Sniffer Detector Report

Stéphane Grundschober

June 1998

IBM Research Division
Zurich Research Laboratory
Global Security Analysis Lab

Abstract

In the current Internet world, many protocols are by nature insecure. A program able to log sessions by monitoring to a network interface may gather sensitive information and compromise the security of a network. Such a program is called a sniffer.

This report describes a sniffer detector system that is based on the spreading of bait.

We first describe what a sniffer is and what threats it poses. With this knowledge in mind, we can define the requirements of the system. A prototype has been developed. We describe the architecture chosen, in addition to implementation aspects. This prototype has been tested in a real-size environment between the IBM Zurich Research Laboratory (Switzerland) and IBM La Gaude (France).

Table of Contents

The only way to get things out of other people is by offering them something first.

Matt Ridley, Time, Special Issue Winter 97/98

A liar should have a good memory.

Quintilian (42-118 A.D), Institutiones Oratorice, IV, 2, 91

1. Introduction

Computer networks are shared communication channels. This means that the network interface of a computer on an ethernet sees all the packets transmitted on that segment. Each packet has a header indicating the recipient of the packet. Only the machine with that address is supposed to accept the packet.

However, it is possible to set an interface to accept any packet. We say it is in *promiscuous* mode.

A program specifically designed to put the network interface in promiscuous mode, to parse and log critical information about insecure protocols is called a *sniffer* [1]. It constitutes the malicious use of information, which is a security problem.

These malicious sniffers pose a threat to the Internet world, as many protocols currently used on the Internet are insecure. For example, Telnet or FTP protocols send passwords over the network in clear text. SNMP community names are also sent in clear text. A sniffer able to log specifically this information can compromise the security of a complete network. Indeed, the sniffer's owner can reuse the gathered login and password pairs to gain unauthorized access.

It is not straightforward to install a sniffer. There are three possible protections:

A software one: Only super users (on a Unix system) have the authorizations to put an interface in promiscuous mode. This feature may not be implemented in other operating systems. On a token ring, if an interface is set in promiscuous mode, it has to advertise this state to the ring master, which could disable it in response.

A physical one: An intruder should not have access to the physical wires of the ethernet bus. Thus he cannot install his own computer sniffer.

A topological one: A switched network transmits the packets only to the recipient computer. Although it is much better than a shared wire, it does not protect the network from a sniffer installed on a router.

Nevertheless, experience has shown that using a sniffer is a very successful method commonly used by crackers [2]. To prevent such intrusion, one could of course avoid the utilization of such insecure protocols. Unfortunately, this may not always be possible for the following reasons:

- no alternative solution may exist,

- it might be too complicated to change the protocol,

- it might be illegal to change the protocol because encryption is restricted.

Therefore, it is of interest to find a way to detect the utilization of a sniffer.

2. Sniffer operation

2.1 Phase 1: Collect information

One thing to keep in mind is that packet capture performed by a sniffer is invisible from a network point of view. This is interesting for the sniffer's owner because it lets him gather information about the network with a low probability of being discovered by managing tools.

What information can be collected by the sniffer on a local ethernet segment?

1) Host IP addresses
   It lets the intruder know what the existing hosts are without dumping the local DNS (query to list all records).

2) Host functionalities
   By analyzing what kind of traffic is exchanged, the sniffer is able to find, for example, which host is a file server, which is the administrator host, and so on.

3) Telnet and FTP sessions
   In both protocols, the password requested by a remote host is transmitted in clear text over the network. The sniffer is able to capture login and password pairs.

4) Rlogin sessions
   It lets the sniffer know which user from which host has rlogin access to which host.

5) SNMP community names
   The SNMPv1 authentication is very weak. Each SNMP-UDP packet contains in clear text a community name which is matched with a local version in the controlled equipment. Knowing the community name for the read/write access allows one to control the network equipment.

2.2 Phase 2: Attack

In some attacks, IP spoofing may be necessary. It is useful either to hide the intruder's host or to go through weak identification schemes based on the source IP address. The IP spoofing technique is easy to perform if the sniffer is somewhere on the route from the target host to the impersonated host (otherwise, one must do TCP sequence number guessing, which is not trivial [3], [4]). In fact, all the replies will be seen by the sniffer, which will then be able to create the required answers.

Once a sniffer has got the information he wanted, there are some simple attacks he can make:

1) Password reuse
This the most straightforward attack. Once the sniffer has got the login and the password of a user by monitoring a Telnet or FTP session, it is trivial to reuse them.

2) Rlogin sessions
By impersonating a user on a host, the intruder can get a shell or execute commands on a remote host.

3) SNMPv1
Once the community name is known, it is straightforward to use it to modify routing table entries of routers, for example. This leads to denial of services attacks, or to host impersonation.

3) NFS, Finger, ...
Many protocols have security holes, especially old versions of daemons not properly patched. The possibility of catching all packets on a subnet facilitates the discovery of such old servers, and leads to the appropriate attack.

3. Sniffer Detector

We have seen that, from a network point of view, a sniffer is passive.

We may try to detect its presence if we catch it putting an interface in promiscuous mode. We could send an ICMP echo request (ping) packet targeted to the suspicious host, but with a bad MAC address (this is easily achieved by manipulating the ARP table). If the interface is in promiscuous mode, it will copy this packet and pass it to the operating system, which will in turn send an ICMP echo reply. If the interface is not in promiscuous mode, it will discard the echo request. Unfortunately, this does not work on all operating systems. Linux has this behavior, for instance, but not Solaris. Moreover, this technique works only if we have access to the subnet where the sniffer is installed. Therefore, we have to find another method.

To discover the presence of a sniffer, one must wait for the sniffer's owner to launch an attack. The problem is how to differentiate a normal Telnet session from one started by an intruder. Our idea is to spread bait that is presumably especially attractive to the sniffer's owner. This report presents a tool we designed and implemented according to the following principles:

- The tool sends packets over the subnet that simulate Telnet or FTP sessions, including false passwords for false accounts. These sessions must contain something of particular interest to the sniffer's owner (more so than ordinary sessions).They therefore include sensitive logins such as *root*, *admin*, ...
- The tool then waits for the intruder to use the information bait. Nobody besides the intruder has knowledge of these accounts and passwords. If someone reuses these pairs, the tool recognizes symptoms of an attack and can trigger an alarm.

There are different ways to achieve this, as discussed in the following sections.

3.1 Packet generation

There are essentially two ways to generate packets. Choosing one or the other has consequences on the hardware architecture choice (see 3.2 ”Hardware architecture” on page 6). The first generation method consists of forging each packet completely. The second one will use real connections.

	Packet Forging	Real Connection
Complexity	Need to forge all packets, from the lowest layer and up.	Simple scripts do the job.
MAC address	Cannot forge MAC addresses.	MAC addresses of the host(s) interfaces.
IP addresses	IP addresses easily forged. They have no physical meaning.	IP addresses of the host(s). They are valid.
TCP seq number	Forged: pseudo-random, time-based, ... Must reproduce the normal behavior of a TCP stack.	*Natural* numbers, generated by a normal TCP stack.
Data transmitted	Need to simulate the complete protocol.	Need to simulate user interaction only.
Resistance to an attack	Attack-proof: the IP addresses used are not valid ones. Even hijacking has no chance of success.	Special care must be taken: the hosts involved should have only the strictly necessary daemon running (even none at some times), they may be prone to be the target of SYN flood attacks,... When necessary, dummy shell or secured login should be implemented.

3.2 Hardware architecture

Several architectures can be used for packet generation and detection. In particular, the system could be implemented on one host, where it could separate the generation from the detection, spread the various components throughout the network, etc.

3.2.1 Single host

Both packet forging and real connection methods can be used. For the real connection method and if we want to use only one interface, we need a kernel able to link different IP addresses to the same interface. The packet forging method seems more appropriate for this configuration. Indeed if the host tries to connect itself, some operating systems will not let the packets go to the network wires and provide a shortcut inside the network stack.

+	-
Interprocess communication facilitated. Low Cost. Easy attack detection.	Static MAC addresses -> requires one interface per simulated host (if we want to go to such detail). Single point of failure. Cannot transmit packets *through* the network. Works on a subnet only.

3.2.2 Two generators, detection located on the generators

Both packet generation schemes could be used. The real connection scheme is easier to implement than the packet forging scheme, because of the problems of synchronization between the two generators.

+	-
No problems with MAC addresses. If placed on two different subnets, there is the possibility to send packets through the network. There are some limited possibilities to locate the sniffer. Add some redundancy.	Need an interhost secure communication channel to synchronize, perhaps with a coordinator. Twice the cost of Single Host.

3.2.3 Many generators, detection centralized on a coordination host

Both packet generation schemes could be used. The real connection scheme is easier to implement than the packet forging scheme. In this configuration, the manager asks the generating hosts to start a transmission (or connection). The various hosts report to the manager which packets they see passing by. It is up to the manager to decide whether it is a consequence of the generation or an attack.

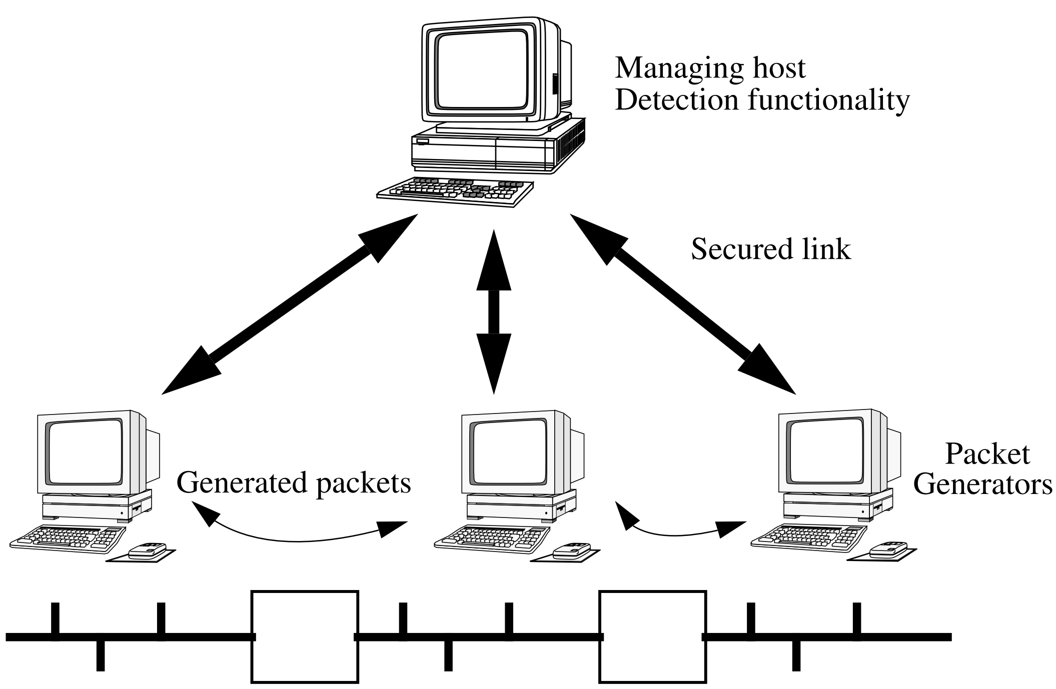

Figure 1. Multiple hosts configuration

+	-
No problems with MAC addresses. If placed on different subnets, there is the possibility to send packets through the network. It is possible to locate the sniffer by sending different information on different subnets. Add some redundancy. Most general solution. Single host configuration could be performed by software developed for multiple host.	Need an interhost secure communication channel to the managing host. Expensive... Add lots of complexity. Detection not trivial.

