denial of service attack against an internet TCP server (CERT 1996). A SYN flood attack is implemented by misusing properties of the TCP connection initiation handshake to create many bogus pending connections in the victim, so many pending connections that the victim's TCP connection resources become saturated with the attacker's bogus connections and the victim must deny legitimate connections, thus denying service to the victim's intended clients. This attack cannot be stopped by a packet filtering firewall that stops all TCP connection requests from clients since the server's job is to accept these connection requests and subsequently serve the clients. Dan Bernstein and Eric Schenk (Bernstein 1996, 1997) responded to this dilemma by adding a feature to TCP, the SYN cookies method to detect and defeat a SYN flood attack. In this report I describe and discuss the SYN flood attack, why SYN cookies are still an important topic, the properties of TCP that allow the attack, and the SYN cookies defense against the SYN flood attack.

Introduction

Kevin Mitnick in 1994 seems to have been the perpetrator of the first widely known SYN flood. In that case the SYN flood took a system off the network so that Mitnick's attacking host could impersonate it. Gulker (1997) quotes Shimomura (Mitnick's victim) on the analysis of the attack. CERT (1996) took official notice of the SYN flood attack in 1996 shortly after an ISP, Panix, suffered a disabling SYN flood attack on September 6. The SYN flood is a low bandwidth attack that can be successfully defended, for example by SYN cookies. Random dropping of pending connections during a SYN flood attack allows some service to be provided even in the absence of SYN cookies. Random dropping has been much improved by storing pending connections in a hash table instead of the original list (Lemon). High bandwidth attacks such as the various distributed denial of service attacks are dominating the news (Gibson 2002a). Nevertheless SYN flood attacks are still receiving attention. Chris Brenton (2002) spent time on the SYN flood attack as part of a talk on firewall techniques in a recent SANS webcast. Steve Gibson (2002b) recently created his own version of stateless SYN processing apparently in the absence of knowledge of the prior existence of SYN cookies. Even though due to SYN cookies and the improved random drop, no system need be shutdown by a Panix style SYN flood (Steenbergen 2001) there may still be many vulnerable systems since SYN cookies, though part of the kernel, are disabled by default in Linux (Bernstein 1997). In

addition, security mavens Brenton and Gibson are still devising defensive strategies. Therefore I feel comfortable in choosing SYN cookies for the topic of this report. SYN cookies have in addition the attraction that they are a good starting point and motivator for developing a deeper understanding of TCP and exploring some of its intricacies. Because the "documentation" for SYN cookies is sparse, written by and for experts, and in some instances inconsistent, I have endeavored to make this report understandable to someone at my own level. Details of the TCP protocol I have largely taken from RFC-793 (Postel 1981). For implementation details I have used Linux as the source.

Background: TCP and Establishing a TCP Connection

TCP is a connection oriented protocol for communication between two processes. Each process is associated with a TCP port, a number that identifies the process to TCP. The port is analogous to a mailbox where the process can deposit outgoing data and look for incoming data. TCP allows communication between different hosts that are each identified by a unique IP address (a number). The port number assigned to a process may be duplicated on a different host. In order to uniquely identify the port it is concatenated with the host IP address to define a unique TCP socket. TCP forms a connection by associating two sockets and delivering data received from one socket to the other socket in a structure called a segment. A segment consists of a header, shown in Figure 1, followed by the data. Therefore a connection is uniquely defined by two sockets, each socket being defined by a port number and host IP address. TCP maintains a connection table in which each connection exists as a data structure called a Transmission Control Block (TCB) that stores the state of the connection.

The TCP header, shown in Figure 1, adapted from RFC-793, contains the source port and the destination port but not the source and destination IP addresses. TCP segments are encapsulated in an IP datagrams for transmission to the destination. It is in the IP datagram header that the source and destination IP addresses are found. When a TCP segment is extracted from an IP datagram, a pseudo-header containing the source and destination IP addresses is prefixed to the TCP header so that TCP will have access to them. IP does not use connections and therefore does not need the port numbers used in TCP, but IP and TCP both need the IP addresses.

Three segments pass between client and server in order to establish a connection. First the client sends a segment requesting a connection. The server replies with a conditional agreement. The condition is met and the connection is established when the client replies. This is called the "three-way handshake" and has been likened to establishing communications in a telephone call ("Ring", "Hello", "Hello").

