An Anti-Replay Window Protocol with Controlled Shift *

Chin-Tser Huang Mohamed G. Gouda

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
{chuang, gouda}@cs.utexas.edu

Abstract- **The anti-replay window protocol is used to secure IP against an adversary that can insert (possibly replayed) messages in the message stream from a source computer to a destination computer in the Internet. In this paper, we discuss this important protocol and point out a potential problem faced by the protocol, in which severe reorder of messages can cause the protocol to discard a lot of good messages. We then introduce a controlled shift mechanism that can reduce the number of discarded good messages by sacrificing a relatively small number of messages. We use simulation to show that the modified protocol is more effective than the original protocol when a severe reorder of messages occurs. In particular, we show that the modified protocol reduces the number of discarded good messages by up to 70%.**

I. INTRODUCTION

A replay attack is an attack in which an adversary inserts into the channel, from the source to the destination in a communication, a copy of a message that was sent before by the source ([2] and [14]). If the destination cannot distinguish replayed messages from normal messages when it is under replay attack, the destination may result in incorrectly authenticating the adversary as the source, or incorrectly granting to the adversary access to some resource or service that the adversary does not have access to.

In order to counter replay attacks, IPsec, the standard protocol suite for adding security features to the IP layer in the Internet ([6], [7], and [8]), incorporates a small protocol that is called anti-replay window protocol. This protocol can provide anti-replay service by including a sequence number in each IPsec message and using a sliding window. According to IPsec, a unidirectional security association can be established between any two computers in their networks: one computer is the source of the association and the other is the destination. On the source end, the source keeps a counter for the sequence numbers used for sending messages and the sequence numbers are always monotonic. When a security association is established, the counter is initialized to zero. Every time the source sends a message to the destination, the source includes in the message the current value of the sequence number counter, and increments the counter by one so that the used sequence number will not be reused again. On the destination end, the destination uses a sliding window to determine whether a received message is a normal message or a replayed message. If the sequence number of the received message is less than the number represented by the left edge of the window, then the message is regarded as a replayed message and is discarded by the destination. If the sequence number of the received message falls inside the window, the destination can determine whether the message is a replayed message or not by checking the information kept in the window. If the sequence number of the received message is larger than the number represented by the right edge of the window, the message is regarded as a fresh message and the window is shifted to the right, making this received sequence number the new right edge of the window. (Note that for a message and its sequence number to be accepted, the message also needs to pass integrity check in the destination.)

The anti-replay window protocol has been shown to be effective in preventing replay attacks [4]. Every replayed message inserted by an adversary is guaranteed to be detected and discarded by the anti-replay window protocol. However, this simple protocol has a potential problem in which severe reorder of messages can cause the protocol to discard a lot of good messages. This problem needs to be solved because message reorder occurs very often in the Internet as reported in [1], [10], [12], and [13].

In the rest of this paper, we first formally specify the anti-replay protocol used in IPsec, and show that the protocol can counter replay attacks. Then, we point out a scenario in which severe reorder of messages can cause the protocol to discard a lot of good messages. Next, we propose to heuristically control the shift of the sliding window used by the anti-replay protocol, thus to effectively reduce the number of discarded messages due to message reorders. We use some simulation to show that our protocol performs better than the original protocol when a severe message reorder occurs.

The protocols in this paper are specified using a version of the Abstract Protocol Notation presented in [3]. We use this notation because it provides a well-defined set of semantics that is suitable for distributed environment and is not provided by programming languages like C/C++. In this notation, each process in a protocol is defined by a set of constants, a set of variables, and a set of actions. For example, in a protocol consisting of two processes x and y, process x can be defined as follows.

```
process x
const   <name of constant>   :   <type of constant>
        …
        <name of constant>   :   <type of constant>
var     <name of variable>   :   <type of variable>
        …
        <name of variable>   :   <type of variable>
begin
        <action>
[]      <action>
```

* This work is supported in part by the DARPA contract F33615-01-C-1901 from the Defense Advanced Research Projects Agency. It is also supported in part by an IBM Faculty Partnership Award for the second author for the academic year 2000 - 2001.

…
[]	<action>
end

The constants of process x have fixed values. The variables of process x can be read and updated by the actions of process x. Comments can be added anywhere in a process definition; each comment is placed between the two brackets { and }.

