Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

Slow scan detector

Sessionizer software design

Glen Henderson

Defence R&D Canada – Ottawa
CONTRACT REPORT
DRDC Ottawa CR 2008-285
April 2009

Canada

Slow scan detector

Sessionizer software design

Glen Henderson
Bell Canada - ICT

Prepared By:
Bell Canada - ICT
333 Preston St.
Ottawa, ON K1S 5N4
Contract number: W7714-071029
CSA: Joanne Treurniet, Defence Scientist, 613-990-7096

Defence R&D Canada – Ottawa

Contract Report
DRDC Ottawa CR 2008-285
April 2009

Scientific Authority

Original signed by Joanne Treurniet

Joanne Treurniet

Defence Scientist

Approved by

Original signed by Julie Lefebvre

Julie Lefebvre

H/NIO

Approved for release by

Original signed by Pierre Lavoie

Pierre Lavoie

H/DRP

15bo01

Abstract

The Network Information Operations Section at DRDC Ottawa has developed a proof-of-concept low-profile scan detection engine using the MATLAB programming environment. It reads network traffic from pcap format files using custom-built interfaces to the pcap libraries and separates the traffic into connections (TCP) or sessions (UDP/ICMP). Anomalous connections are then processed to identify scans, including those that are slow and/or distributed.

In this work, a software framework was designed where the traffic sessionizer module is written in C++ to increase the processing speed for use in an operational environment. The sessionizer module interfaces with MATLAB to perform the scan detection. The system and sessionizer software are described in detail herein.

Résumé

La section Opération d'information de réseau de RDDC Ottawa a développé un moteur de détection de balayages discret au moyen de l'environnement de programmation MATLAB. Il lit le trafic réseau de fichiers en format pcap au moyen d'interfaces personnalisées avec les bibliothèques pcap et il répartit ce trafic entre des connexions (TCP) ou des sessions (UDP/ICMP). Les connexions anormales sont ensuite traitées afin de relever les balayages, y compris les balayages lents et/ou les balayages répartis.

Au cours de ce travail, un cadre logiciel a été conçu dans lequel le module sessionizer de trafic est écrit en C++ afin d'accroître la vitesse de traitement en vue de son utilisation dans un environnement opérationnel. Le module sessionizer est interfacé avec MATLAB pour effectuer la détection des balayages. Le logiciel système et celui du sessionizer sont décrits en détail dans ce document.

This page intentionally left blank.

Table of contents

List of figures

List of tables

This page intentionally left blank.

1 Introduction

1.1 Background

The Network Information Operations Section at DRDC Ottawa has developed a proof-of-concept low-profile scan detection engine using the MATLAB programming environment. It reads TCP, UDP and ICMP traffic from *pcap* format files using custom-built interfaces to the *pcap* libraries and separates the traffic into connections (TCP) or sessions (UDP/ICMP). Any anomalous connections are then sorted and aggregated to identify scans, including those that are slow and/or distributed. The next stage of this work is to develop a prototype that can be deployed operationally.

1.2 Objective

The objective of this project is to convert the module that separates the TCP, UDP and ICMP traffic into sessions ("sessionizes") from its original version in MATLAB to the C programming language. The remaining MATLAB functions will be compiled with this C function to yield a standalone application. Enhanced user interfaces will also be included in the prototype.

2 Solution Requirements

The requirements for the new sessionizing code are as follows:

1. Create an equivalent software implementation of the existing Matlab-based sessionizing code in C in a Linux environment.

2. Maintain all existing functionality provided in the existing Matlab-based sessionizing code.

3. Make calls to the Matlab computational engine to interface with the existing Matlab routines for clustering

4. Provide demonstrable improvements in terms of performance and memory usage.

3 Solution Overview

This section provides a conceptual overview of the significant data flows associated with the Sessionizer process. The following diagram indicates the primary flows from when the application is launched to just prior to the sessionizing process.

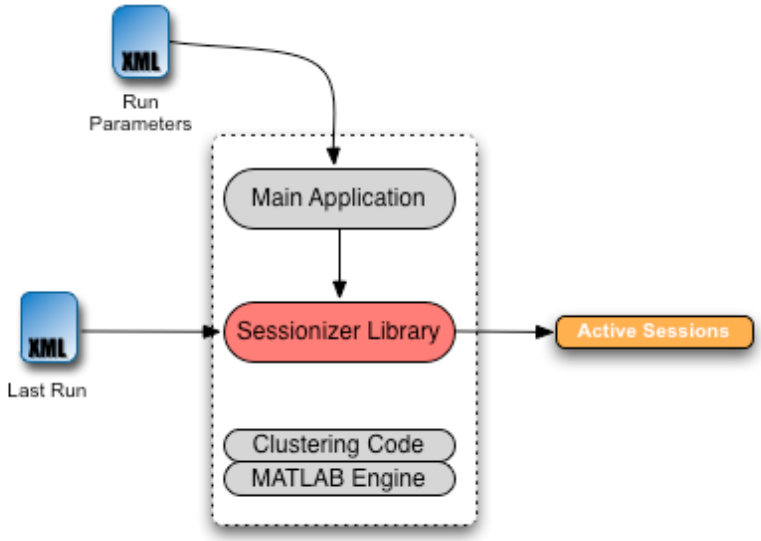

Figure 1: Conceptual Overview: Preloading Session Data

The application itself, when loaded into memory, consists of the main application, the Sessionizer library and the Matlab engine. The critical component in this structure is the Sessionizer library since it contains all the logic for performing the sessionizing functions. The main application, in contrast, is a simple instantiation of the use of the Sessionizer library, as its only functions are to:

1. Take the name of an XML file that is input on the command line. This XML file specifies the process run parameters (see Annex A for XML file formats):

    a. The IP address ranges that define the internal network;

    b. The names of the PCAP files that serve as input to the sessionizing routines;

    c. The name of the file that contains the results of the last run; and

    d. The location of the working directory.

2. Pass this information to the Sessionizer library through the functions: *addNetwork*, *addFile* and *initialize*.

3. Instruct the Sessionizer to commence the sessionizing routine via the *run* routine.

At this point the process control is handed over to the Sessionizer library. The first task in this process is to read in the results from the last sessionizing run to pre-populate the Active Sessions container. This will bring the sessionizing process up to the point where the last sessionizing process left off. Once the session containers have been pre-loaded, new network activity from the PCAP files can be read and processed.

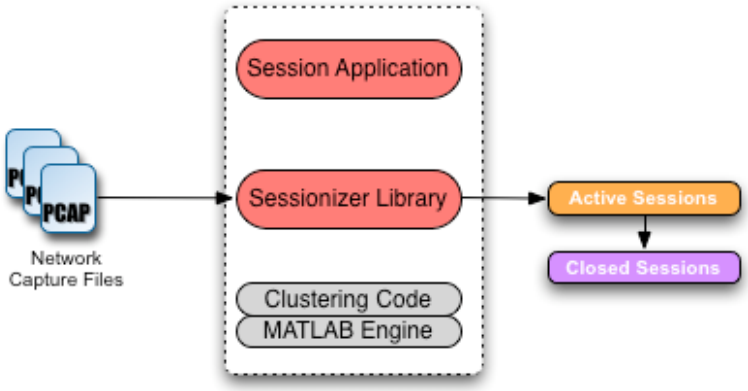

Figure 2: Conceptual Overview: Sessionizing Packet Data

Each PCAP file is processed in turn, with the results for any relevant (i.e. Ethernet based) traffic going into the container entry for the session to which it belongs. Should the network traffic reflect a new session, that is, where there is an existing session in the active container that corresponds to a closed or inactive session, the existing session data is moved to the closed session container and a new entry is made in the active session container for the new session. At the end of the sessionizing process, all network packets will have been processed and placed into a session, which resides in either the active session container or the closed session container.

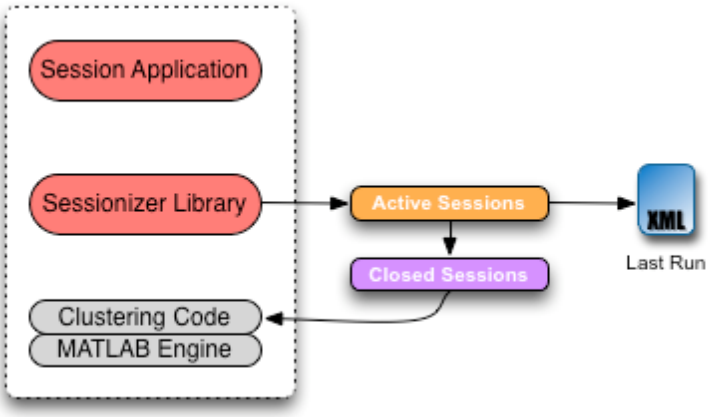

Figure 3: Conceptual Overview: Completing the Sessionizing Process

Finally, to complete the sessionizing process, all sessions in the active container are reviewed and

1. All expired sessions are moved to the closed sessions container; and

2. Copies all remaining sessions from the active session container to the closed session container, so that it contains all sessions found in the specified files.

At this point there is a divergence in the paths the two containers take. The contents of the active session container is written to the last run file, fulfilling its role to save the state of the sessionizing process and allowing this state to be restored at the next run of the sessionizing application.

