Improving TCP/IP security through randomization without sacrificing interoperability

Michael James Silbersack
The FreeBSD Project

Introduction

Over the years, many security problems have been found in the TCP and IP protocols. This is not surprising; the authors of these protocols probably did not anticipate their creations being used on open, chaotic networks like today's internet. Had they envisioned our present reality, they most certainly would have included provisions to prevent spoofing, modification, and interception of data.

In the face of attackers that can intercept packets, not much can be done to improve TCP/IP without moving to IPSec or other protocols which encrypt the entire packet. However, in the face of spoofing attacks where the attacker has only partial information about the target connection, some improvements can be made.

Over the past few years, FreeBSD has moved slowly to make changes to our TCP/IP stack when security issues that required a change in network visible behavior were announced. There is a simple reason for this – almost every time we have made a reactionary change to the TCP/IP stack, users have reported compatibility problems.

This paper aims to describe the changes that FreeBSD has made to improve network security without sacrificing compatibility, and also to propose some new changes that will increase network security even further.

Four major topics are covered: TCP Initial Sequence Numbers, TCP Timestamps, IP ID values, and ephemeral port randomization.

TCP Initial Sequence Numbers

The topic of TCP initial sequence numbers has been written about many times. The Morris worm made news in 1988, Kevin Mitnick's spoofing attack on Tsutomu Shimomura made news in 1995, "The Problem with Random Increments" appeared in 2001, and Paul Watson's "Slipping in the Window" made the news in 2004. Despite these events, and the publishing of many excellent papers on the topic such as [Zal01] and [Zal02], this is still a topic worth discussing for one main reason: Every operating system still uses a different method of ISN generation!

This divergence is seemingly due to the fact that there is no RFC recommendation on initial sequence numbers that takes all security and compatibility issues into account. Additionally, no paper has reexamined all operating systems to see how effective the response to "Slipping in the Window" has been.

The importance of unpredictable TCP initial sequence numbers

The TCP protocol uses a 32-bit sequence number to track the current state of a connection; this sequence number is incremented for each octet of data sent over the connection, and in response to packets with the SYN or FIN flags. TCP connections are bidirectional, so there are

effectively two sequence numbers that must be tracked per connection, although each is effectively independent.

There are three categories of attacks that can be performed if an attacker can guess the current sequence number of a connection: Connection spoofing, connection resetting, and data injection.

Connection spoofing is potentially the most serious of the attacks, and was described first in [Mor88]. In order to spoof a connection, one must send a fabricated SYN packet with a false source IP address, then guess the sequence number than the destination system will respond with in its SYN-ACK packet. If this value can be guessed and put into a fabricated ACK packet, the server will believe that it has established a connection with the false source IP address. While the attacker can not receive data from the victim in this scenario, he can send data. This is a very dangerous attack when services that can grant permissions based on IP addresses, such as rlogin, are attacked.

As connection spoofing requires an exact guess in order for success to occur, increasing each initial sequence number by a random positive increment over the previously used ISN provides moderate protection from the attack. If a random positive increment in the range of one to one million is used, the attacker must send on average five hundred thousand packets to successfully spoof a single connection. Due to this difficulty and also due to the removal of IP-based authentication in most programs today, connection spoofing is not any longer a serious threat.

Connection resetting attacks have the modest goal of interrupting an existing connection between two hosts. As pointed out in [Wat04], a spoofed RST packet need not be exact, and must only have a value within the TCP sliding window. With many operating systems using a 64K or larger window, this means that a brute force attack on the entire sequence space of a connection would only require 2^32 / 2^16 (65536) packets. When random positive increments are used, and the general range of a connections sequence numbers can be narrowed down, the attack becomes trivial.

Data injection attacks take advantage of the same TCP flaw/feature, but instead of sending a RST packet, they send a data payload. In the case of encrypted connections like SSH/SSL, this will merely result in the connection being terminated. In the case of more free-form protocols such as telnet, commands could probably be successfully injected.

Although most connections are too short lived and unimportant to be worth resetting / injecting data into, [Wat04] points out that BGP sessions are valuable enough to be targeted.

As the result of [Wat04], many improvements to TCP which would make these attacks less likely by reducing the range of sequence values accepted were proposed. Also suggested was the randomization of ephemeral ports to add an additional barrier to the attack. Unfortunately, the complexity, potential compatibility issues, and legal issues surrounding the proposed fixes have caused many operating systems (including FreeBSD) to only partially implement them.

Initial Sequence Number requirements

The original TCP document, RFC 793 states:

```
RFC 793, page 27:
To avoid confusion we must prevent
segments from one incarnation of a
connection from being used while
the same sequence numbers may still
be present in the network from an
earlier incarnation.  We want to
assure this, even if a TCP crashes
and loses all knowledge of the
sequence numbers it has been using.
When new connections are created,
an initial sequence number (ISN)
generator is employed which selects
a new 32 bit ISN.  The generator is
bound to a (possibly fictitious) 32
bit clock whose low order bit is
```

```
incremented roughly every 4
microseconds.  Thus, the ISN cycles
approximately every 4.55 hours.
Since we assume that segments will
stay in the network no more than
the Maximum Segment Lifetime (MSL)
and that the MSL is less than 4.55
hours we can reasonably assume that
ISN's will be unique.
```

Other than stating that the sequence numbers of connections which share the same IP/port tuple should have non-overlapping sequence numbers within the same MSL, which is defined to be 2 minutes, no other requirements are stated.

The goal of having monotonically increasing initial sequence numbers of course only matters when an IP/port tuple is reused within a short period of time. A system reboot (or NAT machine reboot) is one reason this can occur.

