Chapter 3

TCP Ambiguity Scrubber Engine

3.1 Evasion Technique against TCP Protocol

Stateful TCP filtering is widely deployed in both commercial and open source network filtering devices like firewalls and network intrusion detection systems (NIDSes) [16, 32, 42]. Most researches before focus on the integrity check of header fields, maintenance of TCP states, and protocol normalization. In recent years the network security devices not only inspect network layer and transport layer (in our case, IP and TCP headers) but also dig into the application layer. One of the well-known methods in NIDSes is misuse detection which adapts a technique called "signature matching" [13]. Signature matching uses pattern-matching algorithms to detect a certain string within a payload stream. For example, an NIDS will try to find the string "big@boss.com" in a SMTP connection to detect the SoBig worm. A signature matcher which detects malicious string in a per-packet basis without paying attention to packet ordering and segment overlapping works in many cases if the attackers don't use any evasion technique to interfere the NIDS. However it's very easy to evade such systems by reordering the packet sequences, chopping a large packet into several small ones, and so on [40]. The TCP filtering engine proposed by this dissertation focuses on providing shaped data stream to the signature matcher and eliminating ambiguities to keep from attack evasion. In the following we will describe why per-packet-basis signature matcher cannot detect malicious data in many circumstance and how can we overcome these problems by proposed methodologies.

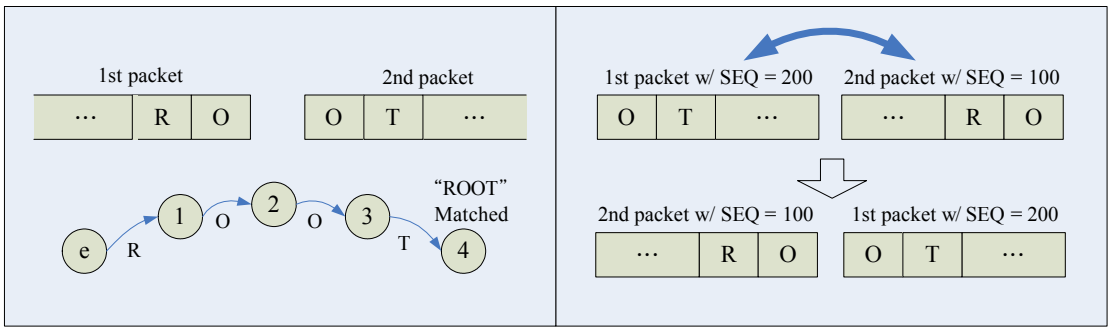

Figure 7. (a) Pattern occurs in packet boundary and (b) packet comes out of order

The first obvious problem happens when the target string occurs in packet boundaries. For example, Figure 7 (a) shows a string "root" divided by two packets. This string cannot be detected if these two packets are processed separately without keeping any information about their relation. If the pattern-matching engine is automata-based, then one apparent solution is to keep the corresponding state of the first packet and set it to the initial state of the second packet instead of starting from state ε.

The second problem happens if the packets arrive out of order. As demonstrated in Figure 7 (b), if the packets are not reassembled correctly, the target string cannot be detected still.

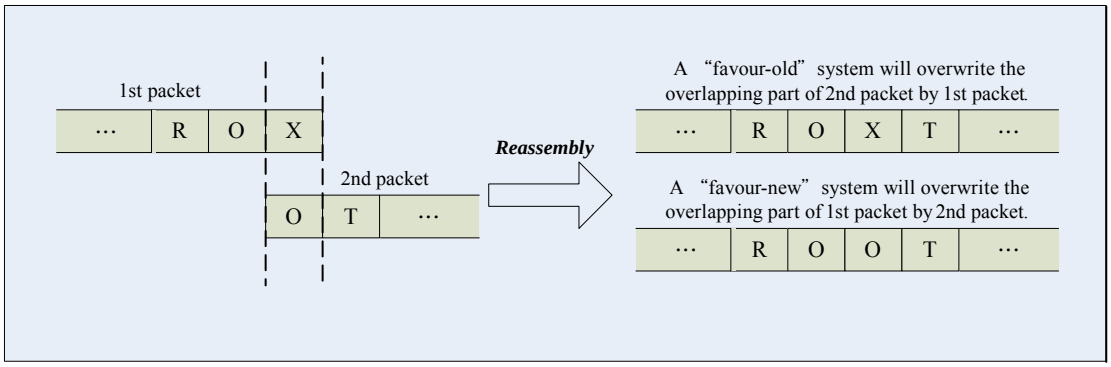

Figure 8. Attackers try to evade the NIDS by retransmitting different data.