The content of the container is sent to the Matlab engine. This is a multi-stage process, using the M-code functions available in the Matlab C API and exposed from the Matlab libraries. The process is performed as follows:

1. A Matlab session is created via *engOpen*.

2. The contents of the closed session container are re-created in Matlab data constructs, that is, in the form of Matlab compatible data matrices.

3. The clustering code (i.e. *cluster_anomalies.m*) is called inside the existing Matlab session.

4. The Matlab session is closed via *engClose*.

At this point the processing of the network session is handled exclusively by the content of the DRDC supplied Matlab code.

As can been seen, the sessionizing design incorporates a loopback process, where the active sessions at the end of the process become the input to the next sessionizing run.

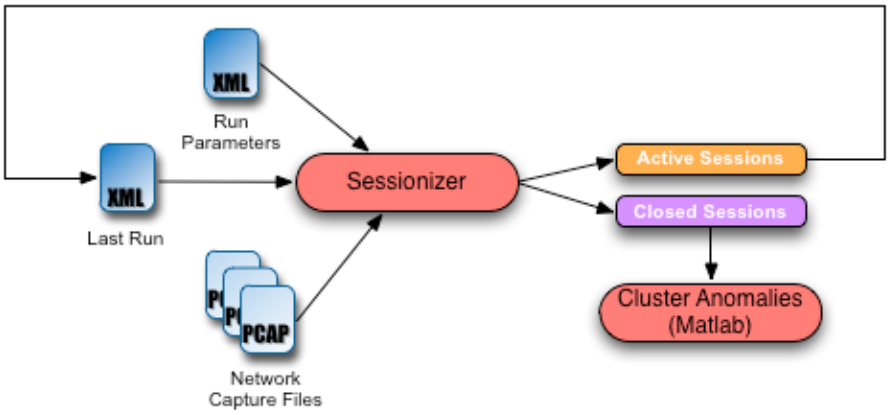

Figure 4: Conceptual Overview: Complete View

3.1 The Sessionizing Process In-Depth

The following diagram provides a more detailed data-centric view of the Sessionizing process.

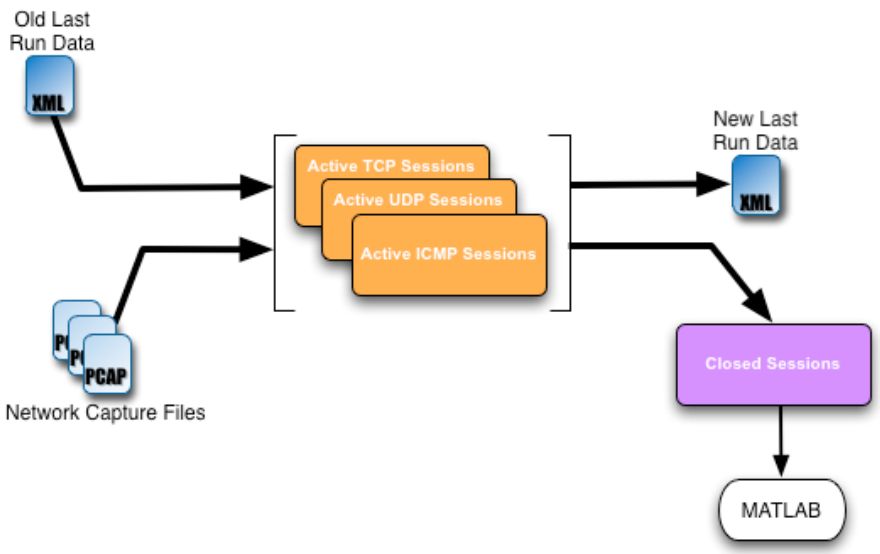

Figure 5: Data Centric View of the Sessionizing Process

The significant data flows are as follows:

1. During the last Sessionizing run, the active sessions were written to an XML file (see Annex A for XML file formats). This XML file represents the network state, that is, the state of all active session at the time the last Sessionizing run completed. In order to run a new Sessionizing process for the next set of data capture files, this network state must be restored. The XML file is read and the TCP, UDP and ICMP session containers are populated with the session data from this file.

2. The Network capture files are processed (packet by packet) and the TCP, UDP and ICMP session containers are expanded to include new sessions from the network capture data.

3. Once the packet processing is completed, any sessions that are closed are moved to the Closed Sessions container. These sessions will be processed by the Matlab routines.

4. The remaining active sessions are stored in the new Last Run XML data file.

The core Sessionizing logic is the code that places individual packets into the correct session. This logic is examined in greater detail in the following sections.

3.2 Packet Processing Logic

The core of the Sessionizer library is the manner by which each packet is processed. The specific activities that lead to the sessionizing of a packet will depend on IP protocol of the packet in question. In general, the processing logic follows the same pattern for each type of packet:

1. Ensure there is a session object in the Active Container for the given packet

2. Update the session object to reflect the current packet's data

The logic for handling each protocol described in the following sections.

3.2.1 TCP Packets

The following diagram describes how TCP packets are handled.

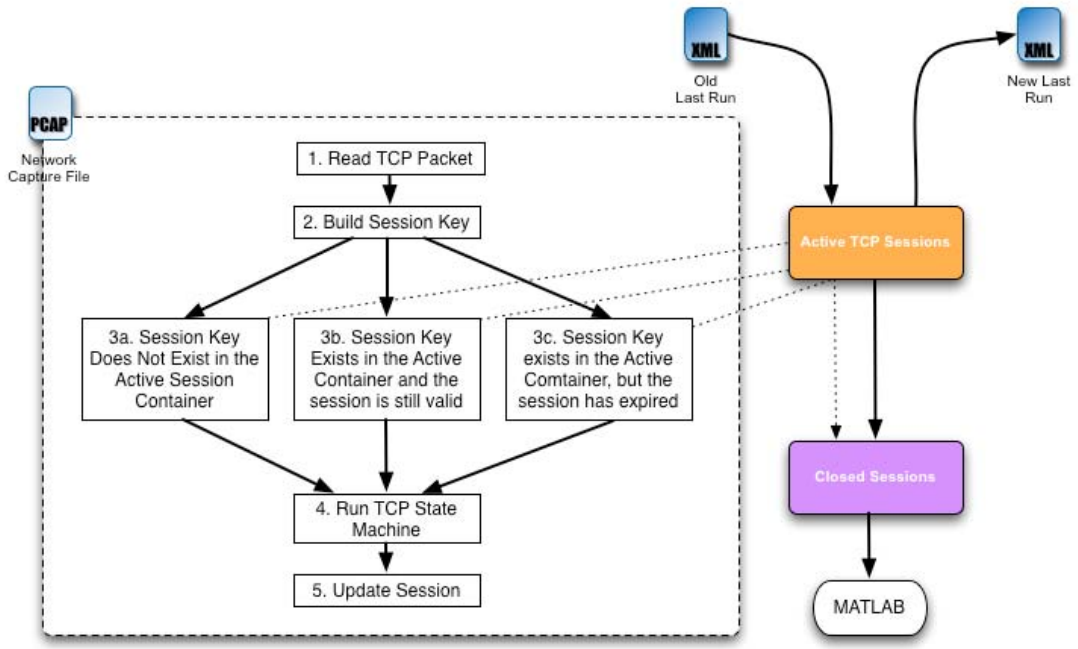

Figure 6: TCP Packet Handling Logic

Note that the processing content for the sessionizing activities is given. The last run data (*Old Last Run*) from the previous sessionizing process has been loaded into the Active TCP Session Container so that the current list of active TCP sessions has been populated. At this point the packets are read from the Network Capture File. Each stage in the processing of a given packet is described below:

1. Read TCP Packet: Each packet is read from the file sequentially. The TCP sessionizing process is only called for TCP packets.

2. Build the Session Key: The Active TCP Session container stores sessions as an STL Map container construct where **session keys** are mapped to **session objects**. A session key, for TCP sessions, is the IP/port pair (source/destination). The session object contains all relevant information about the currently active session for that key. To build a session key for the packet, the direction of the packet must be ascertained. The source (i.e. originating) IP for the packet is compared to the Internal IP address ranges. If the source IP is internal, that is, if the source IP falls within the network ranges that have been identified as Internal, the packet is outgoing, otherwise it is incoming. Session keys are always expressed as: Internal address, Internal port, External address, External port.

For outgoing packets, the session key is:

a. Source IP, source port, destination IP, destination port

For incoming packets the session key is:

b. Destination IP, destination port, source IP, source port

With the determination of the packet's session key, it is possible to determine whether there is an existing session for the given packet.

3. Placing a packet into a session. Once of three possible scenarios apply

   a. This is a new TCP session (stage 3a): If there is no session in the Active TCP session container for the session key, a new session object is created and this object is added to the container and mapped to the session key. This new session will be deemed to be in the LISTEN state and have zero payload size.

   b. This is a continuation of an existing session (3b): If there is a session in the Active TCP session container, a check will be made to ensure that the session is still active. Three criteria must be met for a session to be expired:

      i. the session must have exceeded the session lifetime (30 minutes since last packet for this session);

      ii. the session must be in the CLOSED state and the current packet must have the SYN flag

      iii. If neither of these conditions apply, the session is considered to be active

   c. An expired session (3c): If one of the above conditions does apply, the session has expired. The 'old' session is moved to the Closed Session container (dotted line) and a new session object (in the LISTEN state with 0 payload size) is created in the Active TCP Session container and mapped to the current session key.