Another situation in which port reuse will occur is when a client machine makes frequent connections to a server, going through its entire ephemeral port range in the process. If the client quickly runs through this range and reuses the first ephemeral port, the SYN packet reaching the server will find a socket still in the TIME_WAIT state. In order to maintain the "quiet time" of the TIME_WAIT state, but to still allow new connections on that IP/port tuple to be accepted, the authors of the 4.2BSD TCP/IP stack added a simple check. If the ISN of a SYN packet coming in was greater than the value of the last sequence number used in the connection that previously occupied that IP/port tuple, the old socket would be discarded and a new connection would be established. Given the mod 1 arithmetic used in sequence number calculations and the 32-bit size of sequence numbers, this means that any value up to 31 bits in size greater than the previously used value would be accepted, and any value up to 31 bits less in size would be ignored until the TIME_WAIT socket timed out on its own.

This sequence number check, although originally a quick hack, made its way into many TCP/IP stacks over the years. An operating system that ignores this rule and attempts to send out SYN packets with random ISN values will find that roughly 50% of connections will fail in situations where TIME_WAIT recycling comes into play.

An attempt to emulate the monotonic increase algorithm from RFC 793 while making sequence number prediction hard is what presumably led to the random positive increment algorithms used by many operating systems in the 1990s.

In 2001, as a result of the information in [New01], this author did an ad-hoc survey of the ways in which open source operating systems validated initial sequence numbers, and determined that the BSD-derived TIME_WAIT check is the only actual requirement imposed on a TCP/IP stack author. SYN-ACK packets should exhibit a sequence value greater than the one used by the previous incarnation of a connection on that port, but no known operating system actually checks. Additionally, there is no requirement that an operating system use the same sequence space for SYN and SYN-ACK packets.

An improvement: RFC 1948

In RFC 1948, Steven Bellovin proposed a ISN generation algorithm that would create monotonically increasing ISN values that differed per IP/port tuple:

```
RFC 1948, page 2 - 3
...Instead, we use the current 4
microsecond timer M and set

ISN = M + F(localhost, localport,
remotehost, remoteport).

   It is vital that F not be
computable from the outside, or an
attacker could still guess at
sequence numbers from the initial
sequence number used for some other
connection.  We therefore suggest
that F be a cryptographic hash
function of the connection-id and
some secret data.  MD5 [9] is a
```

```
good choice, since the code is
widely available.
   The secret data can either be a
true random number [10], or it can
be the combination of some per-host
secret and the boot time of the
machine...
```

This algorithm performs exactly as expected, creating a unique value for each IP/port tuple that is then incremented at a constant rate by the system time. Unfortunately, there is one property of this algorithm that precludes it from being used as is. Since the time component increases at a constant rate and the hash component is constant, all future ISNs for a IP/port-tuple may be perfectly predicted once a single value has been observed. This flaw was noted in [Zal01], but no specific improvement was suggested.

Therefore, the following attack (inspired by comments from Robert Watson) is possible when RFC 1948 is used for generating the ISNs sent out in SYN-ACK packets. A spammer with a T1 connection he wishes to keep secret obtains a dial-up connection from an ISP that does not block connections to port 25. The spammer than makes connections from his dynamically assigned IP address to port 25 on each of his intended spam targets, logging the ISN returned and the OS fingerprint detected. Next, the spammer disconnects his modem, and puts the observed data into his mass mailing software. This software then proceeds to use the obtained data to forge connections to each of the target mail servers, causing spam to appear to originate from the dial-up IP address.

RFC 1948 usage in FreeBSD

In the spring of 2001, the results of [New01] showed that the random positive increments algorithm FreeBSD was still using was insecure. As a quick fix, the algorithm used by OpenBSD was ported over. Unfortunately, that algorithm created non-monotonic sequence numbers in SYN packets, and problems with TIME_WAIT recycling were quickly reported by users.

As a result, this author decided to start from scratch, and on August 22nd, 2001 the following ISN generation algorithm was committed to the FreeBSD TCP/IP stack:

ISN values in SYN-ACK packets are given random values, as returned by arc4random().

ISN values in SYN packets are generated by the RFC 1948 algorithm:

ISN = Time + MD5(remoteport, localport, remotehost, localhost, secret)

Time increments at 1MB/second and the secret is a 128-bit system-wide secret value that is seeded when the first outbound connection establishment occurs.

Two user-adjustable values are present:

net.inet.tcp.strict_rfc1948 – When set to 0 (the default), SYN-ACK values are filled with random data. If set to 1, the ISNs of SYN-ACK packets would be generated by the RFC 1948 algorithm.

net.inet.tcp.isn_reseed_interval – This determines the number of seconds between reseeding of the system-wide secret value. If left at 0 (the default), no reseeding will ever occur.

Not using RFC 1948 for SYN-ACK packets was motivated by the predictability issue described in the section above. Ad-hoc research of how other operating systems generated and interpreted SYN-ACK ISNs showed that no common operating system actually cared about monotonicity, so the most secure option was adopted – purely random ISNs.

On the other hand, SYN ISNs needed to be monotonic, due to the TIME_WAIT recycling sequence number requirement discussed above. No secure algorithm other than RFC 1948 could be found that satisfied this requirement. After some consideration, it was determined that prediction of SYN ISNs would not be a commonly abused problem. Such prediction would not allow

for connection spoofing, but would only allow for connection reset/data injection. However, the only connections vulnerable to this would be ones made from a static-IP server to the dynamic IP address which the attacker had previously occupied.

In order to see the results of this implementation, see the graphs in Appendix A.

In the four years since this algorithm was added to FreeBSD, the only major change that has occurred is the addition of TCP syncookies by Jonathan Lemon in December of 2001, as described in [Lem01]. When enabled, as they are by default, syncookies replace the random value in SYN-ACK packets. The algorithm used in SYN packets remains unchanged.

While the use of syn cookies remains controversial (no other operating system uses syn cookies by default), the randomness of the resulting sequence numbers is not in question, as shown by the graphs in Appendix A.

