Network Working Group Xiangyang Zhang
Internet-Draft Tina Tsou
Intended status: Informational Futurewei Technologies, Inc
Expires: December 8, 2011 June 10, 2011

IPsec anti-replay algorithm without bit-shifting
draft-zhang-ipsecme-anti-replay-01

Status of this Memo

Copyright Notice

Abstract

This document presents a new method to do anti-replay check and update, which becomes one alternative to the anti-replay algorithm in RFC 4302 and RFC 4303. The new method will deem the bit-shifting unnecessary. It will reduce the number of times to slide the window. In addition, it makes bit-check and bit-update easier as it does not depend on the low index of the sliding window. It is especially beneficial when the window size is much bigger than 64 bits, for example, 1024 bits.

IPsec employs one anti-replay sliding window protocol to secure against an adversary that can insert the messages inside the network tunnel. This method still inherits the sliding window protocol, but use one or more redundant bytes to ease the update of sliding window. The bit-shifting is deemed unnecessary with updating the high and low index of the window, which is especially efficient in case of the big window size. Thus the method reduces the number of times to update the window.

In addition, the bit location is fixed for one sequence number, thus makes the bit check and update easier and faster.

Table of Contents

1. Introduction

IPsec standard defines the anti-replay sliding window protocol, where the receiver must maintain an anti-replay window of size W. This window will limit how far out of order a packet can be, relative to the packet with the highest sequence number that has been authenticated so far. The window can be represented by a range [WB, WT], where WB=WT-W+1. The whole anti-replay window can be thought of as a string of bits. The value of each bit indicates whether or not a packet with that sequence number has been received and authenticated, so that replay packet can be detected and rejected. If the packet is received, the receiver will get the sequence number S in the packet. If S is inside window (S<=WT and S>=WB), then check the corresponding bit (location is S-WB) in the window to see if this S has already been seen. If S<WB, the packet will be dropped. If S>WT and is validated, the window is advanced by (S-WT) bits. The new window will become [WB+S-WT, S]. The new bits in this new window are set to indicate that no packets with those sequence numbers have been received yet. The typical implementation (for example, RFC-2401 algorithm) is done by shifting (S-WT) bits.
In normal cases, the packets arrive in order, which results in constant update and bit shifting operation.

The minimum window size can be 32 or 64. But no requirement is established for minimum or recommended window sizes beyond 64-packet. The actual window size is required to be based on reasonable expectations for packet re-ordering. For high-end multi-core network processor with multiple crypto cores, the window size must be much bigger than 64bits or 128bits, due to the varied IPsec processing latency caused by different cores. In such a case, the window sliding is tremendous costly even with hardware acceleration to do the bit shifting. Here we recommend one method to make the bit-shifting unnecessary.

2 Description of new anti-replay algorithm

Here we present an easy way to only update the window index and also reduce the times of updating the window. The basic idea is illustrated in the following figures. Suppose that we configure the window size W, which consists of M-1 blocks, where M is power of two (2). Each block contains N bits, where N is also power of two (2). It can be a byte (8 bit) or word (32bit), or multiple words. We hide one block from the window. So the supported window size if (M-1)*N. All these M blocks are circulated and become a ring of blocks, each with N bits. In this way, the configured window (M-1 blocks) is always a subset window of the actual window when window slides.

Initially the actual window is defined by low and high end index [WB, WT], as illustrated in Figure 1.

```
+--------+--------+--------+--------+
|xxxxxxcc|cccccccc|cccccccc|ccccc100|
+--------+--------+--------+--------+
       ^                          ^
       |                          |
       WB                         WT
```

Figure 1: the sliding window [WB, WT],
in which WT is last validated sequence number
and the supported window size W is WT-WB+1.
(x=don’t care bit, c=check bit)

