Resisting SYN flood DoS attacks with a SYN cache

Jonathan Lemon jlemon@FreeBSD.org
FreeBSD Project

Abstract

Machines that provide TCP services are often susceptible to various types of Denial of Service attacks from external hosts on the network. One particular type of attack is known as a SYN flood, where external hosts attempt to overwhelm the server machine by sending a constant stream of TCP connection requests, forcing the server to allocate resources for each new connection until all resources are exhausted. This paper discusses several approaches for dealing with the exhaustion problem, including SYN caches and SYN cookies. The advantages and drawbacks of each approach are presented, and the implementation of the specific solution used in FreeBSD is analyzed.

1 Introduction

The Internet today is driven by machines that communicate using services layered on top of the TCP/IP protocols, these include HTTP, ftp and ssh, among others. The accessibility of these services is dependent on how well the underlying transport protocol performs, which in this case is TCP. If TCP is unable or unavailable to deliver the layered service to a remote machine, the user perceives the site as being dead or inaccessible. Perhaps merely an inconvenience in the past, this is a much more serious problem today as machines are being used for commerce and business.

One way that a malicious host can attempt to deny services provided by a server machine is by sending a large number of TCP open requests. These are known as SYN packets, named after the specific bit in the TCP specification, hence this type of Denial of Service attack is often called SYN bombing or SYN flooding. When the server receives this packet, it is interpreted as a request by the remote host to initiate a TCP connection, and at this point, the OS on the server machine commonly allocates resources to track the TCP state. By sending these requests in rapid succession, an attacker can exhaust the resources on the machine to the point where it becomes unresponsive, or crashes.

The server can attempt to reduce the impact of the flooding by changing the resource allocation strategy that it uses. One approach is to allocate minimal state when the initial request is received, and only allocate all the resources required when the connection is completed; this is termed a SYN cache. Another approach is to allocate no state, but instead send a cryptographic secret back to the originator, called a cookie; hence the name SYN cookie. Both approaches are intended to allow the machine to continue to provide its services to valid users.

The rest of this paper is structured as follows: Section 2 examines the details involved in the SYN flood Denial of Service attacks and examines the approaches of different defenses. Section 3 details the experimental setup used for testing, while Section 4 describes the current system behavior and motivation for change. Section 5 discusses the SYN cache implementation and presents the performance measurements from the change, while Section 6 does the same for SYN cookies. Section 7 discusses related work, and the paper concludes with a summary in Section 8.

2 TCP Denial of Service

A traditional TCP 3 way handshake for establishing connections is shown in Figure 1, where state is allocated on the server side upon receipt of the SYN to hold information associated with the incomplete connection. The goal of a SYN flood is to tie up resources on the server machine, so that it is unable to respond to legitimate connections. This is accomplished by having the client discard the returning SYN,ACK from the server and not send the final ACK. This results in the server retaining the partial state that was allocated from the initial SYN.

The attacker does not necessarily have to be on a fast machine or network to accomplish this. Standard TCP

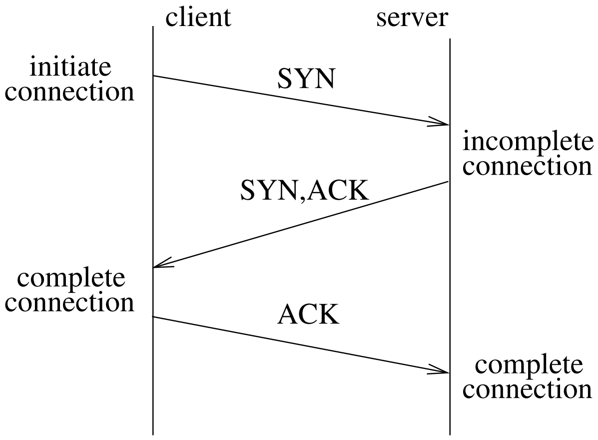

Figure 1: Standard TCP 3 way handshake.

will not time out connections until a certain number of retransmits have been made, which usually is a total of 511 seconds[7]. Assuming a machine permits a maximum of 1024 incomplete connections per socket, this means an attacker has only to send 2 connection attempts per second to exhaust all allocated resources. In practice, this does not form a DoS attack, as existing incomplete connections are dropped when a new SYN request is received. The time required for the server to send a SYN,ACK and have the client reply is known as the round trip time (RTT); if an ACK arrives at the server but does not find a corresponding incomplete connection state, the server will not establish a connection. By forcing the server to drop incomplete connection state at a rate larger than the RTT, an attacker is able to insure that no connections are able to complete.