Next we consider in more detail the process of establishing a TCP connection since a SYN flood exploits by misuse the three-way handshake. The goal of a SYN flood is to exhaust a necessary resource and thereby create a denial of service attack. SYN cookies

are a method of allowing TCP to function even when the resource is exhausted thus protecting against that denial of service attack.

When a process wishes to provide a service to other processes via TCP, TCP associates the server process with a port and opens the port for listening. A listening port is a TCB containing the host IP address and port number of the server process but an unspecified (all zeros) port and IP address for the as yet unknown client. The TCB of a listening port has the connection state set to LISTEN. When a segment arrives to a listening port, the segment specifies the source and destination sockets, thereby specifying a unique connection. TCP first searches the TCBs of the existing connections (TCBs with state not equal to LISTEN) in the connection table for a match to the incoming segment. If a match is found there is no need to establish a new connection and the segment is handled in the context of the existing connection. If no match is found then TCP searches the TCBs with state equal to LISTEN for a match to the destination socket. If a match is found then TCP will enter the process of establishing a new connection. This process is shown as a diagram in Figure 2 and described below.

<table>
<tr><td colspan="8">Source Port (16 bits)</td><td colspan="4">Destination Port (16 bits)</td></tr>
<tr><td colspan="12">Sequence Number (32 bits)</td></tr>
<tr><td colspan="12">Acknowledgement Number (32 bits)</td></tr>
<tr><td>Data Offset (4bits)</td><td colspan="2">Reserved (6 bits)</td><td>U R G</td><td>A C K</td><td>P S H</td><td>R S T</td><td>S Y N</td><td>F I N</td><td colspan="4">Window (16 bits)</td></tr>
<tr><td colspan="8">Checksum (16 bits)</td><td colspan="4">Urgent Pointer (16 bits)</td></tr>
<tr><td colspan="10">Options (24 bits)</td><td colspan="2">Padding</td></tr>
</table>

Figure 1. The structure of the TCP header adapted from RFC-793. The fields involved in the SYN Cookie defense are discussed in the text.

There are six flag bits in the TCP header (Figure 1), URG, ACK, PSH, RST, SYN, and FIN. The ACK, RST, SYN, and FIN flags are set at the source to tell the recipient how to change the state of the connection. The client indicates its desire to have a new connection with the server by sending a segment with the SYN flag set and the ACK and RST flags unset to a listening port on the server. A segment with one or more flags set is called by the names of all the set flags so a segment with SYN only set will be called a SYN segment. SYN is short for “synchronize sequence numbers” and indicates the importance of sequence numbers as part of the state of a connection. Each direction of data flow in a connection has an independent sequence number that counts the bytes of data sent. For every TCP segment the current sequence number of the source appears in the header field called sequence number. In the header field called acnowledgement number the sender puts the sequence number of the next byte it expects to receive thereby acknowledging the last byte received.

