ï»¿

Independent Submission
Request for Comments: 9187
Category: Informational
ISSN: 2070-1721

J. Touch
Independent Consultant
January 2022

Sequence Number Extension for Windowed Protocols

Abstract

Sliding window protocols use finite sequence numbers to determine segment placement and order. These sequence number spaces wrap around and are reused during the operation of such protocols. This document describes a way to extend the size of these sequence numbers at the endpoints to avoid the impact of that wrap and reuse without transmitting additional information in the packet header. The resulting extended sequence numbers can be used at the endpoints in encryption and authentication algorithms to ensure input bit patterns do not repeat over the lifetime of a connection.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction
2. Background
3. Related Discussion
4. Using SNE in Protocol Design
5. Example Code
6. Validation Suite
7. Security Considerations
8. IANA Considerations
9. Informative References
Acknowledgments
Author's Address

1. Introduction

Protocols use sequence numbers to maintain ordering and, in sliding window systems, to control the amount of outstanding unacknowledged information. These sequence numbers are finite and thus commonly wrap around during long connections, reusing past values.

It can be useful for protocols to keep track of this wrap around in a separate counter, such that the sequence number and counter together form an equivalent number space that need not wrap. This technique was introduced as "Sequence Number Extension" in the TCP Authentication Option (TCP-AO) [RFC5925]. The example provided there was intended to introduce the concept, but the pseudocode provided is not complete.

This document presents the formal requirements for Sequence Number Extension (SNE), a code example, and a check sequence that can be used to validate this and alternate implementations. Sequence numbers are used in a variety of protocols to support loss detection, reordering, flow control, and congestion control. Limitations in the size of a sequence number protocol field can limit the ways in which these capabilities can be supported.

Under certain conditions, it is possible for both endpoints of a protocol to keep track of sequence number rollover and effectively extend the sequence number space without requiring modification of the sequence number field used within protocol messages. These conditions assume that the received sequence numbers never vary by more than half the size of the space of the field used in messages, i.e., they never hop forward or backward by more than half that space. This constraint is typical in sliding window protocols, such as TCP. However, although both ends can track rollover unambiguously, doing so can be surprisingly complex. This document provides examples and test cases to simplify that process.

This document is intended for protocol designers who seek to use larger sequence numbers at the endpoints without needing to extend the sequence number field used in messages, such as for authentication protocols, e.g., TCP-AO [RFC5925]. Use of extended sequence numbers should be part of a protocol specification so that both endpoints can ensure they comply with the requirements needed to enable their use in both locations.

The remainder of this document describes how SNE can be supported and provides the pseudocode to demonstrate how received messages can unambiguously determine the appropriate extension value, as long as the reordering is constrained. Section 2 provides background on the concept. Section 3 discusses currently known uses of SNE. Section 4 discusses how SNE is used in protocol design and how it differs from in-band use of sequence numbers. Section 5 provides a framework for testing SNE implementations, including example code for the SNE function, and Section 6 provides a sequence that can be used by that code for validation. Section 7 concludes with a discussion of security issues.

2. Background

Protocols use sequence numbers to maintain message order. The transmitter typically increments them either once per message or by the length of the message. The receiver uses them to reorder messages and detect gaps due to inferred loss.

Sequence numbers are represented within those messages (e.g., in the headers) as values of a finite, unsigned number space. This space is typically represented in a fixed-length bit string, whose values range from 0..(2^N)-1, inclusive.

The use of finite representations has repercussions on the use of these values at both the transmitter and receiver. Without additional constraints, when the number space "wraps around", it would be impossible for the receiver to distinguish between the uses of the same value.

As a consequence, additional constraints are required. Transmitters are typically required to limit reuse until they can assume that receivers would successfully differentiate the two uses of the same value. The receiver always interprets values it sees based on the assumption that successive values never differ by just under half the number space. A receiver cannot detect an error in that sequence, but it will incorrectly interpret numbers if reordering violates this constraint.

The constraint requires that "forward" values advance the values by less than half the sequence number space, ensuring that receivers never experience a series of values that violate that rule.

