

Target-Based TCP Stream Reassembly

Authors

Judy Novak
Steve Sturges

Revision 1.0
August 3, 2007

Prepared by:

Sourcefire, Incorporated
9770 Patuxent Woods Drive
Columbia, MD 21046

Introduction

In their landmark 1998 paper, “Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection,”¹ Thomas Ptacek and Timothy Newsham exposed some weaknesses in intrusion detection systems (IDS). The authors revealed that intrusion detection systems cannot be effective and accurate because they do not necessarily process, or perhaps, even *observe* network traffic exactly as the destination host that receives the message does. This flaw exists in several layers of evaluation and processing of the packets including the IP, TCP, and application layers. As an example of the problem, consider traffic that has overlapping TCP segments that are sent to a given host. Because different operating systems have unique methods of TCP segment reassembly, if an intrusion detection system uses a single “one size fits all” reassembly method, it may not reassemble and process the packets the same way the destination host does. An attack that successfully exploits these differences in TCP reassembly can cause the IDS or intrusion prevention system (IPS) to miss the malicious traffic and fail to alert or block.

Remarkably, years later, many of these problems exposed by Ptacek/Newsham still exist. One answer to the TCP reassembly dilemma is a savvy IDS/IPS that is aware of the operating system and applications on the hosts that the IDS/IPS protects. The term “target-based” has been coined to identify an intelligent IDS or IPS that is informed about hosts residing on the protected network and is capable of analyzing traffic sent to those hosts as the host itself analyzes the traffic. This does not solve all of the problems discussed by Ptacek and Newsham, but it certainly improves the accuracy of the IDS/IPS. This can eliminate false positives about irrelevant alerts such as a Windows exploit bound for a Unix host. As well, deliberately mangled packets do not dupe the IDS/IPS, since it processes those packets exactly as the receiving host does.

Similarly, overlapping IP fragments present a problem for an IDS/IPS because the IDS/IPS may not examine or honor the same IP fragment as the destination host. Yet, an administrator may be able to configure an IPS to drop any overlapping IP fragments. Realistically, this should pose no problems for legitimate traffic. Wholly overlapping TCP segments, however, may appear in normal traffic as a retransmission of unacknowledged data. What about blocking only partial overlaps since they may be an evasion attempt? Even this is not foolproof since a host that uses the TCP option selective acknowledgements may “repacketize” multiple contiguous missing packets when retransmitting them. This may appear as a partial overlap if examined purely in terms of starting and ending TCP sequence numbers. This means that it is imperative for any IDS/IPS to reassemble overlapping TCP segments using the same policy as the destination host; otherwise, evasion is trivial.

The open source IDS/IPS Snort has begun to implement target-based analysis with the stream5 and frag3 preprocessors. Stream5 is able to reassemble overlapping TCP segments using the same policy as the destination host. A user configures Snort to apply specific TCP reassembly policies for individual hosts or networks. Then, when Snort sees overlapping TCP segments bound for any of these hosts, it knows the appropriate reassembly policy to apply—allowing both Snort and the destination host to reassemble the segments identically. This successfully precludes evasion attacks that use overlapping TCP segments.

This paper discusses TCP overlapping segment attacks, a model for identifying TCP reassembly policies, and a method and code used to determine a given host’s TCP reassembly policy.