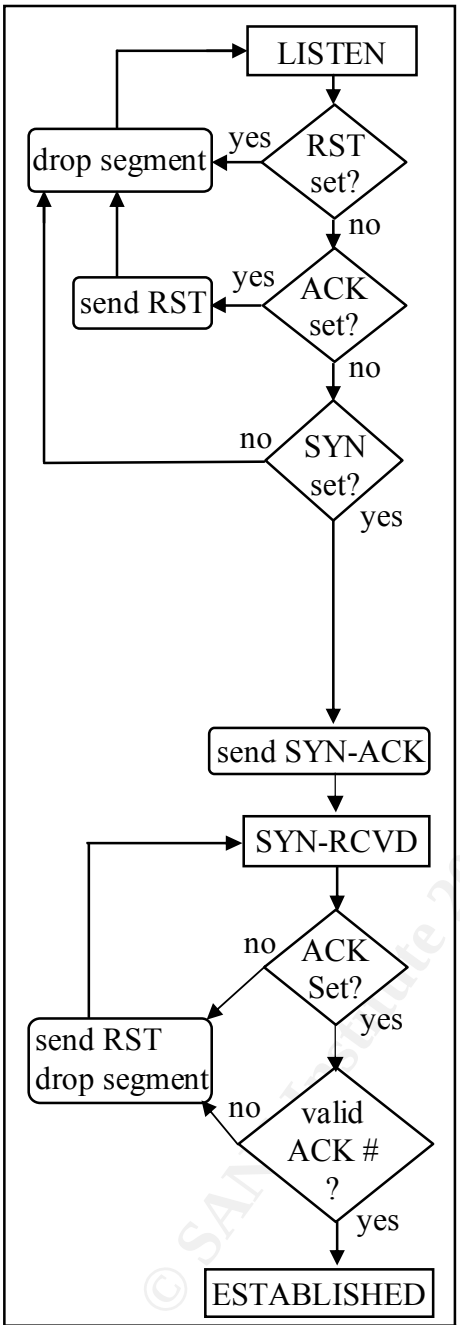

Figure 2. SYN Cookies disabled: Establishing a TCP connection.

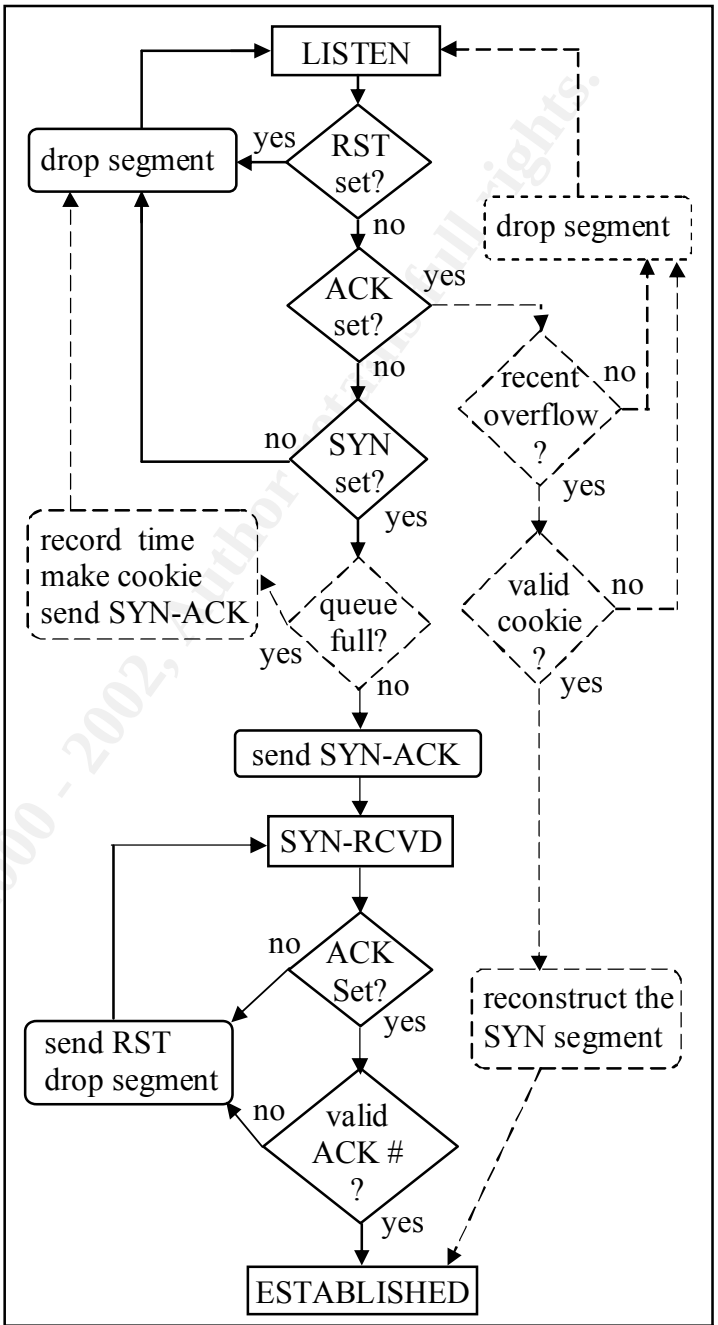

Figure 3. SYN Cookies enabled: Establishing a TCP connection.

