RUN-TIME PROTOCOL-CONFORMANCE VERIFICATION IN FIREWALLS

Ulrich Ultes-Nitsche & InSeon Yoo

University of Fribourg

Department of Computer Science, Chemin du Musée 3, CH-1700 Fribourg, Switzerland

{uun, in-seon.yoo}@unifr.ch

ABSTRACT

Today, business continuity depends significantly on the continuous availability of information systems. It is well-known that such systems must be protected against intrusion and denial of service attacks. Historically, many of such attacks used ill-formed data-packets and/or protocol runs, which did not conform to the protocols' standards. Attackers exploited vulnerabilities of the protocols' implementations in the servers' operating systems: conformance with protocol standards was not tested properly. Prominent examples are: the *ping of death*, the *land attack*, the *SYN flood attack*.

To protect information systems better, one should aim to recognize and block such attacks as early as possible, i.e. already in a firewall at a company network's border. We will discuss in this paper the design of a run-time protocol-verifier and data-packet sanity-checker we will use to complement the Intelligent Firewall, which is currently developed in the *Janus* project. The presented concepts are, however, generic and applicable to any firewall.

KEY WORDS

Firewalls, DoS protection, conformance to protocol standards, protocol specification, SDL

RUN-TIME PROTOCOL-CONFORMANCE VERIFICATION IN FIREWALLS

1 INTRODUCTION

Today, business continuity depends significantly on the continuous availability of information systems. It is well-known that such systems must be protected against intrusion and denial of service attacks. Historically, many of such attacks used ill-formed data-packets and/or protocol runs, which did not conform to the protocols' standards. By doing so, they caused servers to crash or perform tremendously poorly at best. Attackers exploited vulnerabilities of the protocols' implementations in the servers' operating systems: conformance with protocol standards was not tested properly. Prominent examples are: the *ping of death* (using fragmented PING requests exceeding the maximum IP packet size), the *land attack* (using TCP segments with equal source and destination address as well as equal source and destination port number), the *SYN flood attack* (blocking server resources by using incomplete protocol runs of TCP connection establishments). We could list several more. To protect information systems better, one should aim to recognize and block such attacks as early as possible, i.e. already in a firewall at a company network's border. We will discuss in this paper the design of a run-time protocol-verifier and data-packet sanity-checker we will use to complement the Intelligent Firewall, which is currently developed in the *Janus* project. The presented concepts are, however, generic and applicable to any firewall.

The aim of the discussed security sub-system is easy to formulate: check arriving data packets for conformance with protocol standards, including the state of the protocol run *and* the structure of the packet itself. In case of any non-conformance detected, block the packets. Obviously this can be achieved easily by hard-coding protocol information into the firewall. However, it should be equally obvious that this would be a very inflexible solution: Each new version of a protocol would require a re-implementation of the protocol conformance part of the firewall. A far more flexible solution will be considered here: The core implementation of the protocol verifier and sanity checker will be generic. They are then configured by feeding them with protocol specifications in a suitable format. We have decided to use a formal notation frequently used by the European telecoms industry to specify communication protocols. This language is called SDL (SDL is also used outside Europe). In SDL, we can easily specify the protocols to which we want the observed data traffic to conform. From the SDL specification, an extended finite-state machine (EFSM) description can be generated automatically to configure the protocol-verifier, which then checks observed data traffic against the EFSM description of the relevant protocol. As SDL was created to describe *conceptually* the exchange of messages in communication protocols, it is, unfortunately, lacking a rich enough data description language to express conformance of data packets with their standardized formats. We are therefore currently working on a separate extension to the SDL protocol specification, in which the standardized structure of *data-packets* can be described. As these descriptions will constrain all possible packet formats to the ones allowed by the standards documents, we assume that in our case a first-order constraint language will be a suitable extension to the SDL specifications. In the constraint language, we will make, for instance, statements about how parts of a data packet relate to other parts. This will enable us to implement the packet sanity-checking component of the firewall in a similarly flexible way as the run-time protocol-verifier (it needs to inspect data packets and interpret the corresponding constraints on the packets' structures). This is, however, future work.

We believe that our work will be beneficial to increase the capability of any packet-filtering firewall by better protecting the secure network from attacks based on ill-formed protocol runs or data packets.

