Adventures in TCP/IP

The Handling of SYN Segments in FreeBSD

BY RANDALL STEWART AND MICHAEL TÜXEN

TCP Connection Setup

The Transmission Control Protocol (TCP) is a connection-oriented transport protocol providing a reliable bidirectional byte stream service. The TCP connection setup requires three TCP segments to be exchanged, which is called a three-way handshake. The TCP endpoint initiating the TCP connection and sending the first TCP segment, the SYN segment, is called the client. The TCP endpoint waiting for the first TCP segment is called the server and responds to the received SYN segment with a SYN ACK segment. When the client receives this SYN ACK segment, it completes the handshake by sending an ACK segment.

The TCP handshake is not only used to synchronize the state between the two endpoints including the initial sequence numbers for providing reliability, but also to negotiate the use of TCP extensions via TCP options. With today's Internet, the most widely deployed TCP options during the handshake (contained in the SYN and SYN ACK segments) are:

1. The maximum segment size (MSS) option
   The MSS option contains a 16-bit number (between 0 and 65535), which is the maximum number of payload bytes the sender of this option is willing to receive in a single TCP segment. For this number, it is assumed that no options at the IP layer and the TCP layer are used. In case such options are used, the number must be decremented by the size of the options. This helps the TCP sender to avoid sending TCP segments requiring fragmentation at the IP layer.
2. The SACK-permitted option
   This option announces that the sender can handle selective acknowledgments (SACK options). This improves the performance in case of packet loss.
3. The TCP Window Scale option
   This option contains a natural number between 0 and 14. If both sides send this option, the receive window scaling is enabled. This allows the use of a larger receive window than would be allowed by the format of the TCP header, where the receive window is limited to 16 bits (and therefore 65535 bytes). This avoids the fact that the size of the receiver window field in the TCP limits the throughput of a TCP connection.
4. The TCP Timestamp option
   This option contains two 32-bit numbers, which often encode some timing information in millisecond granularity. It is used to improve the TCP performance.

TCP is specified using a state event machine. Initially, an endpoint is in the CLOSED state. When the endpoint is willing to accept TCP connections (on the server side), the TCP endpoint is moved to the LISTEN state. When receiving the SYN segment from the client side and replying with a SYN ACK segment, the endpoint enters the SYN RECEIVED state. Once the TCP endpoint receives the ACK segment sent by the client, the TCP endpoint enters the ESTABLISHED state. These states can be observed using the `netstat` or `sockstat` command line tools.

The Application Programmers Interface (API) used to control TCP endpoints is the socket API. Programs normally use a listening socket, which tells the TCP implementation that it is OK to accept TCP connections on this endpoint, and for each accepted TCP connection the programs use a separate socket for each TCP connection. Applications can set parameters on the listening socket and most of the time these settings are then inherited by the accepted sockets. This article focuses on the TCP connection setup on the server side. It should be noted that this functionality applies to all TCP stacks (default, RACK, BBR, …).

This article focuses on the TCP connection setup on the server side.

SYN Flooding Attacks

When TCP was initially implemented, a new TCP endpoint was created whenever a SYN segment was received for a TCP endpoint in the LISTEN state. This required a memory allocation and resulted in a new TCP endpoint in the SYN RECEIVED state. All necessary information including the information related to the TCP options received in the SYN segment was stored in the TCP endpoint. This was done without any verification of the information provided, the IP address, and the TCP port number.

This allowed an attacker to send many SYN segments to a server and the server would allocate TCP endpoints until it ran out of resources. Therefore, the attacker could perform a denial-of-service attack, since once the server has no resources available anymore, it would not accept SYN segments from valid clients. The attacker only needs to send SYN segments, in particular, the attacker would not respond to any SYN ACK segments received. The attacker can even use spoofed IP addresses (IP addresses the attacker does not own).

The goal of this SYN flooding attack is that the receiver runs out of resources and is therefore not able to provide the service it is intended to provide. In FreeBSD, there are two mitigations implemented in the TCP stack for dealing with SYN flooding attacks:

1. Reducing the amount of memory allocated when a TCP endpoint moves from CLOSED to SYN RECEIVED state. This is done by using the SYN cache described in the next section.
2. Not allocating any memory when processing the incoming SYN segment. This is done by using SYN cookies as described in the section following SYN cache.