As far as anyone in the FreeBSD project is aware, there have been no reports of compatibility problems caused by this method of ISN generation.

During an audit of the syn cookie code in 2003, one security flaw was found. The secret value used when generating syn cookies was only 32 bits in length, allowing an attacker with a fast processor to perform a brute force hash attack and find out the secret. Once found, the secret could be used to perform perfect connection spoofing attacks against the victim until the secret expired (for about 60 seconds.) The flaw was fixed by simply increasing the secret size to 128 bits, making the attack infeasible. There have been no reports of this flaw being exploited in the wild.

A minor improvement to the FreeBSD algorithm

One oversight in the algorithm currently used by FreeBSD to generate SYN-ACK packets is that it tries to be too random. Specifically, when TIME_WAIT recycling occurs on a socket, a totally new ISN value is chosen. While this works properly under normal circumstances, it means that with certain values of ISN and certain old duplicate packets on the network, old data can be injected into the new connection.

As can be seen in the graph in Appendix B, the proposed change to SYN-ACK generation uses the existing scheme for ISN generation, except when a socket in the TIME_WAIT state is being recycled. In that case, a random positive increment from the previously used sequence number is used. However, if the TIME_WAIT socket expires, as occurs in the 130 second idle time shown above, a fresh ISN is once again chosen. Note that a few ports change their sequence numbers over the 30 second idle period; this appears to be the result of the case where TIME_WAIT sockets are created and expired on the client side of the connection rather than on the server side. This will be examined more thoroughly before the final implementation is completed.

Improving RFC 1948

During a discussion with Jeffery Hsu, which unfortunately can not be located, the idea for an improvement upon RFC 1948 was developed. The basic idea was this: Use two MD5 hashes in RFC 1948, slowly switching between them by averaging these values. This would allow the resulting sequence value to be monotonic, yet unpredictable over long periods of time. Such an algorithm would look like this:

```
md5_1 = MD5(remoteport, localport,
remotehost, localhost, secret1)
md5_2 = MD5(remoteport, localport,
remotehost, localhost, secret2)
ISN = Time +
(md5_1 * (reseed_interval -
elapsed_time)) / (reseed_interval)
+ (md5_2 * (elapsed_time)) /
(reseed_interval)
```

In order for this to work, a few additional

parameters must be specified. The rate of increase of time must be greater than the maximum rate of decrease from md5_1 to md5_2. If this premise is violated, ISNs will actually decrease when certain large values of md5_1 and small values of md5_2 occur.

Reseeding is straightforward in this algorithm. When the elapsed time catches up to reseed_interval, 100% of the value will be from md5_2, and 0% of the value will be from md5_1. At this point, the contents of secret2 should be transferred to secret1, secret2 should be filled with a new random value, and elapsed time should be reset to zero.

To see a visual representation of the ISN values generated by this dual hash algorithm, see Appendix B.

Although this algorithm is an improvement over RFC 1948, it is still predictable until the next reseed occurs. The possibility of using a non-linear function to transition between the two hash values is being investigated.

TCP Timestamps

TCP Timestamp values, as specified in RFC 1323, are intended to improve the performance of TCP by increasing the accuracy of RTT measurement, especially in the case of lost packets, and allow systems to determine if a wrapped sequence number is the result of an old packet or a new connection.

The simplest way to implement TCP timestamps is to use a single global time value for all connections. This is the basic implementation that FreeBSD and most other operating systems use. Unfortunately, this global counter leaks information in two ways. First, as this counter is derived from system uptime, it allows an attacker to know how long the system has been up. Such uptime information could be abused in a variety of ways. A simple scan of a network reveals which systems have long uptimes – and are therefore probably behind on security patches. A more patient attacker who logs this data over a long period of time could learn that a company performs weekly restarts and use this information as part of a timed attack.

The second piece of information leaked by a global counter is a system's identity. Given an range of IP addresses, an attacker looking at timestamp values will be able to determine which IP addresses belong to independent systems, and which IP addresses are aliases belonging to a single machine. This information could be very useful for an attacker – if no obvious security holes are found on one IP address of a machine, he could search all the other IP addresses of the machine for weaknesses, confident that he is still investigating the target machine and not wasting time on a honeypot or some other diversion.

There are three simple solutions to these information leakage problems. First, uptime monitoring may be partially foiled by initializing the global counter to a random value at boot time. Unfortunately, this will be ineffective if an attacker simply probes once per day and records his results. As such, it is an almost useless change.

Solution number two is to switch to using separate timestamp counters for each TCP connection, and to initialize a new connection's timestamp value to 0. This prevents an adversary from learning about a system's uptime, or determining if two IP addresses are hosted by the same computer.

Solution number three differs from number two in that the connection counter is initialized to a random value instead of to zero each time. This change is intended to prevent future attacks which might rely on predicting timestamp values.

While changes number two and three defeat the information leaks listed above, they also go against the spirit of RFC 1323, and may cause problems in certain situations. Section 4 of the RFC discusses how timestamps can be used for PAWS – Protection Against Wrapped Sequence numbers.

Section 1.2 of RFC 1323 describes a case where PAWS would ideally come into play:

> (2) Earlier incarnation of the connection
>
> Suppose that a connection terminates, either by a proper close sequence or due to a host crash, and the same connection (i.e., using the same pair of sockets) is immediately reopened. A delayed segment from the terminated connection could fall within the current window for the new incarnation and be accepted as valid.

If timestamps are generated from a global counter, the PAWS mechanism would have no problem determining that timestamps on packets delayed in the network are old. However, if each connection starts with the timestamp counter at 0, PAWS will be totally foiled, unable to tell new from old packets. In the case that random timestamp initialization is used, PAWS might work in some cases, but be fooled in others – the effects would be unpredictable.

