SOURCEfire

Target-Based TCP Timestamp Stream Reassembly

Authors
Judy Novak
Steve Sturges

Revision 2.0
July 30, 2007

Prepared by:
Sourcefire, Incorporated
9770 Patuxent Woods Drive
Columbia, MD 21046

Abstract

This paper explores the use of the TCP timestamp option and associated timestamp values to comprehend how different operating systems react to manipulated timestamp values. This is valuable knowledge for an intrusion detection system (IDS) or intrusion prevention system (IPS) to possess and implement to avoid evasions that employ TCP timestamp value mutations.

Introduction to Timestamps

The TCP timestamp option is used by many current operating systems. There are two timestamp values associated with the TCP timestamp options field – the sender's timestamp followed by the receiver's echoed timestamp. Each timestamp value represents the respective computer's "up time", the number of units that have passed since the last reboot. According to RFC 1323 "TCP Extensions for High Performance" [1], the unit selected to mark the passing of time is as follows - "we choose a timestamp clock frequency in the range 1 millisecond to 1 second per tick".

TCP timestamps are used to measure the round trip time (RTT) of a given TCP segment. As well, TCP timestamps provide an indication of when to discard delayed segments – a process known as Protection Against Wrapped Sequence Numbers (PAWS). There is a finite range of 32-bit TCP sequence numbers available to mark the chronology of TCP segments, so it is possible for sequence numbers to wrap from the largest possible back to zero in a particular TCP session. Theoretically, it is possible, though not likely, for a sending host or network to delay a segment, for that same host to send additional segments containing TCP sequence numbers that wrap, and for the delayed segment to arrive coincidentally during a current exchange where the delayed TCP sequence number is in the current window and conflicts with the valid segment containing a TCP sequence number in the current window. The receiving host discards the delayed segment because the timestamp is older than the one the receiving host most recently acknowledged. RFC 1323 summarizes the timestamp as "From the receiver's viewpoint, the timestamp is acting as a logical extension of the high-order bits of the sequence number."

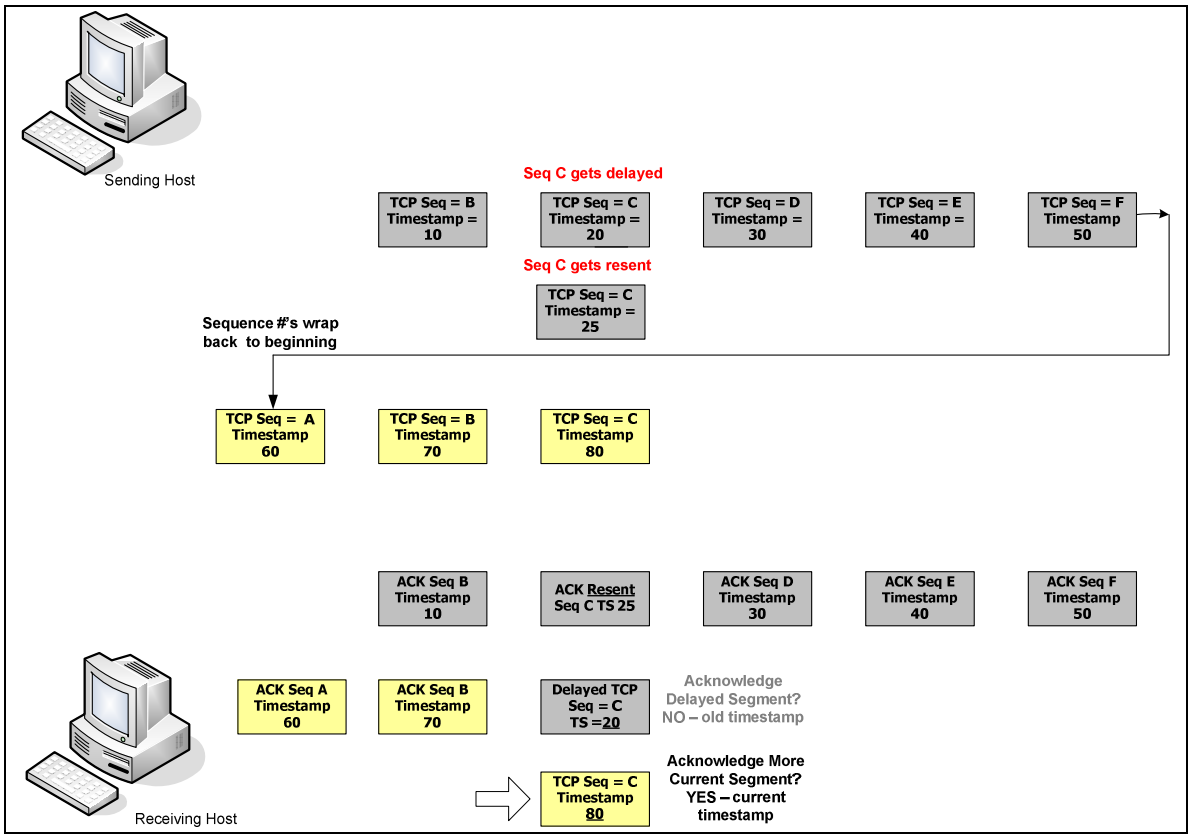

Figure 1. Illustration of Protection Against Wrapped Sequence numbers

In Figure 1, the sending host first sends a series of segments with chronologically increasing TCP sequence values of "B" through "F". We purposely use letters to denote the sequence values to focus on the chronology and not specific numbers, per se. Assume that the range of TCP sequence values available for use is "A" through "F", analogous to real TCP sequence numbers 0 through $2^{32} - 1$. TCP timestamps also increase on each of the segments that the host sends.

Suppose that the segment with TCP sequence "C" and a TCP timestamp of 20 gets delayed and the receiving host does not acknowledge it. Eventually, the sending host resends TCP sequence "C" segment, but with a timestamp of 25 that reflects the later timestamp clock. The receiving host acknowledges all the segments that arrive including the one with sequence "C" and a timestamp value of 25. Since sequence "F" is the final sequence value in our range of possible values, the sequence value wraps back to "A" and the sending host then transmits segments with sequence values of "A", "B", and "C". The receiving host acknowledges segments with sequence "A" and sequence "B". Even though the host received a sequence "B" before; it already acknowledged it and the current sequence "B" once again becomes the next expected sequence value.

Now comes the trickery. Coincidentally, the delayed original sequence "C" segment with timestamp 20 arrives concurrently with current sequence "C" with timestamp 80. The receiving host most recently acknowledged sequence "B" with a timestamp of 70. That means that the delayed sequence "C" segment has an old timestamp of 20 compared to 70. The receiving host discards it and acknowledges sequence "C" segment with a timestamp of 80. As you probably

surmised, such an occurrence is a rare coincidence, but the authors of RFC 1323 included the Protection Against Wrapped Sequence numbers nevertheless.