SYN Cache

The initial implementation of the SYN cache was added to the FreeBSD source tree in November 2001. It reduces the memory overhead of TCP endpoints in the SYN RECEIVED state by not allocating a full TCP endpoint, but a TCP SYN cache entry (`struct syncache` as defined in `sys/netinet/tcp_syncache.h`) instead. A TCP SYN cache entry is smaller than a TCP endpoint and only allows storage of information relevant in the SYN RECEIVED state. This information includes:

- The local and remote IP address and TCP port number.
- The information relevant for performing timer-based retransmissions of the SYN ACK segment.
- The local and remote TCP initial sequence number.
- The MSS reported by the peer in the MSS option of the received SYN segment.
- The local and remote window scaling shift values exchanged in the relevant window scaling options of the SYN and SYN ACK segments.
- Whether window scaling, timestamp, and SACK support were negotiated.
- Accurate ECN state.
- Additional IP layer information.

When a SYN segment is received for a listening endpoint, a SYN cache entry is allocated, the relevant information is stored in it and a SYN ACK segment is sent in response. If SYN cookies are disabled and there is a bucket overflow, the oldest SYN cache entry in the bucket is tossed. If the corresponding ACK segment is received, a full TCP endpoint is created with the data from the SYN cache entry and then the SYN cache entry is freed. The SYN cache also assures that the SYN ACK segment is retransmitted in case the corresponding ACK segment is not received in time.

When a SYN segment is received for a listening endpoint, a SYN cache entry is allocated.

The `sysctl`-variable `net.inet.tcp.syncookies` (which defaults to 1) control whether SYN cookies, as described in the next section, will be used in combination with the SYN cache to cover the case where no SYN cache entry can be allocated or looked up.

The SYN cache is vnet specific and organized as a hash table. The number of buckets is controlled by the loader tunable `net.inet.tcp.syncache.hashsize` (default 512). The maximum number of SYN cache entries in each hash bucket is controlled by the loader tunable `net.inet.tcp.syncache.bucketlimit` (default 30). There is also an overall limit of SYN cache entries given by the loader tunable `net.inet.tcp.syncache.cachelimit` (default is 15360 = 512 * 30). The number of currently used SYN cache entries is reported by the read-only `sysctl`-variable `net.inet.tcp.syncache.count`.

There are additional `sysctl`-variables relevant to the SYN cache. These are:

- `net.inet.tcp.syncache.rst_on_sock_fail`
  Controls for sending an RST segment or not in case a socket can't be created successfully (which defaults to 1).
- `net.inet.tcp.syncache.rexmtlimit`
  The maximum number of retransmissions of a SYN ACK segment (which defaults to 3).
- `net.inet.tcp.syncache.see_other`
  Control the visibility of the SYN cache entries (which defaults to 0).

The TCP SYN cache allows the server side to perform a fully functional handshake with a minimized memory resource. There is no functional difference to using a full TCP endpoint for TCP endpoints in the SYN RECEIVED state. Even tools like `netstat` or `sockstat` will report the entries from the SYN cache.

Supporting additional TCP options is not a problem, since the TCP SYN cache entry can be expanded.

The `sysctl`-variable `net.inet.tcp.syncookies_only` (defaulting to 0) can be used to disable the use of the SYN cache. In this case, only SYN cookies described in the next section will be used.

SYN Cookies

An additional level of protection against SYN flooding attacks was added to the SYN cache implementation in December 2001. Instead of allocating a smaller amount of memory when processing a received SYN segment, the relevant information is stored in a so-called SYN cookie and sent to the client in the SYN ACK segment. Then the client is expected to reflect the SYN cookie in the ACK segment. When the ACK segment is finally processed by the server, all relevant information is in the SYN cookie and the ACK segment. So, the server can create a TCP endpoint in the ESTABLISHED state. This way a SYN flooding attack does not result in any memory exhaustion. However, the generation of the SYN cookie mustn't require too many CPU cycles. If the SYN cookie generation is not cheap CPU-wise, it might allow a denial-of-service attack: this time not against the memory resource, but against the CPU resource.