2 EXAMPLES OF PROTOCOL EXPLOITATIONS

In this section, we present examples of previous attacks and discuss how they exploited anomalous protocol runs, including anomalous packet structures.

2.1 Ping of Death

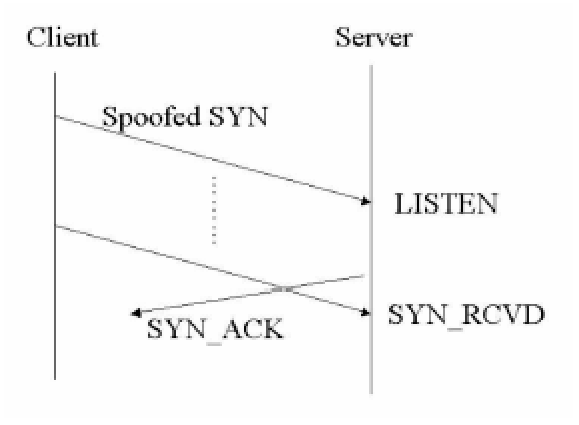
Attackers send fragmented PING requests that exceed the maximum IP packet size of 64KB, causing vulnerable systems to crash. The idea behind the Ping of Death (Fyodor, 1996) and similar attacks is that the user sends a packet that is malformed in such a way that the target system will not know how to handle the packet. Carefully programmed operating systems can detect and safely handle abnormal IP packets, but some failed to do so. Besides ICMP *ping*, also UDP and other IP-based protocols could transport the Ping of Death.

2.2 Land Attack

The land attacks (CISCO, 1997) are also known as IP DoS (*Denial of Service*) (Fyodor, 1997). The land attack involves sending a stream of TCP SYN packets in which the source IP address and TCP port number is set to the same value as the destination address and port number (i.e. of the attacked host). Some implementations of TCP/IP cannot handle this theoretically impossible condition, causing the operating system to go into a loop while trying to resolve repeated connections to itself.

2.3 SYN flood attack

The client system begins by sending a SYN message to the server (CERT/CA-1996-21, 2000)). The server then acknowledges the SYN message by sending a SYN-ACK message to the client like in the figure below. The client then finishes establishing the connection by responding with an ACK message. The connection between the client and the server is then open, and the service-specific data can be exchanged between the client and the server.

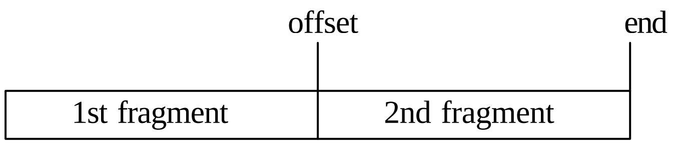


SYN Flooding

The TCP SYN attack exploits this design by sending SYN packets with random source addresses to a victim host. The victim host sends a SYN ACK back to the random source address and adds an entry to the connection queue. Since the SYN ACK is addressed to an incorrect or nonexistent host, the last part of the three-way handshake is never completed and the entry remains in the connection queue until a timer expires, typically within about one minute. By generating such SYN packets from random IP addresses at a rapid rate, it is possible to fill up the connection queue and deny TCP services to legitimate users.

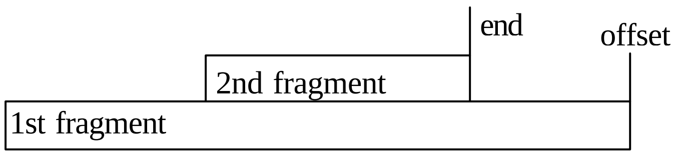
2.4 Teardrop attack

The Teardrop attack (Hoggan, 2000) exploits a vulnerability of the reassembly of segmented IP packets. Fragmentation is necessary when IP datagrams are larger than the maximum transmission unit (MTU) of a network segment. In order to successfully reassemble packets at the receiving end, the IP header for each fragment includes an offset to identify the fragment's position in the original unfragmented packet. In a Teardrop attack, packet fragments are deliberately fabricated with overlapping offset fields causing the host to hang or crash when it tries to reassemble them. The normal usage of offsets if presented below:



IP TearDrop Attack - Correct reassembling

However, the teardrop attack sends a fragment that deliberately forces the calculated value of the end pointer to be less than the value of the offset pointer. This can be achieved by ensuring that the second fragment specifies a FRAGMENT OFFSET that resides within the data portion of the first fragment and has a length such that the end of the data carried by the second fragment is short enough to fit within the length specified by the first fragment. Diagrammatically, this can be shown as follows:



IP TearDrop Attack - incorrect reassembling

When the IP module performing the reassembly attempts to store a copy of the fragment into the buffer assigned to the complete datagram, the calculated length of data to be copied (that is the end pointer minus the offset pointer) yields a negative value. The memory copy function expects an unsigned integer value and so the negative value is viewed as a very large positive integer value. The result of such an action depends on the IP implementation, but typically causes stack corruption, failure of the IP module, or a system hang.