The segments in our contrived “A” through “F” range of sequence values appear to wrap immediately. Segments with sequence values of “B” and “C” are once again in the current window almost as soon as they are recycled from the first iteration of segments with identical sequence values. In reality, the recycling is not immediate, but instead a host sends millions of segments before it reuses duplicate TCP sequence numbers in the same session.

Timestamps and Intrusion Detection

Ordinarily, TCP timestamps have little or no effect on accurately detecting intrusions since a host, hence IDS/IPS that receives a segment with an old timestamp should discard it in favor of one with a current timestamp. It stands to reason that if an attacker attempts some kind of evasion by sending wholly overlapping segments, one with a valid timestamp containing a payload of an attack, and the other with an old timestamp, but no attack payload; the IDS/IPS should discard the segment with the old timestamp and honor and examine the one with the current timestamp thus mimicking the destination host behavior.

Yet, extensive examination of the use of crafted eccentric TCP timestamps combined with the ambiguity of RFC 1323 expose unique behavior by different operating systems. We have observed this target-based behavior after we send overlapping TCP segments when unacknowledged segments exist. In other words, when we place unexpected or old timestamps in overlapping segments that arrive before some delayed segment, different operating systems manifest unique responses. Unless an IDS/IPS is aware of and understands the target's behavior, it is susceptible to evasions by use of overlapping TCP segments with different or abnormal TCP timestamps.

One philosophy regarding accurate handling of old timestamps and an appropriate IDS/IPS reaction that conforms to the destination host's behavior is that it is even more important for an intrusion prevention system than an intrusion detection system to understand the implications of TCP timestamps. While it is acceptable for an intrusion detection system to alert about an exploit after-the-fact, the intrusion prevention system must block the entry of malicious packets. Realistically, what this means is that an intrusion detection system can determine how a destination host processed one or more segments by examining the TCP acknowledgement numbers that the receiving host returns. Of course, this becomes more involved when overlapping segments are sent, requiring the intrusion detection system to be familiar with target-based TCP stream reassembly. An acknowledgement from the receiving host in this case does not necessarily disambiguate which of the overlapping segments the receiving host favors. In contrast, an intrusion prevention system does not get the opportunity to examine feedback from the destination host; it must have a priori knowledge and react appropriately.

An IPS that incorrectly processes TCP segments with unusual timestamps may be duped into generating a false positive and unnecessarily blocking a segment that the destination host would ultimately drop. Worse yet, it may permit a malicious segment that the IPS should block to reach the destination host and possibly cause some harm. A second philosophy about intrusion preventions systems is that scrutiny before a packet enters the network affords the IPS the opportunity to perform packet scrubbing for abnormal or unusual characteristics. In the case of expired timestamps, an intrusion prevention system could choose to drop the packet or scrub it to have no timestamp, examine the payload, and permit entry if it finds no malicious payload.

The Reality of TCP Timestamps

According to RFC 1323 "It is important to note that the timestamp is checked only when a segment first arrives at the receiver, regardless of whether it is in-sequence or it must be queued for later delivery." If there is a delayed segment, once it arrives, "the timestamps of the queued segments are *not* inspected again at this time, since they have already been accepted." Given this guidance, our expectation is that TCP will inspect segments with TCP timestamps as they arrive. TCP does not wait to compare the just-arrived segment's timestamp with the delayed segment's timestamp. Instead, it compares the just-arrived segment's timestamp with those of segments that have already arrived.

The reality of how hosts react to old timestamps is much more convoluted than a simple one-size-fits-all solution. This is especially so when we fabricate the TCP segments that we send to contain unusual timestamp values, or contain no timestamp, or delay the arrival of the first segment that usually follows the three-way handshake. Because there are so many exceptions and tangled logic forks found in timestamp processing, we attempt to make the explanation of timestamp processing more coherent by employing the dreaded flow-chart to depict some of the phases and issues involved.

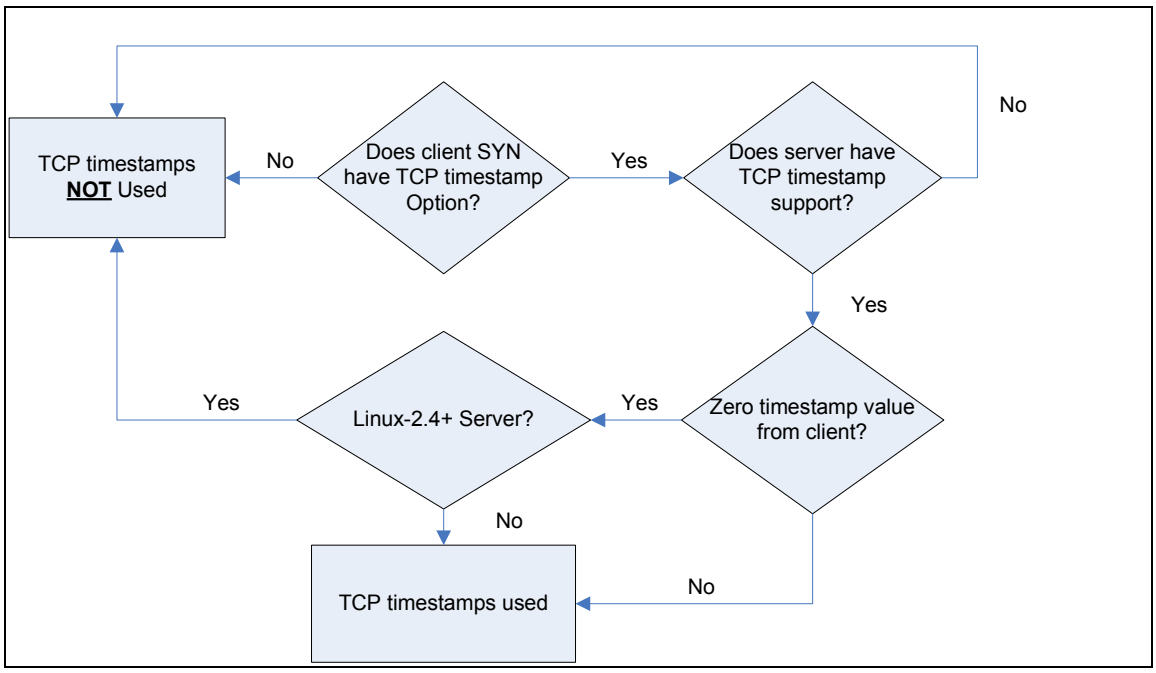

Figure 2. TCP timestamp processing on the three-way handshake

First, let's examine what transpires on the three-way handshake shown in Figure 2. Both client and server must support the TCP timestamp option for either to employ it. Each signals its intent to use the TCP timestamp by including it in the TCP options field when it sends the segment where the SYN flag is set. A client that uses a timestamp on the three-way handshake typically sets its timestamp value to the current timestamp clock value and sets the receiver's timestamp to zero. However, some operating systems, such as Windows set both the sender's and receiver's timestamps to zero and wait until after the completion of the three-way handshake to begin to record their own timestamp and echo the receiver's timestamp. Servers that run different operating systems uniquely respond to a client initiation where the TCP timestamp values are zero.