3.3 Detection

There are two levels of alarm:

1) IP level, easy to trigger, prone to misfire

   This alarm is triggered by the reception of a packet containing at least one of the IP addresses that have been generated, either in the source or in the destination field of the IP packet.

   The detector must be aware that a session is currently generated in order to detect only the packets coming from the intruder. If this functionality is implemented in the manager, it lets us realize a fault tolerant mechanism (if the manager does not see a notification when generating a session, it indicates a problem with a detector).

   A misfire may happen if someone on the network sends by error a packet containing one of the IP addresses.

2) Application level, more difficult to implement, no misfire

   This alarm is triggered by the reception of information generated at the application level. For example, we capture as password in a Telnet session the (false) password sent in a session generated previously.

   The detector must be aware of a running packet generation in order to detect only sessions issued by the intruder.

   No misfire can happen because the information we sent are not used anywhere in the network. Only a person having used a sniffer could know this information.

   This method can detect an attack targeted to any host of the network (as long as we have a detector somewhere on the route), not only the hosts involved in the generation of the sessions.

The first method is simple to implement, whereas the second is difficult. In the first case, a simple packet capture utility such as tcpdump [5] can be used. In the second case, a complex sniffer able to reconstruct sessions up to the application level is required.

Tracking down the sniffer could be possible if we have many generators and detectors. By sending different information on different paths and observing the trials of the intruder, we should be able to point out on which segment the sniffer is located.

4. Solution Choice

4.1 Architecture

The chosen architecture of the system is the following:

- Multi host - Most general solution, allows a good modularity of the system and enhanced detection methods. A **manager** host configures and synchronizes **client** and **server** hosts in order to generate the sessions. A complex **network monitor** signals to the manager network events. The manager has the knowledge to discriminate events caused by the generation or by real attacks.
- Real connections - The traffic is generated by real connections, simplifying the generating process.

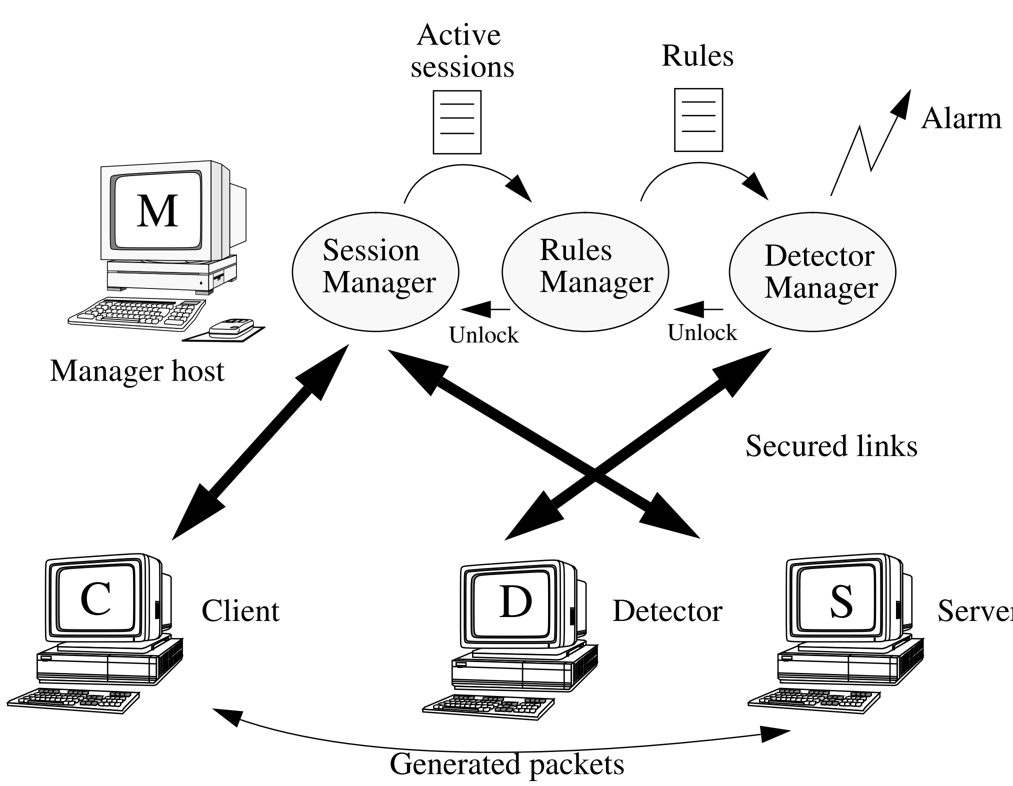

Figure 2. Architecture

See 5 ”Implementation” on page 14 for implementation details.

4.2 Manager design

The manager is built of three components.

The first one, the Sessions Manager, decides when and what session to generate. It generates sessions either automatically, or with the help of the user. It configures the server, gives the information to the Rules Manager and finally launches the session from the client after a synchronization with the Rules Manager. This synchronization is necessary to ensure that the rule list is up-to-date. At the end of the session, it unconfigures the server and tells the Rules Manager that this session is over.

The second component, the Rules Manager, is in charge of deducing from the generated sessions which information should be monitored by the detectors. It receives from the Sessions Manager a structure containing all the parameters of the generated session. From this information, the Rules Manager will generate various rules[1]. For example, there could be two rules designed to catch packets with one of the IP addresses used, a third one looking at the Telnet login and password pair, and a fourth one looking at the same login/password pair but for FTP sessions. It is the Rules Manager that has the knowledge to create intelligently these rules from each generated session. They are recorded in a rules table, and given to the Detectors Manager. It also removes old entries (the aging time may be of the order of one week). If an update of the rules table is necessary, it signals this fact to the Detectors Manager and waits for an update confirmation. When it receives this confirmation, or if no update was necessary, it unlocks the Sessions Manager, letting him proceed with the session generation.

The third component, the Detectors Manager, configures the detectors by sending the rules table. It receives updates from the Rules Manager. When an update has been sent to all the detectors, it unlocks the Rules Manager. It also receives the *Hot* messages[2] from the detectors. Through the knowledge of which session(s) are active and which rule(s) triggered the *hot* message, it can evaluate the seriousness of the alarm, and if needed transmits it to a human operator.

4.3 Session generation

The manager sends a request to a client/server pair to start the generation of a session.

They both use a new IP address for this session through an IP-aliasing [10] method implemented into the OS kernel. We used Linux[3] as operating system, because it has this IP-aliasing capability. This lets the system generate sessions between many virtual

1.See 5.1.2.1 "Manager messages" on page 17
2.See 5.1.2.4 "Detector messages" on page 20
3.Slackware and RedHat 4.2

hosts from only two physical hosts. If the sniffer is somewhere on the route, it can not see the mac addresses of these two hosts, and has therefore no way to discover that they are in fact the same hosts. On the other hand, the intruder will not be able to connect to these virtual hosts, because these temporary addresses are removed at the end of a session. It helps us to protect the system (we would like to avoid denial of service attacks against our system), but may give a hint to the intruder of the existence of a sniffer detection system. Fortunately, we should have detected his actions before he becomes suspicious.

These sessions are real sessions, involving non-modified protocol stacks and services daemons. See 7.2.1 ”Attack resistance” on page 44 for a discussion about why it could be interesting to run modified daemons.

This is achieved by using Expect [11] scripts[1] reproducing user interaction.

4.4 Detection

The detector engine consists of a packet filter associated with a session reconstruction functionality. The Detectors Manager sends rules[2] to the detector.

When the detector engine detects a packet or a session matching at least one rule, it sends a hot message to the manager, transmitting information about the session and which rule(s) let it trigger the message. The manager will then interpret this message and decides if this is a result of a current session or if it is an attack.

The detector engine is difficult to implement. Several solutions are possible:

- Write from scratch a packet and session monitor, based on the libpcap [6] library.

- Write some scripts or programs parsing tcpdump [5] output.

These two solutions represents too much work, and are a master’s thesis by themselves. Fortunately, there is a third solution.

- Use as foundation a third party network monitor:

- NFR (Network flight recorder) [8] is a powerful application, with good managing functionalities (archiving of log files, ...). Unfortunately, a lack of instructions in the built-in language makes it inappropriate for filtering tcp sessions. For example, in a Telnet session, options can be sent at any time. If we want to reconstruct what the user did, we have to filter out these options. They always start with the ASCII code 255. NFR doesn’t have an instruction to compare a character with a numerical value. It is therefore impossible to filter out the Telnet options.

1.See 5.2.2 ”Client component” on page 22
2.See 5.1.2.1 ”Manager messages” on page 17

- Bro [9] is much simpler than NFR, but still very powerful, and has its own programming language. C++ classes parse the packets stream and generates events corresponding to states of the protocol. These events are then caught by scripts, where the policy or high level functionalities are defined. This is a very flexible and powerful architecture.

We have chosen to use Bro as detection engine. We needed to add some interprocess communications functionalities to Bro, so that the manager is able to transmit requests to the engine. See 5.2.4 ”Detector component” on page 27 for more implementation details.

4.5 Course of operations

This paragraph describes which component is aware of what information, and how they interact with each other.

When the user request the generation of a session, the Sessions Manager creates an entry in the sessions table. This entry stores the IP addresses of the client and the server, the respective temporary IP addresses used for the generation of that session, and protocol is to be generated with its options.

The Sessions Manager then starts the server and transmits it this entry.

If the setup is successful, the Sessions Manager gives the table to the Rules Manager. If not successful, the Sessions Manager removes the entry from the table and informs the user that something went wrong.

The Rules Manager analyses the session entry, and creates new rules. These rules are stored in a rules table. They are of three types (see 5.1.2 ”Protocol description” on page 16). Each rule possesses a time tag, indicating when it has been created. There is no duplication of rules. Instead, the time tag is reset.

This rules table is given to the Detectors Manager along with the sessions table. The Detectors Manager transmits the new rules to each detector. If there are fewer rules, it sends a RESET request and retransmits the whole rules table.

Once the detectors have been updated, the Sessions Manager starts the client and transmits the session entry.

The client configures its interface with the transmitted IP address, and starts the generation of a session. It uses the transmitted protocol options to generate the session.

During the generation, a detector will see this session. Because its rules table has been updated, it will send a HOT messages (see 5.1.2 ”Protocol description” on page 16) to the Detectors Manager, informing it that a packet or a session did fulfill a rule. The HOT message includes the session 4-uple (IP_source, IP_destination, Port_source, Port_destination) identifying this session, and the list of rules at the origin of this notification.

