Internet Draft H. Heikkila
<draft-heikkila-ip-checksum-00.txt>
Intended Status: Informational
Updates: 1071, 1141, 1624 (once approved)
Expires: May 9, 2013 November 5, 2012

Practical Math and Algorithms for TCP/IP checksum
<draft-heikkila-ip-checksum-00.txt>

Abstract

This document reformulates the definition of TCP/IP checksum. The new formulation is equivalent to the traditional one, but it uses much simpler mathematics, avoiding concepts like "one's complement sum". This document attempts to be helpful for both newbies and seasoned engineers when considering checksum problems. Practical calculation and software examples are included.

Table of Contents

1. Introduction

RFC 793 defines the TCP checksum briefly as "the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text" [RFC0793]. RFC 1071 contains helpful details and examples that make the checksum understandable, and mentions the useful method of "end around carry". However, it can be argued that the presentation of RFC 1071 is still more complicated than necessary [RFC1071].

This document argues that the checksum is best described in terms of more elementary mathematics, namely remainders modulo 65,535. It may be easier to avoid the concept of one's complement sum, which is nowadays not widely used for computer arithmetic. This document also attempts to show that the proposed new description is, in general, sufficient for practical purposes. In what follows, the equivalence of the old and new definition is proved, example calculations are included to show how the known mathematical properties of the checksum can be derived from the new definition, and software code examples are included.

2. Old Definition of Checksum

RFC 1071 uses the notation +' for one's complement sum. Here is a definition of it:

(Definition of binary operation +')
For any two integers a and b such that 0 <= a,b <= 65,535,

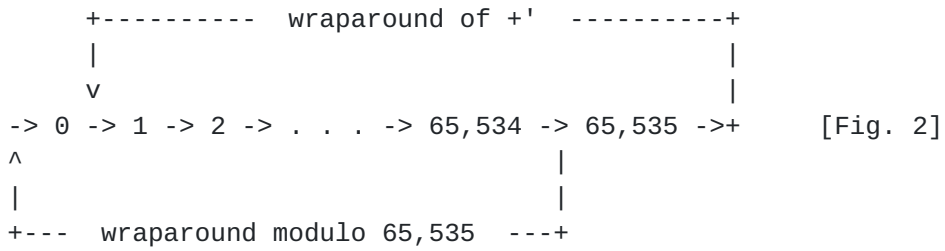
a +' b = a+b, if a+b <= 65,535; [Eq. 1]

= a+b-65,535, otherwise.

Textually, this definition appears to differ from that in RFC 1071; but if the reader understands the +' operation in RFC 1071, he or she can easily verify that the two formulations of +' are equivalent. RFC 1071 calls this operation "1's complement addition". But in the definition of this document, the operands a,b and the result are definitely non-negative integers (actual 1's complement calculations are not involved).

Note that this definition uses elementary school mathematics: the right-hand side of the definition uses just the well-known addition and subtraction operators. Overflow or wraparound properties are not used as such and wraparound of +' is defined explicitly. Also, the decimal constant 65,535 is chosen (instead of the hexadecimal constant ffff) to emphasize the elementary character of the proposed mathematical operations.

Operation +' wraps around to prevent results larger than 65,535, but the wraparound is slightly different from ordinary addition modulo 65,535. The following figure illustrates the wraparound properties of operator +' and compares it to the normal wraparound modulo 65,535 (wrap-to-one versus wrap-to-zero). Of course, both wrap methods differ from the familiar 16-bit register addition modulo 2^16, since 65,535 = 2^16-1.



[Fig. 2]

Following RFC 1071 very closely, we define IP checksum as follows:

(1) Data consists of octets, which are paired to form 16-bit unsigned integers, with most significant octet first (network byte order). Within the data, there is a field for 16-bit checksum, in an even offset location. If the total number of octets is odd, an additional zeroed octet is added to the end of the octet string for checksum calculation and removed after the calculation.

(2) To generate a checksum, the checksum field itself is cleared, the sum B is computed over the octets concerned using operation +', and the value (65,535-B) is placed in the checksum field. (Note that 65,535-B means just the elementary subtraction.)

(3) To check a checksum, the sum B is computed over the octets concerned using operation +'. If the result is 65,535, the check succeeds.

Note some differences to RFC 1071: We have 16-bit *unsigned* integers, while RFC 1071 has just "integers" (probably to make it possible to interpret these 16 bits as a signed integer with one's complement representation). Instead of (65,535-B), RFC 1071 uses bit complementation, but the reader can easily see that the results are the same. Thus, the definition presented above is equivalent to that in RFC 1071.

