## Anti-Packet replay Algorithm

This is a diagram I made to explain how sliding window algorithm is used to prevent packet replay attack in typical stateless HTTP communication between client and web host.

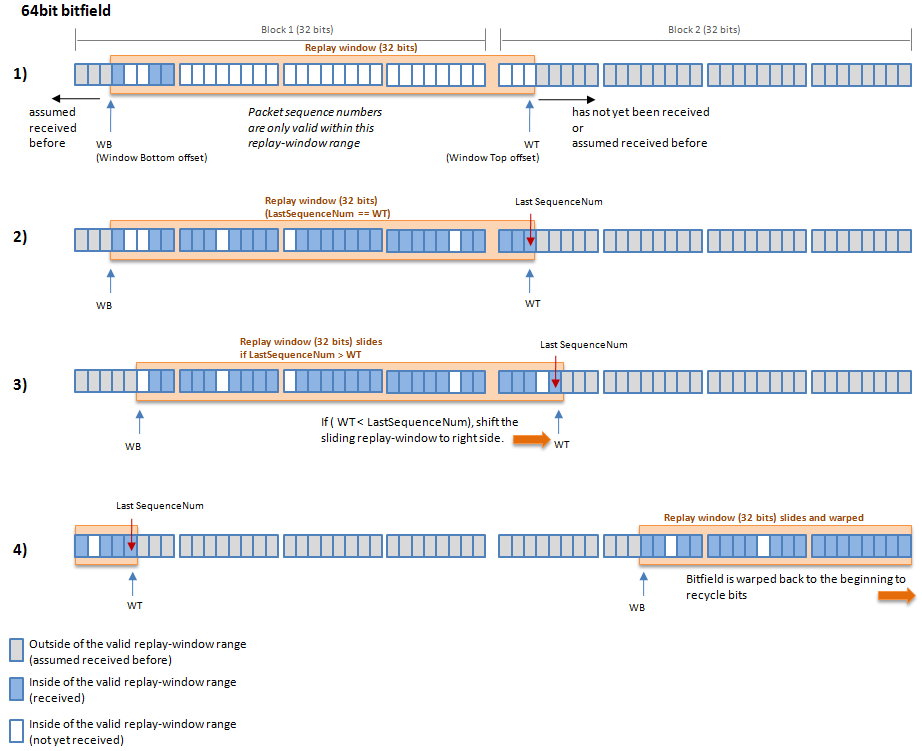
I use two 32-bits bit fields to store session sequence index and slide it through in circular fashion.

The referenced algorithm is ‘An Anti-Replay Window Protocol with Controlled Shift’. It’s not perfect defence against packet replay attack, but it was a reasonably efficient solution.

For detailed algorithm description, reference the separate tech document AntiPacketReplayAlgorithm.pdf

And there are more descriptions in step by step in code sample below.

*Sam Lee*



public class AntiPacketReplay

{

/\* the configured window = (M-1 blocks) \* (N bits)

\* The window can be represented by a range [WB, WT], where WB=WT-W+1

\*

All these M blocks are circulated and become a ring of blocks, each with N bits.

The configured window (M-1 blocks) is always a subset window of the whole window when the supported window slides.

+--------+--------+--------+--------+

|xxxxxccc|cccccccc|cccccccc|ccccc100|

+--------+--------+--------+--------+

^ ^

| |

WB WT

Figure 1: the sliding window [WB, WT],

in which WT is last sequence number

and 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.