2.5 Code Red

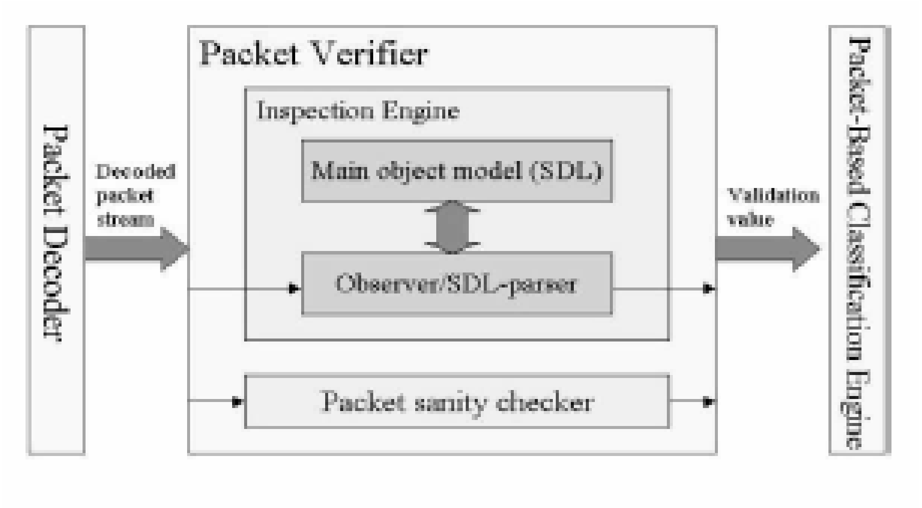
The Code Red worm is a malicious self-propagating code (CERT/CA-2001-23, 2002) that spreads surreptitiously through a hole in certain versions of the Microsoft Internet Information Server (IIS). Infected systems may experience performance degradation as a result of the scanning activity of this worm. The beginning of the Code Red's attack packet looks as follows (CERT/CA-2001-19, 2001), (CERT/CA-2001-09, 2001):

```
GET/default.ida?NNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNN%u9090%u6858%ucbd3
```

This shows that binary characters are used in an HTTP header. However, the HTTP standard prohibits binary characters in HTTP headers.

3 THE PACKET VERIFIER

The purposes of the packet verifier are validating compliance to standards, and validating expected usage of protocols. The packet verifier checks the protocol header of packets, verifies packet size, checks TCP/UDP header length, verifies TCP flags and all packet parameters, does TCP protocol type verification, and analyses TCP Protocol header and TCP protocol flags. In addition, the packet verifier contains an inspection engine, including an observer and a main object model, to validate expected usage of protocols with SDL (the Specification and Description Language) (ITU-T, 1992) specifications:



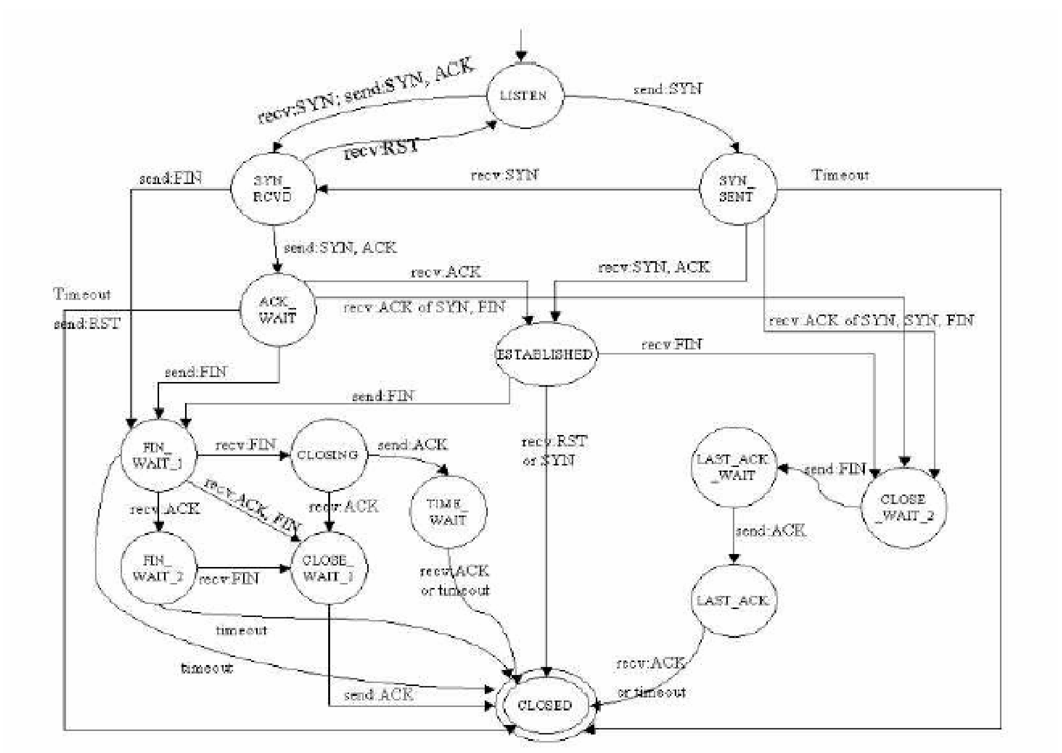
Components of the Packet Verifier

Using SDL specifications we define the possible behaviours of protocols. While investigating the encoded packets, the observer/SDL-parser validates whether or not each sequence of packets follows what is required by the SDL specifications. Finally, this packet verifier sends the result of validation to the packet-based classification engine.

3.1 Protocol Specification

We use an abstract specification of the TCP protocol, where the state machines accept a superset of what is permitted by the standards, and is still sufficient to deal with incomplete protocol runs meeting the standards (such as in the case of the SYN flood attack). We describe the TCP state machine in this sub-section.

A TCP connection is always initiated with the three-way handshake, which establishes and negotiates the actual connection over which data will be sent. The whole session begins with a SYN packet, then a SYN/ACK packet and finally and an ACK packet to acknowledge the whole session establishment. Our TCP specification is depicted pictorially below. A new session starts in the LISTEN state. Data transfer takes place in the connection ESTABLISHED state. If the TCP connection is initiated from an external site, then the state machine goes through SYN_RCVD and ACK_WAIT states to reach the ESTABLISHED state. If the connection is initiated from an internal machine, then the ESTABLISHED state is reached through the SYN_SENT state. In order to tear down the connection, either side can send a TCP segment with the FIN bit set. If the FIN packet is sent by an internal host, the state machine waits for an ACK of FIN to come in from the outside. Data may continue to be received till this ACK to the FIN is received. It is also possible that the external site may initiate a closing of the TCP connection. In this case we may receive a FIN, or a FIN + ACK from the external site. This scenario is represented by the states FIN_WAIT_1, FIN_WAIT_2, CLOSING, TIME_WAIT_1, and TIME_WAIT_2 states. Our state machine characterizes receive and send events. If the connection termination is initiated by an external host, note that the TCP RFCs do not have the states CLOSE_WAIT, LAST_ACK_WAIT, and LAST_ACK since they deal with packets observed at one of the ends of the connection. In that case, it is reasonable to assume that no packets will be sent by a TCP stack implementation after it receives a FIN from the other end. In our case, we are observing traffic at an intermediate node, e.g. firewall, so the tear down process is similar regardless of which end initiated the tear down. In addition, time-out values must be taken into account. They may change according to the security policy, however, the default values should be fairly well established in practice. To reduce clutter, the following classes of abnormal transitions are not shown: conditions where an abnormal packet is discarded without a state transition, e.g. packets received without correct sequence numbers after connection establishment and packets with incorrect flag settings. These parts will be checked by the SanityChecker.