For completeness, we add a fourth rule, mandated by [RFC0768] and [RFC2460]:

(4) To generate a checksum for UDP header, calculate as above, but if the resulting checksum is 0, place 65,535 instead in the checksum field.

One observation is still in order. If some data consists of nothing but octets where all bits are cleared, we call the data "zeroed". Real protocols never send such data; for example, IPv4 header starts with version number 4 and so IPv4 header checksum is never calculated over zeroed data [RFC0791]. However, the algorithm may still face zeroed data, as errors are possible. The following facts are notable.

- If checksum is generated over zeroed data, the checksum will be 65,535. After that checksum has been placed to the checksum field, the resulting data will be acceptable to the checking algorithm.

- If checksum checking is done over zeroed data (meaning that even the checksum is zeroed), the checking will reject the data. (This is appropriate, because no reasonable protocol would send such data anyway.)

In what follows, we occasionally need to consider three special cases for checksum calculation: UDP data, zeroed data with valid checksum 65,535, and zeroed data with invalid checksum 0. (This is slightly annoying, because without these cases the checksum analysis would be easier.)

3. Mathematical Observations

This section contains basic mathematical analysis of the checksum algorithm. While this section is not really difficult, the reader may anyway choose to skip it. The later part of this document uses only simple and well-known concepts like division with remainder and bit shift.

3.1. Basics

We use the symbol % for remainder division, in accordance with the C programming language. While our divisors are positive, we allow the dividends to be negative. The remainder is always positive; for example, (-2) % 65,535 = 65,533. Computers are known to be unpredictable with division of negative integers; so we avoid such divisions in algorithms.

As expected, we need the concept of "congruence modulo M"; we use symbol =' for the congruence relation. Thus

a =' b (mod M)

means simply that a and b have the same remainder when divided by M. Equivalently, this means that the difference a-b is an integral multiple of M. For checksum, the modulus M is always 65,535, and we omit the (mod M) from the notation, from now on.

One well-known property of congruence is this. If

a =' b and c =' d

then

a+c =' b+d and a-c =' b-d and ac =' bd. [Eq. 3]

This fact makes it possible to calculate with congruences almost as easily as with equalities.

Obviously a+b =' a+b-65,535, so that definition [Eq. 1] yields this:

Fact. a+'b =' a+b.

Corollary. If B is the sum calculated with +' over some 16-bit unsigned integers and S is the sum calculated with ordinary addition over the same data, then B =' S.

According to definition item (3), the sum with +' yields 65,535 when calculated over correctly checksummed data; since 65,535 =' 0, we see:

Corollary. If S is the sum calculated with ordinary addition over some data that has correct checksum, S =' 0; equivalently, S is divisible by 65,535.

[Eq. 4]

Now we see that [Eq. 4] is an almost complete definition of the checksum, without referring to one's complement sum. From it, we can derive both generation and checking algorithms for checksum, with the following open issues: for generating a checksum, [Eq. 4] does not tell whether 0 or 65,535 should be chosen if both are possible; for checking a checksum, [Eq. 4] considers both 0 and 65,535 equal as checksums, while sometimes they are not.

3.2. The Shift Property

Obviously, 65,536 =' 1 and 65,536 = 2^16. From this we see that for any integer a, we have a*2^16 =' a. But for non-negative integers, multiplication by 2^16 means bit shifting to the left by 16 bits.

Using the C notation for bit shift (left shift is <<, right shift is >>), we have the following basic property for any non-negative integer a:

$$a << 16 \;=' a. \qquad \text{[Eq. 5]}$$

As a first application of this property, consider four-octet integers, using the notation of RFC 1071, sec. 2(C).

$$\begin{aligned}[A,B,C,D] + [E,F,G,H] &= [A,B,0,0] + [C,D] + [E,F,0,0] + [G,H] \\ &= ([A,B]<<16) + [C,D] + ([E,F]<<16) + [G,H] \\ &=' [A,B] + [C,D] + [E,F] + [G,H]\end{aligned}$$

This is what RFC 1071 calls "parallel summation": we can accumulate the sum in 32-bit pieces. The resulting block sum is not the same as the real block sum, but still congruent to it. It is fairly easy to see that we can do parallel summation even for eight bytes, if we have 64-bit arithmetic support in hardware.