\*

\*

+--------+--------+--------+--------+

|xxxxxxcc|cccccccc|cccccccc|ccccc110|

+--------+--------+--------+--------+

^ ^

| |

WB WT

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

+--------+--------+--------+--------+

|ccc10000|xxxxcccc|cccccccc|cccccccc|

+--------+--------+--------+--------+

^ ^

| |

WT WB

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

\*

\*/

const int NUM\_BLOCKS = 2;

const int NUM\_BITS\_PER\_BLOCK = 32; // in terms of 32 bit integer

const int WINDOW\_SIZE = (NUM\_BLOCKS - 1) \* NUM\_BITS\_PER\_BLOCK;

const int WINDOW\_INDEX\_MASK = (WINDOW\_SIZE-1);

const int REDUNDANT\_BIT\_SHIFTS = 5;

const int REDUNDANT\_BITS = (1<<REDUNDANT\_BIT\_SHIFTS);

const int BITMAP\_LOC\_MASK = (REDUNDANT\_BITS - 1);

private int windowTailSeqNum = 0;

private int[] windowBitmaps = new int[NUM\_BLOCKS];

public AntiPacketReplay()

{

}

public AntiPacketReplay(int **lastSequenceNum**, int[] **bits**)

{

windowTailSeqNum = **lastSequenceNum**;

windowBitmaps = **bits**;

}

public int[] ReplayWindowBits

{

get { return this.windowBitmaps; }

set { this.windowBitmaps = value; }

}

public int ReplayWindowLastSeqNum

{

get { return this.windowTailSeqNum; }

set { this.windowTailSeqNum = value; }

}

public bool IsWithinReplayWindow (int **sequenceNum**)

{

/\*

Case i. s is smaller than all sequence numbers in the window:

q cannot determine whether it has received this message before(discards)

Case ii. s is one of the sequence numbers in the window

q can determine whether it has received this message before(discards)

or it has not received this message before(deliver)

Case iii. s is within w positions to the right of the window

q determines that it has not received this message before(deliver)

q slides the window such that s becomes the new right edge of the window

Case iv. s is more than w positions to the right of the window:

q determines that it has not received this message before

\*

\*/

int **bit\_location**;

int **index**;

// first == 0 or wrapped

if (0 == **sequenceNum**) {

return false;

}

// first check if the sequence number is in the range

if (**sequenceNum** > windowTailSeqNum) {

return true; // larger is always good

}

// The packet is too old and out of the window

if (WINDOW\_SIZE < (windowTailSeqNum-**sequenceNum**)) {

return false;

}

// The sequence is inside the sliding window

// now check the bit

**bit\_location** = **sequenceNum** & BITMAP\_LOC\_MASK;

**index** = ((**sequenceNum** >> REDUNDANT\_BIT\_SHIFTS) & WINDOW\_INDEX\_MASK) % NUM\_BLOCKS;

// this packet already seen

if (0 != (windowBitmaps[**index**]&(1<<**bit\_location**)) )

{

return false;

}

return true;

}

public bool UpdateReplayWindow(int **sequenceNum**)

{

int **bit\_location** = 0;

int **index\_next**, **index\_cur** = 0;

int **index\_diff**, **index** = 0;

if (**sequenceNum** == 0) {

return false; // first == 0 or wrapped

}

// too old

if (WINDOW\_SIZE <

(windowTailSeqNum-**sequenceNum**)) {

return false;

}

// now update the bit

**bit\_location** = **sequenceNum** & BITMAP\_LOC\_MASK;

**index\_next** = (**sequenceNum** >> REDUNDANT\_BIT\_SHIFTS) & WINDOW\_INDEX\_MASK;

// first check if the sequence number is in the range

if (**sequenceNum** > windowTailSeqNum) {

**index\_cur** = (windowTailSeqNum >> REDUNDANT\_BIT\_SHIFTS) & WINDOW\_INDEX\_MASK;

**index\_diff** = **index\_next** - **index\_cur**;

for (int **id** = 0; **id** < **index\_diff**; ++**id**)

{

**index** = (**id** + **index\_cur** + 1) % NUM\_BLOCKS;

windowBitmaps[**index**] = 0;

}

windowTailSeqNum = **sequenceNum**;

}

if (0 == (windowBitmaps[**index**]&(1<<**bit\_location**)) )

{

windowBitmaps[**index**] |= (1 << **bit\_location**);

}

return true;

}

}