The TCP state machine

3.2 Protocol SanityChecker

To cover other protocol aspects apart from TCP state specification, we are building a sanity checker. This performs layer 3 and layer 4 sanity checks. These include verifying packet size, checking UDP and TCP header lengths, dropping IP options and verifying the TCP flags to ensure that packets have not been manually crafted by a malicious user, and that all packet parameters are correct. In the IP protocol, according to the Internet Protocol Standard (RFC791, 1981), an IP header length should always be greater than or equal to the minimal Internet header length (20 octets) and a packet's total length should always be greater than its header length. IP address checks will also be important since land attacks use the same IP address for source and destination. According to the TCP standard (RFC793, 1981), neither the source nor the destination TCP port number can be zero, and TCP flags, e.g. URG and PSH flags, can be used only when a packet carries data. Thus, for instance, combinations of SYN and URG or SYN and PSH become invalid. In addition, any combination of more than one of the SYN, RST, and FIN flags is also invalid.

SanityChecker examines every packet within a 10 second window, and at the end of each window it will record any malicious activity it sees using syslog. SanityChecker currently detects some attacks; all TCP scans, all UDP scans, SYN flood attacks, Land attacks, and ping of death attacks. SanityChecker assumes any TCP packet other than an RST may be used to scan for services. If packets of any type are received by more than 7 different ports within the window, an event is logged. The same criteria are used for UDP scans.

If SanityChecker sees more than 8 SYN packets to the same port with no ACK's or FIN's associated with the SYN's, a SYN flood event is logged. Any TCP SYN packets with equal source and destination address and ports are identified as a land attack. If more than 5 ICMP ECHO REPLIES are seen within the window, SanityChecker assumes it may be a Smurf attack (CERT, 1998). Note that this is not a certainty. SanityChecker also assumes that any fragmented ICMP packet is bad. This catches attacks such as the ping of death. To increase the confidence in the reported events, the SanityChecker cooperates with the protocol inspection engine, which uses the SDL protocol description. Furthermore, when the SanityChecker cooperates with the protocol inspection engine, current TCP packet's sequence numbers are matched against a state kept for that TCP connection. For example, in the teardrop attack, fragmented packets can be identified by packet's IP id and sequence number. The inspection engine will examine all protocol transitions triggered by arriving packets and remember IP id and sequences number. So the SanityChecker can detect reassembly problems by cooperating with that engine.

4 GENERATION OF A STATE-MACHINE DESCRIPTION

We made the state-machine specification presented in this section by hand. Our next step will be applying SDL and a suitable tool to create such a specification. SDL is an International Telecommunication Union (ITU) standard, based on the concept of a system of Communicating Extended Finite State Machine (CEFSM) (Hopcroft and Ullman, 1979). To understand how SDL can be applied, we address briefly the dynamic semantics of the finite state machine, and an underlying SDL model subsequently.

4.1 Dynamic Semantics of Finite State Machines

SDL is based on the concept of Communicating Extended Finite-State Machines (CEFSM), which communicate with each other and their environment by signals in an asynchronous manner via possibly delaying communication paths. These signals are buffered on arrival at a process. A finite state machine (FSM) is defined in a standard way as a 4-tuple $< S, s_0, E, f >$, where S is a set of states, s_0 is an initial state, E is a set of events with their parameter lists, f is a state transition relation. However, the construction of FSM, is limited by the state-explosion problem. An *extended* finite-state machine (EFSM) solves this problem by introducing variables in addition to explicit states of the process instance. These variables become implicit states, being able to take on a number of values themselves.

Each EFSM is defined as a FSM with addition of variables to its states: it is a 5-tuple $< S, s_0, E, f, V >$, where V is a set of local variables along with their types and initial values, if any. Each state in an EFSM is defined by a set of variables, including state names. The transition T of an EFSM becomes $[< s, v_1, \ldots, v_n > + \text{input}^*, \text{task}^*; \text{output}^* + < s', v_1', \ldots v_n' >]$, where s and s' are the names of the states, $< v_1, v_2, \ldots, v_n >$ and $< v_1', v_2', \ldots v_n' >$ are the values of extended variables, n is the number of variables, "+" means coexistence, ";" means sequence of events such as tasks and outputs, and "[,]" denotes a sequenced pair. The difference between an EFSM and an FSM is that an EFSM associates each transition not only with input and output actions but also with assignment action and condition (Wang, 1993).

A *communicating* extended finite-state machines (CEFSM) includes the definitions of EFSMs and signals (Ellsberg et al., 1997): it is a 6-tuple $< S, s_0, E, f, V, X >$, where X is a set of signals. In CEFSM, signals are responsible for communicating information from within the CEFSM to other automata, some of which may be located in the environment of a system. The signals account for the observable behaviour, which is more important than the actual model for a specification. In SDL, CEFSM processes use signals to communicate with other CEFSMs and the environment.