Before we examine how a server handles a client SYN segment with zero timestamps, you must have a general understanding of how a server behaves after receiving a client SYN segment that contains TCP options. Servers reflect client TCP options. Not all operating systems support all available or offered TCP options. If a server receives a session initiation segment from a client that contains fewer TCP options than the server supports, the server reflects only the TCP options that the client sent. Conversely, if a server receives a session initiation segment from a client that contains more TCP options than the server supports, the server reflects only the TCP options that it supports. When some operating systems such as Linux 2.4 and later kernels receive a Windows client SYN segment with zero timestamp values, they omit the TCP timestamp option in the TCP options reflected back to the client. Even though the client offers, and the server supports the timestamp option, a value of zero in the client's timestamp causes the server to reject the use of the TCP timestamp option altogether.

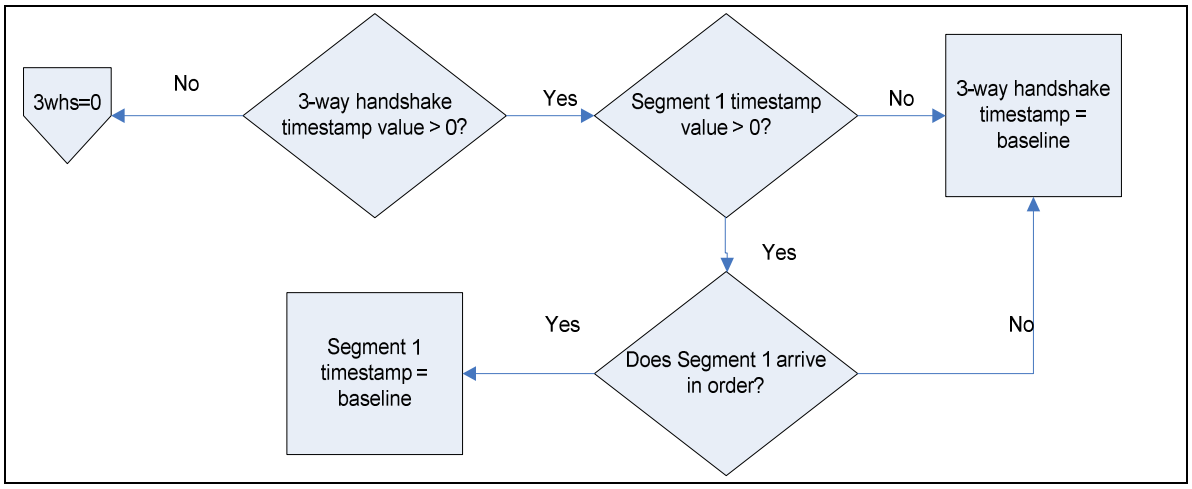

Figure 3. Processing after receiving three-way handshake with a non-zero TCP timestamp value

Next, let's look at the logic when the three-way handshake timestamp value is non-zero as displayed in Figure 3. This is the simpler logic path than a zero timestamp value since receiving hosts use this non-zero timestamp value as the "baseline" to compare subsequent segment timestamps. The client segment with the TCP sequence number following the three-way handshake, labeled "Segment 1", has special significance in TCP timestamp processing, especially in the next case where we examine the TCP timestamp from the three-way handshake has a value of zero. To clarify, a non-zero timestamp value on segment 1 means that there must be a TCP timestamp in the segment and it must be greater than zero. Segment 1 becomes the baseline when it has a non-zero timestamp value and arrives directly after the three-way handshake. Otherwise, the timestamp from the three-way handshake remains the baseline.

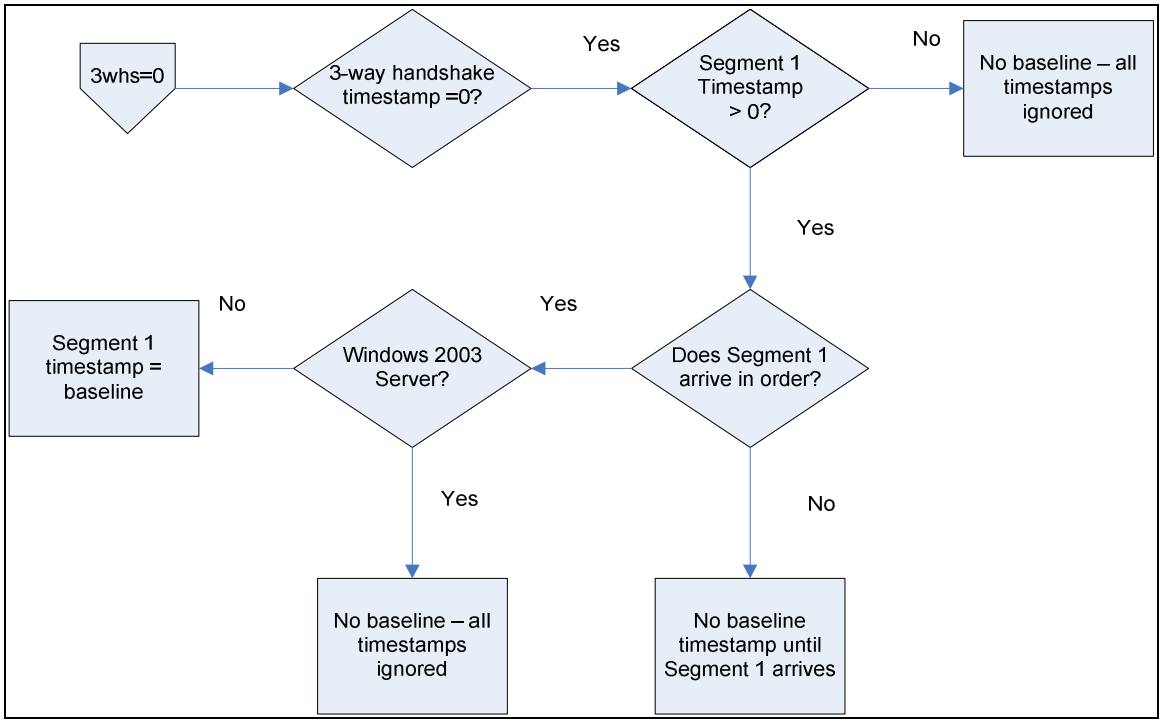

Figure 4. Processing after receiving three-way handshake with a zero TCP timestamp value

Figure 4 shows the processing flow when the three-way handshake TCP timestamp value is zero. In general, it appears that a receiving host uses segment 1 as a baseline for TCP timestamp values when the three-way handshake timestamps are zero. This is true if the timestamp value on segment 1 is non-zero and it arrives in order. One notable exception is Windows 2003 which appears to ignore timestamps completely even when segment 1 arrives in order and has a non-zero value.