If we receive a packet with the sequence number (S) greater than WT, we slide the window. But we only change the window index by adding the difference (S-WT) to both WT (WB is automatically changed as window size is fixed). So S becomes the largest sequence number of the received packets. Figure 2 shows the case that the packet with sequence number S=WT+1 is received.

```
+--------+--------+--------+--------+
|xxxxxxcc|cccccccc|cccccccc|ccccc110|
+--------+--------+--------+--------+
        ^                          ^
        |                          |
        WB                         WT
```

Figure 2: the sliding window [WB, WT] after S=WT+1

If S is in the different block from where WT is, we have to initialize all bit values in the blocks to 0 without bit shifting. If S passes several blocks, we have to initialize several blocks instead of only one block. Figure 3 shows that the sequence number already pass the block boundary. Immediately after update, all the check bits should be 0 in the block where WT resides.

```
+--------+--------+--------+--------+
|ccc10000|xxxxcccc|cccccccc|cccccccc|
+--------+--------+--------+--------+
    ^         ^
    |         |
    WT        WB
```

Figure 3: the sliding window [WB, WT] after S pass the boundary

After update, the new window still covers the configured window. This means the configured sub-window also slides, conforming to the sliding window protocol. The actual effect is somewhat like shifting the block. In this way, the bit-shifting is deemed unnecessary.

It is also easier and much faster to check the window with the sequence number because the sequence number check does not depend on the lowest index WB. Instead, it only depends on the sequence number of the received packet. If we receive a sequence number S, the bit location is the lowest several bits of the sequence number, which only depends on the block size (N). The block index is several bits before the location bits, which only depends on the window size (M).

We do not specify how many redundancy bits needed except that it should be power of two (2) for computation efficiency. If microprocessor is 32 bit, 32 might be a better choice while 64 might be better for 64 bit microprocessor. For microprocessor with cache support, one cache line is also a good choice. It also depends on how big the sliding window size is. If we have N redundancy bits (for example, 32 bit in the above description), we only need 1/N times update of blocks, comparing to the bit-shifting algorithm in RFC 4302.

The cost of this method is extra byte or bytes used as redundant window. The cost will be minimal if the window size is big enough. Actually the extra redundant bits are not completely wasted. We could reuse the unused bits in the block where index WB resides, i.e. the supported window size could be (M-1)*N, plus the unused bits in the last block.

3 Example of new anti-replay algorithm

Here is the example code to implement the algorithm of anti-replay check and update, which is described in the previous sections.