4.2 SDL Underlying Model

The language SDL is intended to formally specify complex, event-driven, real-time, interactive applications involving many concurrent activities that communicate using discrete signals. It is especially well suited for specification of communication protocols, reactive systems such as switches, routers and distributed systems. SDL has been designed for the specification and description of the behaviour of such systems, i.e. the internetworking of the system and its environments. SDL allows the hierarchical description of systems. The description starts from a construction called system, where functional blocks are inserted. A block is a component composed by one or more processes and/or other blocks. A block consists of processes connected by signal routes. A process contains a sequential behaviour and concurrency is modelled by a set of processes. Each process is a CEFSM. These machines or processes run in parallel. They are independent of each other and communicate with discrete messages, called signals. A process can also send signal to and receive signals from the environment of the system. The behaviour of a state machine is characterized by a set of transitions. A transition to another state or the same state occurs whenever an input is consumed. When a process is in a state it accepts input. This input can be a signal received by the input port or timers. When a process enters a new state, it means that a transition terminates. CEFSM enables decisions to be made in transitions based on the value associated with a variable so that the state which follows when a specific input is consumed is not only determined by the existing state and input.

The SDL language supports two equivalent notations: the graphical notation (SDL-GR) and the textual notation (SDL-PR). The SDL-GR is a standardized graphical representation of the system. SDL elements such as system, block, process, signal etc. are drawn using standardized graphical symbols. The SDL-PR is a textual phrase representation of the SDL system, or in other words, it is a SDL source code.

- **Process Model** The Z.100 ITU-T standard defines that the SDL underlying model is CEFSM (Communicating Extended Finite State Machine), where all processes are CEFSMs. For each process, a finite number of states, inputs and outputs determine its behaviour. Non-determinism allows representing spontaneous transitions, which are transitions without any signal causing them. This is useful to describe unpredictable system characteristics. In SDL, only one input signal can be consumed/evaluated at each instant. This means that each input signal consumed corresponds to one state transition in an SDL description.

- **Communication Model** The concurrency model used in SDL allows independent and asynchronous process operation. There is no guaranteed relative ordering of operations in distinct processes, except the ordering created by explicit synchronization among processes through the use of shared signals. Shared signal events are then the means by which a precise ordering of events in distinct processes can be achieved.

  The communication between processes is reliable. It is assured that the receiving process will consume every signal produced by a sender process. However, it is not guaranteed that the ordering of the signals generated by all processes is the same as of their consumption. This model is adequate to describe events without precise ordering, like systems that can have their normal flow interrupted. Handshaking or unlimited queues, in practice bounded queues, are used to implement the communication model. For both cases, each SDL state results in a set of protocol communication signals and area overhead to implement the protocol. This characteristic may cause large communication overhead, which can penalize the implementation.

4.3 Specification Development

We present how we specify TCP state transition with CEFSM in this section. A CEFSM is defined as a 6-tuple $< S, s_0, E, f, V, X >$, as we mentioned above.