The next trick is what RFC1071 calls "folding"; long integers can be shortened with bit shifting.

$$\begin{aligned}[A,B,C,D] \;&=\; [A,B,0,0] + [C,D] \;=\; ([A,B]<<16) + [C,D] \\ &=' [A,B] + [C,D]\end{aligned}$$

3.3. Negation and Subtraction

Negation and subtraction are easy to do with unsigned (non-negative) numbers if the modulus 65,535 is first added to the operands sufficiently many times. To manipulate the negative of A, first find some N such that N*65,535 >= A; then (-A) =' N*65,535-A, and the latter value is non-negative but still congruent to the mathematically correct value.

Similarly, to do subtraction A-B, substitute (N*65,535+A-B), where N is chosen so that the result is positive and calculations do not overflow.

3.4. Byte Order Independence

We use the notation [a,b] = a*256+b for a 16-bit integer that consists of two octets a and b, as RFC 1071. If we multiply [a,b] by 256, we can calculate:

$$\begin{aligned}256*[a,b] \;&=\; 256*(a*256+b) = a*256*256 + b*256 \\ &= a*2^16 + b*256 = b*256 + (a << 16) \\ &=' b*256 + a.\end{aligned}$$

From this we see: 256*[a,b] =' [b,a] and 256*[b,a] =' [a,b]. Byte swapping is thus equivalent to multiplying by 256, modulo M. But multiplication is distributive over addition, so that

```
256*([a,b]+[c,d]+ ... + [y,z])
      =  256*[a,b]+256*[c,d]+ ... +256*[y,z]
      =' [b,a] + [d,c] + ... + [z,y]
```

From this we see that if we deliberately swap the bytes of terms of a sum, calculate the sum, and then swap the bytes back, we get a result that is (of course) in general not identical to the correct sum but at least congruent to it modulo 65,535. And in checksum calculation the critical results 0 = [0,0] and 65,535 = [255,255] are immune to swapping. So the congruence method yields rather elegantly the well known property of IP checksum: it can be calculated with inverted byte order.

4. New Definition of Checksum

Here is a modified checksum definition. The mathematical considerations above show that it is equivalent to the traditional one.

We define the *blocksum* over a set of 16-bit unsigned integers to be the arithmetic sum of these integers.

(1) Data consists of octets, which are paired to form 16-bit unsigned integers, with most significant octet first (network byte order). Within the data, there is a field for 16-bit checksum, in an even offset location. If the total number of octets is odd, an additional zeroed octet is added to the end of the octet string for checksum calculation and removed after the calculation.

(2) To generate a checksum, the checksum field itself is cleared, the blocksum B is computed over the octets concerned, then B is divided by 65,535, yielding remainder R. The value C is placed in the checksum field, computed as follows.

(2a) If the blocksum B is zero, let C = 65,535. (This is the theoretical case of zeroed data.)

(2b) Otherwise, if R = 0 and the checksum is for UDP header, let C = 65,535 [RFC0768] [RFC2460].

(2c) Otherwise, if R = 0, let C = 0.

(2d) Otherwise, let C = 65,535 - R.

(3) To check a checksum, the blocksum B is computed over the octets concerned. If the result B is non-zero and divisible by 65,535, the check succeeds.

Note for the UDP case: UDP protocol examines the checksum field before applying algorithm (3); if the checksum is zero, (3) is not applied at all, as there is no checksum. (Instead, the zero-checksum data is either accepted as in [RFC0768] or rejected as in [RFC2460].)

Note also that the definition above is formulated to be exactly identical to that in RFC 1071. For example, if IPv4 header contains checksum 65,535 (hex-ffff), algorithm (3) accepts it (as RFC 1071 would do) although algorithm (2) would not produce such a checksum except in case (2b). Also, in algorithm (3) the condition "B is non-zero" is usually unnecessary, because B=0 implies that all data is zeroed, and reasonable protocols reject such data on other grounds (for example, IPv4 header is rejected if its protocol version is not 4).

5. Examples of Algorithms

All examples use the C programming language. Naturally, identical algorithms can be implemented in other software languages and even in hardware.