Each connection is dropped with $1/N$ probability, and if the goal is to recycle every connection before the average RTT, an attacker would need to flood the machine at a rate of N/RTT packets per second. For a listen queue size of 1024, and a 100 millisecond RTT, this results in about 10,000 packets per second. A minimal size TCP packet is 64 bytes, so the total bandwidth used is only 4Mb/second, within the realm of practicality.

As the sender may forge their source IP address, a defense that relies on filtering packets based on the source IP will not be effective in all cases. Using a random source IP address will also cause more resources to be tied up on the server if a per-IP route structure is allocated.

Often it is not possible to distinguish attacks from real connection attempts, other than by observing the volume of SYNs that are arriving at the server, so the machine needs to be able to handle them in some fashion.

In order to defend against this type of attack, the amount of the amount of state that is allocated should be reduced, or even eliminated completely by delaying allocation until the connection is completed. Two known approaches to do this are known as SYN cache and SYN cookies. The caching approach is similar to the existing behavior, but allocates a much smaller state structure to record the initial connection request, while the cookie approach attempts to encode the state in a small quantity which is returned by the client when the TCP handshake is completed.

2.1 Defenses

SYN caching allocates some state on the machine, but even with this reduced state it is possible to encounter resource exhaustion. The code must be prepared to handle state overflows and choose which items to drop in order to preserve fairness.

The initial SYN request carries a collection of options which apply the TCP connection, these commonly include the desired MSS, requested window scaling for the connection, use of timestamps, and various other items. Part of the purpose of the allocated state is to record these options, which are not retransmitted in the return ACK from the client.

SYN cookies do not store any state on the machine, but keep all state regarding the initial TCP connection in the network, treating it as an infinitely deep queue. This is done by use of a cryptographic function to encode all information into a value that is sent to the client with the SYN,ACK and returned to the server in the final portion of the 3 way handshake. While this approach appears attractive, it has the drawback of not being able to encode all the TCP options from the initial SYN into the cookie. These options are then lost, denying the use of certain TCP performance enhancements.

A secondary problem related to cookies is that the TCP protocol requires unacknowledged data to be retransmitted. The server is supposed to retransmit the SYN,ACK before giving up and dropping the connection, whereupon a RST is sent to the client in order to shutdown the connection. When SYN,ACK arrives at a client but the return ACK is lost, this results in a disparity about the established state between the client and server. Ordinarily, this case will be handled by server retransmits, but in the case of SYN cookies, there is no state kept on the server, and a retransmission is not possible.

SYN cookies also have the property that the entire connection establishment is performed by the returning ACK, independent of the preceding SYN and SYN,ACK transmissions. This opens the possibility of flooding the server with ACK requests, in hopes that one will contain the correct value which allows a connection to be established. This also provides an approach to bypass firewalls which restrict external connections by filtering out incoming packets which have the SYN bit set, since initial SYN packet is no longer required to establish a connection.

Another difficulty with cookies is that they are incompatible with transactional TCP[6]. T/TCP works by sending monotonically increasing sequence numbers to the peer in the TCP options field, and uses previously received sequence numbers to establish connections on the initial SYN, eliminating the 3 way handshake. However, use of the T/TCP sequence numbers is mandatory once a TCP connection is initiated, and this requires the server to record the initial sequence number, and whether the T/TCP option was requested.

Thus cookies cannot be used as the normal line of defense in a high performance server. The usual approach is to use a state allocation mechanism, and fall back to using cookies only after a certain amount of state has been allocated. This is the approach taken by the the Linux kernel implementation.

3 Experimental Setup

The code base used was FreeBSD 4.4-stable, from sources as of November 14th, 2001. The target machine used for testing was an Intel PIII/850, with 320MB of memory, and was equipped with an onboard Intel EtherExpress 100Mb/s chip, an Intel 1000/Pro Gigabit adapter and a NetGear GA620 Gigabit adapter. The NetGear adapter was attached directly to a second machine that acted as a packet source, while the Intel adapter was directly attached to a third machine that acted as a packet sink. A fourth machine was connected via the 100Mb port and was used for taking timing measurements of real connection requests to the test machine.