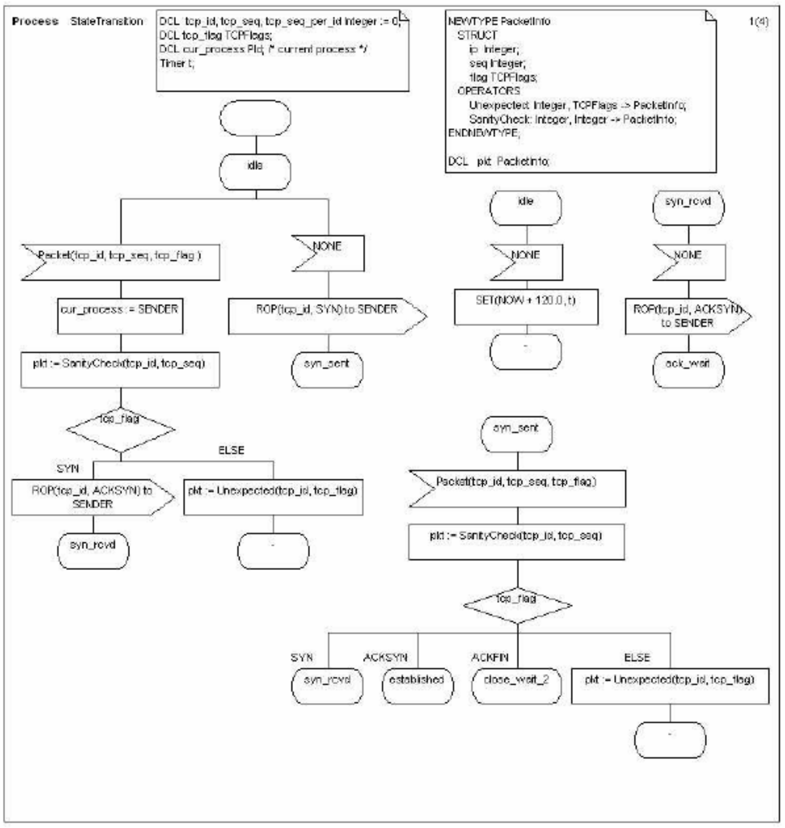
- S is a set of states
- s_0 is an initial state
- E is a set of events with their parameter lists
- f is a state transition relation
- V is a set of local variables along with their types and initial values, if any
- X is a set of signals

For a state, an input event, and a predicate composed of a subset of V, the state transition relation f has a next state, a set of output events and their parameters, and an action list describing how the local variables are updated.

The purpose of SDL in our project is to verify whether the TCP transition follows the standards. To do this, we made very simple TCP transition using SDL based on Figure *The TCP state machine*. For TCP state transitions, our CEFSM is as follows:

- S = { listen, syn_rcvd, syn_sent, ack_wait, established, fin_wait_1, fin_wait_2, closing, close_wait_1, close_wait_2, time_wait, last_ack_wait, last_ack, closed }
- s_0 = listen
- E = { send(ip_id, flags), recv(ip_id, flags) }
- f : { f(listen, recv(ip_id, SYN), ip_seq_per_id = 0) -> (syn_rcvd, ip_seq_per_id = ip_seq_per_id + ip_seq, {SYN, ACK}), f(listen, send(ip_id, SYN), SYN, ip_seq_per_id != 0) -> (syn_sent, , {}), … }
- V = { ip_seq_per_id, ip_seq }
- X = { ACK, SYN, FIN }

In this SDL specification, Timeout, and other flags e.g., RST, PSH, URG are not included. Timeout and RST, PSH, URG flags can be dealt with in a low-level implementation part. To detect packet fragmentation, the SDL specification part can tell the packet sequence and proper flag, and the low-level implementation part cooperates with this SDL specification, other flag combinations, and the timeout part. Figure \ref{Figure: TCPProcess} shows the StateTransition process which we built in SDL. To the graphical SDL-GR spec (Figure *Process StateTransition of the TCP Protocol State Machine in SDL* below), the corresponding textual SDL-PR representation can be found in Figure *Part of SDL-PR Source in the process StateTransition*.



Process StateTransition
1(4)
idle
Packet(tcp_id, tcp_seq, tcp_flag)
NONE
cur_process := SENDER
ROP(tcp_id, SYN) to SENDER
pkt := SanityCheck(tcp_id, tcp_seq)
syn_sent
tcp_flag
SYN
ELSE
ROP(tcp_id, ACKSYN) to SENDER
pkt := Unexpected(tcp_id, tcp_flag)
syn_rcvd
-
idle
NONE
SET(NOW + 120.0, t)
-
syn_rcvd
NONE
ROP(tcp_id, ACKSYN) to SENDER
ack_wait
syn_sent
Packet(tcp_id, tcp_seq, tcp_flag)
pkt := SanityCheck(tcp_id, tcp_seq)
tcp_flag
SYN
ACKSYN
ACKFIN
ELSE
syn_rcvd
established
close_wait_2
pkt := Unexpected(tcp_id, tcp_flag)
-

```
DCL tcp_id, tcp_seq, tcp_seq_per_id Integer := 0;
DCL tcp_flag TCPFlags;
DCL cur_process PId; /* current process */
Timer t;
```

```
NEWTYPE PacketInfo
  STRUCT
    ip Integer;
    seq Integer;
    flag TCPFlags;
  OPERATORS
    Unexpected: Integer, TCPFlags -> PacketInfo;
    SanityCheck: Integer, Integer -> PacketInfo;
ENDNEWTYPE;

DCL  pkt PacketInfo;
```