The Detectors Manager receives this message. It creates an entry in the hot sessions table. This table records each notified session, which rules have been triggered for this session, and when it learned the existence of this session for the first time. The Detectors Manager looks at its active sessions table. If the IP addresses and the detected protocol options (including the well-known server port) are the same as one of the current sessions, the Detectors Manager decides that this notification is normal, as the system is generating this session. It therefore discards this message.

If the Detectors Manager receives a notification of a session issued from the same two hosts of a current session, but with different ports, it supposes that this is a *side-effect* of the current session. This happens when an FTP session opens data connection back to the client host on the port 20, or when a server tries to identify the remote user through an ident (port 113) connection. Such connections are tagged as Warnings only.

Once the client sees on the application layer that the session is over, it informs the Sessions Manager wether the generation process went successfully.

When the detector catches the end of the session (FIN or RST TCP packet), it sends a FIN message to the Detectors Manager. When the Detectors Manager receives this message, it frees the memory allocated for tracking this connection.

The Sessions Manager, upon receiving the acknowledgment of the client, sends the request to the server to remove its configuration. This action is performed whatever the result of the client side is. If it was not successful, it informs the user that something went wrong.

The server informs the Sessions Manager if it manages to remove its configuration. If it was not successful, this is a major problem and the Manager waits for a human operator to resume operations.

Upon success, the Sessions Manager removes the session entry in the active session table and gives this new table to the Rules Manager, which in turn gives it to the Detectors Manager.

If one detector detects now a session triggering at least one of his rules, it sends a HOT message to the Detector Manager. The latter will print an Alarm, because no currently active sessions is responsible for this message.

5. Implementation

5.1 Inter-components communication

5.1.1 Scenario example

This is an example of a typical scenario, showing the exchanged messages between the different components, and which actions they trigger. The formal description of the protocol is given in 5.1.2 "Protocol description" on page 16.

SM: Sessions Manager RM: Rules Manager DM: Detectors Manager

S: Server C: Client D: Active Detector

1) SM➨S Start: Sid=123, IPsrc=C, IPdst=S, PortSrc=-1, PortDst=-1, Prot=Telnet, Opt={user, passwd}

The manager opens a session to the server and tells him to be ready to receive a Telnet connection from the client C, using the user name *user* with the password *passwd*. The server configures itself to receive that connection. It inserts a new line in the */etc/passwd* file and creates a home directory for this user. It also creates a new network address S (IP-aliasing [10]). The new session identifier is 123. A timer is started: the server's answer should come before the timeout.

2) S➨SM Server_OK: Sid=123

If the server configured itself correctly, it sends the confirmation to the manager. The manager timer is disabled (the answer did come on time[1]).

2a) The Sessions Manager updates the Active Session table[2]. It then informs the Rules Manager that a new session will be generated, and waits for a confirmation. The Rules Manager updates the rules table by using the information transmitted in the active session table. It inserts new rules, resets the aging timer for existing rules and expands the information to different protocols. This behavior is coded into the Rules Manager. It then informs the Detectors Manager of the update, and waits for a confirmation.

2b) The Detectors Manager compares the old rules table with the new one. If the update consists of the addition of new rules only, it sends these new rules to the detectors. If there are deletions, it resets the detector (this is achieved simply by killing and restarting the detector) before sending the whole rules table again.

3) DM➨D Look_host: Rid=1, IP=C

1.If not, it sends an ERROR message. The Sessions Manager would then abort the generation process.
2.See 4.5 "Course of operations" on page 12 for a description of an active session entry.

4) DM➨D Look_host: Rid=2, IP=S

5) DM➨D Look_protocol: Rid=3, Prot=Telnet, Opt={user,passwd}

6) DM➨D Look_protocol: Rid=4, Prot=FTP, Opt={user, passwd}

3-6: The Detectors Manager tells the detector what it should look for on the network. This is any packet containing the IP address of the client or of the server, and any Telnet or FTP connection showing user and passwd. Each rule has a rule identifier (Rid). When firing, the detector will send with the *hot* message a list of the rule ids corresponding to the alarm.

6a) Upon successful transmission, the Detectors Manager unlocks the Rules Manager.

6b) Unlocked, the Rules Manager signals the Sessions Manager to proceed.

7) SM➨C Start: Sid=123, IPsrc=C, IPdst=S, PortSrc=-1, PortDst=-1, Prot=Telnet, Opt={user, passwd}

The Sessions Manager, unlocked by the Rules Manager, tells the Client to start the session.

7a) The client creates a new IP address C (IP-aliasing), and starts the session.

8) C➨S Telnet session.

A way to generate such a session is through the use of *Expect [11]* scripts executed on the client host.

9) D➨DM HOT: IPsrc=C, IPdst=S, PortSrc=ccc, PortDst=23, Rid=1,2

The detector signals that he received a "hot" packet, corresponding to the Rule Id 1 and 2. It's up to the manager to decide if this is part of the session or if it is an intrusion. In this special case, the manager knows there is a session running. This early warning is therefore discarded.

10) C➨SM Client_OK: Sid=123, ended

If everything went fine, the client tells the manager that the session generation has ended.[1]

10a) The Sessions Manager removes the entry in the active session table and signals the modification to the Rules Manager.

10b) The Rules Manager updates (if necessary) the rules table, and signals the update to the Detectors Manager.

1.If something went wrong (the client sent an ERROR message, or the Sessions Manager timed out), the Session Manager would try to abort the session, display an error message and proceed with the server to remove its configuration.

10c) If necessary, the Detectors Manager updates the detectors.

11) SM➨S STOP: Sid=123

The manager asks the server to delete its special configuration for this session. A timer is started. The server removes the entry in the */etc/passwd* file, the home directory and the IP address. This is to prevent the intruder from entering the system by using the information he sniffed.

12) S➨SM Server_OK: Sid=123

The de-configuration has succeeded. The timer is disabled, because the answer did arrive before the time-out[1].

13) D➨DM HOT: IPsrc=X, IPdst=Y, PortSrc=xxx, PortDst=23, Rid=3

Later, the detector tells the Detectors Manager he has found a hot Telnet session containing the user name *user* and the password *passwd*. The manager, by looking in the active session table, knows he has not instructed such a session, and triggers a high-level alarm.

5.1.2 Protocol description

Reliable connections are needed between the different hosts of the system. These channels are used by the manager to coordinate all the operations.

The channels need to be secure: We do not want the intruder to sniff the passwords used to connect to our system! This can be ensured by a physical separation from the monitored network, but this leads to the same security problems as those for the original network, and may not be possible if the system is distributed on a large scale. It is therefore more judicious to authenticate and encrypt each stream.

We use *ssh [13]*: It is a replacement of *rsh*, and provides strong authentication and encryption. It is reasonably secure[2], although interactions with X11 may cause some vulnerabilities [14]. The communication functionalities required of the system are reduced to standard input/output code, the networking part being supported by ssh.

The protocol is ASCII and line-oriented. This has been useful for the debugging phase, where we have been able to enter commands *by hand* for testing purposes.

In the following paragraphs, we describe the messages generated by the various components of the system.

1.If this was not the case, this is considered a fatal error: The server stays in an unknown state and requires a human operator to reset it to its default configuration.
2.One should always use the latest version. At the time of this writing, it is v1.2.25.

5.1.2.1 Manager messages

START: *START, ID, IPsrc, IPdest, PortSrc, PortDst, Protocol, Options*

ID: Session identifier

IPsrc: IP address of the client which will initiate the session

IPdest: IP address of the server

PortSrc: Source port that should be used (-1 means OS chosen)[1]

PortDst: Destination port that should be used (-1 means well-known port)[2]

Protocol: The well known port assigned to the protocol[3]

Options: Protocol specific options. Examples:

Telnet: user name, password, root (super-user) or user[4]
FTP: user name, password

Audience: Server, Client

Purpose: Tells the systems what is the next session which will be generated.

STOP: *STOP, ID*

ID: Session identifier

Audience: Server

Purpose: Informs the server that the session generation is over, and that it must remove its configuration.

RESET: *RESET*

Audience: Detector

Purpose: Resets (clear) the rules table of the detector.

1.This functionality has not been implemented in the scripts. The src port is always chosen by the OS.
2.The server scripts does not implement this functionality. The port used by the server is always the default port for the considered protocol.
3.Only Telnet (23) and FTP (21) are currently supported. The well-know port numbers are defined in the */etc/services* file
4.This third option modifies only the prompt. The users are never UID 0.

QUIT: *QUIT*

Audience: Detector

Purpose: Asks the detector front-end to kill the detector engine and exit.

Look_host: *Look_host, Rid, IP*

Rid: Rule identifier

IP: IP address to look for

Audience: Detector

Purpose: Tells the detector to look for packets containing this IP address as source or destination.

Look_port: *Look_port, Rid, server_port*

Rid: Rule identifier

server_port: port of the server side of the connection

Audience: Detector

Purpose: Tells the detector to look for packets containing this server side port. This kind of filter is useful in combination with other rules, and should be installed cautiously.

Look_protocol: *Look_protocol, Rid, Protocol, Options*

Rid: Rule identifier

Protocol: Well known port assigned to this protocol.

Options: Protocol specific options. Same as in the START message.

Audience: Detector

Purpose: Tells the detector to look for sessions served by a well known port, for which we know the protocol.

5.1.2.2 Server messages

Server_OK: *Server_OK, ID*

ID: Session identifier

Audience: Sessions Manager

Purpose: Acknowledges the last request from the manager, indicating that everything went fine.

ERROR: *ERROR, ID*, *Error*

ID: Session identifier

Error: Describes the error

Audience: Sessions Manager

Purpose: Alerts the manager that something went wrong.

5.1.2.3 Client messages

Client_OK: *Client_OK, ID, Status*

ID: Session identifier

Status: Status of the session generating process

Start - The session will start[1]

Ended - The session completed successfully

Audience: Sessions Manager

Purpose: Indicates that everything is going fine, acknowledgment of manager requests.

ERROR: *ERROR, ID*, *Error*

ID: Session identifier

Error: Describes the error

Audience: Sessions Manager

Purpose: Alerts the manager about something going wrong.

1.This message has not been implemented

5.1.2.4 Detector messages

HOT: *HOT, IPsrc*, *IPdst*, *PortSrc*, *PortDst, Rid list*

IPsrc: IP address of the host which has initiated the packet

IPdest: IP address of the host which will receive the packet

PortSrc: TCP/UDP source port

PortDst: TCP/UDP destination port

Rid list: List of the rule ids that triggered this message

Audience: Detectors Manager

Purpose: This indicates the Detectors Manager that packets fulfilled at least one rule.

FIN: *FIN, IPsrc, IPdst, PortSrc, PortDst*

IPsrc: IP address of the host which has initiated the packet

IPdest: IP address of the host which will receive the packet

PortSrc: TCP/UDP source port

PortDst: TCP/UDP destination port

Audience: Detectors Manager

Purpose: This message tells the Detectors Manager that the TCP session for which an alarm has been sent is now terminated (a FIN or RST has been received).

DUMP *Dump*

Audience: Manager

Purpose: Asks the manager to send the whole rules table.

ERROR *Error*

Error: Describe the error

Audience: Manager

Purpose: Alerts the manager about something going wrong.