Zeroing or randomizing timestamp values also causes a neat trick used by the Linux TCP/IP stack to break. In Linux, the TIME_WAIT sequence number check has been improved to allow a port to be recycled if the ISN is greater than the previously used value or if the timestamp is greater than the previously used value. This check allows operating systems that used randomized ISN values in SYN packets with a standard timestamp implementation to still recycle ports. However, an operating system that has modified ISN values and timestamps will be out of luck.

The unfortunate part about these changes is that the incompatibilities they cause might not be noticed except under carefully crafted test conditions. While the occurrence of these problems in actual usage is unlikely, the probability is that the problem will occur for some users at some time, which is why these changes have not been implemented in FreeBSD.

Using RFC1948 to improve timestamps

Luckily, there is one potential method of retaining compatibility with the PAWS mechanism while still defeating the information leaks discussed previously. The solution is simple – use the algorithm described in RFC 1948 to generate per-connection timestamps!

Using RFC 1948's algorithm to generate timestamps is not a perfect solution; as with its use in ISNs, it suffers from the issue that it is perfectly predictable to someone who can reconnect with the same IP and port pair. Therefore a service like netcraft, which probes on a regular basis, could determine uptime simply by looking for discontinuities in timestamp values. Someone attempting to determine if two IP addresses were hosted on a single computer could look for matching discontinuities to determine that a reboot of that single machine occurred.

The dual hash improvement on RFC 1948 unfortunately can not be used with timestamps. The differing slopes of each connection would make time measurement more difficult, and the extra math required to generate each timestamp would slow overall throughput.

One additional caveat when implementing RFC 1948 style TCP timestamps is that at least one heuristic in the Linux TCP stack compares the timestamp value of an incoming packet to the timestamp value of other packets to determine if that packet is legitimate when a syn flood is in progress. Assuming that other systems make similar assumptions, perhaps instead of using timestamps that are unique per IP/port pair it would be better to use timestamps that are just unique per IP.

Using timestamps to resist data reset/injection attacks

If TCP Timestamps are made per-tuple unique using the RFC1948 algorithm or

simply randomized at connection start time, using timestamps to greatly improve resistance to blind reset/injection attacks becomes simple to implement. RFC 1323 specifies in section 4.2.2 that timestamps are monotonically incremented at a constant rate between 1 and 1000 ticks per second. This allows a receiver to interpret the sender's timestamps, and use them as additional spoof protection.

Assuming that the sender is following RFC 1323, all a receiver must due in order to make blind spoofing connections on timestamped connections very difficult is to ensure that the following is true for each received packet:

(idle_seconds < 30) && (abs(TScurrent – TSlast) < 32 * 1024)

This still allows any legitimate packet that is up to 30 seconds late in arriving in, while blocking spoofed packets that do not fall into this window. As this algorithm accepts a window of 65536 timestamps out of a possible 2^32 at any point in time, an attacker who attempts to try a brute force reset/injection attack would be required to send an additional 2^16 times as many packets. This increases the difficult of any such attack significantly.

Note that this technique is perfectly compatible with senders using system-wide timestamps and timestamps zeroed at connection start time, but will provide very little added security in those cases.

Unfortunately, the timestamp check must be skipped on idle connections due to the possibility of a host rebooting, losing its timestamp counter, and attempting to reestablish a connection on the same ip/port tuple.

IP ID issues

The problems of sequential IP ID values were described first in [San98] and later in [Fyo] and other places. As of now, FreeBSD has not yet implemented any changes due to the perceived lack of importance of this issue and due to the performance penalties that would be incurred by some of the solutions.

Three main solutions have been implemented in different operating systems to solve the problems of predictable IP ID values.

The simplest option, implemented in Linux, was to use an IP ID value of zero for all packets that had the DF (Don't Fragment) bit set. Unfortunately, while this idea would work if all network devices were RFC compliant, it was discovered that certain network devices would fragment DF packets anyway, leading to a stream of fragments, all with an ID of 0. As a result of such misbehaving devices, the idea of zeroing the IP ID field has been abandoned.

A second solution, now implemented in both Linux and Solaris is to track per-IP state, setting up a seperate IP ID counter for each IP the system communicates with. Unfortunately, this solution would be expensive to implement in FreeBSD; FreeBSD has moved away from looking up per-IP state on every packet reception and transmission. The TCP hostcache, which now stores per-IP information such as MTU, RTT, and other information could be used for this purpose, but it would reduce performance.

A third solution was chosen by the authors of OpenBSD's IP stack. They use a linear congruential generator (LCG) to generate sequences of IP ID values that repeat only after the entire sequence has been cycled. So that the LCG may be reseeded after each cycle without causing a quick reuse of any value, the 16 bit space is split into two 15 bit spaces; the space used is toggled after each cycle. This system will defeat idlescan detection, but may not be as effective at masking packet transmit rate or masking if two IP addresses are hosted by the same machine. If one watches how often a system cycles between the two 15-bit addresses spaces, rough estimates on traffic rates can be gathered. If one notices that two IP addresses always switch IP ID spaces

simultaneously, then they are probably running on the same machine.

One common goal of all of these solutions is to make the time before an IP ID is reused as great as possible. This ideal is mentioned in many documents discussing the topic of IP ID abuse. Fyodor mentions in [Fyo], "This is difficult to get right -- be sure the sequence does not repeat and that individual numbers will not be used twice in a short period."

Despite the pervasiveness of Fyodor's belief, there is in fact no reason why quick recycling of IP ID values is a serious problem.

If two fragmented packets with the same IP ID value are put onto the wire at the same time, there are two possibilities that can occur.

The first possibility is that packet #1 will arrive intact at the destination before packet #2. When this occurs, packet #1 will be reassembled successfully, the reassembly queue will be cleared out, packet #2 will arrive, and it too will be reassembled successfully.