A Simple TCP Overlapping Segment Attack

Suppose that an IDS has a signature that alerts on an attempted buffer overflow attack that occurs when an attacker supplies an overly long username value for FTP authentication. For example, Snort has the following rule to alert on such an attack:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP USER overflow attempt";  
flow:to_server,established,no_stream; content:"USER"; nocase;  
isdataat:100,relative; pcre:"/^USER\s{100}/smi";)
```

This rule looks for traffic that originates outside of and bound for the protected network to the FTP command port. The alert fires when an established stream to port 21 contains a payload that begins with the content of “USER” with no linefeed character in the following 100 bytes.

Further, suppose an attacker wants to evade the IDS by sending wholly overlapping segments with TCP headers containing the identical beginning TCP sequence number, but whose payload contents differ by a single character. The original TCP segment contains a content of “USER”; the overlapping segment contains the content of “XSER”. TCP/IP implementations elect to regard either the original or subsequent segment as the valid one and discard the other. Windows hosts use a policy of accepting the first segment and ignoring all wholly overlapping subsequent ones. An IDS that considers the subsequent segment the valid one will miss an attack that targets a Windows host because it sees the content of “XSER” instead of USER. This is a simple and tidy illustration of overlapping TCP segments; thorough analysis must consider many more complications such as partially overlapping TCP segments and the location of the overlap of the segments. Vern Paxson and Umesh Shankar examined this problem superficially in the paper titled “Active Mapping: Resisting NIDS Evasion Without Altering Traffic”². Our research has uncovered that there are many more reassembly policies than they posited.

Sturges/Novak Model

In the paper “Target-Based Fragmentation Reassembly,”³ Sourcefire employees, Steve Sturges and Judy Novak discuss a paradigm to use for overlapping IP fragmentation reassembly based on the work of Paxson and Shankar. This model accounts for all possibilities of IP fragment overlap. This same model is used in TCP reassembly. Use of this model exposes many different TCP reassembly methods that modern operating systems employ.

This model consists of a series of many different TCP segments of varying offsets, content, and length. Refer to Table1 and Figure 1 for the following explanation.

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

Table 1 – Segment Overlap Possibilities

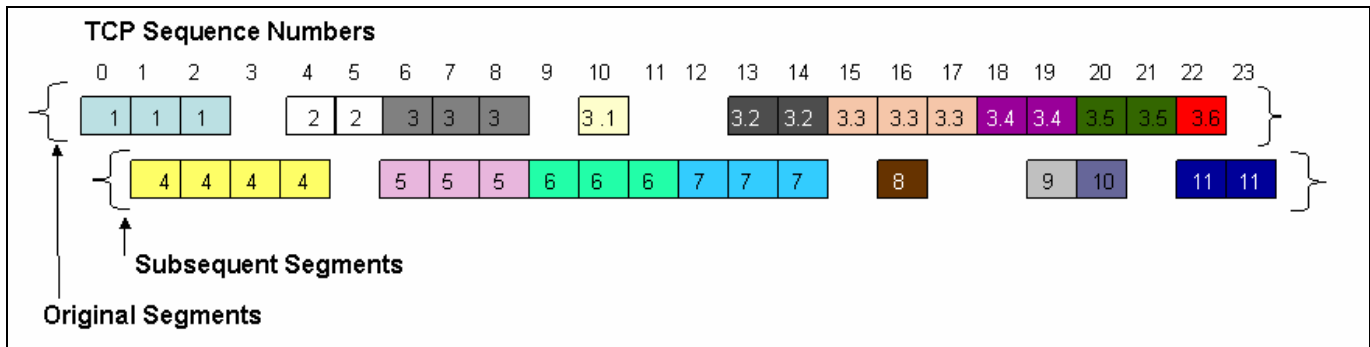


Figure 1 – Sturges/Novak Model

Let's define some terms before proceeding. The term original segment is one that is sent chronologically before a subsequent one. A subsequent segment is one that is sent after an original segment and overlaps an original segment. The label of original or subsequent refers to the chronology in time when a segment is sent. It has nothing to do with relative TCP sequence numbers. The relative TCP sequence numbers are considered in an individual pair of overlapping segments in comparison to where each begins and ends.

Look at the left column in Table 1. Each row contains a different TCP segment overlap configuration. The next column considers segment overlaps from the perspective of the original segment versus the subsequent segment. It lists the segment numbers/labels (the first segment listed is an original segment, the second a subsequent

segment) from the model in Figure 1 that match the row's overlap conditions. For instance, the first configuration is "Starts before, ends after" and contains original segment 3.3 and subsequent segment 8. Now, examine Figure 1 for the original segment labeled 3.3 that begins at relative TCP sequence number 15 and ends at relative sequence number 17 (3-byte payload). Look for the subsequent segment labeled 8 to see that it begins at relative TCP sequence number 16 (1-byte payload) and overlaps original segments 3.3. As you can see, original segment 3.3 begins before and ends after subsequent segment 8.

Figure 1 illustrates the Sturges/Novak model as a series of seventeen uniquely colored TCP segments, labeled 1 through 3, 3.1 through 3.6, and 4 through 11. A segment's number indicates its arrival order. The use of decimal numbers 3.1 through 3.6 is a carryover from the fragmentation paradigm that relies on an expanded Paxson/Shankar model. Original segments 1, 2, 3, 3.1, 3.2, 3.3, 3.4, 3.5 and 3.6 are in the first row with relative TCP sequence numbers ranging from 0 through 22. These are chronologically first. Next, we send subsequent segments 4-11 that overlap the original segments with relative TCP sequence numbers ranging from 1 through 23.

Segment block1 has a beginning TCP sequence number from which all other segments are computed. Segment block 2 has a beginning TCP sequence number 4 more than the beginning sequence number of segment 1; segment block 3 has a beginning TCP sequence number 6 more than the beginning sequence number of segment 1, etc. The receiving host uses the TCP sequence number to properly reorder the segments during reassembly.

Segments may wholly or partially overlap. In the model, segment 5 wholly overlaps segment 3. They both start at relative TCP sequence number 6 and are 3 bytes long. They differ only in content, which we use to determine which segment a particular receiving host's operating system favors. Most segments partially overlap; that is, they do not begin and at the same relative TCP sequence numbers.

The model also requires one or more subsequent segments that overlap original segments and start after one original segment and start before another. Segment 4 is unique because it serves a dual purpose by overlapping two original segments as follows:

- Overlaps both original segments 1 and 2
- Has a beginning relative sequence number of 1, which
 - Is larger than segment 1's beginning relative sequence number of 0 – starts after
 - Is smaller than segment 2's beginning relative sequence number of 4 – starts before