4. TCP State Machine: At the end of this process, there will be an active session object in the TCP active session container that corresponds to the session key of the current packet. At this point, the session is put through the TCP state machine which will:

   a. Determine the event type based on the meaning of the TCP flags of the packet.

   b. Take the session's old state and determine the session's new state, based on the event.

   c. If appropriate, determine the type of erroneous transition if a non-standard event has taken place on the session.

5. Update the Session: The current session in the Active TCP session container will be updated to reflect the new session state and accumulated session payload.

This procedure will be repeated for each TCP packet in the network capture file. As previously stated, once all packets have been processed, the sessionizing process will continue by:

Moving any remaining packet from the Active container to the Closed Session container (solid line)

Creating a new 'last run' file to reflect the state of the network at the end of the capture process

Calling Matlab to cluster the anomalies.

3.2.2 UDP Packets

The following diagram describes how UDP packets are handled.

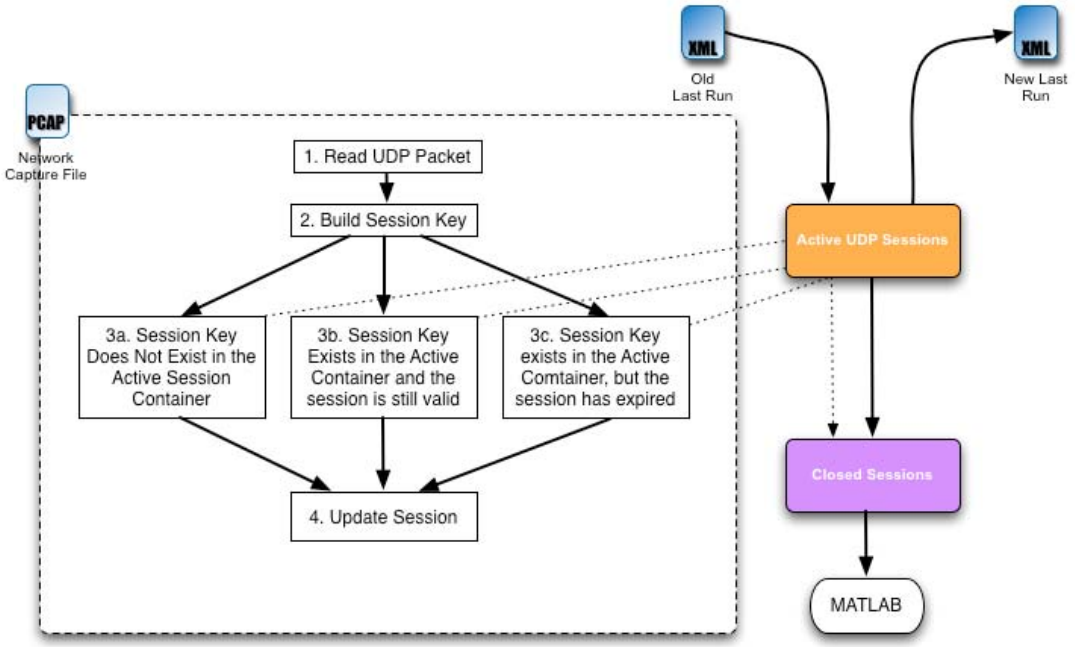

Figure 7: UDP Packet Handling Logic

The UDP data packet handling process matches the process for handling TCP packets. Since all IP packets are read sequentially from the same source, TCP and UDP data packet handling operates concurrently. It is also important to note that TCP and UDP sessions are stored in separate containers; however, the closed session container is shared by all IP sessions. The UDP packet handling process is as follows:

1. Read UDP Packet: Each packet is read from the file sequentially. The UDP sessionizing process is only called for UDP packets.

2. Build the Session Key: The Active UDP Session container stores sessions as an STL Map container (a different container than the one used for the TCP sessions) construct where **session keys** are mapped to **session objects**. A session key, for UDP sessions, is the IP/port

pair (source/destination). The session object contains all relevant information about the currently active session for that key. To build a session key for the packet, the direction of the packet must be ascertained. The source (i.e. originating) IP for the packet is compared to the Internal IP address ranges. If the source IP is internal, that is, if the source IP falls within the network ranges that have been identified as Internal, the packet is outgoing, otherwise it is incoming. Session keys are always expressed as: Internal address, Internal port, External address, External port.

For outgoing packets, the session key is:

a. Source IP, source port, destination IP, destination port

For incoming packets the session key is:

b. Destination IP, destination port, source IP, source port

With the determination of the packet's session key, it is possible to determine whether there is an existing session for the given packet.

3. Placing a packet into a session. Once of three possible scenarios apply:

   a. This is a new UDP session (stage 3a): If there is no session in the Active UDP session container for the session key, a new session object is created and this object is added to the container and mapped to the session key. This new session will have an initial zero payload size. UDP sessions have a single UDP state that does not change through the life of the session (since UDP is a connectionless protocol).

   b. This is a continuation of an existing session (3b): If there is a session in the Active UDP session container that matches the session key, a check will be made to ensure that the session is still active. Unlike TCP sessions, there is only one condition where an existing session expires and is no longer active:

      i. the session must have exceeded the session lifetime (30 minutes since last packet for this session);

   c. An expired session (3c): If the session has expired, the 'old' session is moved to the Closed Session container (dotted line) and a new session object (with 0 payload size) is created in the Active TCP Session container and mapped to the current session key.

4. Update the Session: The current session in the Active UDP session container will be updated by updating the session payload and setting this packet's time as the time of the last packet seen during this session.

This procedure will be repeated for each UDP packet in the network capture file. As described in the TCP packet handling section, once all packets (of all types) have been processed, the sessionizing process will continue, eventually sending all sessions to Matlab for clustering.

3.2.3 ICMP Request/Response Packets

The following diagram describes how non-error ICMP packets are handled.

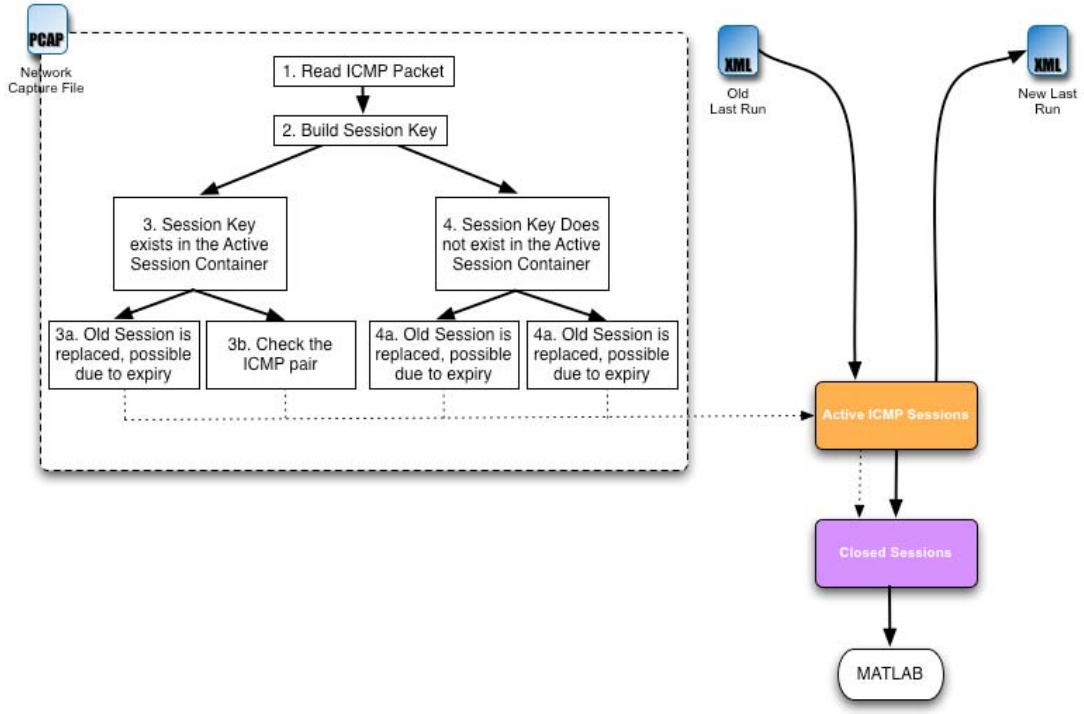

Figure 8: ICMP Packet Handling Logic

As would be expected, processing of ICMP packet has significant differences from the TCP and UDP packet handling routines. However, there are some similarities. Since all packets are read sequentially from the same source, TCP, UDP and ICMP data packet handling is done concurrently. As would be expected, TCP, UDP and ICMP sessions are stored in separate containers; however, the closed session container is shared by all sessions regardless of protocol. The ICMP packet handling process is as follows:

1. Read ICMP Packet: Each packet is read from the file sequentially. The ICMP sessionizing process is only called for ICMP request/response (non-error) packets, specifically: ECHO, TIMESTAMP, ADDRESS MASK, INFORMATION REQUEST and ROUTER SOLICIATION request or reply messages.