The second possibility is that one of the fragments of packet #1 is lost in transit, and/or the fragments of packet #1 and #2 arrive in some jumbled order. If any of these problems occur, the reassembly process will create a reassembled packet that contains portions of both packets. This corrupt packet will then be handed up to either the TCP or UDP layer, where its checksum will fail verification, and the packet will be discarded. The only way a corrupt packet could be reassembled and passed to an application is if two fragments happen to have the same checksum or if the receiving operating system fails to verify the checksum.

What this means is that in the case where two packets to the same destination are sent with identical IP ID values, the loss of one of the fragments of the first packet will effectively result in the loss of the second packet as well.

Therefore, using a PRNG to generate IP ID values may cause a few extra packet drops in certain unlucky situations where packet loss already exists. These extra packet drops can be considered just like any packet loss – a nuisance, but nothing that TCP and UDP can't handle.

On the positive side, using a PRNG to generate IP ID values totally eliminates any possibility of using a machine as an idlescan drone, estimating traffic rate, or determining how many IP addresses belong to a single host.

Ephemeral Port randomization

In order for a blind spoof attack on a TCP connection to be successful, one of the pieces of information that the attacker must guess is the ephemeral source port used by the client end of the connection. As most operating systems sequentially allocate ephemeral port numbers, narrowing the port used by a recently established connection is relatively easy. All the attacker must do is cause the client to connect to the attacker's machine and determine the ephemeral port used. If the client is running services that perform ident checks, this will be easy to trigger. Other methods of inducing a connection may include sending a message that will bounce to a SMTP server running qmail, connecting to a ftp server using passive mode, or forcing the DNS server on the client machine to perform a TCP DNS lookup.

Randomizing the order of ephemeral port allocation is an obvious method of improving the difficulty of a blind attack. Due to the randomization, the attacker will now have to spoof packets from all ports in the ephemeral port range, rather than just the last 5 to 10. In the case of OSes using the classic ephemeral port range (1024 to 5000), this makes the attack 500 times more difficult, assuming an attack range of 10 before randomization. Operating systems that use large ranges of ephemeral ports

(possibly as large as 1024 to 65535) will require an even greater number of packets to be sent.

Paul Watson's paper “Slipping in the Window” led to a quick implementation of port randomization in FreeBSD. This changed seemed safe, as OpenBSD has randomized ephemeral ports since July of 1996 (revision 1.6 of in_pcb.c.) Unfortunately, a few users started reporting problems soon after the change was made to FreeBSD.

The problem reported was that an accelerating webcache that had been upgraded to include port randomization was suddenly seeing failed connections to the backend web server it connected to. One of the failed connections can be seen in Appendix C. Both the webcache and the webserver were running an up to date version of FreeBSD, and no problems were experienced once the sysctl to disable port randomization was toggled off, eliminating the possibility of an unrelated change that broke the system.

This failure case was not seen prior to port randomization because sequential allocation of ephemeral ports leaves a noticeable amount of time before a port is reused. Randomization, due to its nature, will sometimes cause a port to be reused much more quickly - less than a hundreth of a second in the trace shown here.

While the issue shown here is not directly port randomization's fault – something clearly went wrong in the webserver's TCP state machine – it is also true that just an additional second or two before the port in question was reused would probably have avoided the problem.

The number of TCP stack interactions that will see similar problems to the one captured here is unknown, but these results indicate that if one were to magically add simple port randomization to every machine on the planet at once, many breakage situations such as the one here would be seen.

In order to reduce the likelihood of this problem while retaining the security benefits of port randomization, a method to randomize port use but to ensure that ports are not reused too quickly is needed. Unfortunately, using a linear congruential generator to choose ephemeral ports would not be effective – the length for which a connection stays open is not constant, so a port could still be reused quickly if the previous connection is terminated just before the LCG cycles through all other ports and returns to it.

At present, FreeBSD attempts to avoid this quick port recycling problem by falling back to sequential port allocation whenever the machine is making more than 10 outbound connections per second. This solution is more of a hack than anything, and has been slated to be replaced as soon as a better method can be found.

In discussions with Brooks Davis at BSDCan 2005, a workable system of ensuring that ports would not be recycled too quickly was sketched out. The basic concept is to allocate an array with one slot per ephemeral port. At the time that a connection is terminated, the current time and an amount of buffer time (10 seconds) would be added and stored in the slot for that port. This timestamp would make the first time at which the port could be reused. Port allocation would occur randomly at all times, skipping ports which were marked as not yet ready to reuse. One drawback to this solution is that it would not allow hosts to use the same ephemeral port on two different local IPs simultaneously. As a result, a more creative solution may need to be found.

This system has not yet been implemented. Once it has been implemented and passed preliminary testing, the owner of the troubled accelerator proxy will be one of the first users asked to test the change.

Future Work

Preliminary analysis of the TCP ISN

generation systems of other open source operating systems indicates that they may not meet the security and compatibility criteria set forth in this paper. Research into how these operating systems can be improved will take additional time, and unfortunately can not be put into this edition of the paper.

Also, many of the proposed algorithms in this paper have only had proof of concept implementations, and are not ready for inclusion in the FreeBSD source tree yet. After this paper is presented to wider peer review at EuroBSDCon, work on incorporating the changes can proceed.

Finally, the attacks discussed in [Wat04] and [Gont05] have not been addressed in all operating systems equally, and in some cases have not been addressed at all. A test suite similar that can perform all the described attacks should be created and all operating systems should be put to the test, including FreeBSD.