One of the reasons regarding that TCP protocol is relatively robust than other stateless protocol is its retransmission mechanism. This method is quite useful when packets lost in the way from the source to the destination, and the source will resend the identical data again. However, some attackers might fool the NIDSes by

retransmitting packets with different data. In Figure 8, the last three bytes of the first packet is “ROX” while the first two bytes of the second packet are “OT”, and there is one byte overlapping between these two packets. The end system which receives these two packets either considers it as “ROOT” or “ROXT” depending on whether its operation system is a “favour-new” system or a “favour-old” one. In the next section we will introduce how we deal with both these two kinds when retransmission occurs. The packet boundary issue arises again as far as retransmission is concerned. As discussed above, an automata-based pattern matcher can still detect target string in packet boundaries by memorizing the state of last byte of previous packets. However, retransmission doesn’t have to start from the previous packet boundaries. It’s impossible to know which state should be kept to concatenate with the first byte of the transmission packets. This design resolves this issue by collecting the previous *MAX_PATTERN_LENGTH – 1* bytes to append to the head of the retransmission packet payload, where *MAX_PATTERN_LENGTH* is the maximum length of signature patterns.

3.2 Related Works in Segment Reassembly

The evasion technique against the NIDSes is well-studied in many researches [31, 32, 40]. However, how to manipulate packet buffer to present an unambiguous stream for signature matching has not been discussed a lot. Among them, [55] proposed two practical mechanisms to make sure the end systems see the same thing as the NIDS. The first is dropping any out-of-order packets to prevent from ambiguity. The second one is to store all the unacknowledged packets, and when it finds an overlapping between the stored packets and the incoming new packet, it overwrites the new one by the stored data. By the way it can eliminate the ambiguity successfully. However, the drawback of this mechanism is its data movement operation. Coping data is always

less desirable because of its impact on system performance. Another defect is its only ability to decode favour-old-type attack. If the attack is targeted on a favour-new system, it can only stop this attack but cannot decode it successfully.

Snort [48, 49] is the most well-known open source NIDS. Originally Snort operated in the per-packet basis without supporting the concept of data stream. The Preprocessor Stream4 [48] was then developed to accommodate the requirement of stream-based processing. However, since Snort is not an inline IDS, the design of Steram4 is not suitable for inline processing due to its lack of real-time reaction. Their design first copies and stores all the received packets in a binary tree. Every time it receives an ACK packet, it verifies whether the offset from last un-scanned point to least acknowledged byte exceeds a randomized number. If the offset does exceed the randomized number, it will collect packets from the binary tree and reassemble them into a data stream and then perform a pattern matching. The randomized offset is to avoid the packet boundary problem. Since the attacker has no idea the boundary between each scanning, it's impossible to evade the detection in this way. However the Snort may still miss the pattern if it does occur in scan boundaries. Besides, the whole operation involves a lot of data movement, and therefore the system performance is downgraded. Moreover, the reassembly operation is triggered by the ACK packets. It implies when the attacking data may arrive the end system already.

Bro [39], another open-source NIDS, also handles the stream reassembly issue well. If the incoming packet is in order (its sequence number is the sum of the sequence number and the length of previous packet), it will be scanned immediately. On the other hand, if the packet arrives out of order, it will then be queued until there is no gap between the previously processed one and itself. However, this design cannot solve the packet boundary problem comprehensively still.

3.3 The Design and Implementation of the Ambiguity Scrubber Engine

The purpose of the proposed TCP engine is to be a Normalizer which can accomplish the following tasks: maintaining TCP state machine, checking the correctness of SEQ and ACK numbers in TCP headers, keeping packet processed in-order, eliminating the ambiguity of the overlapping parts of data segments, and locating the malicious data even they're exactly on the packet boundaries.