More specifically, when a segment is delivered to a listening port, also referred to as a TCB in the LISTEN state, the port first checks the RST (reset) flag and if set drops the

segment; then it checks the ACK (acknowledge) flag and if set sends a RST segment to the client thereby telling the client to reset (close) the connection since an ACK would be sent to acknowledge what the server has sent and the server hasn't sent anything. Then the port checks for the SYN (ring) flag and if set sends the client a SYN-ACK (hello) segment and allocates a new TCB for the connection in the SYN-RECEIVED state. In the SYN-RECEIVED state a connection is half open or half connected and waiting for the appropriate response (hello) from the client. The server's TCB for this connection now stores the SYN arrival time, both sockets, both sequence numbers, and other information from the client's TCP header. The connections in the SYN-RECEIVED state are sometimes called the backlog queue, the SYN queue, the SYN-RCVD queue, or the open request queue. (Whether these connections truly are a queue would depend on implementation details though there doesn't seem to me to be any value in implementing them as a queue data structure since they will not necessarily be processed in the order they are received. In fact the SYN queue now seems to be implemented as a hash table.)

The connection in the SYN-RECEIVED state is waiting for a reply from the client. If the connection receives a segment with the ACK flag unset then drop the segment and send a RST. If the ACK flag is set then verify the acnowledgement number (the field in the TCP header). If the acnowledgement number is incorrect then drop the segment and send a RST and do not change state. If the acnowledgement number is one more than the sequence number sent in the SYN-ACK then accept the segment and change the connection to the ESTABLISHED state. The SYN flag set in the SYN-ACK counts as one byte sent to the client so the client must increment the sequence number to form its acnowledgement number.

We have now passed through the process of establishing a new connection on the server for a client. The process is summarized in Figure 2, which is adapted from RFC-793 (Postel 1981). RFC-793 contains many other details of this process of less relevance to the SYN flood attack.

The SYN Flood Attack

The number of TCBs (connections) in the SYN-RECEIVED state has a maximum value. The choice of a small maximum is justified in that for a legitimate connection request the connection will typically remain in the SYN-RECEIVED state for only milliseconds, since the client will usually send its ACK immediately. And in that a very large maximum SYN queue size could still be filled by an effective SYN flood and would use more of the finite resources available at the expense of other functions. If a SYN segment arrives and matches a TCB in state LISTEN but the number of TCBs in state SYN-RECEIVED is at the maximum, then no new connection is created and the SYN segment is dropped. This is the basis of the SYN flood denial of service attack: First the attacker sends the server many SYN segments with incorrect (spoofed) IP addresses. The server then allocates a TCB for each half open connection and sends a SYN-ACK to the spoofed

client host. If the attacked server receives no replies to the SYN-ACKs, it will continue to allocate TCBs in the SYN-RECEIVED state until the maximum number is reached. The server then must drop all requests, including legitimate requests, for new connections while waiting for the bogus connections to timeout.

Since the spoofed clients did not send the SYN segments they will never send ACKs and the server will never promote these connections to the ESTABLISHED state. If the server's SYN-ACK reaches a listening (open) or a non-listening (closed) port on the spoofed client, the port will send a RST causing the server to de-allocate the TCB in the SYN-RECEIVED state. If the server's SYN-ACK reaches a firewall protected port the response depends on the firewall. The usual firewall drops segments destined for protected ports. In this case no response is sent to the server and its TCB will remain allocated until the timeout. The lack of response makes the SYN flood attack more effective. Brenton suggested in a SANS webcast (Brenton 2002) that firewalls could be configured to reject segments destined for protected ports which still entails dropping the packet but in addition sending a RST. This is a "good neighbor" firewall configuration which if universally applied would make SYN flood attacks ineffective. If the spoofed client does not exist it cannot reply but the router for the subnet of the spoofed IP address should return an ICMP host-unreachable to the server causing the de-allocation of the TCB in the SYN-RECEIVED state (Stevens). The SYN flood attack is effective at a low rate of successful creation of orphan TCBs that must timeout, roughly max size of SYN queue/20 per second (Schenk in Bernstein 1996) and many hosts are protected with firewalls that drop segments destined for protected ports. Therefore the attack strategy of using random IP addresses for the spoofed clients can work since even a single protected port on a spoofed client could provide enough traffic to SYN flood the victim. It probably wouldn't be hard to develop a list of firewall protected ports that drop segments to make the SYN flood attack maximally effective.