We define a sequence space as follows:

* An unsigned integer within the range of 0..(2^N)-1, i.e., for N bits.

* An operation that increments values in that space by K, where K < 2^(N-1), i.e., less than half the range. This operation is used exclusively by the transmitter.

* An operation that compares two values in that space to determine their order, e.g., where X < Y implies that X comes before Y.

We assume that both sides begin with the same initial value, which can be anywhere in the space. That value is either assumed (e.g., 0) before the protocol begins or coordinated before other messages are exchanged (as with TCP Initial Sequence Numbers (ISNs) [RFC0793]). It is assumed that the receiver always receives values that are always within (2^N)-1, but the successive received values never jump forward or backward by more than 2^(N-1)-1, i.e., just under half the range.

No other operations are supported. The transmitter is not permitted to "backup", such that values are always used in "increment" order. The receiver cannot experience loss or gaps larger than 2^(N-1)-1, which is typically enforced either by assumption or by explicit endpoint coordination.

An SNE is a separate number space that can be combined with the sequence number to create a larger number space that need not wrap around during a connection.

On the transmit side, SNE is trivially accomplished by incrementing a local counter once each time the sequence number increment "wraps" around or by keeping a larger local sequence number whose least-significant part is the message sequence number and most-significant part can be considered the SNE. The transmitter typically does not need to maintain an SNE except when used in local computations, such as for Message Authentication Codes (MACs) in TCP-AO [RFC5925].

The goal of this document is to demonstrate that SNE can be accomplished on the receiver side without transmitting additional information in messages. It defines the stateful function compute_sne() as follows:

```
      SNE = compute_sne(seqno)
```

The compute_sne() function accepts the sequence number seen in a received message and computes the corresponding SNE. The function includes persistent local state that tracks the largest currently received SNE and seqno combination. The concatenation of SNE and seqno emulates the equivalent larger sequence number space that can avoid wrap around.

Note that the function defined here is capable of receiving any series of seqno values and computing their correct corresponding SNE, as long as the series never "jumps" more than half the number space "backward" from the largest value seen "forward".

3. Related Discussion

The DNS uses sequence numbers to determine when a Start of Authority

(SOA) serial number is more recent than a previous one, even considering sequence space wrap [RFC1034][RFC1035]. The use of wrapped sequence numbers for sliding windows in network protocols was first described as a sequence number space [IEN74].

A more recent discussion describes this as "serial number arithmetic" and defines a comparison operator it claimed was missing in IEN-74 [RFC1982]. That document defines two operations: addition (presumably shifting the window forward) and comparison (defining the order of two values). Addition is defined in that document as limited to values within the range of 0..windowsize/2-1. Comparison is defined in that document by a set of equations therein, but that document does not provide a way for a receiver to compute the correct equivalent SNE, especially including the potential for sequence number reordering, as is demonstrated in this document.

4. Using SNE in Protocol Design

As noted in the introduction, message sequence numbers enable reordering, loss detection, flow control, and congestion control. They are also used to differentiate otherwise potentially identical messages that might repeat as part of a sequence or stream.

The size of the sequence number field used within transferred messages defines the ability of a protocol to tolerate reordering and gaps, notably limited to half the space of that field. For example, a field of 8 bits can reorder and detect losses of smaller than 2^7, i.e., 127 messages. When used for these purposes -- reordering, loss detection, flow control, and congestion control -- the size of the field defines the limits of those capabilities.

Sequence numbers are also used to differentiate messages; when used this way, they can be problematic if they repeat for otherwise identical messages. Protocols using sequence numbers tolerate that repetition because they are aware of the rollover of these sequence number spaces at both endpoints. In some cases, it can be useful to track this rollover and use the rollover count as an extension to the sequence number, e.g., to differentiate authentication MACs. This SNE is never transmitted in messages; the existing rules of sequence numbers ensure both ends can keep track unambiguously -- both for new messages and reordered messages.