For the operating systems in our tests, if segment 1 has no timestamp or a timestamp value of zero, then the receiving host ignores all subsequent timestamps for the duration of the TCP session. Even if segment 2 arrives with a valid timestamp, the operating system does not consider segment 2 the baseline timestamp. Another anomalous situation is depicted in Figure 4 where segment 1 is delayed. If this is the case, the timestamps of segments that arrive before it are irrelevant – the receiving host treats them as if they have no timestamps at all. All segments with an old timestamp value that arrive before this special segment 1 are not discarded. Let's assume that we have segments 1, 2, and 3 chronologically in TCP sequence number order. Now, let's say segment 2 has a valid timestamp but it arrives immediately after the three-way handshake, segment 3 follows with a timestamp value of zero, and finally delayed segment 1 arrives with a valid timestamp. The receiver accepts segment 3 even though it has an old timestamp relative to segment 2. Tested TCP stacks do not appear to have a baseline timestamp value until segment 1 arrives. Once segment 1 arrives, the receiving host compares timestamp values of segments that follow to segment 1's timestamp value. While RFC 1323 does not explicitly state this, the different operating system TCP stacks that we tested conform to this convention.

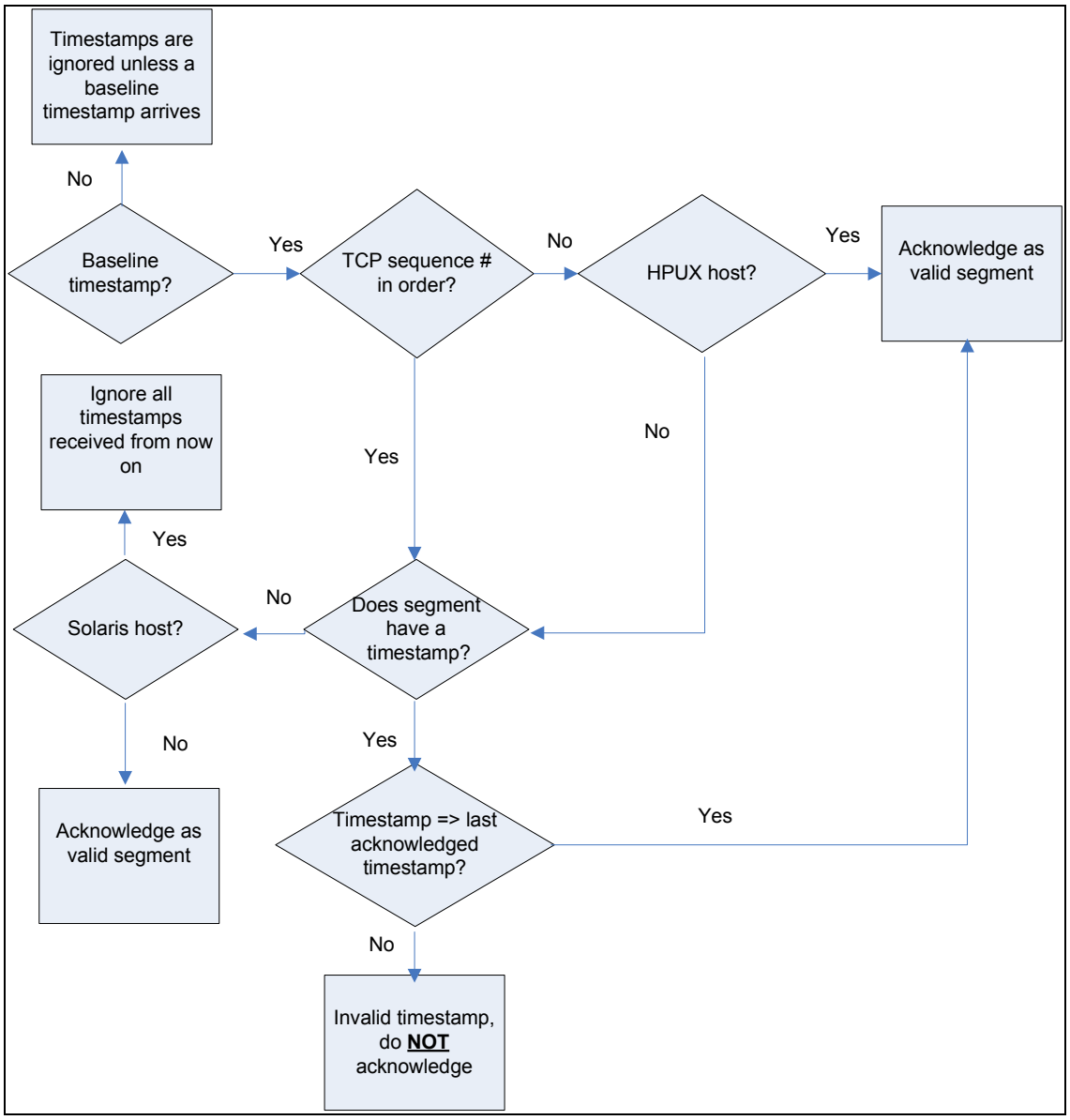

Figure 5. General processing of timestamps after baseline timestamp established

Now, let's examine what happens once a baseline timestamp has been established. A segment arrives and the receiving host compares its TCP sequence number to the most recently acknowledged one. If the receiving host is HPUX and the TCP sequence number is not the next expected one, it acknowledges the segment regardless of the TCP timestamp value. Stated differently, an HPUX host does not discard a segment with an old timestamp value if the segment arrives out of order even if there is a legitimate baseline timestamp value for comparison. All other operating systems continue timestamp processing whether or not the TCP sequence number is the next expected one.

If an aberrant segment with no timestamp arrives in the middle of a TCP exchange where the use of timestamps has been negotiated, most operating systems simply acknowledge the segment if it has the next expected sequence number. However, Solaris hosts respond in a unique fashion and ignore all timestamps in subsequent segments. It's as if the lack of a TCP timestamp in any

segment sent to a Solaris signals it to turn off timestamp tracking for the remainder of the TCP session.

Finally, if the timestamp is greater than the most recently acknowledged one, the receiving host acknowledges it. If the timestamp is older than the most recently acknowledged one, the receiving host discards the segment.

Sample Timestamp Responses

Before we proceed, let's illustrate some specific examples of timestamp processing. In this first example, the client establishes the three-way handshake with the timestamp option and the server supports the timestamp option too. Both segments of the client SYN and the ACK of the three-way handshake have a timestamp of 10. All subsequent segments must have a timestamp greater than 10 to be valid. Segment 1 which is next in terms of TCP sequence number is delayed, and segment 2 arrives with a timestamp of 20. According to RFC 1323, the receiving host should examine the timestamp and queue segment 2 pending the arrival of segment 1. Segment 1 arrives with an old timestamp of 5 which is less than the timestamp found on segments from the three-way handshake. The server never acknowledges segments 1 or 2 because of this; this indicates that it queued segment 2, then received segment 1 that contains the old timestamp and discarded it and any segments that followed it with greater TCP sequence number values.