The SYN Cookies Defense

The key to the SYN flood attack is the filling of the victim's SYN queue. The SYN cookies defense allows the victim to continue accepting connection requests when the SYN queue is full. The SYN cookies approach is to detect when the SYN queue is full and if full, create a cryptographic cookie (a 32 bit number) from information in the SYN segment and then drop the SYN segment. The cookie will be used as the initial sequence number in the SYN-ACK sent to the client. The cookie (plus one) will be returned to the server as the acknowledgement number in the ACK from a legitimate client. The returned cookie can be validated and the most important parts of the SYN segment can be reconstructed from the cookie thus allowing the attacked port to bypass the SYN-RECEIVED state (the SYN queue) and to allocate a TCB for a validated (legitimate) connection directly in the ESTABLISHED state. Since the spoofed clients of the SYN flood never send ACKs, no TCBs are allocated for them in any state when SYN cookies are in use. Thus the SYN cookies defense allows the port to accept new connections

when the SYN queue is full, defeating the SYN flood attack. Figure 3 shows a diagram of this process in which the dotted lines indicate paths and actions taken only during a perceived SYN flood attack.

Detecting the SYN Flood Attack

When a SYN arrives and SYN cookies are enabled the port decides whether a SYN flood is in progress based on whether the SYN queue is full and will only use SYN cookies if the queue is full. Before generating a cookie to send to the client, the time of the current overflow event is recorded for use in ACK processing. When an ACK arrives that does not belong to an existing connection, further processing will occur only if there has been a recent overflow. If an ACK is not preceded by a remembered SYN, a new connection should only be considered if we are, or have recently been, under attack as indicated by the last overflow time recorded during SYN processing.

Construction of the Cookie

The key to not allocating storage while waiting for the ACK from the client is in the construction of the cookie. Since the cookie will be returned to the server in the client's ACK, part of the cookie encodes some recoverable information from the SYN segment so that the server will not need to allocate a TCB to hold that information while waiting for the ACK. This information consists of a three bit value that is an index into a table of commonly used MSS (maximum segment size) values. (The MSS option of the TCP segment header contains a 16 bit value for the MSS which if included in the 32 bit cookie would not leave enough bits in the cookie for its cryptographic functions.) The MSS option is important in using as large a segment as possible without causing fragmentation but it is normally used only in the SYN containing segments used in the initial handshake establishing the connection. Therefore in order not to always use the default MSS while SYN cookies are in use, the largest value in the table not larger than the client's MSS is encoded in three bits of the cookie. Unlike MSS, the Window field of the TCP header can be specified in each segment, but the commonly used Window Scale Factor Option (that provides a multiplier for the value of Window field) is normally only used in the SYN containing segments of the initial handshake (Stevens 1994). No information about the Window Scale Factor Option is encoded in the cookie; therefore the usefulness of the Window field is greatly reduced when SYN cookies are in use. Because not all features of TCP are available when SYN cookies are in use, when SYN cookies are enabled they are used only when the SYN queue is full indicating that the system is probably under attack.

Another component of the cookie is a time based counter to allow rejection of non-recent or stale cookies. In addition the cookie contains cryptographic information to allow validation of the cookie and make cookie forgery very difficult. If stale cookies are

accepted, a forger has a bigger window of opportunity. By predicting a series of cookie values, an attacker could send a large number of valid ACKs that would result in the same large number of connections in the ESTABLISHED state on the server. The forged ACK attack in conjunction with the SYN flood attack elevates the SYN flood attack (being successfully defended by SYN cookies) to a connection table overflow attack possibly causing denial of service. Therefore it is necessary to make prediction of the series of cookie values very difficult while still allowing the server to validate the returned cookie values. When SYN cookies are in use, there is no TCB storing the SYN-ACK sequence number (the cookie) or the time the SYN arrived. Therefore the server must determine from the client's ACK whether the acnowledgement number is valid, that is, whether the acnowledgement number is the cookie (plus one) that the server sent.

The statements by Dan Bernstein on his web site disagree on the exact formula for the cookie (Bernstein 1996, 1997). I'm considering the following formula as implemented by Andi Kleen (1997) and used in Red Hat Linux 7.2:
Cookie = hash1 + sseq + count << 24 + ((hash2 + mssindex) & ((1 << 24) – 1)).

- Where hash1 is an SHA hash of the source port, source IP, destination port, destination IP, and a SYN cookie secret chosen at first call of the function.
- Where sseq is the sequence number of the client's SYN.
- Where count is the current value of a 32 bit minute counter.
- Where hash2 is a an SHA hash of count, the source port, source IP, destination port, destination IP, and a different secret also chosen at the first call of the function.
- Where mssindex is the 3 bit index coding an approximation of the MSS value from the client's SYN.