A receiving operating system selects between overlapping segments based on two factors – chronology and where the segments overlap. Chronology simply refers to whether the segment is an original or subsequent one. As mentioned previously, segments may overlap by starting or ending before, after, or the same as the overlapping segment. For instance, Windows favors an original segment, except when the subsequent segment begins before the original segment.

Before discussing the reassembly policies, an important insight into host reassembly is that typically the receiving host favors an entire segment block over the overlapping one. In other words, it does not take some bytes from one segment and some from the other. However, there are exceptions. The first is a configuration like overlaps 3.1 and 6 where gaps in TCP sequence numbers exist if overlap 3.1 is used exclusively. If the receiving host favors segment 3.1, it must also take the first and third bytes from segment 6 to fill in the gaps. The second exception is that subsequent segment 4 overlaps two original segments – segments 1 and 2. A receiving may favor partial bytes from segment 4 to favor over either segment 1 or segment 2. Segment 4 is the only segment from the model that overlaps two different segments.

The different reassembly policies are:

- **Windows/BSD** favors the original segment, **EXCEPT** when the subsequent segment begins before the original segment.
- **First/Windows Vista** favors an original segment
- **Linux** favors an original segment, **EXCEPT** when the subsequent begins before the original, or the subsequent segment begins the same and ends after the original segment.
- **Solaris** favors the subsequent segment **EXCEPT** when the original segment ends after the subsequent segment, or begins before the original segment and ends the same or after the original segment.
- **Linux-old** favors the subsequent segment **EXCEPT** when the original segment begins before, or the original segment begins the same and ends after the subsequent segment.
- **Last** favors a subsequent segment

TCP segments from Figure 1 are reassembled as follows:

Windows/BSD:

<1><1><1><4><4><2><3><3><3><6><6><6><7><7><7><3.3><3.3><3.3><3.4><3.4><3.5><3.5><3.6><11>

First/Windows Vista:

<1><1><1><4><2><2><3><3><3><6><3.1><6><7><3.2><3.2><3.3><3.3><3.3><3.4><3.4><3.5><3.5><3.6><11>

Linux:

<1><1><1><4><4><2><3><3><3><6><6><6><7><7><7><3.3><3.3><3.3><3.4><3.4><3.5><3.5><11><11>

Solaris:

<1><4><4><4><2><2><5><5><5><6><6><6><7><7><7><3.3><3.3><3.3><3.4><3.4><3.5><3.5><11><11>

Linux-old:

<1><1><1><4><4><2><5><5><5><6><6><6><7><7><7><3.3><3.3><3.3><3.4><3.4><3.5><3.5><11><11>

Last:

<1><4><4><4><4><2><5><5><5><6><6><6><7><7><7><3.3><8><3.3><3.4><9><10><3.5><11><11>

To better understand how reassembly is performed, let's examine a particular policy, Windows/BSD. Windows/BSD favors the original segment, except when the subsequent segment begins before the original segment.

When...	Overlaps...	This Segment is favored...	Because...
subsequent segment 4	original segment 1	original segment 1	segment 1's relative sequence number 0 is less than segment 4's relative sequence number 1
	original segment 2	subsequent segment 4	segment 4's relative sequence number 1 is less than segment 2's relative sequence number 4
subsequent segment 5	original segment 3	original segment 3	segment 3's relative sequence number is equal to segment 5's relative sequence number
subsequent segment 6	original segment 3.1	subsequent segment 6	segment 6's relative sequence number 9 is less than segment 3.1's relative sequence number 10
subsequent segment 7	original segment 3.2	subsequent segment 7	segment 7's relative sequence number 12 is less than segment 3.2's relative sequence number 13
subsequent segment 8	original segment 3.3	original segment 3.3	segment 3.3's relative sequence number 15 is less than segment 8's relative sequence number 16
subsequent segment 9	original segment 3.4	original segment 3.4	segment 3.4's relative sequence number 18 is less than segment 9's relative sequence number 19
subsequent segment 10	original segment 3.5	original segment 3.5	segment 3.5's relative sequence number is equal to segment 10's relative sequence number
subsequent segment 11	original segment 3.6	original segment 3.6	segment 3.6's relative sequence number is equal to segment 11's relative sequence number

Table 2. Windows/BSD Reassembly Policy Explanation

Implementation of the Sturges/Novak Model

Implementation of the Sturges/Novak model requires an API that can send network traffic. The Python API *scapy*, written by Philippe Bionde is an excellent tool to craft packets. While this task is fairly trivial, the problem arises when you attempt to assess how a given host responds to overlapping TCP segments. The easiest way to accomplish this is to install netcat on any host where you have root or administrator access. Netcat is known as the “Swiss army knife” since it is a multi-purpose tool. Netcat can listen for data at a given port and echo the output to the screen or redirect it to a file. This is precisely what we need to show TCP reassembly. Implementation simply requires fashioning overlapping TCP segments based on the model with appropriate TCP sequence numbers and unique content for each TCP segment block.

Let’s see exactly how this works on some actual traffic. The model segments are given unique payloads.

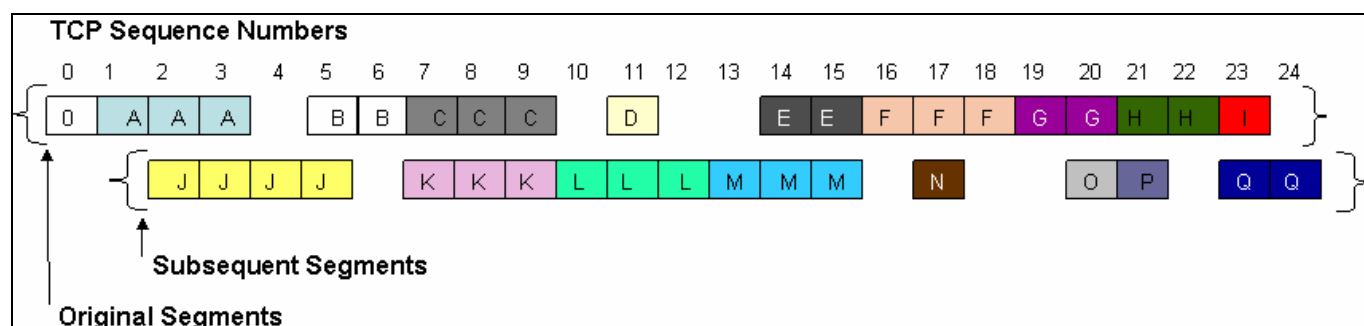


Figure 2 – Model with Payload

Perhaps you’ve noticed that we’ve added a “byte 0” in the Figure 2 model. This is not actually part of the model; “byte 0” is sent after all the other segments to delay the receiving host from reassembling the overlapping segments as they arrive. If the receiving host reassembles segments as they arrive, it will always honor the first one that arrives and never wait for or buffer the overlap. We force it to wait since there is a “missing” segment – byte 0 – that must arrive before the receiving host performs stream reassembly.

We direct the segment overlaps from Figure 2 to a Windows 2000 host that has a netcat listener on port 80. netcat shows how it reassembled the segments.