Once additional work is completed, an updated copy of this paper will be posted at http://www.silby.com/eurobsdcon05/

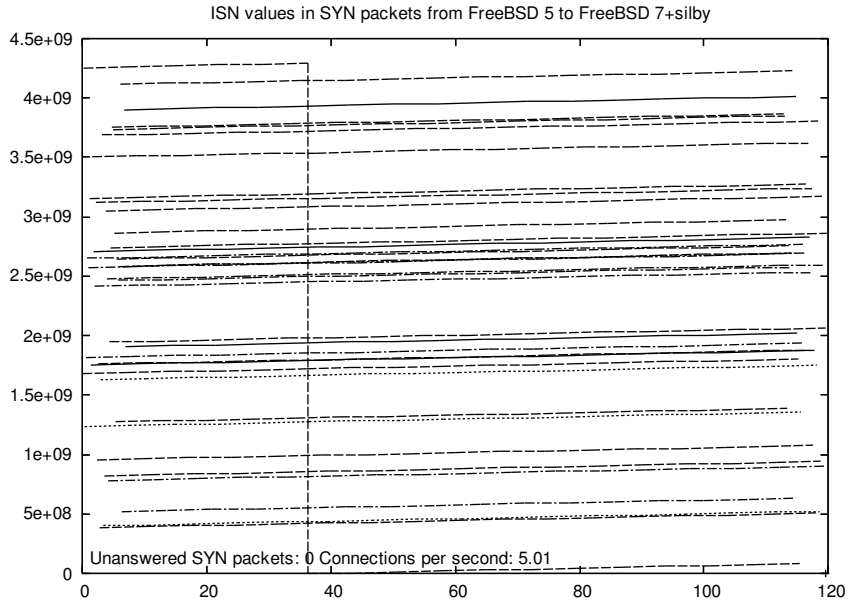
Conclusion

This paper has demonstrated that untested changes to the TCP/IP stack of an operating system can often cause unexpected compatibility issues. However, careful analysis can solve almost any problem, leading to security improvements which do not reduce interoperability.

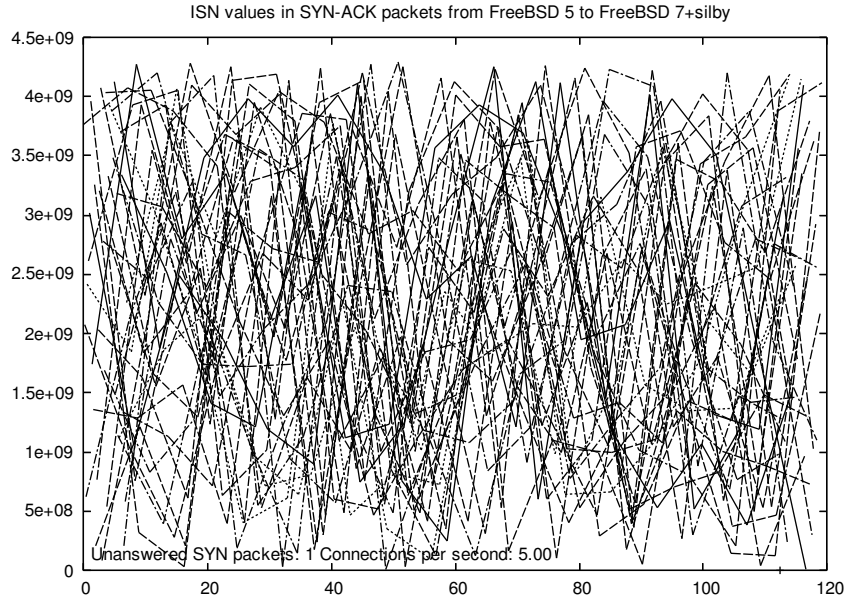
While the new algorithms proposed here have not yet been tested under a wide range of circumstances, it is hoped that the release of this paper will spark a broad discussion on the topic of TCP/IP security, hopefully leading to a new round of standardization that has been sorely lacking in the past few years.

Appendix A: Graph views of FreeBSD 5.4 ISN values

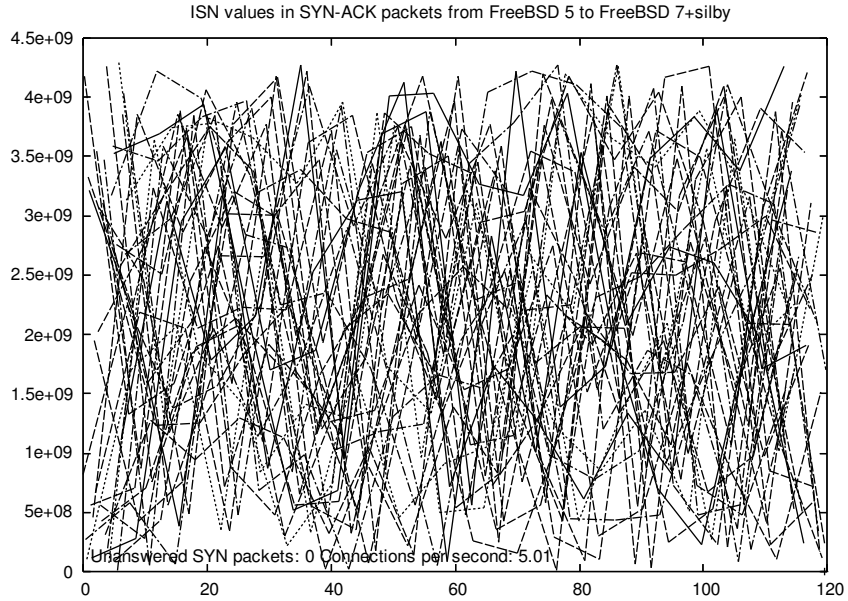
The graphs in Appendices A and B were generated by running a web server on the machine acting as the server in each test, and a http benchmarking tool on the client. To force TIME_WAIT recycling to occur so that the ISN values can be seen per port, the ephemeral port range on the clients was set from 65535 to 65550. The http benchmarking tool was then sent to request a very short HTML page roughly 5 times a second, thereby creating a set of datapoints which was fed into gnuplot. Points are connected together with lines, which is why the graphs of psuedorandom data appear as a graph of haphazard lines rather than a cloud of dots.