Each <action> of process x is of the form:

<guard> → <statement>

The guard of an action of x is either a boolean expression over the constants and variables of x or a receive guard of the form **rcv** <message> **from** y.

Executing an action consists of executing the statement of this action. Executing the actions (of different processes) in a protocol proceeds according to the following three rules. First, an action is executed only when its guard is true. Second, the actions in a protocol are executed one at a time. Third, an action whose guard is continuously true is eventually executed.

The <statement> of an action of x is a sequence of <skip>, <assignment>, <send>, <selection>, or <iteration> statements of the following forms:

<skip> : **skip**
<send> : **send** <message> **to** y
<assignment> : <list of variables of x> :=
<list of expressions>
<selection> : **if** <boolean expression> →
<statement>
…
[] <boolean expression> →
<statement>
fi
<iteration> : **do** <boolean expression> →
<statement>
od

Note that the <assignment> statement simultaneously can assign new values to multiple variables. Consider for example the following <assignment> statement

wdw[j], j := **false**, j+1

In this statement, the j-th element of the boolean array wdw is assigned the value false, and the value of variable j is incremented by one.

II. The Anti-Replay Window Protocol

In the anti-replay window protocol, a process p sends a continuous stream of messages to another process q. The sent messages may be lost or reordered before they are received by q. A message m is said to suffer a reorder of degree w iff the w-th message sent (by p) after m is received (by q) before m.

At any instant, an adversary can insert in the message stream from p to q a copy of any message that was sent earlier by p. Because of the inserted messages, there is a possibility that process q receives and delivers multiple copies of the same message. To prevent this possibility, the two processes p and q are designed such that the following two conditions are satisfied for a given value w.

w-Delivery:

Process q delivers at least one copy of every message that is neither lost nor suffered a reorder of degree w or more after it is sent by p.

Discrimination:

Process q delivers at most one copy of every message sent by p.

To satisfy these two conditions, p attaches a unique sequence number to each message before sending the message to q, and process q maintains a window of w consecutive sequence numbers. For each sequence number s in the window, q maintains a boolean variable indicating whether or not q has already received the message whose sequence number is s.

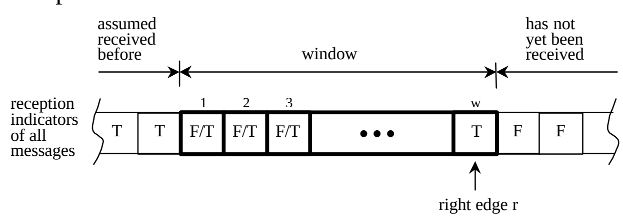
As suggested by Fig. 1, there are three cases to consider when process q receives a message whose sequence number is s.

Case i. s is smaller than all sequence numbers in the window:
In this case, q cannot determine whether it has received this message before. To be on the safe side, q assumes that this message has been received before and discards the message.

Case ii. s is one of the sequence numbers in the window:
In this case, q can determine whether it has received this message before (and so it discards this message) or it has not received this message before (and so it delivers this message).

Case iii. s is larger than all sequence numbers in the window:
In this case, q determines that it has not received this message before and delivers the message. Also q slides the window such that s becomes the new right edge of the window. (This means that some sequence numbers near the old left edge are dropped off the window.)

Next, we present the anti-replay window protocol using the Abstract Protocol Notation described in the Introduction. Process p can be defined as follows.



T: true, the message has been received before
F: false, the message has not yet been received

Fig. 1. The anti-replay window.

```
process p
var   sp    : integer        {sequence number of last msg}
begin
      true →    sp := sp + 1; send msg(sp) to q
end
```

Process q has the following three variables

```
var   wdw   : array [1..w] of boolean,
                                   {window}
      r     : integer,             {right edge of window}
```