```

C:\>nc -l -p 80
0AAAJJBBCCCLLLMMMFFFGGHHIIQ_

```

According to the Sturges/Novak model, a host using the “Windows/BSD” policy reassembles the packets using the following scheme; the shaded numbers are the favored segments:

1	1	1	2	2	3	3	3	3.1	3.2	3.2	3.3	3.3	3.3	3.4	3.4	3.5	3.5	3.6		
	4	4	4	4	5	5	5	6	6	6	7	7	7	8	9	10	11	11		

<A><A><A><J><J><C><C><C><L><L><L><M><M><M><F><F><F><G><G><H><H><I><Q>

Translated, this means that the “Windows/BSD” policy uses all three bytes from segment 1, the second two bytes from segment 4, the second byte from segment 2, all bytes from segments 3, 6, 7, 3.3, 3.4, 3.5, 3.6 and the second byte from segment 11 for the reassembly process. Code for this example is supplied in Appendix A.

What about performing TCP reassembly on a remote host where you have no command line access and most likely is not running netcat for your convenience? It is still possible to identify the TCP reassembly policy as long as there is a listening TCP port and some response from the application running on that port that distinguishes between different data sent. As an example, take an HTTP web server. It is pretty obvious when you send expected and acceptable input because you receive a positive response in the form of some web page data. Conversely, when you send some type of unacceptable input, it responds with negative feedback in the form of an error message. Paxson and Shankar identified TCP reassembly policies using SSH. The only limitation on the service or protocol that you use for reassembly identification is that altered payload that you send must elicit a distinct response. In other words, you must be able to tell from the server's response whether it favors an original or subsequent segment.

One surefire way to receive an HTTP error is by sending a listening web server an incorrect version of HTTP. This version is supplied after the GET statement and the URL name. Typically, version of 1.1 or less is acceptable. Most web servers consider a version of 4.0 invalid. Therefore, if we can craft several different segments that reproduce the overlap conditions in the Sturges/Novak model, we can learn the TCP reassembly policy of the host's operating system running the web server. Unlike the netcat implementation where we could implement the model with one TCP session, we must now craft each of the overlap conditions one by one by starting a new session for an individual overlap pair, and recording the results of each with tcpdump. The problem with using the entire model to craft an HTTP request is that if a web server returns an error, you don't know which of the favored overlapping segments caused the error. Testing that employs a single pair of segment overlaps per HTTP request is certainly more tedious, but it reveals the reassembly policy of the receiving operating system nonetheless. Also, it is possible that the results of testing a pair of overlaps at a time may differ from the results of sending the entire model if we were able to view the model results using netcat. In fact, we did discover some differences in particular operating systems that we discuss later.

Once again, recall the Table 1 that details all the possible overlap conditions. Table 3 below embellishes this by including the segment payloads and relative TCP sequence numbers. These overlapping segments contain payloads that together form a web request – correct or incorrect - depending upon how the receiving web server reassembles them. The web server response indicates to us the segment that it favors.

Segment overlap	Segment Value	Relative TCP Sequence Number
Starts before, ends after	<u>Segments 3.3 & 8</u> seg 3.3 = /index.html HTTP/1.0\r\n\r\n seg 8 = 4	<u>Segments 3.3 & 8</u> seg 3.3 = 4 seg 8 = 22
Starts before, ends before	<u>Segments 1 & 4</u> seg 1 = /index.html HTTP/1.0 seg 4 = index.html HTTP/4.0\r\n\r\n	<u>Segments 1 & 4</u> seg 1 = 4 seg 4 = 5
Starts before, ends same	<u>Segments 3.4 & 9</u> seg 3.4 = /index.html HTTP/1.0\r\n\r\n seg 9 = 4.0\r\n\r\n	<u>Segments 3.4 & 9</u> seg 3.4 = 4 seg 9 = 22
Starts same, ends after	<u>Segments 3.5 & 10</u> seg 3.5 = /index.html HTTP/1.0\r\n\r\n seg 10 = /index.html HTTP/4.0	<u>Segments 3.5 & 10</u> seg 3.5 = 4 seg 10 = 4
Starts same, ends before	<u>Segments 3.6 & 11</u> seg 3.6 = /index.html HTTP/1.0 seg 11 = /index.html HTTP/4.0\r\n\r\n	<u>Segments 3.6 & 11</u> seg 3.6 = 4 seg 11 = 4
Starts same, ends same	<u>Segments 3 & 5</u> seg 3 = /index.html HTTP/1.0\r\n\r\n seg 5 = /index.html HTTP/4.0\r\n\r\n	<u>Segments 3 & 5</u> seg 3 = 4 seg 5 = 4
Starts after, ends after	<u>Segments 2 & 4</u> seg 2 = index.html HTTP/1.0\r\n\r\n seg 4 = /index.html HTTP/4.0	<u>Segments 2 & 4</u> seg 2 = 5 seg 4 = 4
Starts after, ends before	<u>Segments 3.1 & 6</u> seg 3.1 = 1 seg 6 = /index.html HTTP/4.0\r\n\r\n	<u>Segments 3.1 & 6</u> seg 3.1 = 22 seg 6 = 4
Starts after, ends same	<u>Segments 3.2 & 7</u> seg 3.2 = index.html HTTP/1.0\r\n\r\n seg 7 = /index.html HTTP/4.0\r\n\r\n	<u>Segments 3.2 & 7</u> seg 3.2 = 5 seg 7 = 4

Table 3 – Segment Overlap Test Using HTTP Request

Let's look at one particular overlap condition using wholly overlapping segments 3 and 5 where we test whether the receiving host favors original segment 3 over subsequent segment 5. As Table 3 indicates, segment 3 has a payload of "/index.html HTTP/1.0\r\n\r\n" and segment 5 has a payload of "/index.html HTTP/4.0\r\n\r\n". Both of these segments start at the same relative TCP sequence number 4 and are the same length. In this and all other overlap cases, a static segment of "GET " is the true first segment in TCP sequence number order. But, it is sent last to postpone the receiving host's reassembly. The stream is sent as follows:

"index.html HTTP/1.0\r\n\r\n" (segment 3)

"index.html HTTP/4.0\r\n\r\n" (segment 5)

"GET "

We expect to see a positive response if the web server honors original segment 3 is or an error message if, instead it favors subsequent segment 5. Finally, while the Sturges/Novak model has specific segment sizes to implement the test cases, any segment size can be used so long as it conforms to the overlap conditions detailed in the original table.

Model Versus Individual Overlap Test Differences

While performing individual overlap tests, we discovered Windows has some anomalies that are worth noting. First, a Windows IIS web server does not validate the HTTP version number so this cannot be used as a differentiator. It will not generate an error message for HTTP version 4.0. The example that follows uses HTTP version manipulation to resemble the other examples used, solely for ease of understanding. In reality, if you were to attempt to send actual traffic to an IIS server, you would have to manipulate the URL request (e.g. /iisstart.htm) instead.

Also, Windows implementations prior to Vista have a quirk in two special test cases involving overlapping segments 3.1 and 6 and overlapping segments 3.2 and 7. According to the results from the entire model shown on page 6, Windows implementations prior to Vista favor a subsequent segment – in this case segment 6 or segment 7 – when it begins before the original segment. However, in our individual tests of these overlaps, we found that Windows favors the original segment.

Look at the top half of Figure 3 below to see the overlapping segment payloads that caused Windows to favor original segment 3.1 over subsequent segment 6. What is the difference between this and the entire model? The entire model has a left hand “anchor” for subsequent segment 6. The “anchor” is a segment that immediately precedes (its ending TCP sequence number must be one less than the starting TCP sequence number of segment 6) segment 6 and it must arrive at the receiving host before segment 6.

Let’s try to simulate the model by including an “anchor” segment for segment overlaps 3.1 and 6. Look at the bottom half of Figure 3 to see that we’ve added a subsequent segment that precedes segment 6 with a payload of “/index.html HTTP” followed by a segment 6 payload of “/4.0\r\n\r\n”. Windows favors segment 6 as expected from the model behavior. The content in the overlapping segments still conforms to the “starts after, ends before” condition for these segments.

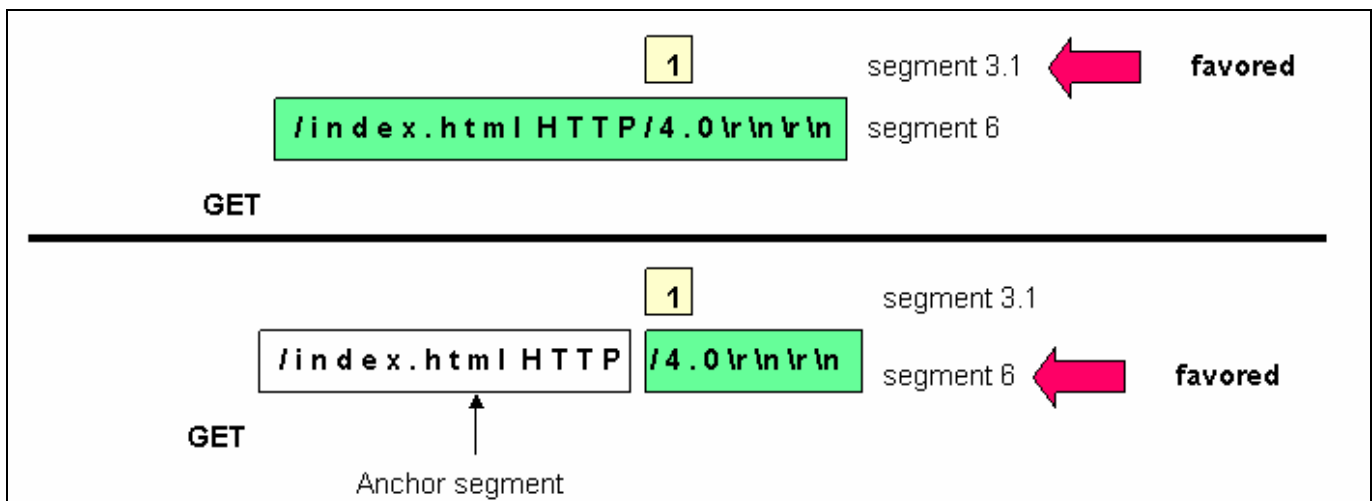


Figure 3– Windows Anchor Quirk

Solaris yields one difference when tested using the entire model in a single stream versus multiple streams, each with a single different overlap pair from the model. The difference occurs in wholly overlapping segments 3 and 5

only. Solaris favors segment 3 in the individual segment pair overlaps but favors segment 5 in the full model test. Let's attempt to recreate the two different outcomes in Figure 4.

The top half of Figure 4 shows the results for the individual test of overlapping segments 3 and 5. Unexpectedly, the receiving Solaris host favors segment 3. In this case, segment 3 immediately follows in TCP sequence number chronology the delayed segment. Our delayed segment is "GET ". Segment 3 content of "/index.html" follows with the next TCP sequence number and is the first to arrive of the overlapping segments.

But, if any segment(s) separates the delayed segment ("GET ") and the original segment ("/index.html HTTP/1.0\r\n\r\n"), then Solaris favors the subsequent segment. Look at the bottom half of Figure 4 to see the insertion of a new segment with the content of "/index.html HTTP" that has the next TCP sequence number after the delayed segment with the content of "GET ". Segments 3 and 5 still wholly overlap, but now segment 3 does not immediately follow the delayed segment causing Solaris to favor segment 5.

The model has segments that are between the delayed segments and segment 3. The model does not show a segment "0" that has the first TCP sequence number, but arrives last to force the receiving host's reassembly to wait for the arrival of all segments. Therefore, model segments 1 and 2 abut segment 0 in terms of sequence numbers and arrive before overlapping segment 3. In this case, as in the example shown on the bottom of Figure 4, Solaris favors subsequent segment 5.

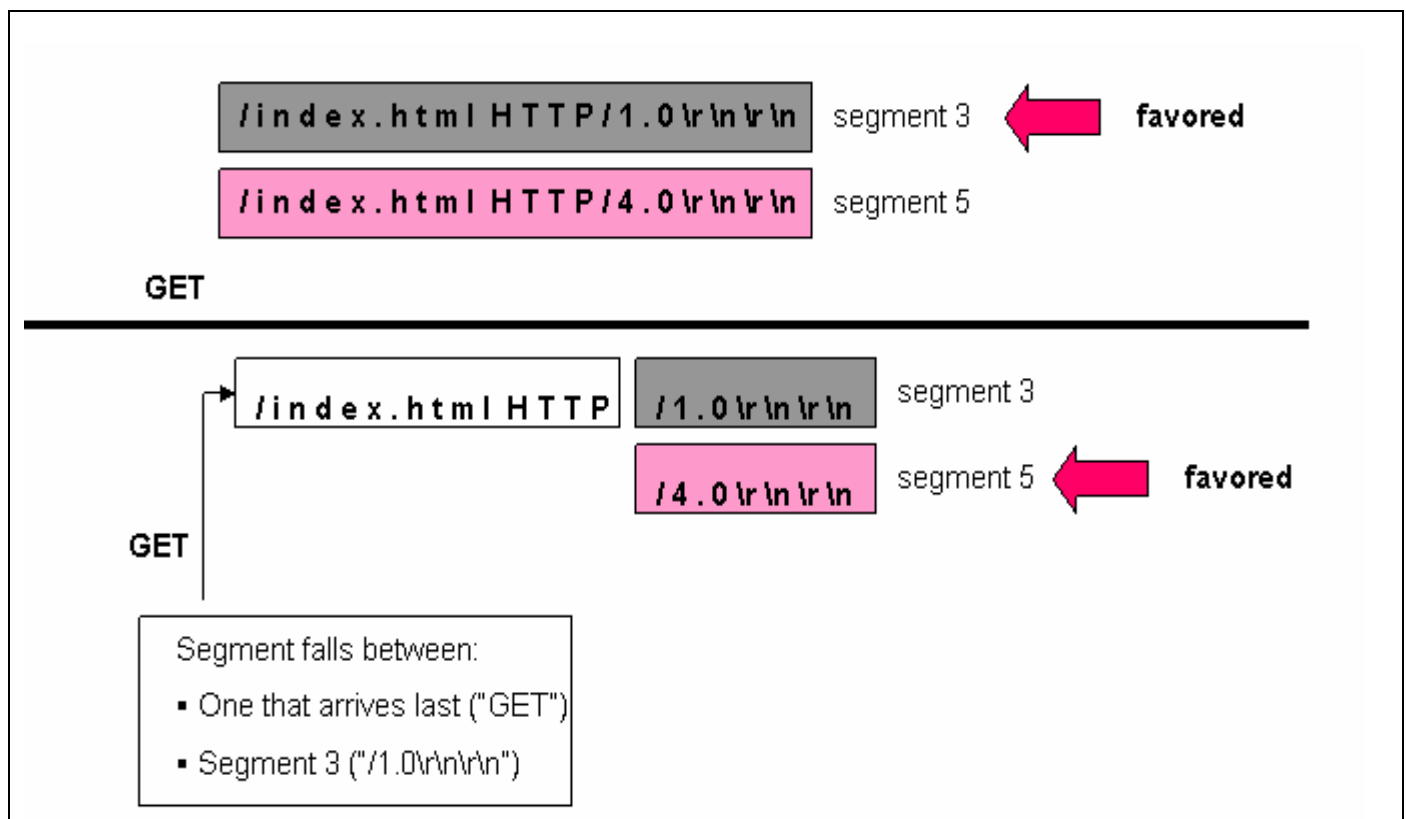


Figure 4 – Solaris Reassembly Quirk

Code to implement individual segment overlaps is found in Appendix B.

Snort stream5 Preprocessor to the Rescue

Steve Sturges, a member of the Sourcefire Snort team, has written an improved TCP stream preprocessor, stream5 that implements target-based TCP stream reassembly policies by allowing a user to identify the TCP reassembly method that is applied to a particular destination IP address or subnet. Assume that you have an entire subnet (10.4.10.x, for example) that consists only of Microsoft Windows hosts and you want to configure Snort to use the appropriate reassembly policy for these hosts. You need only enable the stream5 preprocessor in the configuration file and designate the appropriate stream policy in a stream5_tcp statement as follows:

```
preprocessor stream5_global:  
preprocessor stream5_tcp: policy windows, bind_to 10.4.10.0/24
```

This is fairly straightforward; the only unknown for you as a user is the association of a stream policy, in this case “windows” with an operating system. Table 4 shows the correlation between stream5 policy and operating system.

Stream5 Policy	Platforms
windows	Window OSes prior to Vista
linux	Linux 2.4/HP Laserjet/Cisco IOS
bsd	BSD/FreeBSD/OpenBSD/AIX/OpenVMS/HPUX 10.2
solaris	Solaris 9/10 HP/UX 11
linux-old	Linux-2.2
mac-os	macos
first	Windows Vista
last	No identified OS

Table 4. Stream5 Policies and Associated Operating Systems

Now any overlapping segments that Snort sees destined for subnet 10.4.10.x are reassembled using the “windows” TCP reassembly policy, so that Snort reassembles segments destined to those hosts in precisely the same way as Windows hosts themselves. Note that we have found no operating system or device that uses the “last” policy, but have chosen to include it in case one is discovered that uses a “last” reassembly policy. The “macos” reassembly policy is identical to the “bsd” policy, but a MacOS host accepts data on SYN packets and requires this unique policy to deal with this.

Stream5 target-based reassembly uniquely empowers Snort to handle attempted evasion tactics using overlapping TCP segments.

Summary

This paper discusses target-based TCP reassembly in theory and in application using the Snort stream5 preprocessor. As Ptacek and Newsham noted, an intrusion detection system that is not aware of the reassembly policy used by a destination host cannot possibly know how to perform TCP reassembly appropriately. And, while a firewall or IPS may discard overlapping IP fragments since they are probably either malicious or a by-product of some malfunctioning software or hardware, overlapping TCP segments may reflect normal retransmissions of unacknowledged TCP traffic and a firewall or IPS should not block them.

There is no way for an IDS or IPS to discern whether overlapping TCP segments are normal or malicious unless it is able to apply the appropriate TCP reassembly policy to the traffic based on the destination host's operating system. This makes Snort unique for its ability to perform TCP reassembly properly and prevent evasion attacks that use overlapping TCP segments.

References

- ¹ Thomas H. Ptacek and Timothy N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection", January 1998.
- ² Umesh Shankar and Vern Paxson, "Active Mapping: Resisting NIDS Evasion Without Altering Traffic", 2003.
- ³ Judy Novak, "Target-based Fragmentation Reassembly", 2003.

Appendices

Appendix A – Program to identify TCP reassembly policy when netcat is running on the remote host

```
#-----
#
# Copyright (C) 2007 Judy Novak
#**
#** This program is free software; you can redistribute it and/or modify
#** it under the terms of the GNU General Public License Version 2 as
#** published by the Free Software Foundation. You may not use, modify or
#** distribute this program under any other version of the GNU General
#** Public License.
#**
#** This program is distributed in the hope that it will be useful,
#** but WITHOUT ANY WARRANTY; without even the implied warranty of
#** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
#** GNU General Public License for more details.
#**
#** You should have received a copy of the GNU General Public License
#** along with this program; if not, write to the Free Software
#** Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
#
#-----