5.2 Components specification

5.2.1 General points

As mentioned above (5.1.2 ”Protocol description” on page 16), we will use *ssh* to support the communication. More precisely, the manager application will start ssh sessions between his host and the hosts that support the various components. The ssh sessions will start the remote applications and support an interactive dialog.

This means that the components send and receive the protocol messages on the standard input/output.

A front-end process launched by the ssh command deals with the protocol. When requested, it launches the appropriate program, script or command, depending on the request.

The front end sometimes needs superuser privileges. Therefore, it is either setuid’ed root, or has access to programs setuid’ed root.

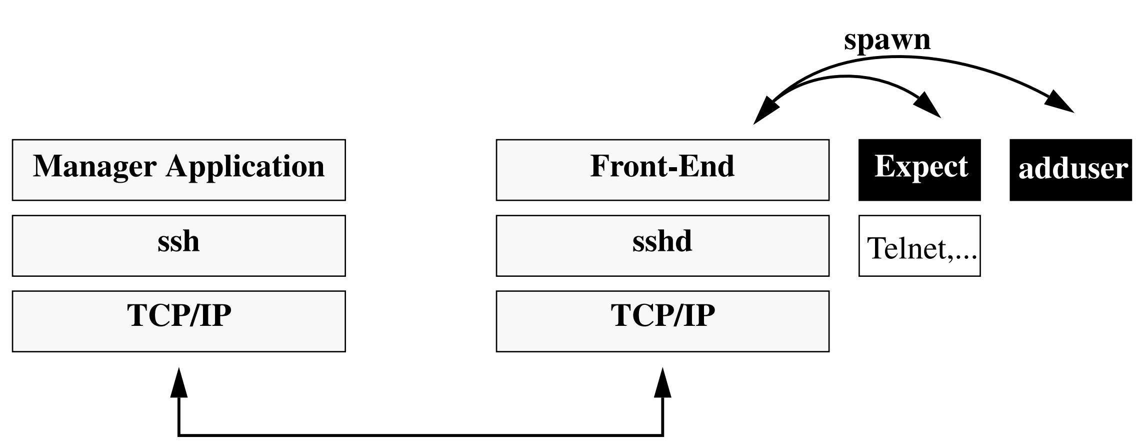

Figure 3. Components Overview

5.2.2 Client component

When the Sessions Manager needs to start a new session, it opens a connection to the client machine and starts the front end (*client_fe.xpt*, an Expect [11] script) with the help of ssh (like the old rsh). The front end then reads the *start* request and configures the network interface with the new temporary IP address (IP-aliasing). It uses a local copy of *ifconfig* and *route* that have been suid'ed to root for this purpose. These are the commands executed by the scripts to configure the new interface:

```
# add an alias of the ethernet interface with a new IP address
ifconfig eth0:0 IPsrc netmask netmask
# add a route to the gateway through this new interface
route add -host gateway eth0:0
# add a route to the server via the gateway and through the new interface
route add -host IPdst gw gateway eth0:0
```

If this configuration is successful, it forks an Expect script, specific to the protocol to simulate (*telnet.xpt* or *ftp.xpt*), and waits for the script to end.

```
if {$prot == 23} {
        if {$portDst == -1} {set portDst 23}
        set spid [spawn ./telnet.xpt $IPdst $portDst $login $passwd]
} elseif {$prot == 21} {
        if {$portDst == -1} {set portDst 21}
        set spid [spawn ./ftp.xpt $IPdst $portDst $login $passwd]
}
```

At this point, it unconfigures the network interface and terminates.

```
wait -i $spawn_id
sleep 5
catch {system ifconfig ${interface}- 0}
send_user "Client_OK: ${sid}, ended\n"
exit
```

Figure 4 shows the Client front end state diagram:

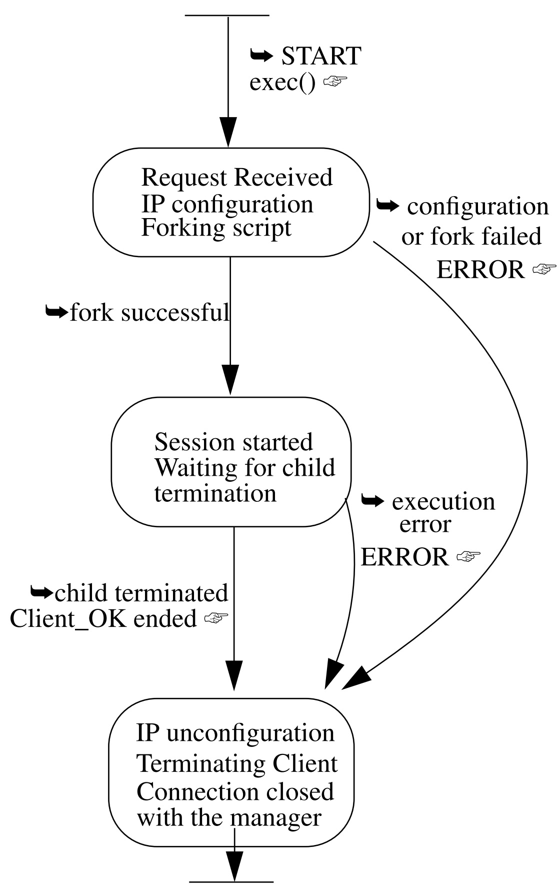

Figure 4. Client state diagram

It is not foreseen to be able to generate more than one session per client at a time, but the addition of this functionality is not blocked by this scheme.

Two protocols are simulated: Telnet and Ftp.

This is what is generated during a Telnet session:

```
peccia:~/sniffer_detector/client$ telnet.xpt peccia 23 user password
spawn telnet peccia 23
Trying 195.212.119.5...
Connected to peccia.zurich.ibm.com
Escape character is '^]'.

Linux 2.0.29 (peccia.zurich.ibm.com) (ttyp4)

peccia login: user
Password:
Linux 2.0.29.
```

```
Last login: Thu May 28 11:54:30 on ttyp4 from internet-gateway.
No mail.

peccia:~$ ls -l
total 0
peccia:~$ cp /etc/passwd ./passwd.old
peccia:~$ gzip passwd.old
peccia:~$ ls -l
total 2
-rw-r--r--   1 user      gsal        1435 May 28 16:46 passwd.old.gz
peccia:~$ exit
Connection closed by foreign host.
Session ok
peccia:~/sniffer_detector/client$
```

This is what is generated during an Ftp session:

```
peccia:~/sniffer_detector/client$ ./ftp.xpt peccia 21 user password
spawn ftp peccia 21
Connected to peccia.zurich.ibm.com
220 peccia FTP server (Version wu-2.4(4) Tue Apr 29 13:38:27 CDT 1997) ready.
Name (peccia:user): user
331 Password required for user.
Password:
230 User user logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> dir
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 2
drwxr-xr-x  16 user      gsal        2048 May 28 16:52 .
drwxr-xr-x  32 root2    root2        1024 May 28 16:23 ..
226 Transfer complete.
ftp> bin
200 Type set to I.
ftp> put passwd.old.gz
local: passwd.old.gz remote: passwd.old.gz
200 PORT command successful.
150 Opening BINARY mode data connection for passwd.old.gz.
226 Transfer complete.
239 bytes sent in 0.000412 secs (5.7e+02 Kbytes/sec)
ftp> dir
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 4
drwxr-xr-x  16 user      gsal        2048 May 28 16:52 .
drwxr-xr-x  32 root2    root2        1024 May 28 16:23 ..
```

```
-rw-r--r--   1 user      gsal         239 May 28 16:52 passwd.old.gz
226 Transfer complete.
ftp> quit
Session ok
```

5.2.3 Server component

When the Sessions Manager wants to configure the server to accept a future session, it starts the server front-end (*server_fe.xpt*) via ssh. The front-end then reads the START request and configures the system. It is started through a wrapper setuid'ed root (*server_wrapper*) to gain enough privileges to update system files (/etc/passwd) and manipulate network interfaces and routing tables. Only one session at a time is allowed (we need to receive a STOP after a START).

The operations performed to configure the server are the following:

- configure the network interface to add the temporary IP addresses (IP-aliasing), in the same way as done by the client front-end
- configure the network routes to reach the client via the new interface, in the same way as done by the client front-end
- Add an entry in the /etc/passwd file, and /etc/shadow if it exists
- Change the password for this user by issuing the 'passwd' command[1].
- Create a directory under /home/username
- Create a customized prompt (# for a pseudo root account, $ for a user account)

After receiving the STOP request, the server front-end removes this setup.

1.The RedHat 4.2 version of 'passwd' does not work with Expect. It requires a live tty. We replaced it with the one from a Slackware distribution.

Following is the state diagram of the server front-end:

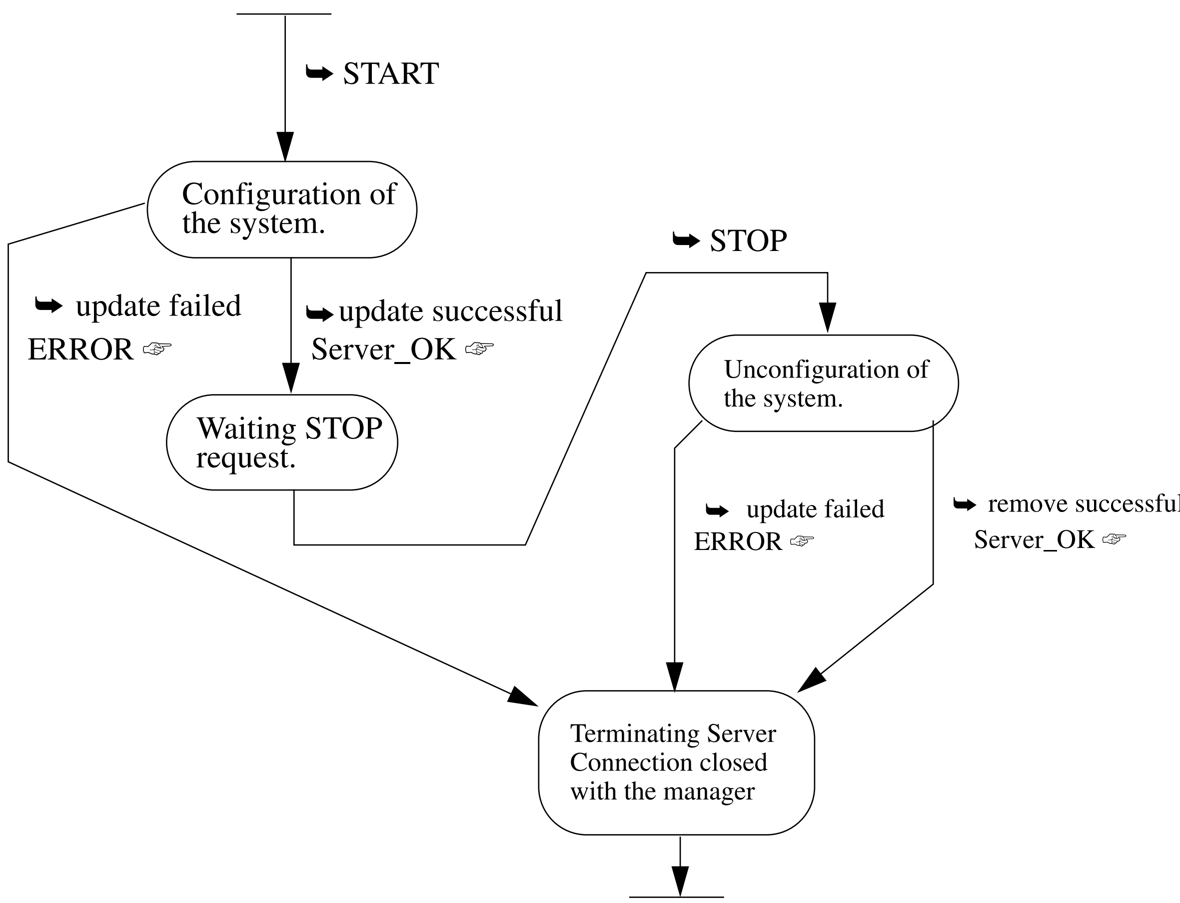

Figure 5. Server state diagram

5.2.4 Detector component

The Detectors Manager starts a new detector front-end through a ssh session. The front-end forks the detector engine, which consists of a packet filter associated with a session reconstruction functionality. The front end transmits the requests from the Detectors Manager to the engine. These requests are the updates of the rules table (*Look_xxx*).