5.1. Basic Calculations

When calculating checksum, the basic method is just to add together the 16-bit unsigned integers over the data, accumulating the sum in a 32-bit variable. The following example is taken from RFC 1071 (where it appears in a slightly more complicated form):

```
register unsigned long sum = 0;
unsigned short *addr = . .;
int count = . . /* octet count */

while (count > 1) {
   sum += *addr++;
   count -= 2;
}
```

If checksum is to be generated, the blocksum is calculated as above, after ensuring that the checksum field is cleared. The basic method to calculate checksum is then:

```
checksum = ((N*65535) - sum) % 65535;
```

(This calculates the remainder of the negative of sum, but the

addition of N*65535 with suitably large N guarantees that dividend is positive without changing the result. For example, N can be 65537.) But if the checksum is calculated for UDP header, the algorithm is this:

```
udp_checksum = 65535 - (sum % 65535);
```

To check the checksum of some received data, blocksum is calculated as above, and verification can be done like this:

```
if (sum % 65535 != 0) goto checksum_error;
```

(Pedantically, the check should be "if (sum == 0 || sum % 65535 != 0)", but usually the sum is not zero, and if it is, other parts of the software reject the packet anyway.)

In practice, many alterations of the algorithm are possible and often necessary, as follows.

- If hardware support for efficient remainder division is not available, some alternative algorithm is needed.
- The summing algorithm usually needs optimizations (as emphasized already in RFC 1071).
- The summing can be done in pieces (usually pieces of even offset and, except for the last piece, even size), and pieces can be summed in any order.
- Computer byte order needs to be taken into account (little-endian machines tend to reverse the byte order when doing arithmetic operations).
- The last odd octet, if any, needs special attention.
- We need algorithms for incremental update of checksum.

The following sections consider these points. We start with some preliminary remarks.

The checksum algorithm is such that usually only the remainder of the sum is significant (when divided by 65,535). There are several operations that change the mathematical value but preserve the remainder. Such operations can be used during the course of calculation. (The mathematical section above proves the validity of these operations.)

- Any multiple of 65,535 can be added or subtracted, as long as

there is no overflow or underflow. In particular, 0xffffffff (the largest unsigned 32-bit integer) is a multiple of 65,535.

- Remainder division can be done: sum = sum % 65535.
- A non-negative value can be bit-shifted left by 16 bits: sum = sum << 16.
- A non-negative value can be folded by shifting the most-significant bits right by 16 and adding to the rest of the number, for example:

```
0x123456789ab =  0x12345670000 + 0x89ab
              =  (0x1234567<<16) + 0x89ab
              -> change to 0x1234567 + 0x89ab
```

In C language, we can define folding essentially as in RFC 1071:

```
#define FOLD(sum)  (((sum) & 0xffff) + ((sum) >> 16))
```

- In assembly programming, the carry bit that overflows after unsigned 16-, 32-, or 64-bit addition can be shifted to the right and added back to the sum ("end around carry" [RFC1071]).

5.2. Integer Division versus Folding

The algorithms, as defined, use the % operator, the remainder division. Of course it is not available in all hardware implementations; on the other hand, even relatively simple microprocessors like the 16-bit Intel 8086 can divide a 32-bit integer by 65,535 and find the remainder with one single machine instruction (DIV). Regarding efficiency, consider the example of Intel386SX (developed in 1980's); in 32-bit mode it can execute DIV instruction, nominally, in 38 clock cycles [386SX]. This should be contrasted with the masking, shifting and addition operations that would be the division-free alternative (folding); they might together take some ten clock cycles.

Also, divisions would not be frequently repeated anyway. Processing a typical incoming TCP segment might require exactly two division operations, one to the check the IPv4 header, the other to check the TCP data. So, compared with other necessary processing, division is likely to be a feasible alternative in some applications.

But there is an alternative to division, folding (defined above). This algorithm reduces any positive integer to 16 bits without changing the remainder (again, it is found in RFC 1071):

```
while (sum>>16)
   sum = FOLD(sum);
```

But usually sum has no more than 32 bits, in which case only two foldings are needed:

```
sum = FOLD(FOLD(sum));
```

Note that folding never produces zero result (unless its argument is zero), so folding has wrap-to-one behaviour; see [Fig. 2]. This changes the algorithms slighly, as folding tends to produce 65,535 when true remainder is zero. So here are the algorithms for checksum generation (UDP considered separately) and checking, without division:

```
checksum = 65535 - FOLD(FOLD(sum));

udp_checksum = FOLD(FOLD(0xffffffff - sum));

if (FOLD(FOLD(sum) != 65535) goto checksum_error;
```

In many systems, operations (65535-x) and (0xffffffff-x) can be easily implemented with bit complement. Such optimizations are left to the reader as exercise.

5.3. Choice of Data Types

We usually prefer unsigned integers, because division of signed integers is unpredictable in computers. (It is worth noting, however, that signed division can be faster than unsigned division in some implementations.)