Array wdw is the window, and variable r is the maximum sequence number in the window. For each i, 1 ≤ i ≤ w, wdw[i] = true iff process q has already received msg(s), where s = r-w+i. Process q can be defined as follows.

```
process q
const w     : integer              {window size, w > 0}
var   wdw   : array [1..w] of boolean,
                                   {window}
      r     : integer,             {right edge of window}
      s     : integer,             {sequence number}
      i, j  : integer
begin
      rcv msg(s) from p →
         if s ≤ r – w →         {s is to the left of window: }
                                {discard msg}
            skip
         [] r – w < s ≤ r →        {s is in window}
            i := s – r + w;
            if    wdw[i] →         {discard msg} skip
            []  ¬ wdw[i] →         {deliver msg}
                  wdw[i] := true
            fi
         [] s > r →             {s is to the right of window: }
                                {deliver msg}
            r, i, j := s, s-r+1, 1;
            do i ≤ w →    wdw[j], i, j := wdw[i], i+1, j+1
            od;
            do j < w →    wdw[j], j := false, j+1 od
         fi
end
```

The proof of the correctness of the anti-replay window protocol is presented in [4].

III. Shortcoming Of The Protocol

The anti-replay window protocol in Section II does not perform well when severe message reorder occurs. Consider the case where process q receives a reordered message msg(s) where s is larger than r+w. In this case, process q slides the window so that s becomes the new right edge of the window. Thus the new window does not overlap the old window (whose right edge is r), and all the information concerning the old window is lost. Later, if process q receives msg(r+1), then q concludes that this message is to the left of the (new) window and discards the message. We call this kind of message reorder as *long-jump reorder* because the distance between the newly received sequence number and the right edge of the old window is very long.

This scenario of long-jump reorder is not uncommon ([1] and [10]). It can arise when p sends several message over some route to process q, then sends subsequent messages over a shorter route, that just became available, to q. Some of the subsequent messages reach q before the earlier messages. When q receives the first subsequent message, q slides its window to the right a long distance. Later, when q receives the earlier messages, it detects that they are to the left of its window and discards them.

In the next section, we introduce the mechanism of controlled shift that can be incorporated in the anti-replay window protocol to reduce the number of discarded good messages caused by long-jump reorders.

IV. Automatic Versus Controlled Shift

In the anti-replay window protocol, if the new sequence number s received by q is larger than r, namely the right edge of the window, then the window in process q is required to shift to the right (after the message passes the integrity check) and s becomes the new right edge of the window. The protocol is designed this way with the thought that if the sequence number of a message does not appear before, then the message is not a replayed message. We refer to this shift as *automatic shift* because the window automatically shifts to the right to cover the newly received sequence number without any consideration of how far the newly received sequence number is ahead of the current right edge of the window.

Unfortunately, using automatic shift makes the anti-replay window protocol vulnerable to the long-jump reorder described in Section III. When a long-jump reorder occurs, the window automatically shifts to far right in order to cover the newly received sequence number, and when those messages whose sequence numbers fall between the right edge of the old window and the newly received sequence number arrive later, they will be discarded because their sequence numbers are less than the left edge of the new window.

The above scenario suggests that once the window shifts to the right, it cannot shift back to the left to cover those late-coming fresh sequence numbers. This fact drives us to the thought that when the newly received sequence number is more than the window size ahead of the right edge of the window, it is possible that sacrificing the message (by sacrificing we mean discarding the message) that owns this sequence number (and perhaps a small number of following messages) and keeping the window staying in the current position can help us save a lot of late-coming fresh messages. However, it is also possible that all of the messages whose sequence numbers fall between the right edge of the window and the newly received sequence number have been lost. In this case, we do not save any message by sacrificing one (or more) messages. Therefore, it takes a wise and appropriate decision on whether to sacrifice a message for the sake of saving a lot of messages. We refer to this decision-making mechanism as *controlled shift*.

Note that the controlled shift becomes effective only when the sequence number of the newly received message is more than the window size ahead of the right edge of the window. With controlled shift added, the anti-replay protocol is still required to satisfy the two conditions given in Section II to keep the protocol correct. When the controlled shift takes place, we require that the anti-replay protocol with controlled shift should have the following three properties.

Adaptability:
Process q adjusts its criteria of determining whether to sacrifice or accept the newly received message according to the current characteristics of the environment, for example the current message loss rate in IP.

Rationality:
Process q sacrifices a newly received message only when the number of messages that could be saved (because of this sacrifice) is larger than the number of consecutively sacrificed messages (including the newly received one).

Sensitivity:
If process q senses that those skipped messages that it means to save by sacrificing some messages with far-ahead sequence numbers are not likely to come, q stops sacrificing messages and shifts the window to the right to cover the newest received sequence number.