A default route was installed on the test machine so that all incoming traffic from the source was sent out to the sink via the other gigabit link. The kern.ipc.somaxconn parameter, which controls the maximum listen backlog, was raised to 1024, while net.inet.tcp.msl was turned down to 30 milliseconds in order not to run out of TCP ports. Mbufs and mbuf clusters were set to 65536 and 16384 respectively, and the system was monitored to insure that the mbuf limit was not reached.

When SYN flooding the box, the source was configured to generate SYN packets at a rate of 15,000 packets per second. This rate was chosen as a load that the box could reasonably handle without becoming susceptible to receiver livelock. Under this load, the box was handling upwards of 30,000 packets per second, incoming and outgoing. The source addresses of the SYN packets were randomly chosen from the 10.x.x.x subnet, and the source port numbers and ISS were also randomly generated.

A small program that accepted and closed incoming connections was run on the test machine, in order to provide a listen socket for incoming packets. Timing measurements were taken on the control machine that was attached to the 100Mb port, which involved taking 2000 samples of the amount of time required for a connect() call to complete to the target machine.

4 Motivation

Initial tests were performed on the target machine using an unmodified 4.4-stable kernel while undergoing SYN flooding. The size of the listen socket backlog was varied from the default 128 entries to 1024 entries, as permitted by kern.ipc.somaxconn. The results of the test are presented in Figure 2.

In this test, with a backlog of 128 connections, 90% of the 2000 connections initiated to the target machine complete within 500ms. When the application specifies a backlog of 1024 connections in the listen() call, only 2.5% of the connections complete within the same time period.

The dropoff in performance here may be attributed to the fact that the sodropablereq() function does not scale. The goal of this function is to provide a random drop of incomplete connections from the listen queue, in order to insure fairness.

However, the queue is kept on a linear list, and in order to drop a random element, a list traversal is required to reach the target element. This means that on average, $1/2$ of the total length of the queue must be traversed to reach the element; for a listen queue backlog of 1024 elements, this leads to an average of $(3 * (1024/2))/2$, or 768 elements traversed for each incoming SYN.

Profiling results show that in this particular case, the system spends 30% of its time in sodropablereq(), and subjectively, is almost completely unresponsive. Examining the graph, we see that there is a considerable dropoff in performance between the backlog cases of 768 entries and 1024 entries, the reason of which is unclear. It is likely that there is a 'knee' in the performance curve is between these points, and system may have reached a point of saturation.

For the rest of the paper, a listen queue backlog of 1024 entries is used, as this is a realistic value used on production systems[4]. It also serves to illustrate the performance gains from a syncache or syncookie implementation.

4.1 Implementation

The new implementation for FreeBSD provides a SYN cache as the first approach for handling incoming connections, and has every connection pass through the