As mentioned in § 4.4, we use Bro [9] as detection engine. This choice led to a design choice of the Detectors Manager and of the detector front-end: It is difficult to remove a rule entry in the internal table of Bro. Therefore, when the Detectors Manager wants to refresh the whole rules table, it sends a *reset* request to the front-end. The front end kills and restarts Bro. The manager can then retransmit the whole rules table, without the deleted entries.

Bro has another problem: Each time a new session is detected, Bro allocates some memory to keep its state. This memory is never released, leading to a memory shortage. We had to implement in the detector front-end a memory monitoring functionality. If Bro reaches 2.8 Mbytes (which is about two and a half its original size)[1], the front-end kills and restarts it, and sends to the manager a DUMP message requesting the transmission of the whole rules table. This is a temporary but unsatisfactory solution, especially because if Bro was monitoring a session, it loses its state. This happens every half hour on our network, loaded at 36 kbytes/s and with a majority of short http connections (37 connections per minute).

1.This value has been determined experimentally in order to avoid a crash of the manager host. It could be different on other systems.

Following is the state diagram of the detector front-end:

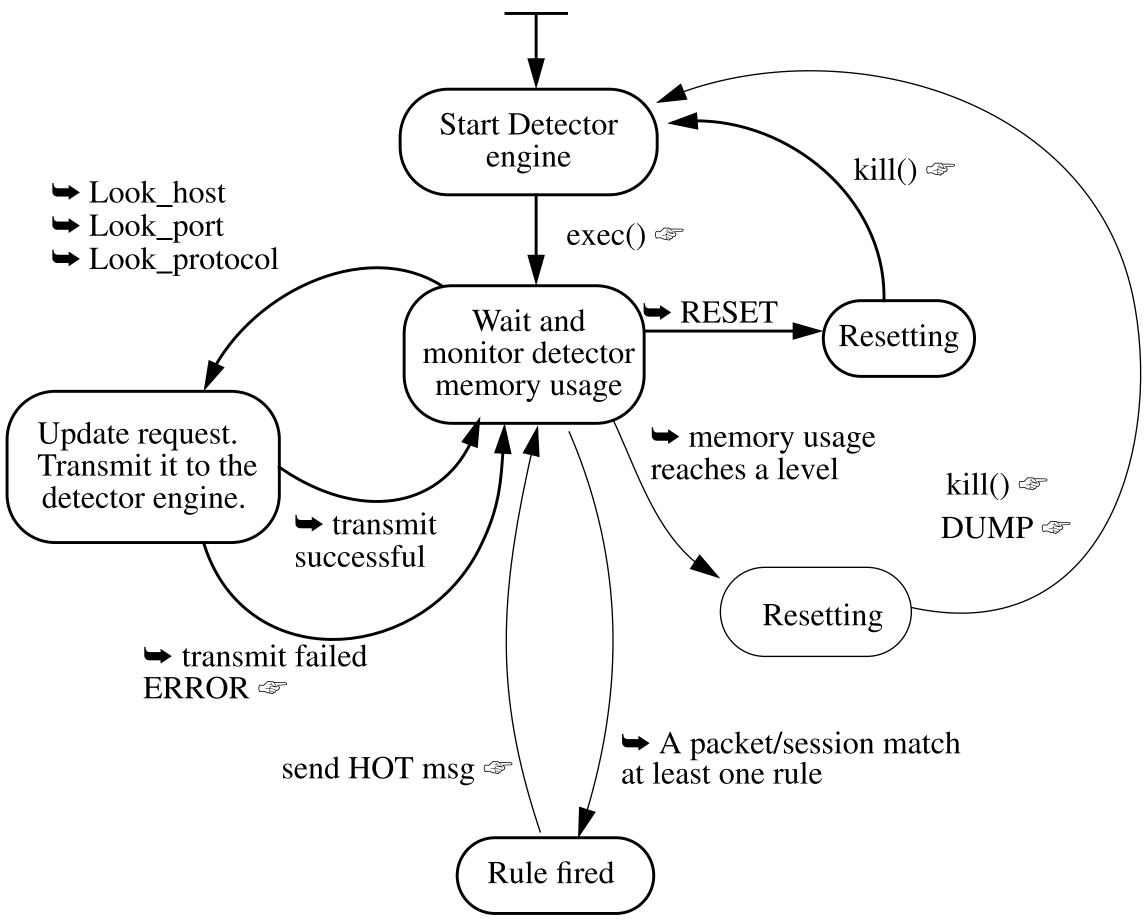

Figure 6. Detector state diagram

Writing a network monitoring tool is a difficult task. Many pitfalls must be avoided, and it may in some cases lead to ambiguity in the interpretation of the captured packets [15]. This is the reason why we chose to use an existing detector engine.

We have used Bro v0.3 alpha as detection engine. Although it is still in alpha, it is already doing what we need to demonstrate the sniffer detector concept. Moreover, the source is free and can be easily upgraded. Adding new detection capabilities requires one to write a C++ class that will parse the stream and generate protocol-specific events. These events are then caught by Bro-scripts, implementing the high level behavior.

In order to integrate it into our system, we have made some modifications of the code:

- To update the internal tables with the rules transmitted from the manager, the detector front-end writes to the standard input of Bro a rule (in the *Look_xxx* format). It then sends a SIGUSR1 to Bro. Bro interrupts its work, parses one line from its standard input, and generates an internal event (look_host, look_port or look_login). This event is captured in the bro script (*det.bro*), and triggers a table update.
- To compile the code in Linux, we had to change some references to packet structures[1]. Some headers files are different from one system to another.
- A new event (connection_new) is generated each time a new 4-uple source IP, destination IP, source Port, destination Port is detected by the capture library.

The current version of Bro filters the following packets:

- Any UDP packet
- Any TCP packet with either SYN, FIN, or RST flag set
- Any TCP packet with a port equal to either 79 (Finger), 21 (FTP), 23 (Telnet), 111 (Portmapper)

As an alpha release, Bro has some drawbacks:

- IP fragmentation is not supported (this is a major deficiency, of which the author is aware)
- ICMP packets are not captured
- Known protocols directed to non-standard ports are not analyzed by Bro
- Many interesting protocols (like DNS requests) are not supported
- There is a memory management problem (the front-end has to monitor its memory usage)

Although we think Bro is a good choice, it is possible to choose another detection engine. Indeed, the manager does not rely on a particular detection engine, as long as it is compliant with the protocol defined in 5.1.2 ”Protocol description” on page 16.

1.For example, in the file *UDP.cc*, *uh_sum* references in a udp structure have been changed to *check.*

5.2.5 Manager component

The manager is split into three components: the Sessions Manager, the Rules Manager and the Detectors Manager. It is a single application that regroups these three components. We avoided a multithreaded application, because it is not well supported under Linux. Regrouping the three components in one application facilitates the transmission of the data structures among the components. See 7.2.2 ”System distribution” on page 45 for improvements.

The manager includes a textual user interface:

```
Sniffer Detector control panel
 1) Dump clients list
 2) Dump servers list
 3) Dump protocols list
 4) Dump rules defined by the Rules Manager
 5) Generate a new session
 6) Generate a random session
 7) Toggle automatic session generation. Status: off
 8) Rules garbage collection
 9) Beep is on
10) Print warning is on
 0) Quit

      Please enter your choice:
```

From this interface the user can start the generation of a session manually, or of repeatedly random sessions.

5.2.5.1 the Sessions Manager

The Sessions Manager controls the client and server hosts. At startup, it reads a configuration file listing all clients and servers available, and the sessions parameters to generate.

When a session is generated, it starts through ssh a server front-end on the server host and sends a START request. When the server front-end acknowledges, it updates its active session table (an array of C structures) and calls the Update method of the Rules Manager. It then starts the client front-end on the client host. Once the session is over, it sends the STOP request to the server, removes the entry in the session table and calls the Update method of the Rules Manager.

The failure to start the server or the client is considered a non-fatal error (this may be caused by a temporary slow network). The failure to stop the server is a fatal-error, because it may compromise the security of the system and leave it in an unknown state.

If this happens, a human operator should log in and repair the system by hand (/etc/passwd, /etc/shadow, network interface and custom directory)[1].

More details on the chain of events can be found in 5.1.1 ”Scenario example” on page 14. The state diagram is represented in Figure 7.

5.2.5.2 the Rules Manager

The Rules Manager has the intelligence to derive the rules from the sessions generated. For example, a Telnet session between two hosts may be mapped to several rules: a pair of *Look_host* rules to detect packets using the client or server IP addresses, a *Look_protocol* rule to detect Telnet session (on any host) showing the user and password generated, and also a rule to detect FTP session showing the same user and password (an intruder could try a FTP session with the information taken from a Telnet session).

On request from the user, the Rules Manager does a garbage collection on the rule list and removes old rules.

Each time the rules table or the session table is updated, the Rules Manager informs the Detectors Manager of this update and waits for an answer. The detector, after processing the update, unlocks the Rules Manager, which in turn unlocks the Sessions Manager if it is the origin of the update.

The state diagram is represented in Figure 8.

5.2.5.3 the Detectors Manager

The Detectors Manager is in charge of starting the detectors and keeping their configuration up-to-date. When the rules table is modified, it sends commands to the detectors to refresh their tables.

It also receives the *hot* messages from the detectors. Because the application is single-threaded, we rely on a wrapper between the manager and ssh. When messages are transmitted by the detectors and reach the wrapper, it copies them to its standard output (which is redirected to a file descriptor by the Detectors Manager). It then sends a SIGUSR1 signal to the manager. The manager interrupts its current task, and process the incoming messages.

1.The current implementation of the manager simply stops when this occurs. The operator should try to log into the server host, edit the files /etc/passwd and /etc/shadow to remove the temporary entry at the end of the file, remove the aliased interface (*ifconfig eth0:0- 0* for an ethernet interface) and remove the temporary home directory (*rm -rf /home/_login_* with ’_login_’ being the temporary login name).