The first property makes the protocol adaptive to the constantly changing environment in the Internet. The second property addresses that the protocol sacrifices only when it figures that it can save more than what is sacrificed. The third property ensures that when those messages that the protocol means to save with controlled shift have been lost, the protocol will not keep sacrificing good messages in vain. In the next section, we present a modified anti-replay protocol to demonstrate how to put controlled shift into reality.

V. A PROTOCOL WITH CONTROLLED SHIFT

In this section, we present a modified anti-replay protocol with controlled shift incorporated and discuss the design in some detail. As stated in Section II, there are two processes p and q in the protocol: p sends a continuous stream of messages to q. The sent messages may be lost or reordered before they are received by q.

To achieve controlled shift, process q needs to be augmented with the following constant and variable in addition to those constant and variables used in the original protocol.

const	dmax	:	**integer**	{maximum number of}
				{consecutively sacrificed}
				{messages}
var	d	:	**integer**	{number of consecutively}
				{sacrificed messages}

Constant dmax is the maximum number of consecutively sacrificed messages allowed by the protocol, and variable d keeps track of the number of consecutively sacrificed messages. The value of d is reset to 0 every time the window is shifted.

As suggested by Fig. 2, there are four cases to consider when process q receives a message whose sequence number is s.

Case i. s is smaller than all sequence numbers in the window:
q cannot determine whether it has received this message before, and so it discards the message.

Case ii. s is one of the sequence numbers in the window:
q can determine whether it has received this message before (and so it discards this message) or it has not received this message before (and so it delivers this message).

Case iii. s is within w positions to the right of the window:
q determines that it has not received this message before and delivers the message. Also, q slides the window such that s becomes the new right edge of the window, and resets d to 0.

Case iv. s is more than w positions to the right of the window:
q determines that it has not received this message before. In this case, q estimates the number of good messages it is going to lose (assuming that these good messages will arrive later), excluding those that are assumed to be lost due to message loss in IP, if q shifts the window to the right and make s the new right edge of the window. q compares this estimate with d+1. If the estimate is larger than d+1, and d+1 is less than dmax, then q discards this message and increments d by 1. Otherwise, q delivers the message, slides the window such that s becomes the new right edge of the window, and resets d to 0.

The first three cases are the same as in the original anti-replay window protocol except that q has to reset d to 0 in the third case. We implement the controlled shift in the fourth case, where s is more than w positions to the right of the window.

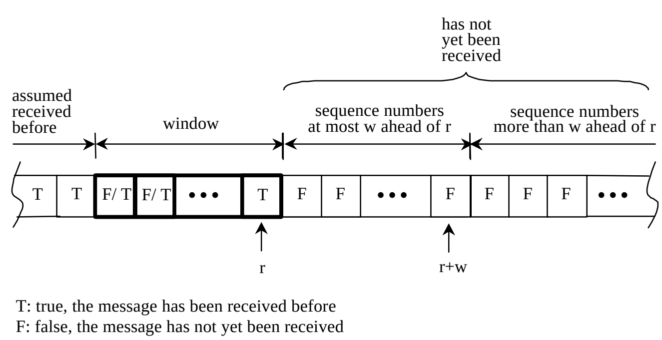


Fig. 2. The anti-replay window for controlled shift.

It is instructive to elaborate how to implement the three properties of controlled shift in the anti-replay protocol. To achieve adaptability, q adaptively estimates how many good messages it is going to lose by making the shift as follows. First, q makes an estimate of the current message loss rate in IP by counting how many sequence numbers in the window have not been received yet and dividing this number by w. Next, q multiplies (1 - estimated message loss rate) by the number of all the sequence numbers sandwiched between the right edge of the current window and the left edge of the new window (if the shift is made). The result is an estimate of the number of good messages q is going to lose by making the shift with message loss in IP excluded. This estimate is adaptive because the number of sequence numbers that have not been received in the current window approximately reflects the current message loss rate in the route from process p to process q.

To achieve rationality, q compares its estimate of the number of good messages it is going to lose by making the shift, with d plus one (the newly received message). q sacrifices the newly received message only when the estimate is larger than d+1, which implies that what can be saved is more than what is sacrificed.