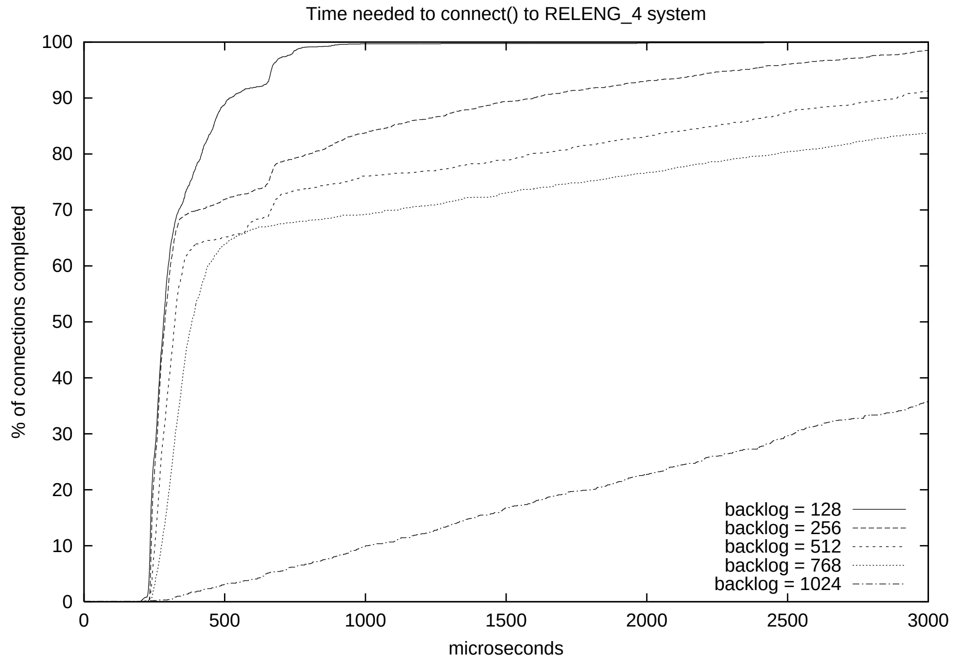

Figure 2: Time needed to connect() to a RELENG_4 system under a SYN flood attack. The kern.ipc.somaxconn parameter on the remote machine was set to 1024, and the size of the listen backlog was varied for each run.

cache. If an existing entry in the cache needs to be evicted, a sysctl tunable controls the optional behavior of sending back a SYN cookie instead of evicting the entry from the cache. In the following discussion, first the implementation of the syncache will be presented, independent of syncookies, with the next section explaining how syncookies modify the behavior of the syncache.

5 SYN Cache

The syncache implementation replaces the per-socket linear chain of incomplete queued connections with a global hashtable, which provides two forms of protection against running out of resources. These are a limit on the the total number of entries in the table, which provides an upper bound on the amount of memory that the syncache takes up, and a limit on the number of entries in a given hash bucket. The latter limit bounds the amount of time that the machine needs to spend searching for a matching entry, as well as limiting replacement of the cache entries to a subset of the entire cache. A global table was chosen instead of a per-socket table as it was felt this would be a more efficient use of system resources. A current implementation restriction that all kernel virtual address space for the memory used at interrupt time must be pre-allocated was also a factor in this decision.

One of the major bottlenecks in the original code was the random drop implementation from the linear list, which did not scale. This bottleneck avoided in the syncache, since the queue is split among hash buckets, which are then treated as FIFO queues instead of using random drop. Another way of viewing this is to consider the original linear list partitioned up into a number of sublists equivalent to the size of the hash table, where choosing a bucket enables us to choose which section of the list to drop. Since the hash distribution across the buckets should be uniform, this is an approximate model of choosing a random list entry to drop.

The hash value is computed on the incoming packet using the source and destination addresses, the source and destination port, and a randomly chosen secret. This value is then used as an index into a hash table, where syncache entries are kept on a linked list in each bucket. The secret is used to perturb the hash value so that an attacker cannot target a specific hash bucket and deny service to a specific machine.

While on the surface it may appear that an attacker could implement a DoS by targeting a hash bucket so that a legitimate connection does not reside on the queue long enough to establish a connection, the risks are migitated by the use of the hash secret. Additionally, since the port number of the connecting machine is used in the hash calculations, a second connection attempt from the client machine tends to result in a second hash bucket chosen, further styming any attempt by an attacker to target a specific bucket.