<CODE BEGINS>

```
/**
 * Copyright (c) 2011 IETF Trust, Xiangyang Zhang and Tina Tsou.
 * All rights reserved.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

/**
 * In this algorithm, the hidden window size must be a power of two,
 * for example, 1024 bits.  The redundant bits must also be a power of
 * two, for example 32 bits.  Thus, the supported anti-replay window
 * size is the hidden window size minus the redundant bits.  It is 992
 * in this example. The size of integer depends on microprocessor
 * architecture.  In this example, we assume that the software runs on
 * 32 bit microprocessor.  So the size of integer is 32.  In order to
 * convert the bitmap into an array of integer, the total number of
 * integers is the hidden window size divided by size of integer.
 *
 * struct ipsec_sa contains the window and window related parameters,
 * such as the window size, the last acknowledged sequence number.
 *
 * all the value of macro can be changed, but must follow the rule
 * defined in the algorithm.
 */
```

```
#define SIZE_OF_INTEGER       32 /** 32 bit microprocessor */
#define BITMAP_LEN            (1024/ SIZE_OF_INTEGER)
                                /** in terms of 32 bit integer */
#define BITMAP_INDEX_MASK     (IPSEC_BITMAP_LEN-1)
#define REDUNDANT_BIT_SHIFTS  5
#define REDUNDANT_BITS        (1<<REDUNDANT_BIT_SHIFTS)
#define BITMAP_LOC_MASK       (IPSEC_REDUNDANT_BITS-1)

int
ipsec_check_replay_window (struct ipsec_sa *ipsa,
                           uint32_t sequence_number)
{
    int bit_location;
    int index;

    /**
     * replay shut off
     */
    if (ipsa->replaywin_size == 0) {
        return 1;
    }

    /**
     * first == 0 or wrapped
     */
    if (sequence_number == 0) {
        return 0;
    }

    /**
     * first check if the sequence number is in the range
     */
    if (sequence_number>ipsa->replaywin_lastseq) {
        return 1;  /** larger is always good */
    }

    /**
     * The packet is too old and out of the window
     */
    if (ipsa->replaywin_size <
        (ipsa->replaywin_lastseq-sequence_number)) {
        return 0;
    }
```

```
    /**
     * The sequence is inside the sliding window
     * now check the bit in the bitmap
     * bit location only depends on sequence number
     */
    bit_location = sequence_number&BITMAP_LOC_MASK;
    index = (sequence_number>>REDUNDANT_BIT_SHIFTS)&BITMAP_INDEX_MASK;

    /*
     * this packet already seen
     */
    if (ipsa->replaywin_bitmap[index]&(1<<bit_location)) {
        return 0;
    }

    return 1;
}


int
ipsec_update_replay_window (struct ipsec_sa *ipsa,
                            uint32_t sequence_number)
{
    int bit_location;
    int index, index_cur, id;
    int diff;

    if (ipsa->replaywin_size == 0) {  /** replay shut off */
        return 1;
    }

    if (sequence_number == 0) {
        return 0;     /** first == 0 or wrapped */
    }

    /**
     * the packet is too old, no need to update
     */
    if (ipsa->replaywin_size <
        (ipsa->replaywin_lastseq-sequence_number)) {
        return 0;
    }
```

```
    /**
     * now update the bit
     */
    index = (sequence_number>>REDUNDANT_BIT_SHIFTS);

    /**
     * first check if the sequence number is in the range
     */
    if (sequence_number>ipsa->replaywin_lastseq) {
        index_cur = ipsa->replaywin_lastseq>>REDUNDANT_BIT_SHIFTS;
        diff = index - index_cur;
        if (diff > BITMAP_LEN) {  /* something unusual in this case */
            diff = BITMAP_LEN;
        }

        for (id = 0; id < diff; ++id) {
            ipsa->replaywin_bitmap[(id+index_cur+1)&BITMAP_INDEX_MASK]
                = 0;
        }

        ipsa->replaywin_lastseq = sequence_number;
    }

    index &= BITMAP_INDEX_MASK;
    bit_location = sequence_number&BITMAP_LOC_MASK;

    /* this packet already seen */
    if (ipsa->replaywin_bitmap[index]&(1<<bit_location)) {
        return 0;
    }

    ipsa->replaywin_bitmap[index] &= (1<<bit_location);

    return 1;
}

<CODE ENDS>
```

4. Acknowledgements

The idea in this document came from the software design on one high-performance multi-core network processor. The new network processor core integrates a dozen of crypto core in distributed way, which makes hardware anti-replay service impossible.

5. Security considerations

This document does not intend to change the IPsec standard, but provide one better option to do anti-replay in a faster and easier way, especially when the sliding window size is bigger.

Some of the anti-replay algorithm specified in this document can be found in those RFCs in reference section.

6. Normative References

[RFC2401] "Security Architecture for the Internet Protocol", RFC 2401.
[RFC4302] "IP Authentication Header", RFC 4302.
[RFC4303] "IP Encapsulating Security Payload (ESP)", RFC 4303.

Author's address

Xiangyang Zhang
Futurewei Technologies
2330 Central Expressway
Santa Clara, California 95051
USA

Phone: +1-408-330-4545
Email: xiangyang.zhang@huawei.com

Tina TSOU (Ting ZOU)
Futurewei Technologies
2330 Central Expressway
Santa Clara, California 95051
USA

Phone: +1-408-859-4996
Email: tena@huawei.com