#!/usr/bin/python

def segment(dip):

    sp = random.randint(1,65535)
    dp = 80
    ISN = 10

    P0   = "0"
    P1   = "A"
    P2   = "B"
    P3   = "C"
    P3_1 = "D"
    P3_2 = "E"
    P3_3 = "F"
    P3_4 = "G"
    P3_5 = "H"
    P3_6 = "I"
    P4   = "J"
    P5   = "K"
    P6   = "L"
    P7   = "M"
    P8   = "N"
    P9   = "O"
    P10  = "P"
    P11  = "Q"

    p1_seqplus = 1
    p2_seqplus = 5
    p3_seqplus = 7
    p3_1_seqplus = 11
    p3_2_seqplus = 14
    p3_3_seqplus = 16
    p3_4_seqplus = 19
    p3_5_seqplus = 21
    p3_6_seqplus = 23
    p4_seqplus = 2
    p5_seqplus = 7
```

```
p6_seqplus = 10
p7_seqplus = 13
p8_seqplus = 17
p9_seqplus = 20
p10_seqplus = 21
p11_seqplus = 23

seg0 = P0
seg1 = P1 * 3
seg2 = P2 * 2
seg3 = P3 * 3
seg3_1 = P3_1
seg3_2 = P3_2 * 2
seg3_3 = P3_3 * 3
seg3_4 = P3_4 * 2
seg3_5 = P3_5 * 2
seg3_6 = P3_6
seg4 = P4 * 4
seg5 = P5 * 3
seg6 = P6 * 3
seg7 = P7 * 3
seg8 = P8
seg9 = P9
seg10 = P10
seg11 = P11 * 2

ip=IP(src=sip, dst=dip)
tcp=TCP(dport=dp, sport=sp, flags="S", seq=ISN, window=5480, options=[('MSS', 1460)])
syn=ip/tcp
synack=srl(syn)

time.sleep(1)
myack=synack.seq + 1
tcpseq=ISN+1
tcp=TCP(ack=myack, dport=dp, sport=sp, flags="A", seq=tcpseq, window=5480)
ackit=ip/tcp
send(ackit)

time.sleep(1)
tcp=TCP(ack=myack, dport=dp, sport=sp, flags="PA", window=5480)
tcp.seq= tcpseq + p1_seqplus
pack1=ip/tcp/seg1
send(pack1)

time.sleep(1)
tcp.seq= tcpseq + p2_seqplus
pack2=ip/tcp/seg2
send(pack2)

time.sleep(1)
tcp.seq= tcpseq + p3_seqplus
pack3=ip/tcp/seg3
send(pack3)

time.sleep(1)
tcp.seq= tcpseq + p3_1_seqplus
pack3_1=ip/tcp/seg3_1
send(pack3_1)

time.sleep(1)
tcp.seq= tcpseq + p3_2_seqplus
pack3_2=ip/tcp/seg3_2
send(pack3_2)

time.sleep(1)
tcp.seq= tcpseq + p3_3_seqplus
pack3_3=ip/tcp/seg3_3
send(pack3_3)
```

```
time.sleep(1)
tcp.seq= tcpseq + p3_4_seqplus
pack3_4=ip/tcp/seg3_4
send(pack3_4)

time.sleep(1)
tcp.seq= tcpseq + p3_5_seqplus
pack3_5=ip/tcp/seg3_5
send(pack3_5)

time.sleep(1)
tcp.seq= tcpseq + p3_6_seqplus
pack3_6=ip/tcp/seg3_6
send(pack3_6)

time.sleep(1)
tcp.seq= tcpseq + p4_seqplus
pack4=ip/tcp/seg4
send(pack4)

time.sleep(1)
tcp.seq= tcpseq + p5_seqplus
pack5=ip/tcp/seg5
send(pack5)

time.sleep(1)
tcp.seq= tcpseq + p6_seqplus
pack6=ip/tcp/seg6
send(pack6)

time.sleep(1)
tcp.seq= tcpseq + p7_seqplus
pack7=ip/tcp/seg7
send(pack7)

time.sleep(1)
tcp.seq= tcpseq + p8_seqplus
pack8=ip/tcp/seg8
send(pack8)

time.sleep(1)
tcp.seq= tcpseq + p9_seqplus
pack9=ip/tcp/seg9
send(pack9)

time.sleep(1)
tcp.seq= tcpseq + p10_seqplus
pack10=ip/tcp/seg10
send(pack10)

time.sleep(1)
tcp.seq= tcpseq + p11_seqplus
pack11=ip/tcp/seg11
send(pack11)

time.sleep(1)
tcp.seq= tcpseq
pack0=ip/tcp/seg0
send(pack0)

import random, time, sys
from scapy import IP, TCP, Ether, Raw, send, srl

dip = sys.argv[1]
sip = sys.argv[2]

segment(dip)
```

Appendix B – Program to identify TCP stream reassembly on a remote host running HTTP

```
#-----
#
# Copyright (C) 2007 Judy Novak
#**
#** This program is free software; you can redistribute it and/or modify
#** it under the terms of the GNU General Public License Version 2 as
#** published by the Free Software Foundation. You may not use, modify or
#** distribute this program under any other version of the GNU General
#** Public License.
#**
#** This program is distributed in the hope that it will be useful,
#** but WITHOUT ANY WARRANTY; without even the implied warranty of
#** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
#** GNU General Public License for more details.
#**
#** You should have received a copy of the GNU General Public License
#** along with this program; if not, write to the Free Software
#** Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
#
#-----

#!/usr/bin/python

def segment(dip, seg2, seg3, seq_seg2, seqplus_seg3):

    sp = random.randint(1,65535)
    dp = 80
    ISN = 10

    seg1 = "GET "

    ip=IP(src=sip, dst=dip)
    tcp=TCP(dport=dp, sport=sp, flags="S", seq=ISN, window=5480, options=[('MSS',1460)])
    syn=ip/tcp
    synack=sr1(syn)

    time.sleep(1)
    myack=synack.seq + 1
    tcpseq=ISN+1
    tcp=TCP(ack=myack, dport=dp, sport=sp, flags="A", seq=tcpseq, window=5480)
    ackit=ip/tcp
    print "Sending ACK"
    send(ackit)

    time.sleep(1)
    tcp=TCP(ack=myack, dport=dp, sport=sp, flags="PA", window=5480)
    tcp.seq = tcpseq + seqplus_seg2
    pack2=ip/tcp/seg2
    send(pack2)

    time.sleep(1)
    tcp.seq = tcpseq + seqplus_seg3
    pack3=ip/tcp/seg3
    send(pack3)

    time.sleep(1)
    tcp.seq= tcpseq
    pack1=ip/tcp/seg1
    send(pack1)

    tcp.seq = tcpseq + len(seg1)
    time.sleep(1)
    tcp=TCP(ack=myack, dport=dp, sport=sp, flags="R", window=5480)
```