Figure 9 shows the packet flow of the TCP engine. First the incoming packet is classified, as long as it's a SYN packet (or others if TCP Cold Start is enabled) a TCP SYN Flood check is performed. After passing the check, a space for recording and tracing this connection is allocated. To detect a SYN Flood event, a threshold for each guarded server is set. Once the number of SYN packets targeted to a guarded server exceeds the threshold in a certain time, we start to drop the first SYN packet of every connection, and enable the semi-transparent-gateway mechanism. We showed in previous chapter that the combination of these two policies mitigate the SYN Flood significantly in a very little cost.

On the other hand, if the packet under processing belongs to an established connection, we will do the header field sanity check. It first verifies the correctness of header control fields, and then checks whether its TCP acknowledgement and sequence numbers are legitimate. The TCP option and PAWS are also taken into consideration. The guideline to check the TCP control flags is shown in [16] while the range of expected sequence and acknowledgement numbers are defined in [42].

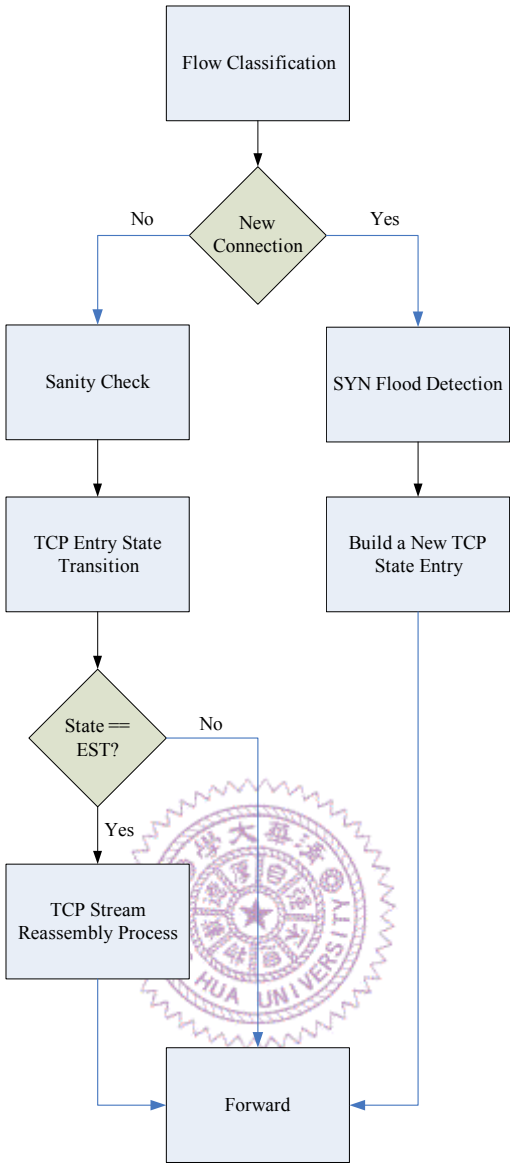

Figure 9. The processing flow of the TCP Ambiguity Scrubber Engine

The state transition diagram of our TCP engine is illustrated in Figure 10. Just like [42], the guideline of this transition diagram is “Never assume anything: the state administration should only be based on facts”. There’re two state records for client and server side respectively. Every time we receive a packet, only the state of the sender of this packet is changed. For example, when we receive a SYN packet from the connection initiator, we will set its state to SYN_SENT but the state of another side remain unchanged. We will set the state of responder to SYN_RECEIVED only after it responds with a SYN/ACK packet. Please note some packets which violate the

state transition diagram will be discarded and cannot go on to the reassembly process.