If necessary, it triggers an alarm for a human operator. The conditions for the display of an alarm are the following:

- If the HOT message contains at least one *Host* rule and if the IP address pair does not belong to a current active session.
- If the HOT message contains two *Host* rules that match an active session but the detected server port does not match the well-known port specified by the active session data structure. This Alarm is tagged as a Warning only, because it may be a side-effect of the generated session.
- If the HOT message contains *Host* rules that match a current active session and if it contains a *Port* rule that is different from the well-known port of the active session.
- If *Host* rules and/or *Port* rules match a current session, and if the HOT message contains a *Protocol* rule that does not match a current active session.

In short: If all the rules of a HOT message can not be found to belong to an active session, the Detector Manager displays an alarm.

This alarm is written in a file:

```
ALARM on Fri May 29 18:21:47 1998 trig. by a session detected 0h 0m 0s ago
        from ipg3: 195.212.119.254[1362] -> 194.196.13.244[23]
        Rule id: 21 type: host generated for the last time 0h 15m 12s ago.
```

When a detector requests the retransmission of the whole rules table with a DUMP message, it is also the Detectors Manager that handles this operation.

The state diagram is represented in Figure 9.

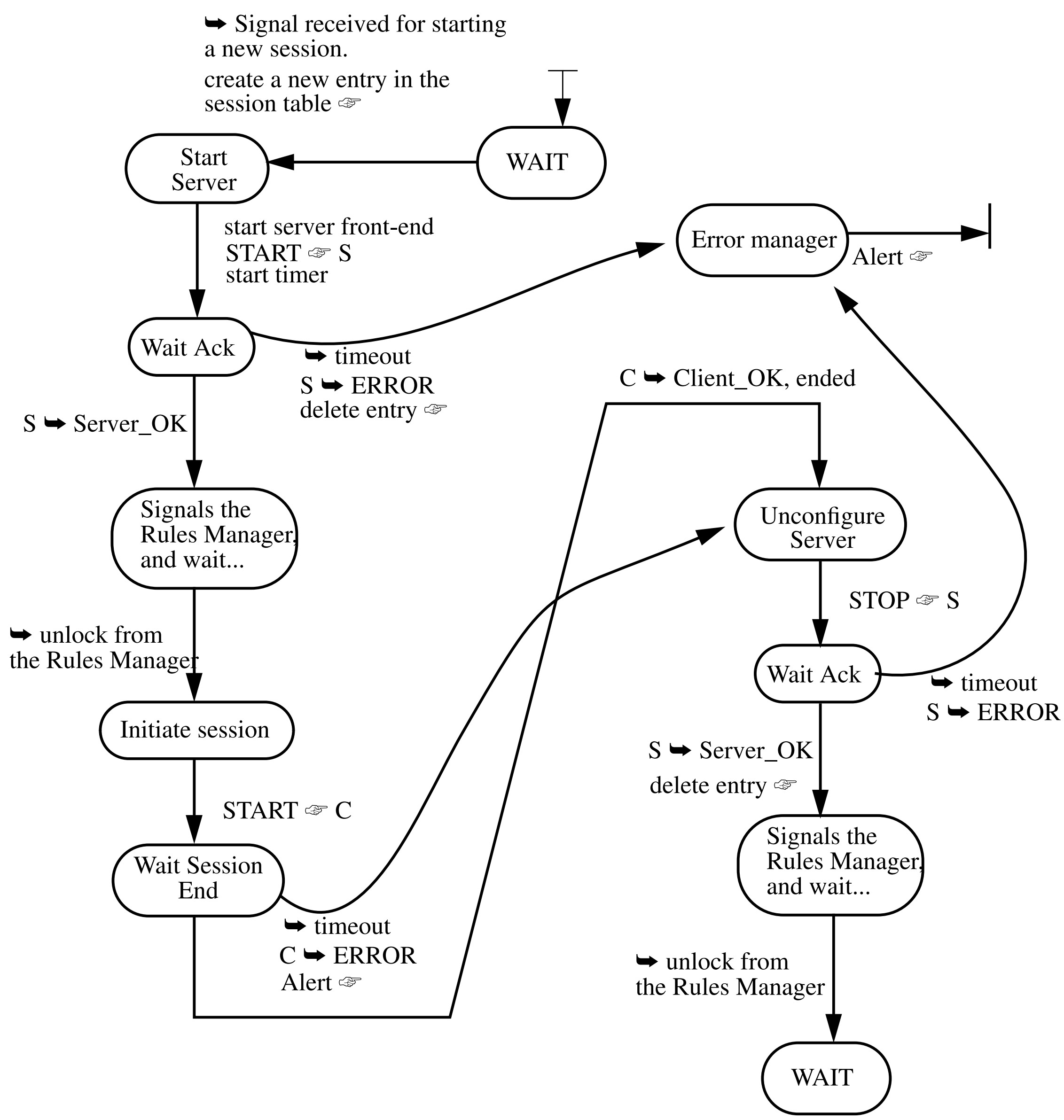

Figure 7. Sessions Manager state diagram

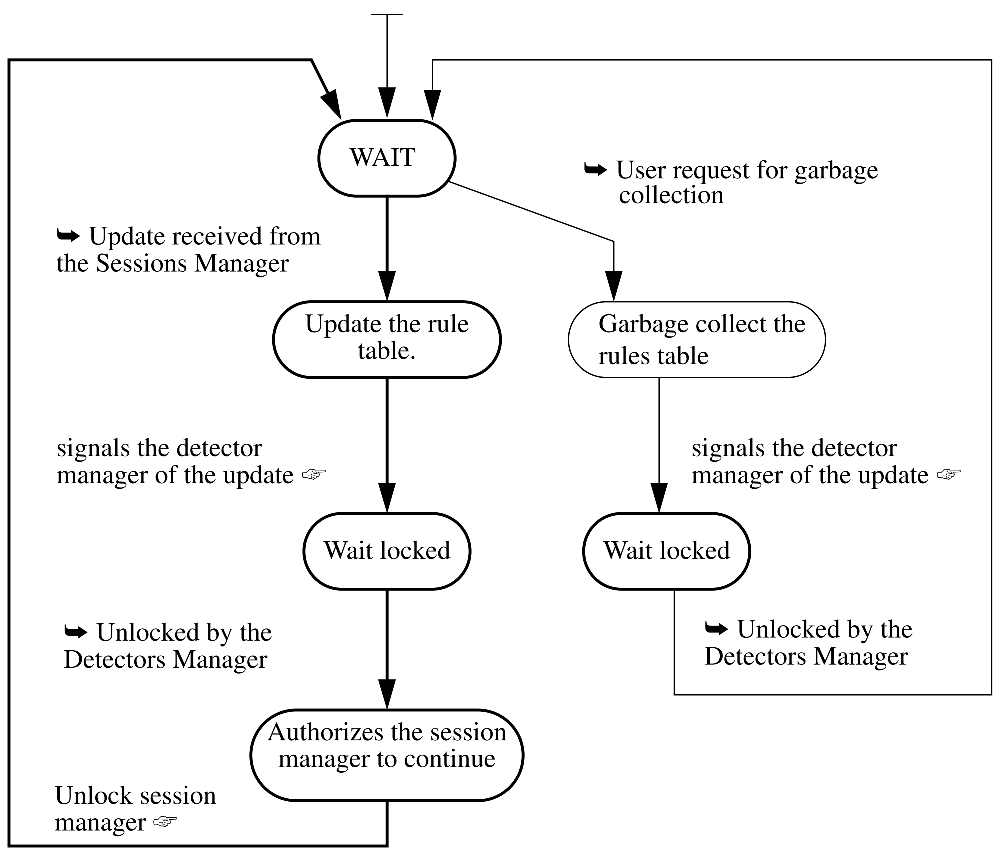

Figure 8. Rules Manager state diagram

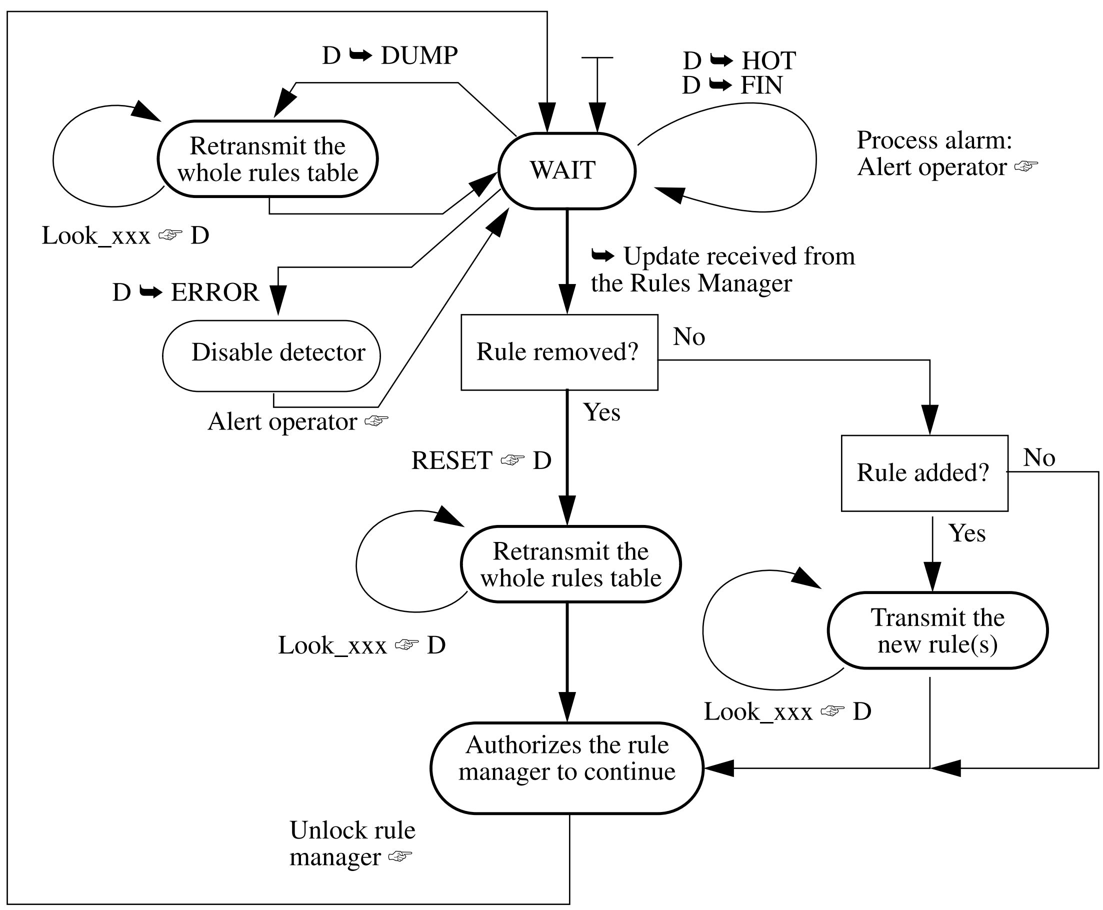

Figure 9. Detectors Manager state diagram

5.3 Overview

At run time, the manager forks many different processes. Figure 10 shows this chain of processes during the generation of a Telnet session.