Here is a slightly graphical formula for the cookie:

```
cookie =
    |                hash1 32 bits                    |
  + |                 sseq 32 bits                    |
  + |count << 24 |(hash2 + mssindex) & 0xFFFFFF 24bits|
      8 bits
```

Validation of the Cookie

When the ACK comes back from a client the acnowledgement number is validated in a series of steps outlined next. Subtract one to get the putative cookie. Calculate hash1 from the ports and IPs in the ACK. Subtract hash1 from the putative cookie to get intermediate1. Subtract one from the ACK's sequence number to get what should be the sequence number of the client's SYN (sseq in the cookie formula). Subtract this from intermediate1 to get intermediate2. Isolate the top 8 bits of intermediate2, shift it down and subtract the result from the low 8 bits of the current count value. The difference represents the time in minutes since the cookie was created. If the difference is greater

than three, the cookie is invalid. Otherwise subtract the difference from the current 32 bit count to reconstruct the 32 bit count used to construct the cookie originally. Calculate hash2 with the reconstructed 32 bit count and the ports and IPs from the ACK. Subtract the low 24 bits of hash2 from the low 24 bits of intermediate2. This result should be the MSS index and if it is out of range (the range is 0-7) the cookie is invalid. Otherwise the cookie is considered valid and a new connection will be created in the ESTABLISHED state.

Attacking the Cookie Cryptography

If the attacker is not sniffing the traffic between the client and server, the attacker must guess the 32 bit cookie value to forge a connection. The probability of success depends on how many of the 4 billion guesses can be tested in the 3-4 minutes available before the cookie expires. At 100 microseconds per guess (ten threads at 1000 Hz each) there would be 2,400,000 guesses or an average of 0.00056 successes in each 4 minute period. This rate is not great enough to mount a denial of service attack but there are other attacks using connection forging (Bellovin).

If the attacker can sniff the traffic then the attacker could attempt to break the secrets by analyzing a series of connections at different times and therefore with different count values. First guess the first secret until the extracted count values increment in a fashion corresponding to the times of collection of the segments. At this point the attacker can predict 8 bit count values for any future time. Now knowing the 8 bit count values, guess the high 24 bits of the count and the second secret until the MSS index is within range for more than one segment. At this point the attacker is in a position to forge connections and, for example, launch the connection table overflow attack described above. Resetting the counter and the secrets, by rebooting if no other method is available, could thwart the attack. I don't know how to evaluate the SHA hash in the context of SYN cookies, but at full power SHA is believed to be infeasible to break (FIPS 1995). SYN cookies secrets 1 and 2 are 448 bits and 416 bits long respectively (Andi Kleen, 2002 personal email communication). An optimistic attacker could maximize the chances of breaking the secrets by using a compromised system to collect cookies on its own connections to the server so the MSS value would be known; and then distribute the job of breaking the secrets to a flotilla of other compromised computers. There are currently easier ways to create a denial of service attack, for example the distributed reflection denial of service (Gibson 2002c).

Cookies as Initial Sequence Numbers

One of the practical requirements of initial sequence numbers and therefore SYN cookies is that a new one should never be slightly smaller than a recent one. This prevents overlap of the sequence number space of two instances of the same connection and therefore

prevents mistaking a delayed segment from a previous instance of that connection for a segment belonging to the current instance. RFC-793 (Postel) says that TCP should use initial sequence numbers that are tied to a clock, which seems to mean monotonically increasing. RFC-793 suggests a 4 microsecond clock that for a 32 bit counter would roll over every 4.77 hours. This statement of the requirement avoids the need to define "slightly smaller". The high 8 bits of a SYN cookie are the sum of the highest 8 bits of hash1, which is constant, plus the highest 8 bits of the client's sequence number, which is assumed to be tied to a 4 microsecond clock, plus the lowest 8 bits (shifted up) of the server's minute clock. The 8 bit minute clock will roll over every 4.27 hours. The sum of the two variable components can be constant for as long as a minute and will roll over at about 2.5 hours. The second hash dominates the lower 24 bits of the cookie. The second hash is not constant since its inputs include the 32 bit minute counter. In fact the second hash and therefore the lower 24 bits of the cookie are essentially random in the range of 0 to about 16 million. When the high 8 bits roll over, the value of the cookie drops by about 4 billion and a new cookie will not be slightly smaller than a recent cookie. However, during a period in which the highest 8 bits are constant and the lower 24 bits are bouncing around, it would be possible to issue a two initial sequence numbers with the second one slightly smaller than the first. And since the actual maximum segment lifetime may be as long as two minutes (at least in 1981 when RFC-793 was written) there could be segments from the first instance of the connection still arriving while the second instance of the connection is active. But, there is a timeout of twice TCP's maximum segment lifetime (MSL) value after the two parties in a connection agree to close it and before TCP actually closes it. Since Linux sets the MSL to 30 seconds (Stevens) the 2MSL value is 1 minute allowing enough time for the high 8 bits of the cookie to be incremented and to therefore prevent issuing two cookies of which the second is slightly smaller. SYN cookies have the advantage that a cryptographic initial sequence number is difficult for an attacker to predict. Normal initial sequence numbers can also benefit from cryptography (Bellovin; T"so).