Client SYN –	TS 10
Server SYN/ACK -	TS 2000
Client ACK –	TS 10
Client Segment 2 -	TS 20
Client Segment 1 -	TS 5

In the next example, we examine the server's behavior where we manipulate the client's three-way handshake values to be zero and delay segment 2.

Client SYN -	TS 0
Server SYN/ACK -	TS 2000
Client ACK -	TS 0
Client Segment 1-	10
Client Segment 3A -	3
Client Segment 3B -	30
Client Segment 2 -	20

The client has timestamp values of zero on the three-way handshake, followed by a segment 1 with a timestamp of 10. Next, segments 3A and 3B wholly overlap each other (they start and end with the same TCP sequence number), but have a different payload and segment 3A has an old timestamp. Finally, delayed segment 2 arrives with a timestamp that is valid for its chronological TCP sequence number. Segment 1 becomes the baseline timestamp and the receiver compares timestamp values found in segments 3A and 3B to it pending the arrival of segment 2.

This example is more complex because it introduces another concern, overlapping segments. This could be a target-based issue since some operating systems favor the original segment sent while others favor the subsequent one. However, it becomes a target-based concern only if both overlapping segments 3A and 3B have valid timestamps since timestamp processing precedes overlap processing. In this instance where segment 3A has an old timestamp, the receiving host discards it and acknowledges segment 3B. Consequently, you no longer have overlapping segments since the host accepts only 3B as valid.

These two examples give you an inkling of some of the issues involved in understanding how hosts perform timestamp processing. We developed a comprehensive set of tests to explore the behavior of receiving hosts and timestamp issues. Yet, the behavior that we expect does not always occur. Under certain conditions, some operating systems appear to suspend examination or ignore the use of timestamps altogether from segments that arrive before a delayed segment.

TCP Timestamp Tests

RFC 1323 does not elaborate how a host should respond when it receives zero timestamp values on the three-way handshake, or when it receives segments with no TCP timestamp option and associated values even though both hosts have negotiated the use of timestamps. We created a myriad of test cases using these considerations along with using old timestamps. For all test cases, the client connects to the server and we manipulate the TCP timestamp values on client segments to observe how the server responds. Since segment 1 seems to have some magical, yet undocumented, value in terms of timestamps, we include tests that alter timestamp values in it.

The whole point of each test is to determine how a server responds to a different combination of TCP timestamp values when it receives overlapping segments. Therefore, the server application we use needs to return a different response depending on whether it honors the original or overlapping segment. A web server meets this prerequisite of responding with a returned web page when a segment contains a valid URL. The web server responds with an error message when a segment contains an invalid URL. A web server responds with a web page when it receives a valid URL and HTTP version. But, a web server (except IIS) responds with an error page when it receives the same URL, but an invalid HTTP version. Each timestamp test has the following basic flow:

```
Client SYN with TCP timestamp -> server port 80
Server SYN/ACK with TCP timestamp
Client ACK with TCP timestamp to server SYN/ACK
Client segment 1 with payload of "GET"
Client segment 3A with payload of  "/index.html HTTP/4.0\r\n\r\n"
Client segment 3B with payload of  "/index.html HTTP/1.0\r\n\r\n"
Delayed segment 2 with payload of " "
```

After we perform many tests with this flow, we conduct a final series of tests that swaps the arrival order of segments 1 and 2 where segment 2 arrives before segments 3A and 3B and segment 1 arrives last.

The existence of, and values associated with TCP timestamps for the above flow varies for each test. Also, overlap configurations for segments 3A and 3B change. In this particular example, segments 3A and 3B start and end with the same TCP sequence number so that segment 3B wholly overlaps segment 3A. If a web server favors segment 3A, it returns an error message. If it favors segment 3B, it returns the requested web page. We'll discuss changes to overlap positioning in more detail in the following section, but let's concentrate on timestamp manipulation for now.

The following tests examine many different aspects and combinations of timestamp values:

Round 1, Case 1: Client timestamp values on the three-way handshake are zero; segment 1 arrives first and has a timestamp value of 11111. Segments 3A and 3B contain various timestamp values and options – no timestamp options, old timestamp or valid timestamps. Delayed segment 2 arrives last with a valid timestamp value of 12345. We expect the receiving host to examine segments 3A and 3B relative to segment 1's timestamp.

	3whs	Segment 1	Segment 3A	Segment 3B	Segment 2
Test 1	ts = 0	ts = 11111	ts = 22222	ts = 33333	ts = 12345
Test 2	ts = 0	ts = 11111	no timestamp	ts = 33333	ts = 12345
Test 3	ts = 0	ts = 11111	ts = 22222	no timestamp	ts = 12345
Test 4	ts = 0	ts = 11111	ts = 22222	ts = 22222	ts = 12345
Test 5	ts = 0	ts = 11111	no timestamp	no timestamp	ts = 12345
Test 6	ts = 0	ts = 11111	ts = 33333	ts = 22222	ts = 12345
Test 7	ts = 0	ts = 11111	ts = 0	ts = 0	ts = 12345
Test 8	ts = 0	ts = 11111	ts = 22222	ts = 3	ts = 12345
Test 9	ts = 0	ts = 11111	ts = 3	ts = 22222	ts = 12345

Round 1, Case 2: Client timestamp values on the three-way handshake are non-zero; segment 1 arrives first and has a timestamp of 11111. Segments 3A and 3B contain various timestamp values and options – no timestamp options, old timestamp or valid timestamps. Delayed segment 2 arrives last with a valid timestamp value of 12345. We expect the receiving host to examine segments 3A and 3B relative to the client's segment 1 timestamp.

	3whs	Segment 1	Segment 3A	Segment 3B	Segment 2
Test 1	ts = 10000	ts = 11111	ts = 22222	ts = 33333	ts = 12345
Test 2	ts = 10000	ts = 11111	no timestamp	ts = 33333	ts = 12345
Test 3	ts = 10000	ts = 11111	ts = 22222	no timestamp	ts = 12345
Test 4	ts = 10000	ts = 11111	ts = 22222	ts = 22222	ts = 12345
Test 5	ts = 10000	ts = 11111	no timestamp	no timestamp	ts = 12345
Test 6	ts = 10000	ts = 11111	ts = 33333	ts = 22222	ts = 12345
Test 7	ts = 10000	ts = 11111	ts = 0	ts = 0	ts = 12345
Test 8	ts = 10000	ts = 11111	ts = 22222	ts = 3	ts = 12345
Test 9	ts = 10000	ts = 11111	ts = 3	ts = 22222	ts = 12345

Round 2, Case 1: Client timestamp values on the three-way handshake are zero; segment 1 arrives first, but has **no** timestamp. Test cases for segments 3A and 3B remain the same as Round 1. Delayed segment 2 arrives last with a valid timestamp value of 12345. But, there is no "baseline" timestamp to compare segments 3A and 3B timestamps. We expect the receiving host to ignore timestamps completely for the entire session.