This figure also shows that each component is on a different host. This may not be the case: The minimal configuration requires two hosts: one with the manager, detector and client/server functionalities, and a second one with client/server (and detector if desired).

One may also use only one machine, but this leads to several restrictions: First, one must ensure that a connection to the same host between two aliased interfaces is transmitted effectively on the wire, and not short-cut by the kernel[1]. Second, one loses the possibility to send packets across a network, which is a major drawback.

These two (or less) host configurations work as long as the manager can connect both hosts via ssh, and there is enough memory and processing power to process all the captured packets.

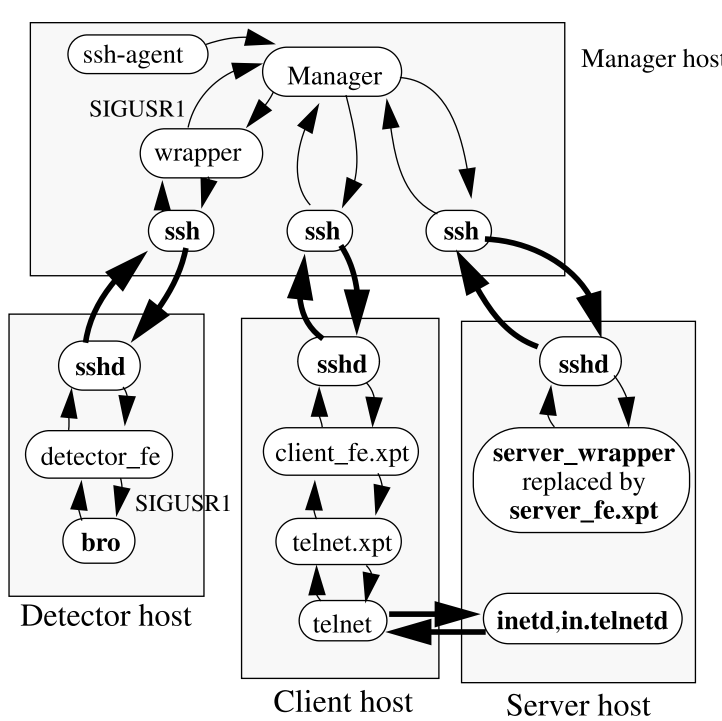

Figure 10. Process chains

1.On Linux, this was the case, but the kernel was not able to read the packet at the same time it transmited it. We were therefore not able to make a connection to the same host via two aliased interfaces.

6. Tests

After the first tests during development performed on the GSAL network (two Thinkpads connected through a router), we moved the system to the IBM network for real-size testing. These tests started on April 29 1998, with the first stable long-run on the weekend of May 17. Three weeks of tests followed to eliminate the last remaining bugs.

6.1 Setup

We have tested our system between IBM Zurich (Switzerland) and IBM La Gaude (France). These two sites are connected via IGN (IBM Global Network). Figure 11 shows the network architecture.

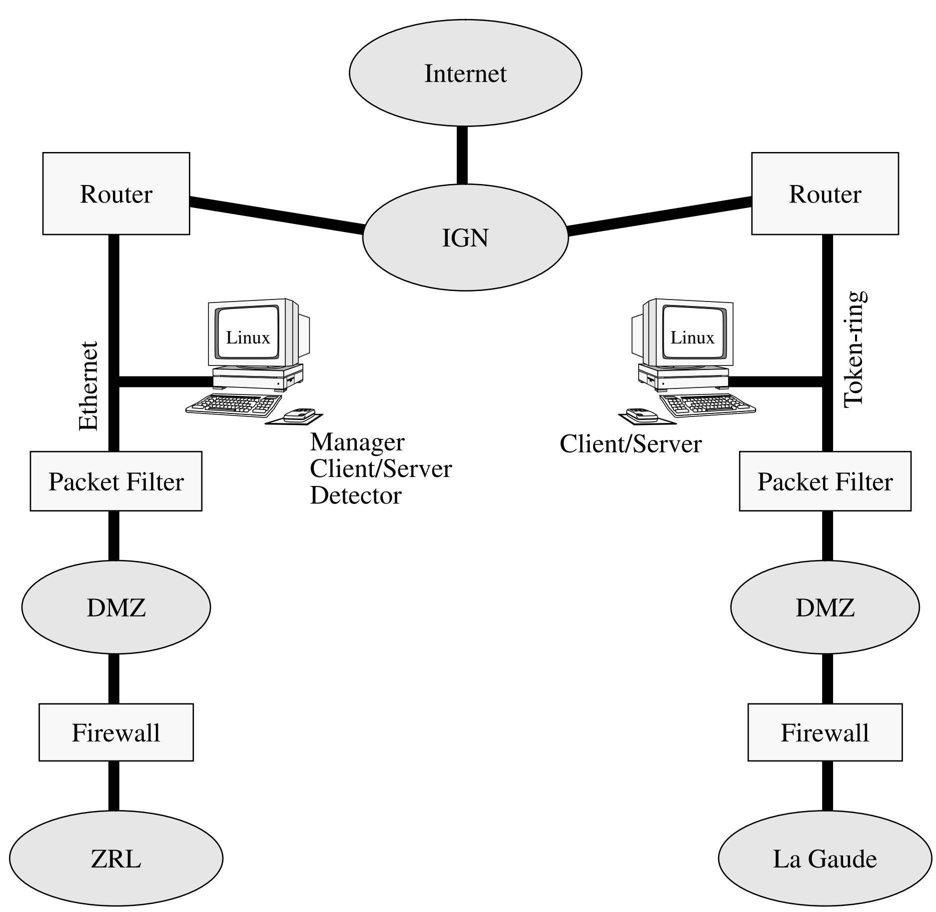

Figure 11. Test network architecture

Our colleagues at LaGaude had only to install a host with Linux (RedHat 4.2) and IP-aliasing compiled into the kernel. Once the host connected to the network, we did all the remaining installation remotely.

We could not install a detector at the La Gaude site because the link is token-ring and the packet capture library of Bro does not support it. Moreover, many token-ring interfaces can not be put in promiscuous mode [1].

6.2 Results

These tests allowed us to discover non-trivial bugs and interactions between the components. This section highlights interesting aspects.

6.2.1 Real size environment

Putting the system on a real-size environment and not a small one as during development revealed several bugs. Some problems related to memory management did not show up during development. In fact, the monitored traffic in the development environment did not reproduce the diversity of the real-size environment. Therefore, the early tests did not explore all the paths of execution of the manager. Following a Murphy's rule, a bug was waiting at that exact place.

The same type of problem arose with Bro. The current implementation of Bro allocates memory for each new connection it is monitoring, but does not release it (this is a non-trivial problem, see [15] for more details). Therefore Bro crashes the detector host after a memory exhaustion, although it never did that on the development network. To prevent this, a mechanism has been implemented in the detector front-end which monitors the memory usage, and kills and restarts Bro when a certain level has been reached. It then asks the manager (by sending a DUMP request) to resend the whole rules table.

The handling of this new DUMP message (not planned in the original specification) led to new bugs linked to signals and alarm management. The solution of this new problem involved the coding of a new timer management class.

6.2.2 Long distance connection

A connection from Zurich to La Gaude goes through UK, Germany, and Paris before going down to La Gaude. This leads to long delays and small throughput, which in turn triggered time-outs too often. These timeout values must therefore be part of the configuration file of the system, and potentially adapted for each new network setup.

6.2.3 IP-aliasing interactions

During a session, and only during that session, a new interface is configured with a temporary IP address (IP-aliasing). The server and client front ends are responsible for this configuration. Removing the interface occurs after receiving the STOP request for the server and after the end of the protocol-dependent script on the client.

Unfortunately, this led to a race condition, where at the application layer the session is over, but not at the transport layer: When the application quits, the session is over and the client removes the interface. The last FIN packet of the TCP stream was sent by the kernel[1] but the corresponding ACK was not received when the interface was removed. This ACK never reaches the transport layer (it does not have the correct address). The TCP state machine will therefore never terminate (stuck in the CLOSE_WAIT or LAST_ACK state, see [18] and Figure 12 for more details of the TCP state machine), and periodically retransmits the last FIN, although the interface is no longer configured[2].

From the manager point of view, the session is over. This means that the next time the manager receives a HOT message from Bro for this session, it will trigger an alarm.

The current version of Bro sends only one[3] HOT message for each 4-uple identifying the connection (IP_src, IP_dst, Port_src, Port_dst). If this is the case, these repeated FINs should not trigger Bro. But if Bro is restarted, after a memory exhaustion for example (see 5.2.4 ”Detector component” on page 27), its internal tables are cleared. Therefore, when the FIN packet is repeated one more time, Bro will send a new HOT message, triggering an alarm on the manager.

1.This occurs on the client host at the end of a Telnet session (passive close, the server initiated the termination of the connection), and on the server host at the end of an FTP session (the client initiated the termination)
2.The kernel is able to send packets with the temporary IP addresses even when the *ifconfig* command no longer reports it. On the other hand, a packet received with this same address is discarded.
3.Of course, it sends other messages when application layer rules are fulfilled.

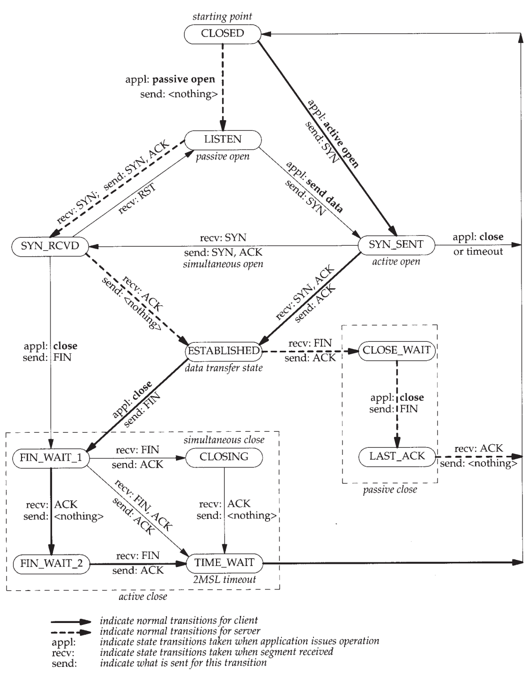

Figure 12. TCP state transition diagram[1]

We said above that this is a race situation, but it is not a tight time schedule: it occurs if the interface is removed before the last ACK has been received, which can be a long time on a long-distance link.

To avoid this problem, we have to wait at the end of a session (as seen by the application layer) before removing the network interface. The duration depends on the network RTT[2]. During our tests, a 5-second wait was not enough, and a 45-second wait removed all false alarms.