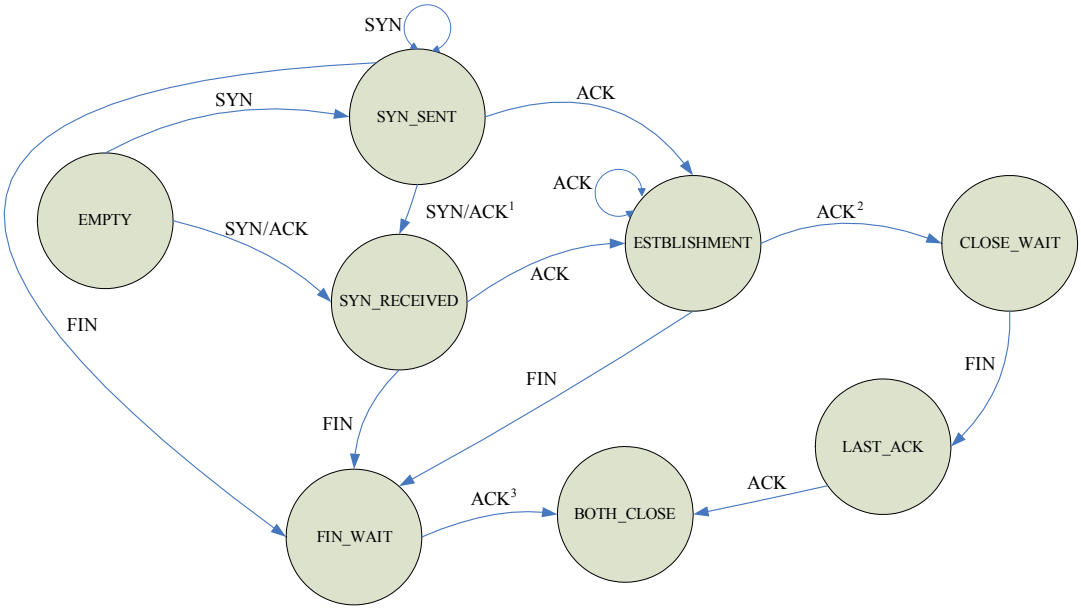

Figure 10. Proposed TCP state transition diagram.

The final stage of the TCP processing engine is stream reassembly process. Figure 11 shows the process in detail. We need two queues for each side (initiator/responder) in this process. Out-of-order Packet Queue (OPQ) stores the packet pointers of those packets which come out of order. OPQ performs packet buffering so that we can scan packets in sequence order. After-Scanned Queue (ASQ) stores the packet pointers of the packets which have been scanned already but not acknowledged by the opposite side. We keep these pointers to resolve the evasion problem by retransmitting an overlapping segment with different data and the pointers where a pattern locates in packet boundary. Every time when a packet of an established connection comes, we first release packet pointers in the ASQ of opposite side according to its ACK number. Since we will drop any further packets which have been acknowledged, it's not necessary to keep the packet pointers anymore. Then we evaluate its sequence number along with the sequence number and payload length of