	3whs	Segment 1	Segment 3A	Segment 3B	Segment 2
Test 1	ts = 0	no timestamp	ts = 22222	ts = 33333	ts = 12345
Test 2	ts = 0	no timestamp	no timestamp	ts = 33333	ts = 12345
Test 3	ts = 0	no timestamp	ts = 22222	no timestamp	ts = 12345
Test 4	ts = 0	no timestamp	ts = 22222	ts = 22222	ts = 12345
Test 5	ts = 0	no timestamp	no timestamp	no timestamp	ts = 12345
Test 6	ts = 0	no timestamp	ts = 33333	ts = 22222	ts = 12345
Test 7	ts = 0	no timestamp	ts = 0	ts = 0	ts = 12345
Test 8	ts = 0	no timestamp	ts = 22222	ts = 3	ts = 12345
Test 9	ts = 0	no timestamp	ts = 3	ts = 22222	ts = 12345

Round 2, Case 2: Client timestamp values on the three-way handshake are non-zero; segment 1 arrives first, but has **no** timestamp. Again, we perform the same timestamp tests for segments 3A and 3B. Delayed segment 2 arrives last with a valid timestamp of 12345. But this time there is a "baseline" timestamp found in the segments of the three-way handshake. We expect timestamps found in segments 3A and 3B to be compared to those.

	3whs	Segment 1	Segment 3A	Segment 3B	Segment 2
Test 1	ts = 10000	no timestamp	ts = 22222	ts = 33333	ts = 12345
Test 2	ts = 10000	no timestamp	no timestamp	ts = 33333	ts = 12345
Test 3	ts = 10000	no timestamp	ts = 22222	no timestamp	ts = 12345
Test 4	ts = 10000	no timestamp	ts = 22222	ts = 22222	ts = 12345
Test 5	ts = 10000	no timestamp	no timestamp	no timestamp	ts = 12345
Test 6	ts = 10000	no timestamp	ts = 33333	ts = 22222	ts = 12345
Test 7	ts = 10000	no timestamp	ts = 0	ts = 0	ts = 12345
Test 8	ts = 10000	no timestamp	ts = 22222	ts = 3	ts = 12345
Test 9	ts = 10000	no timestamp	ts = 3	ts = 22222	ts = 12345

We repeat the above series of four sets of tests, but switch the arrival order of segments 1 and 2. The new tests are known as "Round 3, Case 1 and 2" and "Round 4, Case 1 and 2". We performed this new series of tests to try to understand the role of segment 1 as the baseline timestamp tests. Segment 2 has a valid timestamp of 12345 and it arrives before segments 1, 3A, and 3B. Yet, results reveal that the receiving host does not use segment 2 as a baseline timestamp for later segments 3A and 3B. Segment 1 must arrive first when the three-way handshake values are zero in order for old timestamps in segments 3A and 3B to be discarded.

We should note that while it appears that our concentration in this series of tests is the behavior of the host that receives TCP segments with data and unusual timestamps, the effects of properly evaluating timestamps are more widespread. The salient point is that a host should reject any segment that has an old timestamp whether or not it contains data. For instance, an acknowledgment with an old timestamp should not be accepted. If this is not treated properly by

an IDS/IPS, the IDS/IPS may not correctly analyze the actual data exchanged and acknowledged for the remainder of the stream. This includes knowing when the session is actually closed. If an IDS/IPS is fooled into believing a session is closed, it is easy to perform an evasion.

The Sturges/Novak Overlapping TCP Segment Model

Authors Steve Sturges and Judy Novak have presented their research of the issue of overlapping TCP segments in their paper "Target-based TCP Stream Reassembly". In that paper, we examined operating system behavior using a paradigm of different segment overlaps. We felt that we must examine not only wholly overlapping segments as we referenced in our timestamp tests as segments 3A and 3B, but all the overlap conditions found in the below segment overlap model.

SOURCEfire
Security for the real world.

All Possible TCP Segment Overlap Scenarios

	Original vs. Subsequent Segment	Subsequent vs. Original Segment
Starts before, ends after	Segments 3.3 & 8	Segments 6 & 3.1
Starts before, ends before	Segments 1 & 4	Segments 4 & 2
Starts before, ends same	Segments 3.4 & 9	Segments 7 & 3.2
Starts same, ends after	Segments 3.5 & 10	Segments 11 & 3.6
Starts same, ends before	Segments 3.6 & 11	Segments 10 & 3.5
Starts same, ends same	Segments 3 & 5	Segments 5 & 3
Starts after, ends after	Segments 2 & 4	Segments 4 & 1
Starts after, ends before	Segments 3.1 & 6	Segments 8 & 3.3
Starts after, ends same	Segments 3.2 & 7	Segments 9 & 3.4

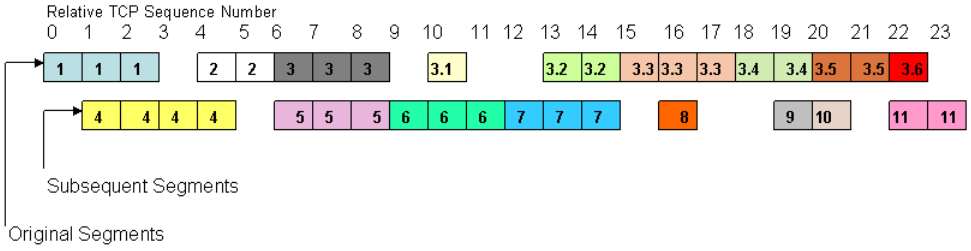

Figure 6. The Sturges/Novak TCP segment overlap model

We had trepidations about exploring the vast number and sheer complexity of the overlap paradigm in conjunction with the TCP timestamp tests developed. We did not want to attempt to develop and test different timestamp combinations in the above segment overlap model. Instead, we conducted all of our TCP timestamp tests using each individual set of the nine segment overlaps listed in the Figure 6 table under the column "Original vs. Subsequent Segment". Now, segments 3A and 3B of our timestamp tests are not necessarily wholly overlapping. Segment 3A may become the incarnation of the above model original segment "3.1" and segment 3B may become the incarnation of the above model subsequent segment "6" where 3A begins after and ends before segment 3B.

Our paper "Target-based TCP Stream Reassembly" explains the model concept in great detail. Briefly, model segment "3.1" or "6" simply refers to the label of an entire segment that is one or more bytes. It has nothing to do with the sequence numbers or payloads of those bytes. For

instance, model segment "3.1" refers to the single byte in the first row of the model labeled original segments. Segment "6" refers to the set of three bytes in the second row of the model labeled subsequent segments. Segments "3.1" and "6" overlap – subsequent segment "6" begins before (relative TCP sequence number less than segment "3.1") and ends after (relative TCP sequence number greater than "3.1) original segment "3.1".