Usually 32-bit integers are suitable for blocksum calculations. For example, the maximum amount of data in a TCP/IPv4 segment is less than 65,535 octets (32,768 octet pairs), which sets an upper limit of block sum to 32,768*65,535 = 2,147,450,880; this means that blocksums fit in 32-bit integers, or even in signed 32-bit integers.

However, 16-bit values can be calculated in parallel as 32-bit values, and then it may be advantageous to use 64-bit values for blocksums. See below for parallel calculation.

5.4. Efficient Calculation of Sum

As RFC 1071 correctly points out, it is usually appropriate to optimize the checksum routine; and obviously the reading and summing loop is *the* most important point to optimize. Note that while the examples shown here are in the C programming language, the most

optimal solution is usually achieved with assembly language.

Data sum should be calculated in the natural byte order, as in example above: "sum += *addr". This means that little-endian machines swap bytes, e.g., data that the network signifies as 0x1234 is summed as 0x3412. As known, and shown in the mathematical section above, this leads to correct results, if the calculated checksum is written to its field in the same natural manner: "*addr = checksum". Also, the important values 0 and 65,535 are immune to byte order. However, see section 5.5 below.

It may be advantageous to read data in bigger pieces, 32 bits (or even 64 bits) as follows. However, addition must not overflow, which is why folding is used in this example:

```
register unsigned long sum = 0;
unsigned long *addr = . .; /* long assumed to have 32 bits */
int count = . . /* octet count */

while (count > 3) {
   sum += FOLD(*addr++);
   count -= 4;
}
```

But if the sum counter has 64 bits, folding as above is not necessary.

As a curiosity, consider the following case, which assumes that sum is a 17-bit (sic!) integer and addr is a pointer to 16-bit integer:

```
while ( . . . ) {
   sum += *addr++;
   sum = FOLD(sum);
   count -= 2;
}
```

The case of a "17-bit" integer may sound theoretical, but actually this is precisely what 16-bit processors often do: the sum is accumulated to a 16-bit register, which, together with the "carry" bit, is effectively a 17-bit counter. The FOLD operation corresponds to what RFC 1071 calls "end around carry".

Often blocksum is calculated in pieces, and sometimes the start of some piece is not at an even offset from the beginning of data. Such calculation swaps the octets in the sum; but this can be corrected so that the blocksum is swapped again before incorporating to the final sum (as noted in RFC 1071).

5.5. Byte Order Issues

As explained above, byte order should usually be ignored in software algorithms, permitting the processor to move data from memory to calculation as efficiently as possible. However, this means that checksum data is in network byte order while ordinary data is in host byte order. Then the standard operations "htons()" (host-to-network swap, short variable) and "ntohs()" (network-to-host) should be used as appropriate [POSIX]. Consider the case of IPv4 header, where the TTL field is decremented by 1 at every router, and this means that one element in the blocksum is decremented by 0x0100. In order not to recalculate the whole header checksum, the checksum is simply incremented by 0x0100, but this value must be modified as follows:

```
new_chksum = ((unsigned long)old_chksum + htons(0x0100)) % 65535;
```

Here, "htons()" is necessary because computers keep arithmetic constants in host byte order. (The pedantic typecast "(unsigned long)" is a C technicality to force the machine to use at least 32 bits during calculation to prevent overflow, even if old_chksum is a 16-bit integer. Here, the typecast is shown also with a documentation purpose, to emphasize that more than 16 bits are indeed necessary during the calculation.)

Note: RFC 1141 and RFC 1624 consider a similar problem, but they emphasize hardware optimization (use bit operations instead of division), and do not mention byte order [RFC1141] [RFC1624].

5.6. Incremental Update of Checksum

Incremental update of checksum is an old idea; it is addressed in [RFC1141] and [RFC1624], and also section 5.5 above uses one case of it as an example. This section contains a new example, ICMPv6 Echo Reply message, to illustrate the method of incremental update.

RFC 4443 defines the important Echo Request message, to which a host replies with Echo Reply (ping6). Suppose a host has received a Request message; naturally, it has to verify the checksum of that message before accepting it. Before sending the Reply message, the host has to calculate also the checksum over the reply. But since the Request and Reply are almost identical, some CPU time can be saved so that the Reply checksum is calculated from the Request checksum. Inspection of RFC 4443 shows that the checksums of Request and Reply differ only with respect to IPv6 addresses (in pseudo-header) and ICMPv6 type (and the checksum itself). So the methods of incremental update are well applicable to the Echo Reply case. We limit our consideration to the IPv6 address case; the update of ICMPv6 type from 128 to 129 is an easy exercise to the reader.