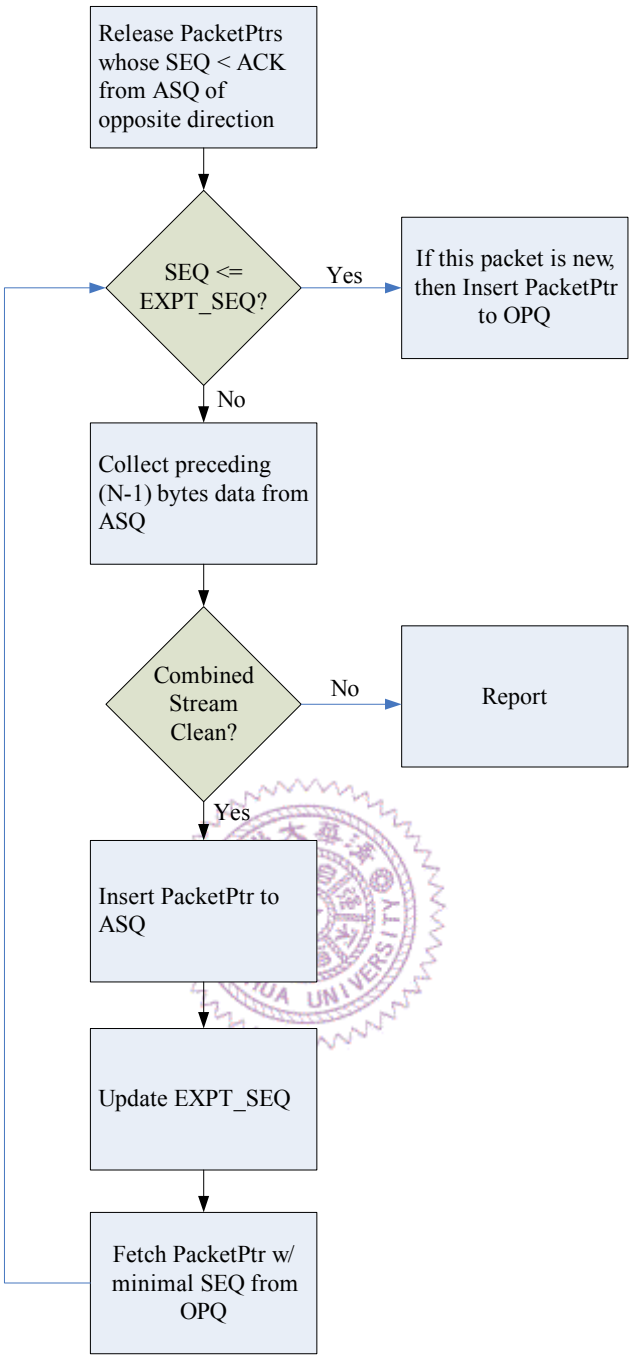

Figure 11. The flow of reassembly process

the previous packet of the same connection in the same direction. There are three possibilities:

$$SEQ > SEQ_{prev} + LEN_{prev}, \quad (1)$$

$$SEQ = SEQ_{prev} + LEN_{prev}, \quad (2)$$

$$SEQ < SEQ_{prev} + LEN_{prev}. \quad (3)$$

SEQ_{prev} indicates the sequence number of the last processed packet of the same connection in the same direction, and LEN_{prev} indicates its payload length. Case (1) means current packet is out-of-order and we will then insert this packet into OPQ for later processing. The reason we want to keep packet scanned in order will be described later. Case (2) means the current packet follows the previous one exactly without overlapping. As mentioned earlier, if the adapted pattern matching algorithm is automata-based, we can continue to compare from the last matching state.