The constraints required to use SNE have already been presented as background in Section 2. The transmitter must never send messages out of sequence beyond half the range of the sequence number field used in messages. A receiver uses this assumption to interpret whether received numbers are part of pre-wrap sequences or post-wrap sequences. Note that a receiver cannot enforce or detect if the transmitter has violated these assumptions on its own; it relies on explicit coordination to ensure this property is maintained, such as the exchange of acknowledgements.

SNEs are intended for use when it is helpful for both ends to unambiguously determine whether the sequence number in a message has wrapped and whether a received message is pre-wrap or post-wrap for each such wrap. This can be used by both endpoints to ensure all messages of arbitrarily long sequences can be differentiated, e.g., ensuring unique MACs.

SNE does not extend the actual sequence space of a protocol or (thus) its tolerance to reordering or gaps. It also cannot improve its dynamic range for flow control or congestion control, although there are other somewhat related methods that can, such as window scaling [RFC7323] (which increases range at the expense of granularity).

SNE is not needed if messages are already unique over the entirety of a transfer sequence, e.g., either because the sequence number field used in its messages never wrap around or because other fields provide that disambiguation, such as timestamps.

5. Example Code

The following C code is provided as a verified example of SNE from 16 to 32 bits. The code includes both the framework used for validation and the compute_sne() function, the latter of which can be used operationally.

A correct test will indicate "OK" for each test. An incorrect test will indicate "ERROR" where applicable.

```
<CODE BEGINS> file "compute_sne.c"
#include <stdio.h>
#include <sys/param.h>

#define distance(x,y)   (((x)<(y))?((y)-(x)):((x)-(y)))

#define SNEDEBUG 1

// This is the core code, stand-alone, to compute SNE from seqno
// >> replace this function with your own code to test alternates
unsigned long compute_sne(unsigned long seqno) {
    // INPUT: 32-bit unsigned sequence number (low bits)
    // OUTPUT: 32-bit unsigned SNE (high bits)

    // variables used in this code example to compute SNE:

    static unsigned long
      RCV_SNE = 0;        // high-watermark SNE
    static int
      RCV_SNE_FLAG = 1;   // set during first half rollover
                          // (prevents re-rollover)
    static unsigned long
      RCV_PREV_SEQ = 0;   // high-watermark SEQ
    unsigned long
      holdSNE;            // temp copy of output

    holdSNE = RCV_SNE;                // use current SNE to start
    if (distance(seqno,RCV_PREV_SEQ) < 0x80000000) {
        // both in same SNE range?
        if ((seqno >= 0x80000000) && (RCV_PREV_SEQ < 0x80000000)) {
            // jumps fwd over N/2?
            RCV_SNE_FLAG = 0;         // reset wrap increment flag
        }
        RCV_PREV_SEQ = MAX(seqno,RCV_PREV_SEQ);
                                      // move prev forward if needed
    } else {
            // both in diff SNE ranges
            if (seqno < 0x80000000) {
                // jumps forward over zero?
                RCV_PREV_SEQ = seqno; // update prev
                if (RCV_SNE_FLAG == 0) {
                    // first jump over zero? (wrap)
                    RCV_SNE_FLAG = 1;
                              // set flag so we increment once
                    RCV_SNE = RCV_SNE + 1;
                              // increment window
                    holdSNE = RCV_SNE;
                              // use updated SNE value
                }
            } else {
                // jump backward over zero
                holdSNE = RCV_SNE - 1;
                              // use pre-rollover SNE value
            }
    }
    #ifdef SNEDEBUG
    fprintf(stderr,"state RCV_SNE_FLAG =        %1d\n",
      RCV_SNE_FLAG);
    fprintf(stderr,"state      RCV_SNE = %08lx\n", RCV_SNE);
    fprintf(stderr,"state RCV_PREV_SEQ = %08lx\n", RCV_PREV_SEQ);
    #endif
    return holdSNE;
```

```
   }

   int main() {
       // variables used as input and output:
       unsigned long SEG_SEQ;        // input - received SEQ
       unsigned long SNE;            // output - SNE corresponding
                                     // to received SEQ

       // variables used to validate the computed SNE:
       unsigned long SEG_HIGH;       // input - xmitter side SNE
                                     // -> SNE should match this value
       unsigned long long BIG_PREV;  // prev 64-bit total seqno
       unsigned long long BIG_THIS = 0;  // current 64-bit total seqno
                                     // -> THIS, PREV should never jump
                                     //    more than half the SEQ space

      char *prompt = "Input hex numbers only (0x is optional)\n\n")
                     "\tHex input\n"
                     "\t(2 hex numbers separated by whitespace,"
                     "each with 8 or fewer digits)";

       fprintf(stderr,"%s\n",prompt);

       while (scanf("%lx %lx",&SEG_HIGH,&SEG_SEQ) == 2) {
           BIG_PREV = BIG_THIS;
           BIG_THIS = (((unsigned long long)SEG_HIGH) << 32)
                      | ((unsigned long long)SEG_SEQ);

           // given SEG_SEQ, compute SNE
           SNE = compute_sne(SEG_SEQ);

           fprintf(stderr,"           SEG_SEQ = %08lx\n", SEG_SEQ);
           fprintf(stderr,"               SNE = %08lx\n", SNE);
           fprintf(stderr,"          SEG_HIGH = %08lx %s\n",SEG_HIGH,
                   (SEG_HIGH == SNE)? " - OK" : " - ERROR !!!!!!!");
           fprintf(stderr,"\t\tthe jump was %16llx %s %s\n",
                   distance(BIG_PREV,BIG_THIS),
                   ((BIG_PREV < BIG_THIS)?"+":"-"),
                   (((distance(BIG_PREV,BIG_THIS)) > 0x7FFFFFFF)
                    ? "ILLEGAL JUMP" : "."));
           fprintf(stderr,"\n");
           fprintf(stderr,"\n");

           fprintf(stderr,"%s\n",prompt);

       }
   }
   <CODE ENDS>
```