2. Build the Session Key: The Active ICMP Session container stores sessions as an STL Map container (a different container than the one used for the TCP and UDP sessions) construct where **session keys** are mapped to **session objects**. A session key, for ICMP sessions, is the IP pair (source/destination) as well as the ICMP *message request type* and *message type*. The

message type is the type of message in the current packet. The request type is the request equivalent for this message type. For example, a ping will have a message type of 8 (ECHO request) and a request type of 8 (ECHO request type). A response to this request would have a message type of 0 (ECHO reply) and a request type of 8 (ECHO request). In effect this gives us the ability to not only record the ICMP message type, but also the type of message that originated the ICMP request/response pair.

The session object contains all relevant information about the currently active session for that key.

To build a session key for the packet, the direction of the packet must be ascertained. The source (i.e. originating) IP for the packet is compared to the Internal IP address ranges. If the source IP is internal, that is, if the source IP falls within the network ranges that have been identified as Internal, the packet is outgoing, otherwise it is incoming. ICMP Session keys are always expressed as: Internal address, request type, External address, and message type. For outgoing packets, the session key is:

a. Source IP, request type, destination IP, my type

For incoming packets the session key is:

b. Destination IP, request type, source IP, my type

For example if an ECHO request went out to IP2 from IP1, the ICMP session key would be: (IP1, 8, IP2, 8) and the response would generate an ICMP session key of (IP1, 8, IP2, 0). As can be seen, the final key element (message type) cannot used in the matching algorithm and is included for consistency and future use.

With the determination of the packet's session key, it is possible to determine whether there is an existing session for the given packet.

3. A session exists in the ICMP Active container: In this case, there was a match in the container for the IP pair given a specific request type. Based on the content of the session object, the packet can be handled in one of two ways:

   a. This is a new ICMP session (stage 3a): If one of the following conditions is met, the original session has expired:

      i. the session must have exceeded the session lifetime (30 minutes since last packet for this session); or

      ii. the message is a request type (as opposed to a response type) and the existing session is in the PAIRED state. That is, a previous session went through a complete request/response cycle and we are now presented with a new request.

      In this scenario, the 'old' session is moved to the Closed Session container and a new session (in the QUERY state) is created using the ICMP session key.

b. This is a continuation of an existing session (3b): The session is still active and in the QUERY state (waiting for a response). We update the session time and (if the message is a response) set the session to the PAIRED state. If it is another request message, the state does not change.

4. There is no session in the ICMP Active session container for this ICMP session key: There are one of two outcomes:

   a. This is an ICMP request message (stage 4a): If this is a request message we create a new ICMP session for this ICMP session key and insert it into the Active container. The session's state is set to QUERY.

   b. This is an ICMP response message (stage 4b): If this is a response message we create a new ICMP session for this ICMP session key and insert it into the Active container. The session's state is set to ERROR as this represents an erroneous situation, namely, a spurious ICMP response without a corresponding request.

This procedure will be repeated for each ICMP packet in the network capture file. As described in the TCP and UDP packet handling section, once all packets have been processed, the sessionizing process will continue.

3.2.4 Lone ICMP Packets

The Sessionizer library also collects lone ICMP messages. These ICMP packets include all ICMP messages that do not have the request/response nature defined in the previous section. These messages will have a payload that references a previous IP packet. Lone ICMP packets are those ICMP messages that have, as a payload, a reference to a non-existent TCP or UDP session.

Since lone ICMP packets are not session related, they are sent immediately to the closed session container as shown in the following diagram.

The following diagram describes how non-error ICMP packets are handled.

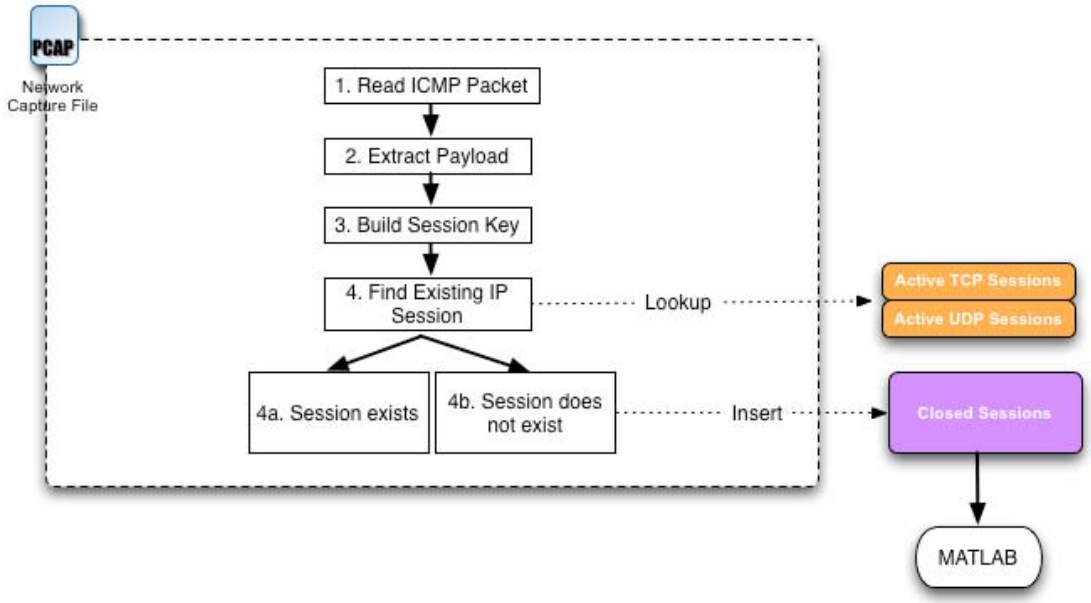

Figure 9: ICMP Error Packet Handling Logic

The lone ICMP packet handling process is as follows:

1. Read ICMP Packet: Each packet is read from the file sequentially. The ICMP sessionizing process is only called for non ICMP request/response packets, specifically, anything not of type: ECHO, TIMESTAMP, ADDRESS MASK, INFORMATION REQUEST and ROUTER SOLICIATION request or reply messages.

2. Extract Payload: The ICMP message will contain, as payload data, an IP packet. The IP header inside the payload will indicate which type of packet it is. Based on the protocol type, the source and destination ports of the packet in the payload data will be extracted.

3. Build the Session Key: Using the IP pair (source, destination) form the ICMP message and the ports from the extracted ICMP payload, a session key is created. In effect, we are trying to rebuild the session key for the TCP or UDP packet contained within the ICMP message payload. To build a session key for the packet, the direction of the packet must be ascertained. The source (i.e. originating) IP for the packet is compared to the Internal IP address ranges. If the source IP is internal, that is, if the source IP falls within the network ranges that have been identified as Internal, the packet is outgoing, otherwise it is incoming. Session keys are always expressed as: Internal address, Internal port, External address, External port. For outgoing packets, the session key is:

    a. Payload source IP, payload source port, payload destination IP, payload destination port

    For incoming packets the session key is:

b. Payload destination IP, payload destination port, payload source IP, payload source port

4. Find Existing Session: With the session key, we determine if there is a corresponding session in the active session container, examining the TCP and UDP containers as appropriate.

   a. Session exists for the ICMP payload (stage 4a): If the session exists, we consider this normal behaviour and the packet processing is complete.

   b. Session does not exist for the ICMP payload (stage 4b): If no session exists for the generated session key, this is a Lone ICMP message and it is added to the Closed Sessions container with a state definition of LONE_ICMP.

Note that since Lone ICMP messages are added directly to the Closed Session container, they are not read from or written to the last run file. This may be a necessary addition in future development.

4 High-Level Design

The following is a brief description of the approach taken by the sessionizer code. Each *pcap* file is read packet-by-packet, as required by the *pcap* library routines, and processed as they are read. Depending on the type of packet (TCP, UDP, or ICMP) different processing routes are called in order to ensure that the packet is handled appropriately. Sessions exist in one of 2 locations within the sessionizer application, depending on their state. The two possible states[1] in this context are:

Active sessions: packet data is still being collected for this session

Closed sessions: packet data for this specific session is completed

A session will move from 'active' to 'closed' based on an expiry time which is customized for each IP session type.

The sessionizer code uses the same definition of a unique session as the original Matlab implementation, namely, each IP/port pair (source and destination addresses with ports) is a unique session. Once a session is closed, a new network connection that uses these IP/port pairs is considered to be a new unique session.

The result of this approach is that, at any one time, there can be only one active instance for each IP/port pair combination. As network packets arrive they will either be applied to an existing active session or generate the creation of a new active session. In the last case, it is possible that a previous active session may be closed as part of the creation of a new active session.

Closed sessions are stored as a combined list of sessions. That is, the list of all sessions includes TCP/UDP and ICMP network session types.

The processing of the network capture files, therefore, populates the active and closed session containers. Once all packets have been processed, the closed sessions are passed to Matlab for further analysis whereas the active sessions are both send to Matlab and stored for future processing with the next set of packet data.

4.1 Module view

4.1.1 General System Interfaces

The following diagram provides a class and external object view of the system and interfaces.

[1] Note that the term state is used in several different contexts. In this context we are referring to the state of the session from the applications point of view (active or closed). This is not to be confused with the state of the session from an IP perspective (SYN, half-open, data transfer, etc…). It is possible for a session that is 'closed' from a network perspective to still be active from the applications point of view, for example, to handle the situation where extraneous packets arrive on a closed session. Conversely a session that is still open may be closed from the applications point of view if it exceeds a timeout value.