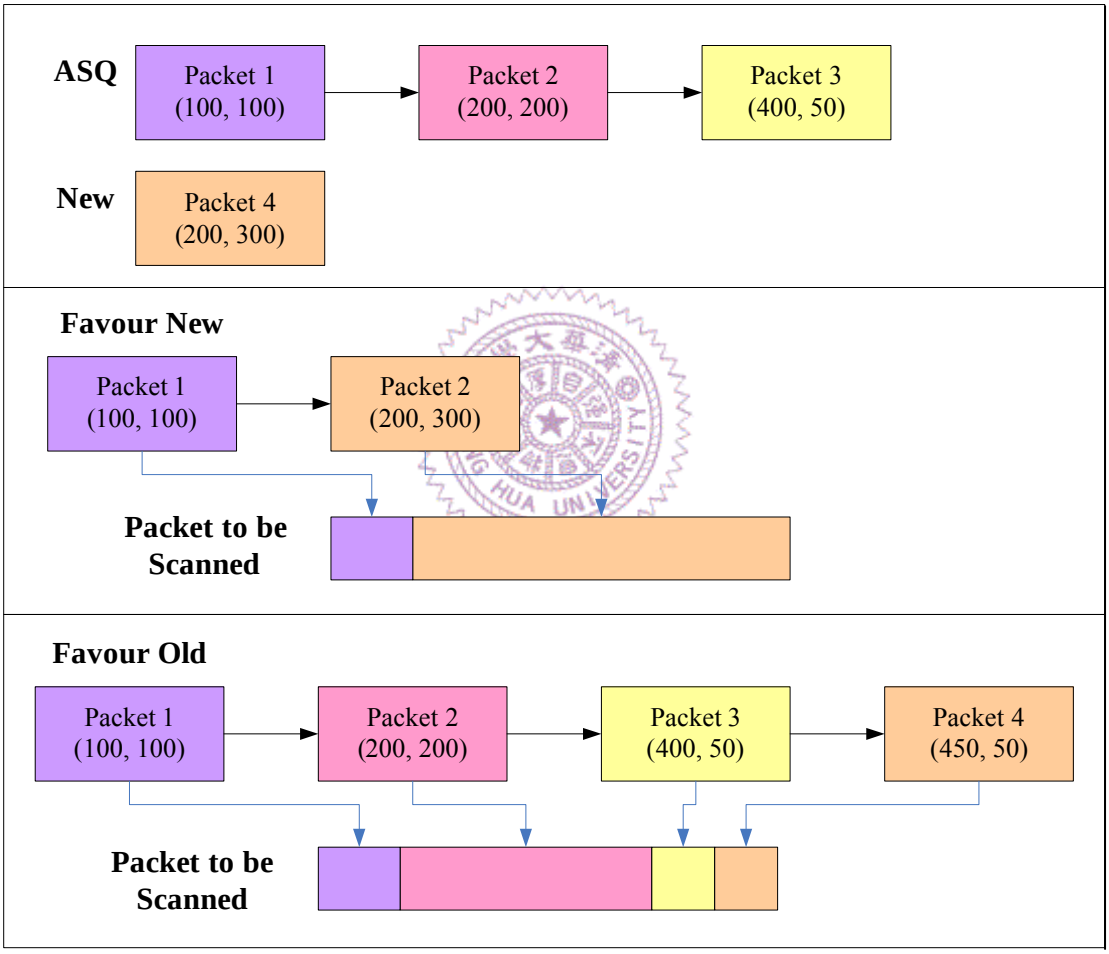

Figure 12. Processing flow for favour-new and favour-old systems

Case (3) means there is an overlapping between the pervious packets and current one. As mentioned above, we shall have different mechanism for both favour-old and favour-new end systems. If the end system is favour-new we will scan the overlapping part of current packet, not the previous ones, and the overlapping data in ASQ will be

released. On the other hand, if the destination system is favour-old, then the overlapping part of the new coming packet should be overwritten by the old data stream collected from ASQ, and the TCP checksum needs to be recalculated as well. Generally data movement/copy operation is less desirable because of its impact on system performance. However, this copy operation makes sure the end system sees the same packet as the NIDS. Even the overlapping part of new packets and saved stream are different, they are eventually synchronized. In both case (favour-new and favour-old), we will collect the preceding *MAX_PATTERN_LENGTH – 1* bytes from ASQ to append to the head of the packet payload to solve the "pattern occurs in packet boundary" problem. Figure 12 shows how we reassemble packet segments for favour-new and favour-old end systems respectively. At completing the reassembly stage, a pattern match operation will be performed to find the malicious data. If the packet doesn't contain any improper data, the packet then will be inserted to the ASQ in case we have another case (3) again later. The reassembly flow will be executed iteratively until case (1) happens.

3.4 Experiments over the TCP Scrubber Engine

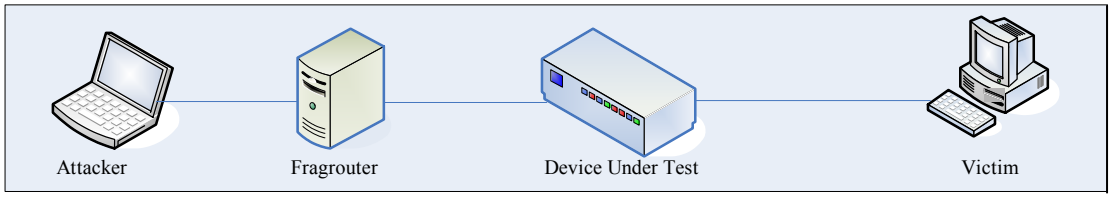

Figure 13. The setup of the experiments over the TCP Scrubber Engine

We implemented the scrubber engine on pSOS+ real-time operating system. Besides, a pattern-matching engine (implemented as [30]) is also developed to find the malicious data from the reassembled stream. Every TCP packet will be checked and reassembled into meaningful stream and provide to the pattern-matching engine

for detection. Figure 13 shows the setup of our experiments over the TCP scrubber engine. The attacker will send the test-cgi exploits (CVE ID: CVE-1999-0070) to the victim web server to discover its file list. The exploit packets will be sent to the Fragrouter first to be chucked into small pieces for evasion. The device under test (DUT) should be able to reassemble those small pieces into a stream and detect the test-cgi exploits before it arrives to the victim.

We followed the test items of NSS test group and the Open Security Evaluation Criteria (OSEC) to fragment the exploits. These two are the most professional and reputable organizations which conduct NIDS tests to generate trustworthy reports. Table II shows the test items regarding the TCP segment evasion in NSS while Table III shows the test items in OSEC.

Table II. Test items in NSS TCP Segment Evasion.