```
fin=ip/tcp
send(fin)

import random, time, sys
from scapy import IP, TCP, Ether, Raw, send, srl

dip = sys.argv[1]
sip = sys.argv[2]

# test 1/4
seg2 = "/index.html HTTP/1.0"
seg3 = "index.html HTTP/4.0\r\n\r\n"
seqplus_seg2 = 4
seqplus_seg3 = 5
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)
# test 4/1
seg2 = "/index.html HTTP/4.0"
seg3 = "index.html HTTP/1.0\r\n\r\n"
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)

# test 2/4
seg2 = "index.html HTTP/1.0\r\n\r\n"
seg3 = "/index.html HTTP/4.0"
seqplus_seg2 = 5
seqplus_seg3 = 4
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)
# test 4/2
seg2 = "index.html HTTP/4.0\r\n\r\n"
seg3 = "/index.html HTTP/1.0"
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)

# test 3/5
seg2 = "/index.html HTTP/1.0\r\n\r\n"
seg3 = "/index.html HTTP/4.1\r\n\r\n"
seqplus_seg2 = 4
seqplus_seg3 = 4
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)
# test 5/3
seg2 = "/index.html HTTP/4.1\r\n\r\n"
seg3 = "/index.html HTTP/1.0\r\n\r\n"
seqplus_seg2 = 4
seqplus_seg3 = 4
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)

# test 3.1/6
seg2 = "1"
seg3 = "/index.html HTTP/4.0\r\n\r\n"
seqplus_seg2 = 22
seqplus_seg3 = 4
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)
# test 6/3.1
seg2 = "4"
seg3 = "/index.html HTTP/1.0\r\n\r\n"
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)

# test 3.2/7
seg2 = "index.html HTTP/1.0\r\n\r\n"
seg3 = "/index.html HTTP/4.0\r\n\r\n"
seqplus_seg2 = 5
seqplus_seg3 = 4
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)
# test 7/3.2
seg2 = "index.html HTTP/4.0\r\n\r\n"
seg3 = "/index.html HTTP/1.0\r\n\r\n"
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)

# test 3.3/8
seg2 = "/index.html HTTP/1.0\r\n\r\n"
seg3 = "4"
```

```
seqplus_seg2 = 4
seqplus_seg3 = 22
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)
# test 8.3
seg2 = "/index.html HTTP/4.0\r\n\r\n"
seg3 = "1"
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)

# test 3.4/9
seg2 = "/index.html HTTP/1.0\r\n\r\n"
seg3 = "4\r\n\r\n"
seqplus_seg2 = 4
seqplus_seg3 = 22
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)
# test 9/3.4
seg2 = "/index.html HTTP/4.0\r\n\r\n"
seg3 = "1\r\n\r\n"
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)

# test 3.5/10
seg2 = "/index.html HTTP/1.0\r\n\r\n"
seg3 = "/index.html HTTP/4.0"
seqplus_seg2 = 4
seqplus_seg3 = 4
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)
# test 10/3.5
seg2 = "/index.html HTTP/4.0\r\n\r\n"
seg3 = "/index.html HTTP/1.0"
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)

# test 3.6/11
seg2 = "/index.html HTTP/1.0"
seg3 = "/index.html HTTP/4.0\r\n\r\n"
seqplus_seg2 = 4
seqplus_seg3 = 4
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)
# test 11/3.6
seg2 = "/index.html HTTP/4.0"
seg3 = "/index.html HTTP/1.0\r\n\r\n"
segment(dip, seg2, seg3, seqplus_seg2, seqplus_seg3)
```

Appendix C – Who Knows?

The text that follows is the GNU General Public License, Version 2 (GPL V2) and governs your use, modification and/or distribution of SNORT.

Section 9 of the GPL V2 acknowledges that the Free Software Foundation may publish revised and/or new versions of the GPL V2 from time to time. Section 9 further states that a licensee of a program subject to the GPL V2 could be free to use any such revised and/or new versions under two different scenarios:

1. "Failure to Specify." Section 9 of the GPL V2 allows a licensee of a program governed by an unspecified version of the General Public License to choose any version of the General Public License ever published by the Free Software Foundation to govern his or her use of such program.

This provision is not applicable to your use of SNORT because we have expressly stated in a number of instances that any third party's use, modification or distribution of SNORT is governed by GPL V2.

2. "Any Later Version." At the end of the terms and condition of the GPL V2 is a section called "How to Apply these Terms to Your New Program," which provides guidance to a developer on how to apply the GPL V2 to a third party's use, modification and/or distribution of his/her program. Among other things, this guidance suggests that the developer attach certain notices to the program. Of particular importance is the following notice:

"This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version."

Thus if a developer follows strictly the guidance provided by the Free Software Foundation, Section 9 of the GPL V2 provides the licensee the option to either use, modify or distribute the program under GPL V2 or under any later version published by the Free Software Foundation.

SNORT is an open source project that is governed exclusively by the GPL V2 and any third party desiring to use, modify or distribute SNORT must do so by strictly following the terms and conditions of GPL V2. Anyone using, modifying or distributing SNORT does not have the option to chose to use, modify or distribute SNORT under any revised or new version of the GPL, including without limitation, the GNU General Public License Version 3.

For ease of reference, the comparable notice that is used with SNORT (contained in the 'README' file) is as follows:

"This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License Version 2 as published by the Free Software Foundation. You may not use, modify or distribute this program under any other version of the GNU General Public License."

If you have any questions about this statement, please feel free to email snort-info@snort.org.