ISN values in SYN packets sent from a FreeBSD 5.4 client to a modified FreeBSD 7 server. Notice how each port has a distinct offset from other ports, but how all have the same rate of increase.

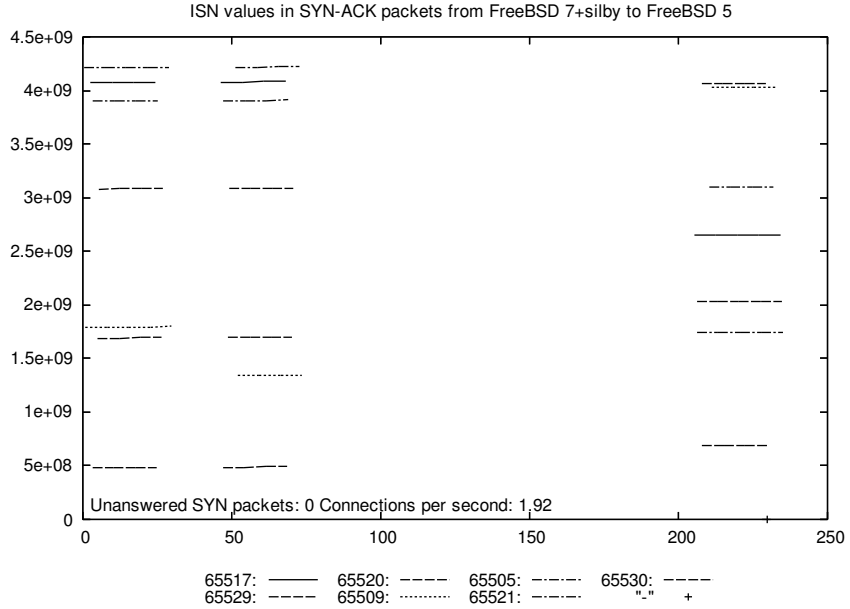


The ISN values in SYN-ACK packets sent by a FreeBSD 5.4 server with net.inet.tcp.syncookies=0. Arc4random is working properly, and prediction of sequence numbers is not possible.

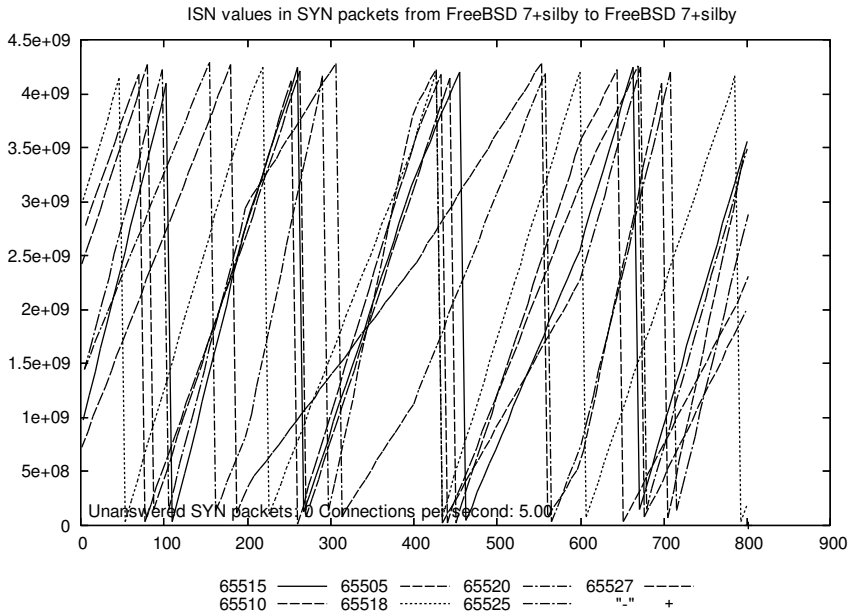


The ISN values in SYN-ACK packets sent by a FreeBSD 5.4 server with net.inet.tcp.syncookies=1. Although syn cookies intentionally create predictability in the short run, it is evident that the long-term effect is similar to randomization.

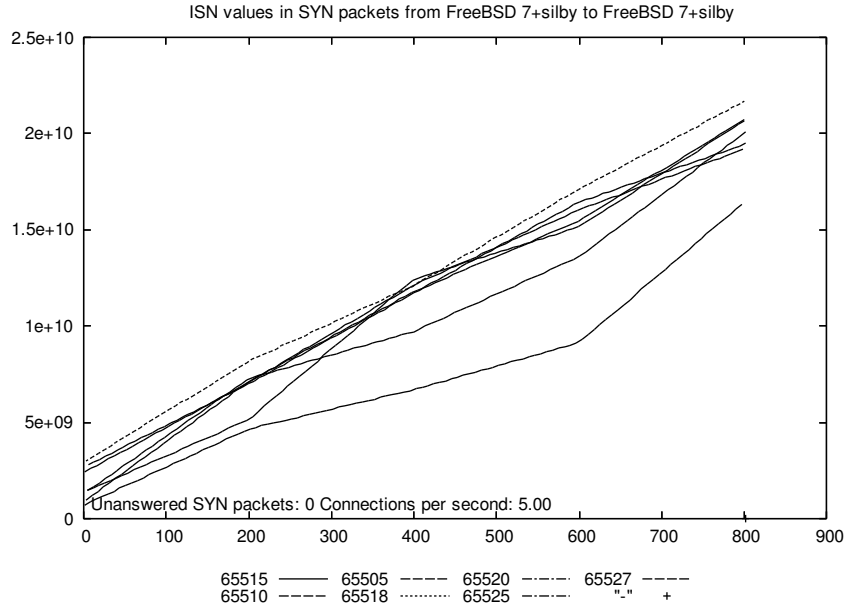
Appendix B: Graphs of proposed changes to FreeBSD's ISN generation schemes