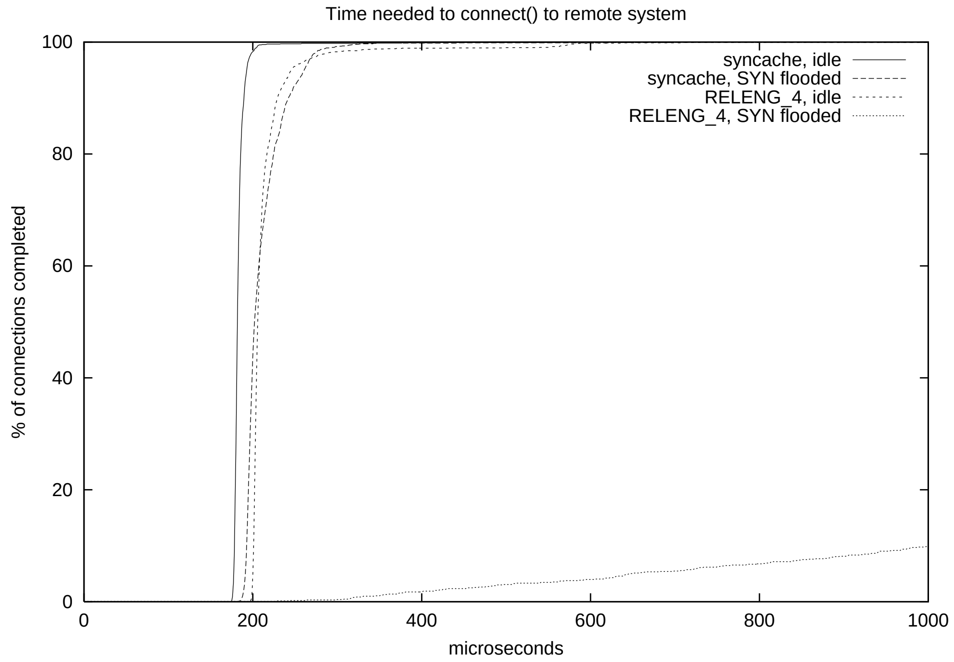

Figure 3: Time needed to connect() to remote system.

If the entry is not found in the bucket, a new syncache entry is created and added to the cache. If the new entry would overflow the per-bucket limit, the oldest entry within that bucket is dropped. If the total number of entries in the cache is exceeded, the oldest entry in the cache is dropped. This way, both the memory usage of the syncache and the amount of CPU time needed to search the hash table are bounded. The user is able to control the sizing of these limits via the following loader tunables established at boot time:

```
net.inet.tcp.syncache.hashsize
net.inet.tcp.syncache.cachelimit
net.inet.tcp.syncache.bucketlimit
```

The *cachelimit* setting determines the maximum number of syncache entries that may be allocated, and bounds the overall memory usage of the system. *hashsize* controls the size of the hash table and should be a power of 2. Finally, *bucketlimit* caps the size of each hash chain, and limits the number of entries that must be searched when looking for a matching SYN entry. However, as the list is handled in FIFO order, an entry must stay on the list for at least one round trip time (RTT) to the remote system in order to successfully establish a connection, so this must be considered when choosing a value for *bucketlimit*.

There are two additional sysctl parameters of interest:

```
net.inet.tcp.syncache.count
net.inet.tcp.syncache.rexmtlimit
```

The first entry is read-only, and indicates how many entries are currently present in the syncache. The second determines how many times a SYN,ACK should be retransmitted to the remote system, and defaults to 3. Three retransmits corresponds to $1 + 2 + 4 + 8 = 15$ seconds, and the odds are that if a connection cannot be established by then, the user has given up.

5.1 Syncache performance

The syncache tests were performed on the target machine using the following system default values: hashsize = 512, cachelimit = 15359, bucketlimit = 30. The results of the test are presented in Figure 3.

As the graph shows, the syncache is effective at handling a SYN flood while still allowing incoming connections. Here, 99% of the incoming connections are completed within 300 microseconds, which is on par with the time required to connect to an idle unmodified system. For comparison, the performance of an unmodified system experiencing a SYN flood is also shown. All of the trials in the test were performed with a listen queue length of 1024.

One interesting result is that the connection latency decreases even when the target box is not experiencing SYN flooding. This is shown by comparing the 'syncache idle' and 'RELENG_4 idle' lines on the graph, which indicate how long it takes to connect to a quiescent system. This result may be attributed to the smaller

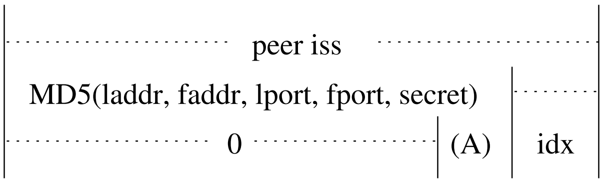

Figure 5: Layers of data in the syncookie.

data structure used to hold the syncache, as the size of the TCP and socket structures allocated and initialized on an unmodified system total 736 bytes, while the smaller syncache structure is only 160 bytes.