Item Number	***Description***
2.2.9	Ordered 1 byte segments, interleaved duplicate segments with invalid TCP checksums
2.2.10	Ordered 1 byte segments, interleaved duplicate segments with TCP control flags
2.2.11	Ordered 1 byte segments, interleaved duplicate segments with requests to resync sequence numbers mid-stream
2.2.12	Ordered 1 byte segments, duplicate last packet
2.2.13	Ordered 2 byte segments, segment overlap (favour new)
2.2.14	Ordered 1 byte segments, interleaved duplicate segments with out-of-window sequence numbers
2.2.15	Out-of-order 1 byte segments
2.2.16	Out-of-order 1 byte segments, interleaved duplicate segments with faked retransmits
2.2.17	Ordered 1 byte segments, segment overlap (favour new)
2.2.18	Out-of-order 1 byte segments, PAWS elimination (interleaved dup segments with older TCP timestamp options)
2.2.20	Ordered 16 byte segments, segment overlap (favour new (Unix))
2.2.21	Ordered 16 bytes segments, segment overlap (favour old (Win32))

The test ***2.2.9***, ***2.2.10***, ***2.2.11***, ***2.2.14***, ***2.2.16***, ***2.2.18*** are TCP sanity check problems in essence. By following the rules defined in [16] and [42], all of these ambiguous packets will be detected and dropped properly. On the other hand, the test ***2.2.12***, ***2.2.13***, ***2.2.15***, ***2.2.17***, ***2.2.20***, and ***2.2.21*** can be overcome by the design of OPQ and ASQ.

Table III. Test items in OSEC TCP Segment Evasion.

Item Number	***Description***
F. 7	The attack connection 3-way-handshake is completed normally, and then the attack is fragmented and sent in ordered 1-byte increments with one fragment sent out of order.
F. 8	The attack connection 3-way-handshake is completed normally, a simulated disconnection is made (with bad TCP checksum), and then the attack is fragmented and sent in ordered 1-byte increments.
F. 9	The attack connection 3-way-handshake is completed normally, and then the attack is fragmented and sent in ordered 1-byte increments with one fragment (the second to last of each packet) duplicated.
F. 10	The attack connection 3-way-handshake is completed normally, and then the attack is fragmented and sent in ordered 1-byte increments with one fragment (the second to last of each packet) repeated, but with null data content.
F. 11	The attack connection 3-way-handshake is completed normally, and then the attack is fragmented and sent in ordered 2-byte increments. Each fragment is preceded by a 1-byte, null data segment that overlaps the latter 2-byte segment.
F. 12	The attack connection 3-way-handshake is completed normally, and then the attack is fragmented and sent in ordered 1-byte increments. Each fragment is followed by a 1-byte, null data segment that has a far out-of-window sequence number.
F. 13	The attack connection 3-way-handshake is completed normally, and then the attack is fragmented and sent out-of-order.
F. 14	The attack connection 3-way-handshake is completed normally, and then the attack is fragmented and sent in ordered 1-byte increments. Each fragment is followed by a repeat SYN for the connection.

F. 15	The attack connection 3-way-handshake is preceded by null data sent in ordered 1-byte segments as if a connection had already completed, then the handshake occurs with the same connection parameters; finally, the attack is fragmented and sent in ordered 1-byte increments.
F. 16	The attack connection 3-way-handshake is completed and immediately closed with a valid RST. Another connection with different ISN is made otherwise with the same parameters, and the attack is sent in ordered 1-byte segments.

The test ***F. 8***, ***F. 12***, ***F. 13***, ***F. 14***, ***F. 15***, and ***F.16*** are TCP field sanity check problems and TCP state checking issues in essence. By following the TCP state transition tables in Figure 13 and rules defined in [16] and [42], all of these ambiguous packets will be detected and dropped properly. On the other hand, the test ***F.9***, ***F.10***, and ***F.11*** can be overcome by the design of OPQ and ASQ. All the out-of-order packets will be kept in proper order, and the overlapping part will be clarified in both favour-old and favour-new ways. The exploit was detected by our engine in all of the tests.

In summary, this design erases the ambiguities of TCP evasion packets in several ways: header field sanity check, state transition check, out-of-order handling, and overlapping resolution. Our design can pass the evasion test of both NSS and OSEC which are the most reputable and professional NIDS test. This design hardly copies data and its memory consumption is relatively lower than the others since we don't need a stream buffer for each connection but a low-cost tree structure. Moreover, it can be easily integrated into existing NIDS implementations since it fulfills the general layer-4 functions comprehensively.