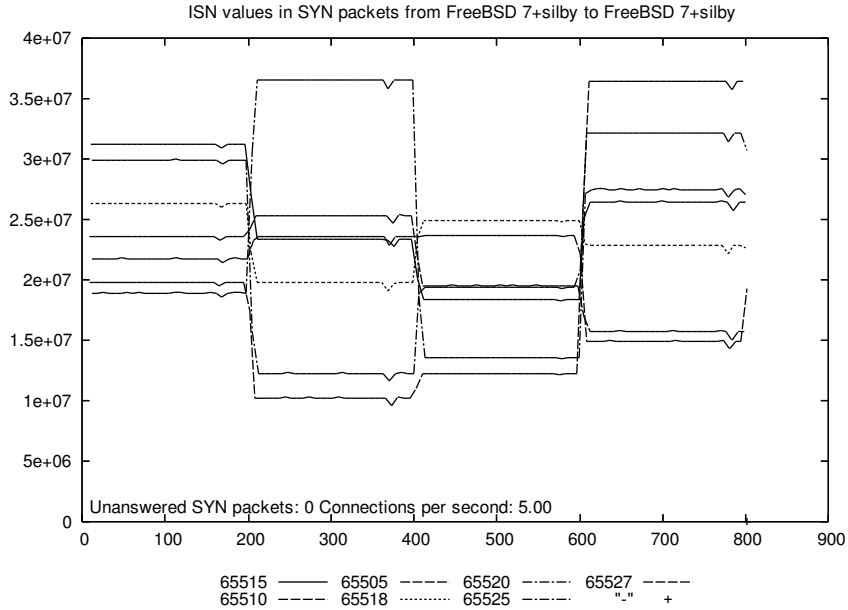
The proposed modification to FreeBSD's SYN-ACK generation is shown. Notice how sequence numbers are the same across the 30 second idle time, but change completely after the 130 second idle time.



A graph of the SYN ISN values from an implementation of the dual hash variant of RFC 1948 using a 200 second reseed interval.



A modification to the dual hash graph so that sequence numbers do not wrap at the 32-bit mark allows for a better view of how the slopes of each port are distinctly different.



A third way of looking at the results of the dual hash algorithm; the first derivative of the ISN values for each port is shown. The slight dips noticeable are due to a glitch in the callout-incremented global time counter.

Appendix C: A failed connection partially due to overly fast ephemeral port recycling

```
17:31:15.372512 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: S 4253937160:4253937160(0) win 8192 <mss
1460,nop,wscale 0,nop,nop,timestamp 152193511 0> (DF)
17:31:15.372642 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: S 1547679919:1547679919(0) ack 4253937161
win 57344 <mss 1460,nop,wscale 0,nop,nop,timestamp 295129972 152193511> (DF)
17:31:15.372656 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: . ack 1547679920 win 8688
<nop,nop,timestamp 152193512 295129972> (DF)
17:31:15.372665 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: P 4253937161:4253937378(217) ack 1547679920
win 8688 <nop,nop,timestamp 152193512 295129972> (DF)
17:31:15.374152 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: . 1547679920:1547681368(1448) ack
4253937378 win 57920 <nop,nop,timestamp 295129972 152193512> (DF)
17:31:15.374243 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: P 1547681368:1547682422(1054) ack
4253937378 win 57920 <nop,nop,timestamp 295129972 152193512> (DF)
17:31:15.374248 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: . ack 1547682422 win 7634
<nop,nop,timestamp 152193515 295129972> (DF)
17:31:15.374253 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: F 1547682422:1547682422(0) ack 4253937378
win 57920 <nop,nop,timestamp 295129972 152193512> (DF)
17:31:15.374257 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: . ack 1547682423 win 8688
<nop,nop,timestamp 152193515 295129972> (DF)
17:31:15.374266 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: F 4253937378:4253937378(0) ack 1547682423
win 8688 <nop,nop,timestamp 152193515 295129972> (DF)
17:31:15.374537 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: . ack 4253937379 win 57920
<nop,nop,timestamp 295129972 152193515> (DF)
17:31:15.389416 XX.XX.XX.XX.1501 > YY.YY.YY.YY.80: S 4253971599:4253971599(0) win 8192 <mss
1460,nop,wscale 0,nop,nop,timestamp 152193545 0> (DF)
17:31:15.389598 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: R 1547682423:1547682423(0) ack 4253937379
win 57920 (DF)
17:31:15.389604 YY.YY.YY.YY.80 > XX.XX.XX.XX.1501: R 0:0(0) ack 4253971600 win 0 (DF)
```

References:

[RFC793] “RFC 793: Transmission Control Protocol”, 1981

[RFC1323] Bellovin, Steven “RFC 1948: Defending Against Sequence Number Attacks”, 1996

[Free03] FreeBSD Security Advisory 3:03 – Brute force attack on SYN cookies

[Fyo] Fyodor, “Idle Scanning and Related IPID games”

[Gont05] Gont, F., "ICMP attacks against TCP", September 2005, Internet Draft

[RFC1323] Jacobson, Braden, & Borman “RFC 1323: TCP Extensions for High Performance”, 1992

[Lem01] Lemon, Jonathan “Resisting SYN flood DoS attacks with a SYN cache”, 2001

[Mor88] Morris, Robert “A Weakness in the 4.2BSD Unix TCP/IP Software, 1985

[New01] Newsham, Timothy “The Problem with Random Increments”, 2001

[San98] Sanfilippo, Salvatore Bugtraq posting: “new tcp scan method”, 1998

[Wat04] Watson, Paul “Slipping in the window: TCP reset attacks”, 2003

[Zal01] Zalewski, Michal “Strange Attractors and TCP/IP Sequence Number Analysis”, 2001

[Zal02] Zalewski, Michal “Strange Attractors and TCP/IP Sequence Number Analysis - One Year Later”, 2002