The only field in the TCP header that can be chosen arbitrarily by the server and is reflected by the client is the initial sequence number of the server. This field is a 32-bit integer and therefore used as the SYN cookie.

The MAC uses a secret key, which is updated every 15 seconds.

In FreeBSD, these 32 bits are split into a 24-bit message authentication code (MAC) and 8 bits, which are used as follows:

- 3 bits for encoding one of 8 MSS values: 216, 536, 1200, 1360, 1400, 1440, 1452, 1460. For MSS values sent by the client in the MSS option not in this list, the largest value not exceeding the given one is used.
- 3 bits for encoding if the peer does not support window scaling or uses one of the 7 values: 0, 1, 2, 4, 6, 7, 8. If the client was sending a value not in the list, the largest value not exceeding the given one is used.
- 1 bit for encoding whether the client sent the SACK-permitted option or not.
- 1 bit for selecting one of two keys.

The MAC uses a secret key, which is updated every 15 seconds. The current and the last secret keys are kept around and used based on the bit in the SYN cookie for selecting the secret key.

The computation of the MAC includes the local and remote IP addresses, the initial sequence number of the client, the above 8 bits, and some internal information. From the MAC, 24 bits are generated and combined with the 8 bits from above, the SYN cookie is constructed.

When the ACK segment of the three-way handshake is received by the server, the MAC is verified. If that is successful, the TCP endpoint is created based on the information in the SYN cookie, which provides an approximation of the MSS option, an approximation of the window shift, and whether the client announced support for the SACK extension. All the other relevant information must be recovered from the ACK segment. This recovered information includes the local and remote IP addresses and port numbers, the local and remote initial sequence numbers whether the TCP timestamp option is used or not, and in the case it is, what the current parameters are.

Comparison of SYN Cache and SYN Cookies

The advantage of SYN cookies compared to the SYN cache is very clear: no memory allocation when a new SYN ACK segment is received. However, using SYN cookies also has its downsides:

- The MSS is approximated by 8 values, all smaller than or equal to 1460. Therefore, there is no support for MTUs larger than 1500 bytes for IPv4.
- The shift used for window scaling is approximated by 7 values, all smaller than or equal to 8. This means that larger window shifts above 8 are not supported and thus the connection will have a smaller window size.
- No support for TCP options other than the ones widely deployed right now. This makes it hard to support new TCP options used to negotiate new TCP features.
- No retransmissions of the SYN ACK segment, if a SYN ACK segment is lost the endpoint initiating the connection will have to retry sending its SYN segment.
- No visibility of TCP endpoints in the SYN RECEIVED state.
- No support for IP-level information.

Using the SYN cache does not have any of these limitations and is transparent but requires a memory allocation for each TCP endpoint in the SYN RECEIVED state.

Combined Usage of SYN Cache and SYN Cookies

Using only SYN cookies provides a better mitigation against SYN flooding attacks than using the SYN cache, but it comes with the drawback of limited functionality. Therefore, the default configuration of FreeBSD enables the SYN cache in combination with SYN cookies. This means that when a received SYN segment is processed, a SYN cache entry is generated, and the SYN ACK segment being sent contains a SYN cookie. If a SYN cache bucket overflows, it is assumed that this happens due to an ongoing SYN flooding attack, and therefore using the SYN cache is paused. During this time, only SYN cookies are used.

This additional functionality, introduced in September of 2019, gives the advantages of the SYN cache during normal operation, but also the improved protection of SYN cookies when a SYN flooding attack is going on.

RANDALL STEWART (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer since 2006. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He is currently an independent consultant.

MICHAEL TÜXEN (tuexen@freebsd.org) is a professor at the Münster University of Applied Sciences, a part-time contractor for Netflix, and a FreeBSD source committer since 2009. His focus is on transport protocols like SCTP and TCP, their standardization at the IETF and their implementation in FreeBSD.