As an example of one test, we took overlapping segments 3.3 and 8 from the above overlap model and ran them through all of the 8 sets (Round 1, Case 1 through Round 4, Case 2) of the nine individual timestamp tests.

Before reviewing some of the results, it is helpful to understand that there are three different kinds of segment overlap conditions that affect timestamp processing. First, there are overlaps where the receiving host must consider both valid to form a complete HTTP request. Specifically, overlaps 1-4 (original segment 1 and subsequent segment 4) and 2-4 (original segment 2 and subsequent segment 4) have one segment that begins before the other, and that same segment ends before the other. If either of these segments is invalid, a gap in sequence numbers and content remains. Next, there are overlaps such as 3.3-8, 3.4-9, 3.5-10, 3.6-11, 3.1-6, and 3.2-7 where one overlap wholly consumes the other by either starting before or at the same sequence number as the other and ending after or at the same sequence number as the other. In this situation, the wholly consuming segment can form the complete content itself. But, if it has an invalid timestamp, and the receiving host honors the small incomplete segment with a valid timestamp, gaps in TCP sequence numbers are formed. Finally, overlap 3-5 is unique because both segments start and end at the same sequence number. If either one is invalid, the other forms a complete content.

Let's look at the results of the Round1, Case1 series of tests performed on an Ubuntu Linux 2.6 kernel host where the three-way handshake timestamp values are non-zero and in-order segment 1 has a valid timestamp.

```
3.3-8:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
1-4:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-rej,9-rej
3.4-9:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
3.5-10:  1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
3.6-11:  1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
3-5:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-3B
2-4:     1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-rej
3.1-6:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
3.2-7:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
```

Figure 7. Results of timestamp tests where three-way handshake and segment 1 have non-zero timestamp values

The first column on the left lists the segment overlap conditions. For instance, "3.3-8" represents segment 3.3 and 8 overlaps from the Sturges/Novak segment overlap model in Figure 6. Each of the nine timestamp tests and overlaps that the receiving Linux host favored follow the segment overlap number (1-, 2-, 3-, etc). A result of "3A" means that the original segment (3.3) is favored; a result of "3B" means that the subsequent segment (8) is favored. Finally, a result of "rej" indicates that either both segments 3A and 3B have old timestamps (test 7) or that the receiving host discards a segment that was vital in creating the entire HTTP request. When the receiving host discards it, a gap in TCP sequence numbers is created and the receiving host acknowledges only valid segments to that point.

Let's examine some of the results from overlapped segments 3.3 and 8. This particular overlap is an example of where segment 3.3 wholly consumes segment 8 because it starts before and ends after segment 8. Linux 2.6 favors segment 3.3 when we conducted our previous tests that had no timestamps. Test 1 for Round 1, Case 1 is where the timestamp values on the three-way handshake are 10000, in-order segment 1 has a valid timestamp of 11111, followed by segment 3A with a timestamp value of 22222 and segment 3B with a timestamp value of 33333, and finally segment 2 arrives with a valid timestamp value of 12345. All timestamps are valid and we need to resort to the overlap model to determine which of the two segments is favored. As we mentioned, Linux 2.6 favors the original segment when the original segment (3.3) begins before and ends after the subsequent segment (8). Our results show that the receiving host favors original segment 3A as we expect since all timestamps are valid.

Timestamp tests 2 through 6 manipulate the values of 3A and 3B to contain no timestamp, the same timestamp, or give 3B has old timestamp. In each of these cases, the receiving host still favors segment 3.3. But now, let's look at what happens in test 7 where timestamps on 3A and 3B are both 0. These are old timestamps compared to segment 1's timestamp of 11111. The receiving host essentially rejects them both. Test 8 is where segment 3B has an old timestamp so segment 3A or 3.3 in our overlap model has a valid timestamp and forms a complete content by itself. Finally, test 9 examines where the timestamp on segment 3A is old and segment 3B is valid. The receiving host favors Segment 3B, but since it alone creates gaps in the TCP sequence numbers, the receiving host never gets an entire HTTP request. The entire set of results for tests on the Ubuntu Linux 2.6 host is found in Appendix A.

In aggregate, for this series of tests, we observe that a Linux host favors the overlap it would if there were no timestamps on the segments for all tests except when a segment with an old timestamp exists. More generally for any operating system, we find anomalous behavior with a given round and case of tests and not with any given timestamp or overlap test within the particular series. For instance, in the results discussion that follows this section in Figure 8, we find unexpected behavior for Windows 2003 when the three-way handshake value is zero and there is an in-order segment 1 with a timestamp. The entire series of tests – all segment overlaps and all nine timestamp tests - share this same deviant behavior. We deduce from this that timestamp processing occurs first before overlap processing. It makes sense when you think about it – a receiving host must first evaluate an incoming segment to ensure that both the timestamp and TCP sequence numbers are valid before processing it.

While performing tests, we unearthed some unexpected differences between the results from individual segment overlap tests and the results from the segment overlap model. The explanation of differences is complex, long-winded, and best not discussed here. If you care to learn more, please reference the paper "Target-Based TCP Stream Reassembly", specifically the section "Model Versus Individual Overlap Test Differences".

Results

We ran these tests against some current operating systems that support the TCP timestamp options - Windows 2000, Windows 2003, AIX, MacOS/BSD, OpenBSD, FreeBSD, HPUX, Linux, and Solaris to evaluate target-based responses. We attempted to run timestamp tests against a Cisco device using SSH as the application protocol instead of HTTP, but not only did it not reflect a TCP timestamp option in the returned SYN/ACK, it would not even return any response if any segment after the server's SYN or ACK had a timestamp on it.