6. Validation Suite

The following numbers are used to validate SNE variants and are shown in the order they legitimately could be received. Each line represents a single 64-bit number, represented as two hexadecimal 32-bit numbers with a space between. The numbers are formatted for use in the example code provided in Section 5.

A correctly operating extended sequence number system can receive the least-significant half (the right side) and compute the correct most-significant half (the left side) correctly. It specifically tests both forward and backward jumps in received values that represent legitimate reordering.

```
   00000000 00000000
   00000000 30000000
   00000000 90000000
   00000000 70000000
   00000000 a0000000
   00000001 00000001
   00000000 e0000000
   00000001 00000000
   00000001 7fffffff
   00000001 00000000
   00000001 50000000
   00000001 80000000
   00000001 00000001
   00000001 40000000
   00000001 90000000
   00000001 b0000000
   00000002 0fffffff
   00000002 20000000
   00000002 90000000
   00000002 70000000
   00000002 A0000000
   00000003 00004000
   00000002 D0000000
   00000003 20000000
   00000003 90000000
   00000003 70000000
   00000003 A0000000
   00000004 00004000
   00000003 D0000000
```

7. Security Considerations

Sequence numbers and their extensions can be useful in a variety of security contexts. Because the extension part (most-significant half) is determined by the previously exchanged sequence values (least-significant half), the extension should not be considered as adding entropy for the purposes of message authentication or encryption.

8. IANA Considerations

This document has no IANA actions.

9. Informative References

[IEN74]   Plummmer, W., "Sequence Number Arithmetic", IEN-74, September 1978.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/info/rfc1034>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

[RFC1982] Elz, R. and R. Bush, "Serial Number Arithmetic", RFC 1982, DOI 10.17487/RFC1982, August 1996, <https://www.rfc-editor.org/info/rfc1982>.

[RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <https://www.rfc-editor.org/info/rfc5925>.

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <https://www.rfc-editor.org/info/rfc7323>.

Acknowledgments

The need for this document was first noted by Juhamatti Kuusisaari in April 2020 during discussions of the pseudocode in RFC 5925.

Author’s Address

Joe Touch
Manhattan Beach, CA 90266
United States of America

Phone: +1 (310) 560-0334
Email: touch@strayalpha.com