To achieve sensitivity, q compares d+1 with dmax before making a decision of whether to sacrifice. If d+1 reaches dmax, which implies that we have seen dmax consecutive subsequent messages without an earlier message, then q senses that the chance of the arrival of earlier messages is extremely small and shifts the window to the right.

Next, we define the modified anti-replay protocol with controlled shift. Process p is the same as specified in Section II. Process q can be defined as follows.

```
process q
const  w     : integer        {window size, w > 0}
       dmax  : integer        {maximum number of}
                              {consecutively sacrificed}
                              {messages}
var    wdw   : array [1..w] of boolean,   {window}
       r     : integer,       {right edge of window}
       s     : integer,       {sequence number}
       i, j  : integer,
       d     : integer        {number of consecutively}
                              {sacrificed messages}
begin
       rcv msg(s) from p →
           if s ≤ r – w →             {discard msg}
               skip
           [] r – w < s ≤ r →
               i := s – r + w;
               if     wdw[i] → {discard msg} skip
               []  ¬  wdw[i] → {deliver msg} wdw[i] := true
               fi
           [] r < s  ≤ r + w →        {deliver msg and shift wdw}
                                      {to make s the new right}
                                      {edge of window}
               d := 0;
               r, i, j := s, s-r+1, 1;
               do i ≤ w →    wdw[j], i, j := wdw[i], i+1, j+1
               od;
               do j < w →    wdw[j], j := false, j+1 od
           [] s > r + w →             {make the decision of}
                                      {whether to sacrifice}
                                      {or deliver message}
               i, j := 1, 0;
               do i ≤ w →
                   if      wdw[i] → skip
                   []  ¬   wdw[i] → j := j+1
                   fi; i := i+1
               od;
               if  (((1-(j/w))*(s-r-w)>d+1) ^ (d+1<dmax)) →
                   d := d+1
               [] ¬(((1-(j/w))*(s-r-w)>d+1) ^ (d+1<dmax)) →
                   d := 0;
                   r, i, j := s, s-r+1, 1;
                   do i ≤ w → wdw[j], i, j := wdw[i], i+1, j+1
                   od;
                   do j < w → wdw[j], j := false, j+1 od
               fi
           fi
end
```

It is straightforward to verify that the modified protocol with controlled shift satisfies the two conditions given in Section II, namely w-delivery and discrimination, by using the same verification methods that are based on auxiliary variables [11], annotation [5], and invariants [9] as used in [4]. However, the verification does not show how effective this protocol is in reducing message losses caused by long-jump reorders. In the next section, we use some simulation results to display the effectiveness of the modified protocol.

VI. Simulation Results

The scenario we simulated can be described as follows. There are two processes p and q, where p sends a continuous stream of messages to q. Each message carries a sequence number with it. The sequence number starts from 1 and is incremented by 1 with every subsequent message sent by process p. Process q uses a window of size 64 to keep track of the sequence numbers it received. However, the sent messages may be lost or reordered before they are received by q. If a message is received when its sequence number falls behind the left edge of the window, this message will be discarded by q. Note that there is no need to simulate message replay because it has been shown that both the original protocol and the modified protocol will discard every replayed message.

To simulate message loss in IP, we set that the message loss rate in our simulation to be 10%. To simulate message reorder in IP, we arranged that 20 small message reorders, where a message suffers a reorder of degree less than 10, occur in the message stream. We also arranged that one long-jump reorder, where a message suffers a reorder of degree larger than 64, occurs in the message stream.

We generated many such message streams to be sent from process p to process q. From each generated stream, process q keeps receiving messages until it receives 1000 messages, then the simulation run terminates. We used each of the generated message streams with process q in the original protocol once, and with process q in the modified protocol ten times, each time with a different value for dmax. Our choices of dmax range from 3 to 12. We recorded the number of messages discarded by q in each simulation run and summed up the numbers. Then, we derived the saving percentage of discarded messages by dividing the number of saved messages (the difference between the number of messages discarded by the modified protocol and the number of messages discarded by the original protocol) by the number of messages discarded by the original protocol. The simulation results for different message loss rates are shown in Fig. 3.