	Windows 2003	**Win2k-Server**	**Linux 2-6**	**Solaris**	**HPUX 11**
R1-C1 **3whs=0, in-order segment 1 w/ ts** Segments 3A/3B relative to segment 1	Unexpected behavior Ignores timestamps	Expected behavior Segment 1 baseline ts	TS not supported[1]	Expected behavior No timestamp quirk[2]	Unexpected behavior Ignores timestamps
R1-C2 **3whs!=0, in-order segment1 w/ts** Segments 3A/3B relative to segment 3-whs	Expected behavior 3-whs baseline ts	Expected behavior 3-whs baseline ts	Expected behavior 3-whs baseline ts	Expected behavior No timestamp quirk[2]	Unexpected behavior Ignores timestamps
R2-C1 **3whs=0, in-order segment 1 no ts** Target-based segment overlap policy	Expected behavior No baseline ts	Expected behavior No baseline ts	TS not supported[1]	Expected behavior No timestamp quirk[2]	Unexpected behavior Ignores timestamps
R2-C2 **3whs!=0, in-order segment 1 no ts** Segments 3A/3B relative to segment 3-whs	Expected behavior 3-whs baseline ts	Expected behavior 3-whs baseline ts	Expected behavior 3-whs baseline ts	Unexpected behavior Ignores timestamps	Unexpected behavior Ignores timestamps
R3-C1 **3whs=0, delayed segment 1 w/ ts** Target-based overlap until segment 1 arrives	Expected behavior No baseline ts	Expected behavior No baseline ts	TS not supported[1]	Expected behavior No timestamp quirk[2]	Unexpected behavior Ignores timestamps
R3-C2 **3whs!=0, delayed segment1 w/ts** Segments 3A/3B relative to segment 3-whs	Expected behavior 3-whs baseline ts	Expected behavior 3-whs baseline ts	Expected behavior 3-whs baseline ts	Expected behavior No timestamp quirk[2]	Unexpected behavior Ignores timestamps
R4-C1 **3whs=0, delayed segment 1 no ts** Target-based segment overlap policy	Expected behavior No baseline ts	Expected behavior No baseline ts	TS not supported[1]	Expected behavior No timestamp quirk[2]	Unexpected behavior Ignores timestamps
R4-C2 **3whs!=0, delayed segment 1 no ts** Segments 3A/3B relative to segment 3-whs	Expected behavior 3-whs baseline ts	Expected behavior 3-whs baseline ts	Expected behavior 3-whs baseline ts	Unexpected behavior Ignores some timestamps	Unexpected behavior Ignores timestamps

Figure 8. Timestamp results of all tested operating systems

[1] Linux 2.4/2.6 does not support a TCP timestamp option when the client TCP timestamp option = 0
[2] Solaris stops returning the TCP timestamp option if it receives a segment with no timestamp option

The first column lists the eight series of tests (R1-C1 is the abbreviation for Round 1, Case 1 etc.) conducted against each destination host. The expected behavior for the test case is listed

underneath – timestamps on segments 3A/3B should have a baseline timestamp from the three-way handshake segments, or from segment 1, or no baseline at all so we expect it to revert to favoring segment 3A or 3B based on the target operating system overlap policy instead of the timestamp.

Windows 2003 behaves as we expect when there is a non-zero timestamp value on the three-way handshake. It totally ignores old timestamps when the three-way handshake has zero timestamps. This is expected behavior when there is no timestamp on segment 1. But, we expect the receiving host to compare timestamps on segments 3A and 3B to the valid timestamp value on the segment 1 that arrived before them.

Windows 2000 Server, AIX, MacOS/BSD/OpenBSD/FreeBSD all respond identically. They all behave as expected. As we mentioned earlier, Linux 2.6 is atypical because it does not reflect the existence of the TCP timestamp option when the client sends a timestamp value of zero in the three-way handshake. Otherwise, it follows the expected behavior. Ironically, Linux 2.2 kernels reflect the use of the TCP timestamp option when they receive a SYN packet with a timestamp value of zero. Solaris has a quirk where it no longer honors or sends the TCP timestamp option if it receives a segment that does not have a timestamp on it. This behavior is present on all test suites, but this oddity alters the expected outcome only when the three-way handshake timestamp values are non-zero and segment 1 has no timestamp. Finally, HPUX 11 ignores the timestamps on any segment that arrives out of order. All of the tests we performed altered the timestamp values on segments with out-of-order TCP sequence numbers so the results appear as if the segments had valid timestamps.

Conclusions

It is apparent that various operating systems respond uniquely to uncommon and common combinations of TCP timestamp values. A savvy attacker who understands a particular target host's behavior can fabricate TCP timestamps to evade an IDS/IPS that is unaware of the subtleties of TCP timestamps. It is not enough for an IDS/IPS to be aware of the use of TCP timestamps, it must also know how a given target-host will react and then respond appropriately.

References:

[1]RFC 1323 – TCP Extensions for High Performance, V. Jacobson, R. Braden, D. Borman, 1992

Appendix A – Example of Results From Tests for Ubuntu 2.6

Ubuntu 2.6 Results

```
================================================================
Round 1:  Inline segment 1 with timestamp
----------------------------------------------------------------
          Zero timestamp on three-way handshake
----------------------------------------------------------------
3.3-8:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
1-4:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.4-9:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.5-10:  1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.6-11:  1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3-5:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
2-4:     1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3.1-6:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3.2-7:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
----------------------------------------------------------------
          Timestamp on three-way handshake
----------------------------------------------------------------
3.3-8:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
1-4:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-rej,9-rej
3.4-9:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
3.5-10:  1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
3.6-11:  1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
3-5:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-3B
2-4:     1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-rej
3.1-6:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
3.2-7:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B

================================================================
Round 2:  Inline segment 1 with NO timestamp
----------------------------------------------------------------
          Zero timestamp on three-way handshake
----------------------------------------------------------------
3.3-8:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
1-4:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.4-9:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.5-10:  1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.6-11:  1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3-5:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
2-4:     1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3.1-6:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3.2-7:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
----------------------------------------------------------------
          Timestamp on three-way handshake
----------------------------------------------------------------
3.3-8:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
1-4:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-rej,9-rej
3.4-9:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
3.5-10:  1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
3.6-11:  1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
3-5:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-3B
2-4:     1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-rej
3.1-6:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
3.2-7:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
```

```
================================================================
Round 3:  Delayed segment 1 with timestamp
----------------------------------------------------------------
          Zero timestamp on three-way handshake
----------------------------------------------------------------
3.3-8:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
1-4:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.4-9:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.5-10:  1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.6-11:  1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3-5:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
2-4:     1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3.1-6:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3.2-7:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
----------------------------------------------------------------
          Timestamp on three-way handshake
----------------------------------------------------------------
3.3-8:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
1-4:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-rej,9-rej
3.4-9:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
3.5-10:  1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
3.6-11:  1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
3-5:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-3B
2-4:     1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-rej
3.1-6:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
3.2-7:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B

================================================================
Round 4:  Delayed segment 1 with NO timestamp
----------------------------------------------------------------
          Zero timestamp on three-way handshake
----------------------------------------------------------------
3.3-8:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
1-4:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.4-9:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.5-10:  1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
3.6-11:  1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3-5:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-3A,8-3A,9-3A
2-4:     1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3.1-6:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
3.2-7:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-3B,8-3B,9-3B
----------------------------------------------------------------
          Timestamp on three-way handshake
----------------------------------------------------------------
3.3-8:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
1-4:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-rej,9-rej
3.4-9:   1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
3.5-10:  1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-rej
3.6-11:  1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
3-5:     1-3A,2-3A,3-3A,4-3A,5-3A,6-3A,7-rej,8-3A,9-3B
2-4:     1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-rej
3.1-6:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
3.2-7:   1-3B,2-3B,3-3B,4-3B,5-3B,6-3B,7-rej,8-rej,9-3B
```

3A – means that the original segment was favored
3B – means that the subsequent segment was favored
rej – means that the receiving host rejected the entire or partial segment 3A/3B and the whole request wasn’t received