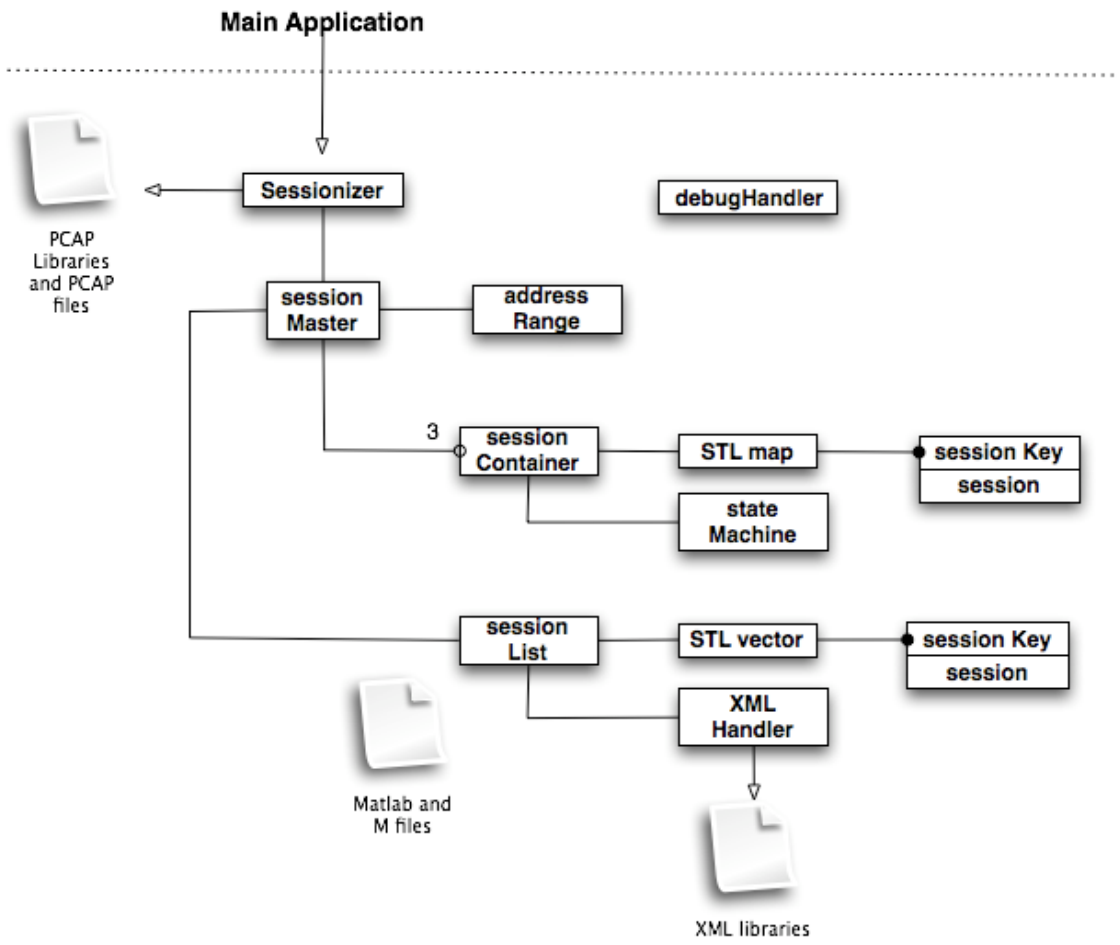

Figure 10: Class Diagram and External Objects

4.1.1.1 The Main Application

The sessionizer code has been written to be a module code component. That is, it is possible to write a software component that can utilize the functionality offered by the sessionizer code. The sessionizer code ships with a sample main application which:

1. Creates a sessionizer object.
2. Defines the target network.
3. Specifies the target *pcap* files.
4. Calls the sessionizer function.

This application demonstrates the main functions a calling application would use[2] to utilize the sessionizer solution. Another application, potentially a graphical interface, would make similar calls to the sessionizer code to perform sessionizing functions.

4.1.1.2 Matlab

The sessionizer code makes calls into the Matlab engine to run the clustering functions in the existing Matlab implementation. Specifically, the sessionizer makes the following calls:

1. Calls to start and stop the Matlab engine.

2. Calls to create, access and destroy Matlab-compatible data constructs

3. Calls to place data constructs inside the Matlab processing environment

4. Calls to run Matlab commands, including customized M-code functions

5. Calls to read the output from Matlab commands

The version of Matlab used by the sessionizer code is Matlab version 6.5 release 14 with service patch 2.

4.1.1.3 XML parser

The sessionizer code reads and writes active session data in an XML file format. The sessionizer uses the open source implementation of the tree and parser APIs from xmlsoft.org. Specifically, the XML library implementation version libxml2 release 2.6.30 was used in the development of the sessionizer code.

4.1.1.4 PCAP Library

The sessionizer solution uses version 0.9.8 of the pcap library.

4.1.1.5 Standard Template Library

The sessionizer code uses the Standard Template Library. STL is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. Using these container classes ensures that the resulting code benefits from optimized and high performance algorithms for accessing data objects.

[2] Note that the main application is a demonstrator for the sessionizer code. It is not part of the sessionizer code itself.

4.1.2 Sessionizer Modules

In this section, each of the main sessionizer modules is described in terms of its function and interfaces.

4.1.2.1 Sessionizer Class

The Sessionizer Class is the primary interface to the user. A user that wishes to utilize the sessionizer functionality will instantiate a Sessionizer object, populate the run parameters and initiate the sessionizing process. All these functions are made through the Sessionizer interface. The user does not need to interact with any other software module. As a result, the Sessionizer object is a singleton, that is, only one instance is created a part of a sessionizing session.

The Sessionizer class is also responsible for interacting with the *pcap* library. The *pcap* library consists of a set of functions to work with the TCPDUMP files, apply packet capture filters and initialize the processing of packets. The actual packet handling call-back functions are not part of the Sessionizer class, but exist in the Sessionzier software module.

4.1.2.2 SessionMaster

The SessionMaster class is also a singleton object as it is created by the Sessionizer object. The main purpose of this class to instantiate the container classes to hold active and closed sessions and call each container class in response to Sessionizer calls. SessionMaster creates 4 container objects: a TCP container object, a UDP container object, an ICMP container object and a sessionList object. The first three container objects are for active sessions and the sessionList object holds closed sessions.

4.1.2.3 SessionContainer

There are, in actual fact, two types of session containers: SessionContainer and icmpContainer. SessionContainer is a class that manages a container for active TCP and UDP sessions and icmpContainer is a class that manages a container for active ICMP sessions. The most important function of the container classes is to perform the data processing for handling new packets as they are read from the *pcap* file.

Active sessions are stored in an STL map container type. A map is a unique mapping of a specific class instance to another specific class instance. In the case of the sessionizer software, the mapping is from a sessionKey to a session. These two classes are described below.

4.1.2.4 SessionKey

This class defines the session key as it applies to sessions. This class has the following data elements:

Internal IP address;

Internal port;

Eternal IP address; and

External port.

This class is primarily a data holding object in that is does not perform any processing on the data itself. It is needed to create a key for the map container that is used by SessionContainer to hold active sessions.

4.1.2.5 Session

This class defines the session data for a given active session. This class has the following data elements:

Direction;

State;

FAPU direction;

Start time;

Stop time; and

Payload size.

This class has some functionality associated with processing sessions, specifically associated with updating session information based on information from new packets and the determination of when sessions have expired.

4.1.2.6 SessionList

The sessionList class, instantiated by sessionMaster, holds the closed sessions in an STL vector container class. This vector holds the sessionKey and session data for each closed session. Once all pcap packets have been processed, the sessionList will be called to send all closed and updated sessions to the Matlab engine for further analysis.

4.1.2.7 Supporting Classes

These classes provide ancillary support for the sessionizing process

addressRange: This class holds the definition of ‘internal’ addresses and allows the sessionizing code to determine if a given IP address falls within the internal address space. This function is critical for determining if an address is incoming (source address is not internal) or outgoing (source address is internal).

stateMachine: This class instantiates the rules for tracking the TCP network state transitions and the determination of the type of TCP error that has occurred: bad flags or transition error. The state machine will:

Map a set of network packet flags to a network event;

Transition a network state from the old state to a new state based on an observed event; and

Determine the type of network transition error that has occurred.

xmlHandler: This class will perform the calls to write the contents of the active session containers to an XML file and read an XML file to populate the session containers.

debugHandler: This is a supporting class to write information, error and debug message to a file for analysis.

4.2 Logic view

The following section describes, at a high level, the logic flow of the sessionizing software.

1. From a custom application, in which a user wants to perform a sessionizing run, the user creates a Sessionizer object and populates the internal address space and list of *pcap* files, through calls to *addNetwork* and *addFile*, respectively.

2. The user executes the run member on the Sessionizer object.

3. The Sessionizer object verifies that all pcap files exist and determines the last packet time for each file

4. The Sessionizer object loads the Last Run data, through a call to the sessionMaster object which calls the xmlHandler object to read and parse the XML data.

5. The Sessionizer object calls the *pcap_dispatch* call-back to read in each packet.

6. The pcap call-back function determines the packet type and calls the specific container object to process the packet. This will create, update, or close active sessions and populate the list of closed sessions as per the packet handling logic.