Conclusion

This report is unified by the desire to understand TCP and SYN cookies. There are several disparate ideas woven together on the TCP and SYN cookies threads. I hope you have been interested in the factual material and amused by the analysis. Even though most TCP implementations have a common ancestor, the details depend on the operating system and even the version of the operating system. Therefore any attempt at a general discussion will need to make some assumptions for the sake of being definite and the conclusions may not apply to all implementations or even in precise detail to any current implementation. For the record, I have taken a Linux centric approach and perhaps I could have made fewer assumptions had I spent more time in the source code.

References

Bellovin, S. “Defending against sequence number attacks.” RFC-1948 1996. URL: http://rfc.sunsite.dk/rfc/rfc1948.html

Bernstein, Daniel. (archive of newsgroup designing SYN cookies) 1996. URL: http://cr.yp.to/archive.html

Bernstein, Daniel. “SYN cookies.” 1997. URL: http://cr.yp.to/syncookies.html

Brenton, Chris. “Firewalls and perimeter protection.” 2002. URL: http://sans.digisle.tv/audiocast_050102/brief.htm

CERT®. “TCP SYN Flooding and IP Spoofing Attacks.” CERT® Advisory CA-1996-21 Original issue date: September 19, 1996. URL: http://www.cert.org/advisories/CA-1996-21.html

FIPS. “Secure hash standard.” FIPS PUB 180-1 1995. URL: http://www.itl.nist.gov/fipspubs/fip180-1.htm

Gibson, Steve, “The Strange Tale of the Denial of Service Attacks Against GRC.COM.” 2002a. URL: http://grc.com/dos/grcdos.htm

Gibson, Steve. “The Genesis of GENESIS.” 2002b. URL: http://grc.com/r&d/nomoredos.htm

Gibson, Steve. “Distributed reflection denial of service.” 2002c. URL: http://grc.com/dos/drdos.htm

Gulker, Chris. “Tsutomu Shimomura's newsgroup posting with technical details of the attack described by Markoff in NYT.” 1997. URL: http://www.gulker.com/ra/hack/tsattack.html

Kleen, Andi. “secure_tcp_syn_cookie() in random.c.” kernel-2.7.4-10.src.rpm 1997. (From the SYN cookie code in Red Hat Linux 7.2) URL: http://distro.ibiblio.org/pub/Linux/distributions/redhat/7.2/en/os/i386/SRPMS/kernel-2.7.4-10.src.rpm

Lemon, Jonathon. “Resisting SYN flood DoS attacks with a SYN cache.” URL: http://people.freebsd.org/~jlemon/papers/syncache.pdf

Postel, Jon. Editor. “Transmission Control Protocol.” RFC-793 1981. URL: http://rfc.sunsite.dk/rfc/rfc793.html

Steenbergen, Richard A. "Understanding Modern Denial of Service." 2001. URL: http://www.e-gerbil.net/ras/projects/dos/dos.txt

Stevens, W. Richard. TCP/IP Illustrated, Volume I 1994. Addison Wesley

T'so, Theodore. "secure_tcp_sequence_number() in random.c." kernel-2.7.4-10.src.rpm 1997. (Red Hat Linux 7.2) URL: http://distro.ibiblio.org/pub/Linux/distributions/redhat/7.2/en/os/i386/SRPMS/kernel-2.7.4-10.src.rpm