6 SYN Cookies

When a syncache bucket does overflow, a fallback mechanism exists which permits sending back a SYN cookie instead of performing oldest FIFO drop of an entry on the hash list. This section explains the syncookie approach, and outlines how the cookie is constructed.

The cookie is sent to the remote system as the system's Initial Sequence Number (ISN), and then returned in the final phase of TCP's three way handshake. As connection establishment is performed by the returning ACK, a secret should be used to validate the connection, which is concealed from the remote system by use of a non-invertible hash. To prevent an intermediate system from collecting cookies and replaying them at a later date, the cookie should also contain a time component. The solution chosen here was to keep a table of secrets which have a bounded lifetime, which has an added benefit of regularly changing the secret which is sent back to the remote system. Figure 5 shows the internal structure of the cookie.

The basis of the implementation is a table of 128 32-bit values obtained from arc4random(). Each entry is used for a duration of 31.25 milliseconds, and has a total lifetime of 4 seconds, which was chosen as a reasonable upper bound for the RTT to the remote system, as SYN,ACK containing the cookie must reach the system and be returned before the secret expires.

In order to generate a cookie, the system tick timer is scaled into units of 31.25 milliseconds by use of divide and shift operations, with the result used to choose the correct window index. If the secret in the current window has expired, a new 32-bit secret is obtained from arc4random(), and the timeout is reset.

The local address, foreign address, local port, foreign port and secret are passed through MD5 to create the initial basis of the cryptographic hash, with 25 bits being used in the cookie, and 7 bits containing the window index. The peer MSS from the TCP options section of the initial SYN is fit into one of 4 predefined MSS values, and the resulting 2 bit index is xor'ed into the mix, as shown by (A) in Figure 5. Finally, the peer's 32-bit ISS is xor'ed in to generate the final cookie, which is sent back to the connecting system as the ISN.

Since no state is kept on the server machine, any returning ACK which contains the correct TCP sequence numbers may serve to establish a connection. Validating the ACK is the reverse of the above process. First the peer's sequence number is removed, and then the 7 bit index is used to select the correct window. If the secret has expired, then the ACK is immediately discarded without further processing. This insures that the system does not have to check every incoming ACK unless a syncookie was recently sent. If the timeout indicates that the secret is valid, it is used in the MD5 hash computation. The ACK is considered valid if the remaining 23 bits evaluate to 0.

In practice, this means that a remote system has 4 seconds to try and brute force a space of 2^{23} entries.

6.1 SYN cookie performance

The syncache tests were performed on the target machine by enabling the following sysctl

```
net.inet.tcp.syncookies
```

and then performing the tests in the usual fashion. The results of the test are presented in Figure 4.

The results show that syncookies provides slightly better performance than syncache alone. This may be due to the fact that the syncache calls arc4random() for every SYN,ACK it sends, while the syncookie routines primarily call MD5(). Investigation into the reason for the performance disparity is ongoing, but the results are not available at this time.

There are also a few unusual results here: There does not appear to be a straightforward explanation for the jump in completed connections at 700 microseconds. This is not due to TCP retransmissions, as the first retransmission timeout is set at one second. A possible explanation is that the system is busy executing the interrupt handler for either of the Gigabit adapters, and is delayed in servicing the 100Mb adapter.

Also of interest to note is that while 100% of the syncache connections have completed in 1 second, the same isn't true for syncookies. This shouldn't happen, as no packet loss on the 100Mb segment was observed, and the system did not run out of mbufs. Upon further investigation, this turned out to be a minor bug in the VM system where the initial boot-time allocation request was rounded improperly, leading to a shortage of syncache structure entries. With the current code, at least one entry is always needed in order to send the SYN,ACK reply.