7. Once all packets have been processed, a call is made to close any active sessions that may have expired, moving them from the active container to the closed session list.

8. The list of active sessions is written to the XML Last Run file to save the state of the active sessions.

9. The closed session list (including updated sessions) is passed to Matlab and the clustering function is called to generate the list of scans.

5 Detailed Design

In this section, each class is described in detail

5.1 Sessionizer Object

5.1.1 Data Objects

The Sessionizer object is created by the calling application. It contains the following data objects:

Table 1: Sessionizer Data Members

Data Object Name	Data Type	Description
filter	STL string	Holds the text which will be passed to the pcap library in order to build the pcap filter. This object is populated through calls to *addNetwork* which also sets the data values in the *addressRange* object
directory	STL string	Holds the text that defines the current working directory. This object is set through the *initialize* call in this class.
pcapfiles	STL vector	This is a vector (or list) of filenames that will be processed by the sessionizing code. Entries in this list are added through the *addFile* call in this class.
master	sessionMaster object	The Sessionizer class creates a single sessionMaster object that will handle all operations related to active and closed sessions.
p	pcap_t *	This is a pointer to the pcap file object which is created during the sessionizing run. The pcap_t pointer is initialized in a call to *pcap_open* and is used in all subsequent pcap commands for that file.

5.1.2 Construction / Destruction

On creation, the Sessionizer object will make the sessionMaster object global in order to ensure that the sessionMaster object is accessible in non-class function calls, such as the *pcap_dispatch* call-back function.

5.1.3 Inline Functions

These functions are in the class header file and perform simple or pass through operations on the data.

Table 2: Sessionizer Inline Functions

Function Name / Definition	Description
Void addFile(char *)	This function takes a string representing a file to be processed by the sessionizer and adds it to the *pcapfiles* vector.

5.1.4 Member Functions

void initialize(char *, int)

This function takes, as input, a string representing the current working directory and integer which represents the desired debugging level (defaults to 0).

This function will perform the following operations:

Empty all class data structures

Set the current working directory

Establish the debugging module

5.1.4.1 addNetwork

void addNetwork(char *)

This function takes, as input, a string that defines a network space in the format: ***network:netmask***. For example, "10.20.30.40:255.255.255.0" will define anything in the address range of 10.20.30.* as internal.

This function will perform the following operations:

Call the sessionMaster object to add the network space to the addressRange object. This call will return host and mask in two separate STL strings.

Append the *host* and *mask* to the filter data element in the format.

The filter object will hold filter text in the format: "net *address* mask *netmask*".

5.1.4.2 run

int run(char *)

The run member function initiates the sessionizing process. It takes, as input, a string which represents the name of the XML file that holds the last run data. This function performs the following operations:

Verify that each of the *pcap* files in the *pcapfiles* exists;

Perform a call to sessionMaster, passing in the current directory and last run file name, in order to have the sessionMaster object load the last run XML data into the appropriate SessionContainer.

For each *pcap* file in the *pcapfiles* list

- Determine the time of the last packet in the list
- Compile and set the pcap filter
- Call the *pcap_dispatch* call-back function for each packet entry
- Close the *pcap* file

Call the sessionMaster to close any active sessions that have expired

Perform a call to sessionMaster, passing in the current directory and last run file name, in order to have the sessionMaster object save the list of active sessions into an XML data representing the last run data.

Call the sessionMaster object to run the clustering Matlab analysis on the closed sessions.

5.1.5 Other Notes

5.1.5.1 processPackets

void processPackets(u_char, pcap_pkthdr *, u_char *)

This function is in the Sessionizer class file, but is not part of the class[3], as a result it is documented here. The declaration of the call-back function is defined by the *pcap* library. This function is called by the *pcap* library in response to the call to the *pcap_dispatch* function performed in the Sessionzier class member function *run*.

This function will determine the type of IP packet that is being processed and call the appropriate container function to process the entry.

- If the packet is a TCP packet, call the sessionMaster object *processEntry* function passing in the packet header, the IP header and the TCP header;
- If the packet is a UDP packet, call the sessionMaster object *processEntry* function passing in the packet header, the IP header and the UDP header;
- If the packet is an ICMP packet and is a non-error ICMP message, call the sessionMaster object *processEntry* function passing in the packet header, the IP header and the ICMP header
- If the packet is an ICMP error message, extract from the message data the IP header that originated the error and the associated data. Call the sessionMaster object *processICMPerror* function passing in the packet header, the IP header, the ICMP header, the error generating IP header and the associate data.

This call back function is not able to return an error code, therefore, all error messages are written to the debug file[4].

[3] It is notoriously difficult to have call-back functions as a member of a class due to the C/C++ name declaration discrepancies. This can be revisited at a later date.
[4] The debug file is stored in the working or scratch directory which is defined in the parameters file.

5.2 sessionMaster Object

5.2.1 Data Objects

The singleton SessionMaster object is created by the Sessionizer object. It contains the following data objects:

Table 3: sessionMaster Data Objects

Data Object Name	Data Type	Description
tcpSessions	sessionContainer object	This is one of three objects that hold active sessions. This object holds the TCP sessions.
udpSessions	sessionContainer object	This is one of three objects that hold active sessions. This object holds the UDP sessions.
icmpSessions	icmpContainer object	This is one of three objects that hold active sessions. This object holds the ICMP sessions.
sessions	sessionList object	This object holds all closed sessions
ranger	addressRange object	This object provides the functionalist for storing internal network ranges and providing a means to identify whether an IP address in within these ranges.

5.2.2 Construction / Destruction

On creation, the sessionMaster object will provide each active session container with a reference to the addressRange object and the sessionList object.

5.2.3 Inline Functions

These functions are in the class header file and perform simple or pass through operations on the data.

Table 4: sessionMaster Inline Functions

Function Name / Definition	Description
void processEntry(pcap_pkthdr*, iphdr*, tcphdr*)	This passthrough function calls the *tcpSession* object *updateEntry* function with these same parameters
void processEntry(pcap_pkthdr*,iphdr*, udphdr*)	This passthrough function calls the *udpSession* object *updateEntry* function with these same parameters
void processEntry(pcap_pkthdr*, iphdr*, icmphdr*)	This passthrough function calls the *icmpSession* object *updateEntry* function with these same parameters
void addRange (char *,	This passthrough function calls the *ranger* object *extract*

string&, string &)	function with these same parameters
Void runMatlab()	This passthrough function calls the *sessions* object *runMatlab* function.

5.2.4 Member Functions

5.2.4.1 processICMPerror

void processICMPerror(pcap_pkthdr*, iphdr*, icmphdr*, iphdr *, u_char *)

TCP, UDP and ICMP (non-error) packets generate session data. ICMP error messages are different in that they are isolated events. Therefore, rather than having a separate container object to handle these packets, they are handled right away in the master object.

The function performs the following functions:

Verify that the error payload holds real data

Extract the source and destination ports from the ICMP message's error data

Create a IP/port sessionKey object using the internal and external addresses and the ports defined above (key1)

Create another IP/port sessionKey object using the internal and external addresses using the ICMP type as source port and the ICMP code as the destination port. (key2)

Determine if there is an entry for key1 in the active sessions list (TCP or UDP)

- If such and entry is found create a new 'closed' session for this ICMP error message

5.2.4.2 expireSessions

void expireSessions(time_t& lastTime)

This is another passthrough function that calls each of the container objects to have them expire their sessions based on the provided time value. This time value represents the time of the last packet.

5.2.4.3 addUpdatedSessions()

void addUpdatedSessions()

This is another passthrough function that calls each of the container objects to have them update their list of updated sessions.

5.2.4.4 loadLastRun

int loadLastrun(String &, char *)

This function is called by the Sessionizer object in order to load the XML data in the last run file, identified by the directory and filename passed in to this call. This function will:

Create a new xmlHandler object

Call the *loadLastRun* function of the xmlHandler object, passing in the active sessions container objects and the name of the XML file.

5.2.4.5 saveLastRun

int saveLastrun(String &)

This function is called by the Sessionizer object in order to save the active sessions into an XML data file representing the last run data. The filename is provided in the call to this function. This function will:

Create a new xmlHandler object

Call the *saveLastrun* function of the xmlHandler object, passing in the active sessions container objects and the name of the XML file.

5.3 sessionContainer Object

5.3.1 Data Objects

The Sessionizer object is created by the Sessionizer object. It contains the following data objects:

Table 5: sessionContainer Data Members

Data Object Name	Data Type	Description
activeList	STL map	This object is an associative container that will map a key to a value object. In the case of the sessionContainer, the key is a sessionKey object and the value object is a session object[5].
ranger	addressRange *	A pointer to the addressRange object created by the sessionMaster
theList	sessionList *	A pointer to the sessionList object created b the sessionMaster

There is also a stateMachine object that is created within the sessionContainer source file

5.3.2 Construction / Destruction

On creation, the sessionMaster object will set the ranger pointer to NULL and initialize the stateMachine object.

[5] The definition of the STL map is map<sessionKey, session>. Using this construct, it is possible to provide a sessionKey object and retrieve the associated session. Note that this type of map is unique, that is, duplicate sessionKey objects cannot co-exist within the map.

5.3.3 Inline Functions

These functions are in the class header file and perform simple or pass through operations on the data.