In some cases, the IPv6 addresses of Request and Reply are the same, just so that the source and destination are swapped; and the order of addresses does not, of course, change the checksum. But if the destination address of the Request is a multicast or anycast address, the source address of the Reply must be a unicast address; see [RFC4443], Sec. 4.2.

In what follows, we imitate the notation of RFC 1624, using ordinary addition and subtraction instead of +' and bit complement. We occasionally use the congruence operation =' (modulo 65,535).

```
HC  - old checksum in ICMPv6 header
C   - blocksum of old data
HC' - new checksum in ICMPv6 header (ICMPv6 type still
      unchanged)
C'  - blocksum of new data (ICMPv6 type still unchanged)
m   - blocksum over the 16-octet old address (multicast
      or anycast)
m'  - blocksum over the 16-octet new address (unicast)
```

Then, if the blocksum of the unchanged part of the data is A, we have:

```
A + m  +  HC  = C
A + m' +  HC' = C'
```

Subtracting the two equations we get this:

```
HC' - HC + m' - m = C' - C
```

Although C and C' need not be identical, they are the same modulo 65,535 (in fact, C =' C' =' 0, if the checksum is correctly calculated), so that C' - C =' 0. Then, doing elementary manipulations, we get these facts, analogously to RFC 1642:

```
HC' =' HC + m - m'
HC' =' -(m' - m - HC)
```

Both of these formulas can be used. The former formula is suitable if we use division, and the latter one if we use folding. (RFC 1642 does not consider division and so prefers the latter formula.) We obtain the following C code (two alternatives):

```
new_chk = (9*65535 + old_chk
           + oldaddr_sum - newaddr_sum) % 65535;

new_chk = 65535 - FOLD(FOLD(9*65535 + newaddr_sum
                            - oldaddr_sum - old_chk));
```

The constant 9*65535 is chosen small enough so that 32-bit addition does not overflow but large enough so that subtraction cannot make the result negative. Note that a sum over a 16-bit IPv6 address cannot exceed 8*65,535 and a checksum cannot exceed 65,535, hence the constant 9.

A final note: if we were to update UDP checksum incrementally, we would need a checksum in the range 1...65,535, and so the following C code would be appropriate (two alternatives, where old_ds and new_ds are sum over old data and sum over new data, respectively):

```
n_udp_chk = 65535 -
            (N*65535 + new_ds - old_ds - o_udp_chk) % 65535;

n_udp_chk = FOLD(FOLD(N*65535 + o_udp_chk + old_ds - new_ds));
```

Also in this case, N must be chosen according to circumstances so that addition and subtraction cannot lead to overflow or underflow in 32-bit arithmetic.

Note that byte order is not an issue in this example, provided that all calculations are done with the same byte order (the octets in IPv6 addresses are calculated with the same algorithm as all other octets in the data).

5.7. The Last Octet

If checksum is calculated over an odd number of octets, the last octet is alone and needs a zeroed octet to form a 16-bit unsigned integer. This is easy, if byte order is taken into account. Here are two ways to add the last octet to a sum:

```
{
   unsigned short tmp = 0;
   ((unsigned char*)&tmp)[0] = ((unsigned char*)last_addr)[0];
   sum += tmp;
}

sum += htons(((unsigned char*)last_addr)[0] << 8);
```

6. Security Considerations

There are no security considerations relevant to this document.

7. IANA Considerations

No actions are required from IANA as result of the publication of this document.

8. References

8.1. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1071] Braden, R., Borman, D., and C. Partridge, "Computing the Internet checksum", RFC 1071, September 1988.

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[RFC4443] Conta, A., Deering, S., and M. Gupta, Ed., "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", RFC 4443, March 2006.

8.2. Informative References

[POSIX] The Open Group Base Specifications Issue 7. IEEE Std 1003.1-2008.

[RFC1141] Mallory, T. and A. Kullberg, "Incremental updating of the Internet checksum", RFC 1141, January 1990.

[RFC1624] Rijsinghani, A., Ed., "Computation of the Internet Checksum via Incremental Update", RFC 1624, May 1994.

[386SX] Intel386(TM) SX Microprocessor Programmers's Reference Manual. Inter Order No. 240331-002, ISBN 1-55512-154-3, 1991.

Authors' Addresses

Heikki Heikkila
Tellabs Oy
Sinimaentie 6 C
02630 Espoo
Finland
EMail: heikki.heikkila@tellabs.com