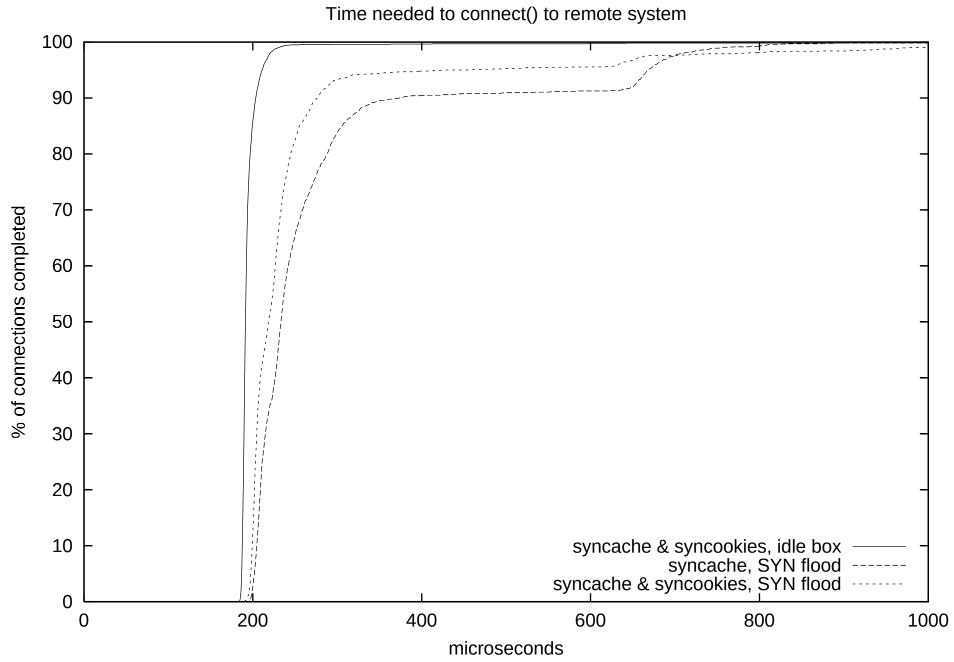

Figure 4: Performance comparison of a system with syncache and syncookies over one using only syncache.

6.2 Round trip performance

Prior measurements were taken by timing how long it takes for a connect() call to complete on the client machine. This corresponds to the time required to complete 2 stages of a TCP handshake, since the client machine enters the ESTABLISHED state as soon as it receives a SYN,ACK. An unanswered question is how long it takes the server to enter the ESTABLISHED state, from the time the initial SYN is sent from the client. This time may be affected by the different processing requirements to verify the ACK, and may fail if the original syncache record no longer exists.

To verify failure was not a concern, the experimental setup was modified to include the time required to read() a byte from the server, which can be viewed as a 4 way handshake: transmit SYN, receive SYN,ACK, transmit ACK, receive data. The results for this test are presented in Figure 6.

On an unloaded box, there is no measurable difference in performance between the syncache and syncookies approaches. However, when the box is loaded, the combination of syncache and syncookies outperforms a pure syncache configuration. Again, as there are no TCP retransmits occurring, the performance difference is not due to entries getting dropped from the syncache hash buckets. This also indicates that the bucket depth of 30 entries that is used in these tests is sufficient to handle the RTT across the local LAN; connections are getting established before they are dropped.

The difference between the two algorithms could be explained by the difference in ISS generation, or by the fact that the standalone syncache needs to perform FIFO drop for a bucket, which is bypassed when syncookies are in use. However, it is not expected that the list management requirements, which consist of few TAILQ_* calls, would be significant. The investigation into the performance difference is still ongoing.

In comparison to the unmodified system presented in Figure 2, there is a dramatic improvement. In this experiment, clients were able to connect to the server and perform useful work (reading one byte), with all attempts completing within 1 second. In the unmodified system, 90% of the connections still had not completed the TCP handshake after 1 second. Even with reduced queue depths, the performance of the unmodified system does not match the new code.

7 Previous Work

David Borman wrote a patch for BSDi which implemented a SYN cache in October 1996, which was released as an official BSDi patch [2]. This implementation used the cache only as a fallback mechanism in case the listen queue overflowed, and did not retransmit the SYN,ACK to the peer. The justification given was that since the host was under attack, performing retransmits would be a waste of CPU time [3].