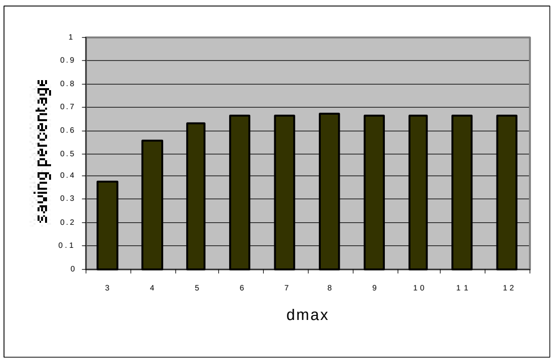


Fig. 3. Simulation results when loss rate=10%.

As shown in the figure, the saving percentage of the modified protocol can be as high as almost 70%. The modified protocol reaches the highest saving percentage when dmax equals 8. The saving percentage remains almost the same when we carried out the simulation with different message loss rates of 5%, 15%, and 20%. This fact shows the adaptability of the modified protocol.

VII. Concluding Remarks

In this paper, we discussed the anti-replay window protocol used in IPsec, and pointed out a potential problem faced by the protocol: when a long-jump reorder occurs, the protocol will end up discarding a lot of good messages. To remedy this flaw, we propose to add a controlled shift mechanism to the protocol. Controlled shift can reduce the harm caused by long-jump reorder with its three properties: adaptability, rationality, and sensitivity. We then present a modified version of the anti-replay protocol that incorporates the controlled shift mechanism, and use simulation to show that the modified protocol loses much less good messages when a long-jump reorder occurs.

In some rare cases where more than w consecutive messages get lost, the modified protocol with controlled shift will be led to discard dmax good messages before it shifts the window to the right. This is the only case when the modified protocol loses more messages than the original protocol does. However, with a good choice of dmax, the loss of the modified protocol can be limited to a small value.

Our simulation results show that the modified protocol has optimal performance when dmax is set to be 8. However, this value has not yet been justified by some analysis as there is currently no commonly accepted model of message reorders in the Internet. Toward this end, more extensive experiments need to be conducted to estimate how message reorders, especially long-jump reorders, occur in the Internet.

References

[1] Bennett, T. C. R., C. Partridge, N. Shectman, "Packet Reordering is Not Pathological Network Behavior", *IEEE/ACM Transactions on Networking*, Vol. 7, No. 6, pp. 789-798, Dec. 1999.

[2] Gong, L., "Variations on the Themes of Message Freshness and Replay", *Proceedings of the Computer Security Foundations Workshop VI*, pp. 131-136, Jun. 1993.

[3] Gouda, M. G., *Elements of Network Protocol Design*, John Wiley & Sons, New York, NY, 1998.

[4] Gouda, M. G., C.-T. Huang, E. Li, "Anti-Replay Window Protocols for Secure IP", *Proceedings of the 9th IEEE International Conference on Computer Communications and Networks*, Las Vegas, October 2000.

[5] Hoare, C. A. R., "An axiomatic approach to computer programming", *Comm. ACM* 12, pp. 576-581, 1969.

[6] Kent, S., and R. Atkinson, "*Security Architecture for the Internet Protocol*", RFC 2401, November 1998.

[7] Kent, S., and R. Atkinson, "*IP Authentication Header*", RFC 2402, November 1998.

[8] Kent, S., and R. Atkinson, "*IP Encapsulating Security Payload (ESP)*", RFC 2406, November 1998.

[9] Manna, Z., and A. Puneli, "*The Temporal Logic of Reactive and Concurrent Systems*", Springer-Verlag, New York, 1991.

[10] Mogul, J., "Observing TCP Dynamics in Real Networks", *Proc. SIGCOMM '92*, pp. 305-317, Aug. 1992.

[11] Owicki, S., and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I", *Acta Informatica*, Vol. 6, No. 1, pp. 319-340, 1976.

[12] Paxson, V., "End-to-End Routing Behavior in the Internet", *IEEE/ACM Transactions on Networking*, Vol. 5, No. 5, pp. 601-615, Oct. 1997.

[13] Paxson, V., "End-to-End Internet Packet Dynamics", *IEEE/ACM Transactions on Networking*, Vol. 7, No. 3, pp. 277-292, Jun. 1999.

[14] Syverson, P., "A Taxonomy of Replay Attacks", *Proceedings of the Computer Security Foundations Workshop VII*, pp. 187-191, Jun. 1994.