Table 6: sessionContainer Inline Functions

Function Name / Definition	Description
Void setRange(addressRange * , sessionList *)	This function sets the value of the addressRange object and the sessionList and is called by the sessionMaster object.
Int insertEntry(sessionKey&, session *)	This function adds the sessionKey and session objects to the list of active sessions by adding them to the STL map container.
void processEntry(pcap_pkthdr*, iphdr*, icmphdr*)	This passthrough function calls the *icmpSession* object *updateEntry* function with these same parameters
void addRange (char *, string&, string &)	This passthrough function calls the *ranger* object *extract* function with these same parameters
Void runMatlab()	This passthrough function calls the *sessions* object *runMatlab* function.

5.3.4 Member Functions

5.3.4.1 doesKeyExist

int doesKeyExist(sessionKey *)

This function returns 0 if the sessionKey exists in the active list, non-zero otherwise.

5.3.4.2 findEntry

iterator findEntry(sessionKey *, time_t&, char &)

Given a sessionKey object, this function determines if the sessionKey exists in the activeList. If it does not exist, the function returns NULL. If it does exist, it verifies if the session has expired, based on the time passed in the time_t parameter. If the session has expired, it will move the session to the closed container and set a flag indicating that a new entry in the session is to be created.

A call to session's *isValid* function will determine if the session has expired.

This is an important point. This function is called for each packet and the time of the packet is passed into this function. If there has not been activity in this session for a while, the time difference between the time for this packet and the time of the last packet o this session will be large. If it is too large, we consider this to be a new session

5.3.4.3 updateEntry (TCP version)

int updateEntry(pkthdr *, iphdr *, tcphdr *)

This function is called by sessionMaster is response to the receiving of a TCP packet. It will perform the following functions:

Determine the direction of the packet by passing in the source address to the addressRange object's IPinRange function.

Build a sessionKey object based on the IP packet data (if it is incoming the source addr is external and the destination address is internal, otherwise, the converse is true)

Call the findEntry function to determine if this is a new session.

If this is a new session

- Create and populate a new session object and add the sessionKey/session map to the container of active sessions

If this is an expired session

- Create and populate a new session object and assign the new session object to the sessionKey map entry

If this is an active session

- Get a pointer to the active session data

Update the TCP session state based on the previous state and the flags of the packet

Update the stop time and payload size for the session

5.3.4.4 updateEntry (UDP version)

int updateEntry(pkthdr *, iphdr *, udphdr *)

This function is called by sessionMaster is response to the receiving of a UDP packet. It will perform the following functions:

Determine the direction of the packet by passing in the source address to the addressRange object's IPinRange function.

Build a sessionKey object based on the IP packet data (if it is incoming the source addr is external and the destination address is internal, otherwise, the converse is true)

Call the findEntry function to determine if this is a new session.

If this is a new session

- Create and populate a new session object and add the sessionKey/session map to the container of active sessions

If this is an expired session

- Create and populate a new session object and assign the new session object to the sessionKey map entry

If this is an active session

- Get a pointer to the active session data

Update the stop time and payload size for the session

5.3.4.5 expireSessions

void expireSessions(time_t&)

This function is called by sessionMaster after all packets have been analyzed; it reviews the list of active sessions to determine if any have expired, based on the time of the last packet in the *pcap* files which has been passed in to the call.

For each active session, get a pointer to the session data

- If the sessions has expired, based on a call to the session object's sessionExpired function

  Move the active session to the closed session container

  Remove the session from the active list

5.3.4.6 addUpdatedSessions

void addUpdatedSessions()

This function is called by sessionMaster after all packets have been analyzed; it adds all remaining sessions to the closed list, even if they are still active, but does not remove them from the active list

For each active session, get a pointer to the session data

- Move the active session to the closed session container

5.3.4.7 writeSessions

int writeSessions(char *, xmlNodePtr)

This function is called by the xmlHandler. The handler passes a node pointer object[6] to the container and the container object writes each of the objects to the output file.

For each active session, get a pointer to the session data

- Create a new child node of the parent node provided above

[6] This is a data construct of the libxml2 library tree API

- Assign each of the session characteristics to the node as properties (IP/port pairs, session data… 9 properties in all)
- Close the child node

5.3.5 Other Notes

5.3.5.1 sessionCompare

An STL map needs a function to help it optimize the finding and sorting of map element. In short, it needs a method to determine which sessionKey object comes before another sessionKey object[7]. A sessionCompare structure has been created in the sessionContainer header file which defines the method by which sessionKeys are compared. The method by which this is done is very important. In essence the logic is as follows, given that a sessionKey consist of an internal address, an internal port, an external address, and an external port:

Comparing 2 sessionKey objects (A and B):

If A and B's internal addresses are not the same

If A's internal address is less than B's, return 1, otherwise return 0;

If A and B's internal ports are not the same

If A's internal port is less than B's, return 1, otherwise return 0;

If A and B's external addresses are not the same

If A's external address is less than B's, return 1, otherwise return 0;

If A and B's external ports are not the same

If A's external port is less than B's, return 1, otherwise return 0;

return 0;

5.4 icmpContainer Object

The icmpContainer object is almost identical to the sessionContainer object, but whereas the sessionContainer object is used for TCP and UDP sessions, the icmpContainer object is used for ICMP (non error) messages. The only significant difference is that although is uses the same sessionKey as the STL map key, port numbers have no significance in ICMP messages. Therefore the internal port object is used to store the ICMP message type and the external port object is used to store the ICMP message code.

[7] Probably because of the tree-like representation for associative containers.

As a result, the sessionCompare function described in the previous section has been replaced by an icmpCompare object which uses the addresses only.

5.5 sessionKey Object

5.5.1 Data Objects

The sessionKey object is created in response to the need to create a new active session object. It is generally created by the SessionContainer object and contains the following data objects, which (taken as a whole) uniquely identify a session:

Table 7: sessionKey Data Members

Data Object Name	Data Type	Description
i_addr	unsigned long	This is the IP address of the internal component of the session in decimal format
i_port	unsigned short	This is the port number of the internal component of the session.
e_addr	unsigned long	This is the IP address of the external component of the session in decimal format
e_port	unsigned short	This is the port number of the external component of the session.

5.5.2 Construction / Destruction

There is a custom constructor that will populate the sessionKey object based on and IP/port pair passed in.

There is also a custom constructor that will populate the sessionKey object based on and IP/port pairs passed in string format. The constructor will convert the data to their unsigned equivalents and populate the data objects. This is used for when data objects are read in from the XML file.

5.5.3 Inline Functions

These functions are in the class header file and perform simple or pass through operations on the data.

Table 8: sessionKey Inline Functions

Function Name / Definition	Description
unsigned long getInternalIP()	Returns the internal IP address

5.5.4 Member Functions

5.5.4.1 setData

void setData(unsigned long, unsigned short, unsigned long, unsigned short)

This function sets the sessionKey data values.

5.6 session Object

5.6.1 Data Objects

The session object is created in response to the need to create a new active session object. It is generally created by the SessionContainer object and contains the following data objects, which describe the state of the session:

Table 9: Session Data Members

Data Object Name	Data Type	Description
State	int	The state of the session (half open, data transfer, etc…)
fapuDirection**	int	In which direction was the first FAPU seen
Direction*	int	In which direction was the first packet
start*	time_t	Which was the time of the first packet
stop	time_t	Which was the time of the last packet
payloadsize	unsigned long	Cumulative size of the traffic on the session

*- these entries are set once at the beginning of the session and are not changed

**- this entry is set once during the session and is not changed

5.6.2 Construction / Destruction

There is a custom constructor that will populate the initial state of the session (start time, direction, state).

There is also a custom constructor that will populate the session object based on arguments passed in string format. The constructor will convert the data to their unsigned equivalents and populate the data objects. This is used for when data objects are read in from the XML file.

5.6.3 Inline Functions

These functions are in the class header file and perform simple or pass through operations on the data.

Table 10: Session Inline Functions

Function Name / Definition	Description
void setDirection(char)	Sets the direction data object of the session
char getDirection()	Gets the direction data object of the session
void setState(int)	Sets the state data object of the session
int getState()	Gets the state data object of the session
Time_t getStartTime()	Gets the start time data object of the session
Time_t getStopTime()	Gets the stop time data object of the session
Unsigned long getPayload()	Gets the payload size data object of the session

5.6.4 Member Functions

5.6.4.1 update

void update(time_t&, unsigned long&)

This function sets the stop time of the session and increments the payload size, based on the parameters passed in to the function call.

5.6.4.2 isValid

int isValid(time_t &)

This function returns zero if the session is active (last time + 1800 seconds is less that the time passed in). Returns non-zero otherwise.

5.6.4.3 updateState

void updateState(int)

This function sets the objects state data object to the value determined by the state machine after we provide the current state and the flags for the current packet.

5.6.4.4 sessionExpired

int sessionExpired(time_t&)

This function returns zero if the session is still active and non-zero otherwise. For some sessions, certain states expire earlier. For example the S, SA states expire after 5 minutes, others expire after 20 minutes.

5.7 sessionList Object

5.7.1 Data Objects

The sessionList object is created by the sessionMaster object and is used to stored closed sessions. Whereas the sessionContainer objects hold active sessions in an STL map where the map uses a sessionKey as the key and a session object as the value, the sessionList stores sessionKey and session information in an STL pair. In an STL pair, two objects are stored as a single object. These pairs are held in an STL vector.