This code was incorporated into NetBSD[5] in May

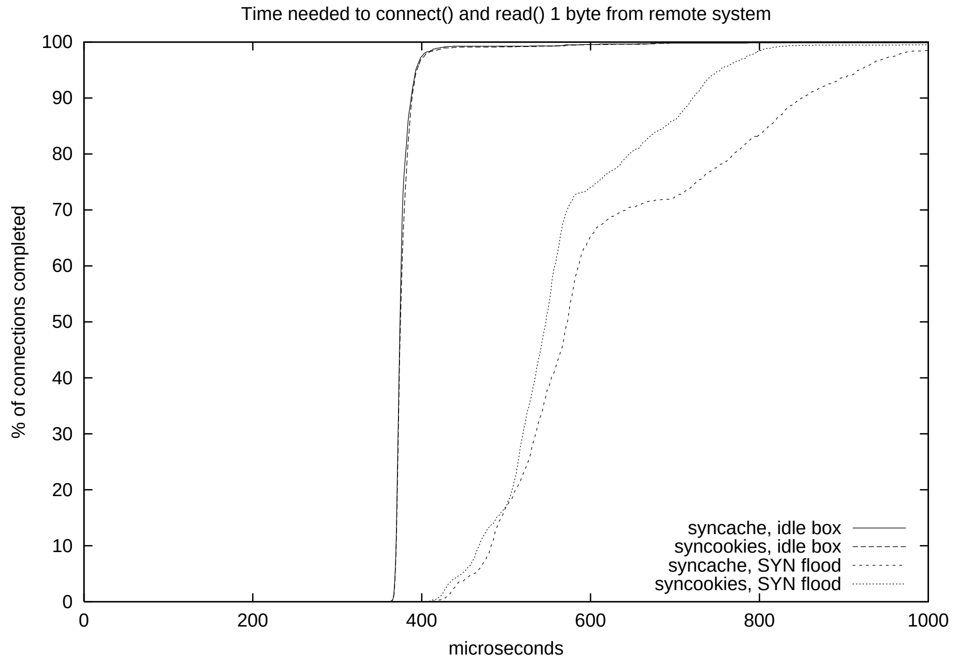

Figure 6: Time required to connect() to remote system and read() one byte in response. No errors at the system call level were observed during the test.

1997 and subsequently enhanced to perform retransmits, as well as having the cache handle all incoming connections, instead of only those which overflow the listen queue. The implementation described in this paper bears a strong resemblance to their existing code.

An alternate approach was taken by Linux, which chose to incorporate syncookies[1] as their defense against this style of attack. On these systems, the syncookie defense mechanism engages only when the normal listen queue overflows.

8 Further Work and Conclusion

When syncookies are enabled, the existing code does not drop any entries from the syncache, choosing to send a syncookie response instead. However, in practice this leads to the syncache being full of bogus entries from a SYN flood, and forces all legitimate connections to be handled by syncookies. Essentially, the system ends up behaving as if there is no syncache, which is not an ideal situation.

An alternate approach that may prove feasible is to use a syncookie as the ISN for all connections, instead using arc4random() in the syncache case. This would permit the replacement mechanism of entries within the syncache to operate as normal, as the returning ACK could be accepted by either by virtue of passing the syncookie check, or by matching an existing syncache entry. This approach is currently under investigation; one issue that needs to be addressed is whether the reduced entropy of a syncookie ISN provides adequate protection from remote attackers as compared to one from arc4random().

In this paper, we have seen that an unmodified machine provides unacceptable response times under a simple 10Mb/s SYN flood attack. Two approaches to handing this load are presented and evaluated, and we show that both are able to extensively migitate the effects of a SYN flood and allow the system to continue operating. This goal is reached by the dual approach of reducing memory consumption and state on the server side, and the use of better algorithms to handle a large number of incompleted connections. With the new code, the same hardware is now able to withstand a SYN flood attack while maintaining an acceptable level of service to legitimate clients.

References

[1] BERNSTEIN, D. J. Syn cookies. `http://cr.yp.to/syncookies.html`.

[2] BORMAN, D. Bsdi implementation of syn cache. `ftp://ftp.bsdi.com/private/44-syn-diffs.gz`.

[3] BORMAN, D. tcpip-impl posting. http://www.kohala.com/start/borman.97jun06.txt.

[4] Freebsd, tuning(7) man page. http://www.FreeBSD.org/cgi/man.cgi?query=tuning&apropos=0&sektion=7&man%path=FreeBSD+4.4-RELEASE&format=html.

[5] Netbsd. http://www.netbsd.org/.

[6] Rfc 1644. Transactional TCP.

[7] STEVENS, W. R. Tcp/ip illustrated.