Process StateTransition of the TCP Protocol State Machine in SDL

```
PROCESS StateTransition ;
NEWTYPE PacketInfo
STRUCT
ip  Integer;
seq Integer;
flag TCPFlags;
OPERATORS
Unexpected: Integer, TCPFlags -$>$ PacketInfo;
SanityCheck: Integer, Integer -$>$ PacketInfo;
ENDNEWTYPE;
DCL pkt  PacketInfo;
DCL  tcp_id, tcp_seq, tcp_seq_per_id Integer := 0;
DCL tcp_flag TCPFlags;
DCL cur_process PId; /* current process */
Timer t; START;

NEXTSTATE  idle ;
STATE syn_sent ;
INPUT Packet(tcp_id, tcp_seq, tcp_flag) ;
TASK pkt := SanityCheck(tcp_id, tcp_seq) ;
DECISION tcp_flag ;
( ACKFIN ):NEXTSTATE  close_wait_2 ;
( ACKSYN ):NEXTSTATE  established ;
( SYN ):   NEXTSTATE  syn_rcvd ;
ELSE:      TASK pkt := Unexpected(tcp_id, tcp_flag) ;
NEXTSTATE - ;
ENDDECISION;
ENDSTATE;
STATE syn_rcvd ;
INPUT NONE;
OUTPUT ROP(tcp_id, ACKSYN) to SENDER ;
NEXTSTATE  ack_WAIT ;
ENDSTATE;
```

Part of SDL-PR Source in the process StateTransition

5 CONCLUSIONS

We have discussed protocol anomalies and address a packet verifier model. The purposes of the packet verifier are to validate compliance to standards, and to validate expected usage of protocols, especially protocol anomaly detection. Considering performance in real-system, we are implementing a SanityChecker to cover protocol header anomalies, and, using SDL, we specify a TCP transition system. The specification will then be compiled into an inspection engine program for observing packets. This engine will be generic and its behaviour will only be configured by feeding it with a specific protocol specification. At the moment, we specified TCP protocol transitions. Through this state machine, we will implement the inspection engine. The system described in this paper is currently under development.

To summarize: We believe that this protocol anomaly analysis and the packet verifier model will be useful to detect protocol anomalies and verify proper usage of protocols. It will be part of the Intelligent Firewall, which we are currently developing in the project Janus (project web-page: http://diuf.unifr.ch/people/yoois/Janus/).

6 REFERENCES

CERT (1998). Advisory ca-1998-01 smurf ip denial-of-service attacks. In *Online Publication*.

CERT/CA-1996-21 (2000). Advisory ca-1996-21 tcp syn flooding and ip spoofing attacks. In *Online publication*.

CERT/CA-2001-19 (2001). Cert advisory ca-2001-19: Code red worm exploiting buffer overflow in iis indexing service dll. In *Online publication*.

CERT/CA-2001-23 (2002). Cert advisory ca-2001-23: Continued threat of the code red worm. In *Online Publication*.

CERT/IN-2001-09 (2001). Cert incident note in-2001-09: Code red ii: Another worm exploiting buffer overflow in iis indexing service dll. In *Online Publication*.

CISCO (1997). Security advisory: Tcp loopback dos attack (land.c) and cisco devices.

C.J.Wang, M. (1993). Generating test cases for efsm with given fault models. In *Proc. of IEEE INFOCOM93, Vol.2, pp.774-781*.

Fyodor (1996). Ping of death attack. In *INSECURE.ORG*.

Fyodor (1997). The land attack(ip dos). In *INSECURE.ORG*.

Hoggan, D. (1994-2000). Teardrop attack. In *The Internet Book: Introduction and Reference*.

Hopcroft, U. (1979). *Introduction to Automata Theory*. Addison Wesley.

ITU-T, C. (1992). *Recommendation Z.100: Specification and Description Language (SDL)*. General Secretariat, Geneva, Switzerland.

J. Ellsberger, D. Hogrefe, and A. Sarma (1997). *SDL 92: Formal Object-oriented Language for Communicating Systems*. Prentice Hall.

RFC791 (1981). Internet protocol. In *DARPA Internet Program Protocol Specification*.

RFC793 (1981). Transmission control protocol. In *DARPA Internet Program Protocol Specification*.