Table 11: sessionList Data Members

Data Object Name	Data Type	Description
Sessions	STL vector of pairs	This is the STL vector that holds the sessionKey, session pairs.

5.7.2 Construction / Destruction

The constructor ensures that the sessions vector is empty.

5.7.3 Inline Functions

These functions are in the class header file and perform simple or pass through operations on the data.

Table 12: sessionList Inline Functions

Function Name / Definition	Description
void addSession(sessionKey&, session *)	Adds the sessionKey and session objects to the vector or paired objects.
int getCount()	Returns the length of the vector object.

5.7.4 Member Functions

5.7.4.1 runMatlab

int runMatlab()

This function will call the Matlab engine to accept the list of closed sessions and process the clustering function (as an M-file). In order to process Matlab functions on data, this data must be translated into Matlab equivalents.

The function performs the following operations:

Open a Matlab session via a call to *engOpen* (a Matlab function)

Create a Numeric Array which has 9 columns (to holds the session data) and is sufficiently large to hold all closed sessions.

For each session in the sessionList

- Copy the session data (IP/port pair, state, direction, start/stop time, payload size)

Send the data to the Matlab engine

Call the clustering function (cluster_anomalies3)

Read the output of the function to a local variable and write the results to the output file.

Clean up the Matlab data structures

Shut down the Matlab engine

5.8 StateMachine Object

5.8.1 Data Objects

The stateMachine object is a helper object that performs all state transition functions and error handling for tracking TCP sessions. There are three types of state transition data types:

1. A mapping of TCP flags to event type
2. A transition table taking oldstate and the event type to determine the new state
3. A transition table taking the previous state and event and generating an error value

Table 13: stateMachine Data Members

Data Object Name	Data Type	Description
Events	STL map	This data object maps TCP flags to an event type.
Transition	STL vector of vectors	This 2 dimensional array, formed of vectors, hosts the state transition table
Errormap	STL vector of vectors	This 2 dimensional array, formed of vectors, hosts the error mapping information

5.8.2 Construction / Destruction

The default constructor ensures that both the transition and error vectors have dimensions of 7x7 and are initialized to transition to the error state.

5.8.3 Inline Functions

These are no inline functions.

5.8.4 Member Functions

5.8.4.1 initialize

void initialize()

This function must be called prior to using the state machine. It populates the state machine objects with the transition data.

5.8.4.2 getEvent

int getEvent(int)

This function takes TCP flags and returns the event type.

5.8.4.3 getNextState

int getNextState(int, int)

This function the old state and TCP flags and returns the next state, based on the transition. The following rules apply:

If the old state is an error state, return the original old state

If the new state is an error state, calculate and return the error state based on the errormap transition.

5.9 xmlHandler Object

5.9.1 Data Objects

The xmlHandler object is used to read last run files and populate the sessionContainer objects with active sessions from the last sessionizing run and write the last run data from active sessions to a file to be used in the next sessionizing run. The output file is in XML format. This object uses the libxml2 tree API to perform parsing and traversing functions.

5.9.2 Construction / Destruction

There are no custom constructors.

5.9.3 Inline Functions

There are no inline functions.

5.9.4 Member Functions

5.9.4.1 loadLastRun

int loadLastRun(sessionContainer &, sessionContainer &, icmpContainer &, string& char *)

This function traverses the XML file defined by the name and directory that were passed in to the function call. As sessions are read, they are passed to the appropriate sessionContainer (TCP, UDP or ICMP) and added to the container object.

5.9.4.2 saveLastRun

int loadLastRun(sessionContainer &, sessionContainer &, icmpContainer &, string&)

This function creates a new XML document where there is a section for each session type (TCP, UDP, and ICMP). Once the parent node is created, each sessionContainer is called in turn to write the data from each container to the XML file. Once all sessions are written to the XML document, the document is written to disk.

5.10 addressRange Object

5.10.1 Data Objects

The addressRange object has two functions: to store the list of networks that define which is 'internal' to the session run and to provide a means by which an IP address can be tested to see if it falls into that range.

Table 14: addressRange Data Members

Data Object Name	Data Type	Description
Range	STL vector of pairs	This is a list of paired objects where each pair consists of a network and a netmask.

5.10.2 Construction / Destruction

There are no custom constructors.

5.10.3 Inline Functions

These functions are in the class header file and perform simple or pass through operations on the data.

Table 15: addressRange Inline Functions

Function Name / Definition	Description
int addRange(char *)	Calls the non-inline double string version of addRange

5.10.4 Member Functions

5.10.4.1 extract

void extract(char *, string &, string &)

This function will break the string into two constituent STL strings hosting the address and netmask that defines the internal address space.

5.10.4.2 addRange

int addRange(string &, string &)

This function will create the two components of the addressRange entry: a network and the netmask. The network is derived by ORing the network with the netmask. The netmask is passed in the argument list. Note that these parameters must first be converted to their integer equivalents.

The network and netmask are then added, as a pair, to the range vector of pairs.

5.10.4.3 IPinRange

int addRange(unsigned long)

This function returns 0 is the address that is passed in the argument list is in the network range for one of the entries in the range vector. For each entry in the range, the netmask is applied to the passed address and compared to the network. If there is a match, the address is in the network space.

6 Outstanding Issues

The following functions have not been implemented in the Sessionizer library as of this writing.

1. The use of sequence numbering to identify spurious socket change.

2. The determination of the direction of the first FIN flag and the modification of the state machine to process state changes based on the direction of this FIN packet.

Additionally, there should be a more formal statement of the interface needs:

3. What is the complete list of run parameters to be set in the XML file and passed to the Sessionizer library? What Sessionizer behaviour should be dynamically set through the parameter file?

4. How should the output from the Matlab analysis be handled? Ideally, this information should be presented visually in a dynamic web application.

References

[1] Daniel Viellard, "The XML C parser and toolkit of Gnome" (Online), http://www.xmlsoft.org/ (Access date: 21 Nov 2008).

This page intentionally left blank.

Annex A XML File Formats

Run Parameter XML File Sample:

```
<run scratch="/home/work/scratch" debug="1">
 <internal>
  <addr host="128.222.0.0" mask="255.255.0.0" />
  <addr host="131.223.0.0" mask="255.255.0.0" />
  <addr host="131.224.0.0" mask="255.255.0.0" />
  <addr host="131.225.0.0" mask="255.255.0.0" />
 </internal>
 <scans>
  <file name="/home/work/data/file1" />
  <file name="/home/work/data/file2" />
 </scans>
</run>
```

Last Run Parameter XML File Sample:

```
<LastRun>
  <Sessions>
    <tcp>
      <Session intaddr="2150301699" intport="35663"
extaddr="3726912677" extport="80" direction="0" fapu="0"
state="10" start="1156276442" ustart="475652" stop="1156276442"
ustop="475652" size="0"/>
    </tcp>
    <udp>
      <Session />
    </udp>
    <icmp>
      <Session />
    </icmp>
  </Sessions>
</LastRun?
```

This page intentionally left blank.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Bell Canada - ICT
Floor 11, 333 Preston St.
Ottawa, ON K1S 5N4

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

UNCLASSIFIED

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

Slow scan detector: Sessionizer software design

4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used)

Henderson, G.

5. DATE OF PUBLICATION (Month and year of publication of document.)

April 2009

6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)

56

6b. NO. OF REFS (Total cited in document.)

0

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Contract Report

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – Ottawa
3701 Carling Avenue
Ottawa, Ontario K1A 0Z4

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)

15bo01

9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)

W7714-071029

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)

10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

DRDC Ottawa CR 2008-285

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

Unlimited

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.)

Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

The Network Information Operations Section at DRDC Ottawa has developed a proof-of-concept low-profile scan detection engine using the MATLAB programming environment. It reads network traffic from pcap format files using custom-built interfaces to the pcap libraries and separates the traffic into connections (TCP) or sessions (UDP/ICMP). Anomalous connections are then processed to identify scans, including those that are slow and/or distributed.

In this work, a software framework was designed where the traffic sessionizer module is written in C++ to increase the processing speed for use in an operational environment. The sessionizer module interfaces with MATLAB to perform the scan detection. The system and sessionizer software are described in detail herein.

La section Opération d'information de réseau de RDDC Ottawa a développé un moteur de détection de balayages discret au moyen de l'environnement de programmation MATLAB. Il lit le trafic réseau de fichiers en format pcap au moyen d'interfaces personnalisées avec les bibliothèques pcap et il répartit ce trafic entre des connexions (TCP) ou des sessions (UDP/ICMP). Les connexions anormales sont ensuite traitées afin de relever les balayages, y compris les balayages lents et/ou les balayages répartis.

Au cours de ce travail, un cadre logiciel a été conçu dans lequel le module sessionizer de trafic est écrit en C++ afin d'accroître la vitesse de traitement en vue de son utilisation dans un environnement opérationnel. Le module sessionizer est interfacé avec MATLAB pour effectuer la détection des balayages. Le logiciel système et celui du sessionizer sont décrits en détail dans ce document.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

scan detection, network traffic analysis

Defence R&D Canada

Canada's leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

www.drdc-rddc.gc.ca