These long waiting times consitute a risk for the system. During waiting times, the server is configured to accept a connection[3]. Moreover, the session is not over from the manager point of view. An intruder can connect to the server, use the sniffed password and enter the system without being detected. This is possible because the Detectors Manager knows only the two IP addresses used to generate that session, not which TCP-ports have been chosen by the operating system. If the intruder spoofs the source

1.From *TCP/IP Illustrated vol.1* [18]

2.Round Trip Time

3.Indeed, the manager waits for the completion of the client script before sending the STOP request to the server.

IP address, the Detectors Manager will discard the HOT message sent by Bro. The Detector Manager does not have the intelligence to remember that it already received a HOT message for this session, see 7.1.3 ”Better interpretation of HOT messages” on page 44 for improvements. Once in, he could use one of the many attacks to gain root access and compromise the host. He has to log in during the wait time (after that, the server is unconfigured and the account removed).

6.2.4 Token-ring driver instabilities

The token-ring driver of Linux causes some unexpected crashes. We can reproduce such crashes by removing an aliased interface while a telnet session is running.

6.2.5 False alarms interpretation

During the first weeks of testing, there were many alarms. By looking at the alarm file and session file, we were able to interpret them. They were two types of alarms:

- The first one is a new session (the manager learns of its existence for the first time) with both IP addresses of a previous session (two *host* rules triggered the alarm) and where the client was on a remote host (from the manager), namely at the LaGaude site. Extract of the *Alarm* file:

```
Detector ipg3 restarted on Mon Jun  1 12:27:31 1998.

ALARM on Mon Jun  1 13:26:05 1998 trig. by a session detected 0h 0m 0s ago
        from ipg3: 194.196.13.241[16190] -> 195.212.119.67[23]
        Rule id: 22 type: host generated for the last time 0h 3m 35s ago.
        Rule id: 20 type: host generated for the last time 0h 3m 35s ago.
```

This previous session was canceled after a timeout of the manager.
Extract of the *Session* file:

```
Mon Jun 1 13:22:29 1998 : Start 194.196.13.241->195.212.119.67 : 23 operator uetl1B root
Mon Jun  1 13:24:37 1998 : Stop with error...
```

This kind of alarm may be caused when the abort signal reaches the client host only after the manager has removed the session from its active sessions table. During this short interval, the script may start a session, which in turn triggers an alarm. A longer timeout on the manager avoided premature session cancelation, and no subsequent alarm was detected.

- The second type of alarm is an already detected session (seen for the first time a couple of minutes before) for which the manager did not receive a FIN message from Bro. Two *host* rules of a previously generated session did trigger the alarm. The detector was restarted about one minute before the display of the alarm.
Extract of the *Alarm* file:

```
Detector ipg3 restarted on Fri May 29 18:20:15 1998.

ALARM on Fri May 29 18:21:47 1998 trig. by a session detected 0h 12m 39s ago
        from ipg3: 195.212.119.65[1362] -> 194.196.13.244[23]
        Rule id: 21 type: host generated for the last time 0h 15m 12s ago.
        Rule id: 17 type: host generated for the last time 0h 15m 12s ago.
```

This second kind of alarm can be explained by the behavior of the TCP stack and IP-aliasing. At the end of a session when removing the interfaces, if the TCP state machine has not received the last ACK, it gets stuck and sends FIN packets with exponential backoff as explained in § 6.2.3. Once the manager has removed the session from its table, one of these late packets will trigger the display of an alarm. This was difficult to determine, because Bro does not send a HOT message for each packet it sees, but for each new 4-uple (IP_src, IP_dst, Port_src, Port_dst). The latter FINs where therefore not mentioned by Bro. Once restarted, after a memory exhaustion, it would resend a HOT message to the manager, triggering the display of the alarm. Increasing the delay between the end of a session and the unconfiguration of the network interfaces suppressed these alarms.

We therefore think that these alarms were false alarms. Moreover, if these alarms reported real attacks, this meant that the attacker conscientiously spoofed the source IP address. The attacker never tried a connection from another host, nor tried to log onto another host with the captured login/password pair. This seems to be a very strange behavior, but intruders may behave strangely.

6.2.6 No attacks detected

If our interpretation of the false alarms is correct, and because we did not see new alarms after the time-out modifications, we can conclude that we did not detect the presence of a sniffer on the IGN network between Zurich and La Gaude. This was actually the expected result, because this network is independent of the Internet and operated by IBM.

7. Improvements

In light of the implementation and tests, the following improvements may be part of future work.

7.1 New functionalities

The current version of the sniffer detector is a prototype and not a finished product. New functionalities must be added before turning this piece of software into a real intrusion-detection system.

7.1.1 Protocols

More protocols must be implemented. This requires two steps.

The first step is to upgrade Bro (the detection engine) to be able to interpret new protocols. Missing protocols are ICMP, DNS request, but also NFS, SMTP or WWW. Moreover, the ability to find these protocols when connecting to non-standard ports would be a very interesting feature. Unfortunately, this leads to a huge amount of traffic to analyse because the network filter must be set up to capture everything.

The second step is the implementation of the client and server scripts. They must parse the START message to get the protocol options and do actions suited to the session to be generated. The server has to configure itself to accept a connection from the client, and the client has to simulate a user interaction with the system.

7.1.2 Messages

The number of messages exchanged between the client or server and the manager must be increased to provide more information to the manager on the states of the front-ends and on the session in progress. For example, the client could send messages when:

- it parsed the START message correctly
- the protocol specific script has been launched
- the session is over
- the interface is unconfigured

This would let the manager distinguish much more easily between session-induced HOT messages and attack induced ones. This also reduces the timing window during which the server is vulnerable and thus the exposure of the system.

The server may also send new messages to the manager (the Sessions Manager could transmit these messages to the Detectors Manager to improve its correlation functionality), indicating for example that someone connected his host. This may also be implemented in a host-based detector (as opposed to a network-based detector such as Bro).

7.1.3 Better interpretation of HOT messages

The implementation of the Detectors Manager checks only whether a HOT message is a direct consequence of a current session. A much more sophisticated algorithm should be used, taking into account which HOT messages it *should* see and which are the messages already received. For example, if we generate a Telnet session, we should see a first Telnet session and then perhaps an Ident one. If we receive a second notification for a Telnet session, we should conclude that we did not generate that one and it could be an intrusion.

This is an extension of what has already been done: if the detector receives a new connection indication that both IP addresses are from a current session, it tags them as Warnings only. Data connections opened during an FTP session are such examples.

7.1.4 Session randomization

It could be interesting to have different scenarios for each protocol. The client would then choose one randomly. This would make the sessions more natural for a human sniffer, contrary to a complex and automated one, which would log only the login and password pair.

7.2 System enhancements

The following enhancements do not involve a modification of the detection method, but provide a better framework.

7.2.1 Attack resistance

We have to account for the fact that our system may be the target of an attack. Although we may detect the attack just before the intrusion, it may compromise the hosts involved in the system and help the intruder go further.

During our tests, only the required services (telnetd, ftpd, identd, sshd) were running.

The system is especially vulnerable when the server is configured to accept a new connection. If the intruder manages to get in at that time and spoofs his source IP address, it would not be detected by the manager. To avoid this, the server could accept only one session instance at a time. A modification of tcpwrapper [16] could do the job.

IP-firewalling could help avoid an intrusion, although it does not protect against source IP-spoofing, or TCP-Hijacking [3], [4].

Another protection would be to modify the service daemons to run in protected and limited environments. If the intruder manages to connect the host without being detected, these crippled daemons will not let him compromise the system. They could also generate additional status messages for the manager, thus improving the detection capability.

7.2.2 System distribution

The manager is a single process that facilitates transmission between the three components of the manager of the sessions and rules tables via memory pointers. This leads unfortunately to problems of concurrency, program interruption and so on.

It would be much more interesting to split the manager into three independent processes. They could then be located on different hosts and enable some kind of replication. With a mechanism able to restart components in case of a failure, we would have a nicely distributed fault-tolerant system. This architecture could also lead to a distributed intrusion-detection system, compliant with the *Common Intrusion Detection Framework* [17]. The detector and the Sessions Manager would generate events to the correlator[1] (which is now part of the Detectors Manager). The correlator would also get other sources of information and distinguish between real and false alarms.

7.2.3 Inter-component communication

We use ssh as a secure communication channel[2]. Indeed, we do not need to reinvent the wheel and build a cryptography, key management and network package. The drawback is that we use pipes to communicate between the processes, inducing buffering problems. A chain of processes is also a chain of point of failure. Dealing with a broken pipe somewhere in the chain is not a trivial task.

Therefore, we have to consider replacing ssh with a custom implementation of a secure communication channel, directly accessible inside the code, without forking a process.

1.The analysis engine, A-boxes, in the CIDF model

2.During the development of this prototype, a vulnerability was discovered by Ariel Futoransky and Emiliano Kargieman of CORE SDI S.A. A new version of ssh (1.2.25) patches this vulnerability. This is the strength of publicly available protocols involved in security: Many people worldwide are tracking bugs and vulnerabilities, resulting in a much better specification.

Such an implementation would improve the manageability and therefore reliability of the channels.

This represents of course a huge amount of work, but this issue is essential for future distributed intrusion-detection systems. The chosen solution should be the result of a profound dialogue between designers and programmers.

8. Conclusion

Intrusion detection is becoming a main component of network security. Until now, the measures to protect a network consisted mainly of the use of firewalls and authentication to implement an access policy.

Intrusion detection goes one step further: it admits that intruders can still get past the protections installed, and that we need to be able to detect them. Moreover, an intruder may not necessarily come from outside of the corporate network but also from inside.

The sniffer detector is one additional component in the intrusion-detection toolbox. Unlike conventional intrusion detection systems, it uses a new *bait* technique. This idea is beginning to make its way into the minds of developers[1].

The sniffer detector prototype has a promising future, because it arouses interest among IBM consulting services. It also let us delve into some of the problems that could arise during the development of a distributed intrusion-detection system.

On a more personal basis, it gave me the opportunity to learn a great deal about the hacker community, intrusion techniques, and detection methods. I was particularly impressed by the fact that approximately five new vulnerabilities are discovered every week in various operating systems.

Working in a research lab such as IBM's, surrounded by top researchers, was a great experience. It gave me a new perspective on research. It helped me with the difficult task of better identifying my future professional objectives.

1.Network Associates are developing a product, code-named Honey Pot, that is essentially a decoy network within a network. The idea is to lure hackers into the trap.
http://www.zdnet.com/intweek/daily/980601k.html

9. Acknowledgments

I am sincerly grateful to the following people for their help and support of my research:

- The GSAL group at IBM Zurich for letting me pursue my Master's thesis
- Marc Dacier and Hervé Debar for their welcome, their help, and their constructive suggestions in order to put some structure in this work and for their review of all the documents I produced
- Refik Molva for his supervision from Eurecom
- Dominique Alessandri for his moral support during the debugging phase, and for his judicious advice about C++ coding
- Andreas Wespi for managing of the GSAL experimentation network
- Pierre-Dominique Bauce, Hugo Reibel and Michael Niksch for letting me install the sniffer detector between Zurich and La Gaude, and for occasionally pushing the reset button as needed...
- Vern Paxson (Network Research Group LBL) for letting us use *Bro*
- Lilli-Marie Pavka for correcting my English
- The entire Security department of IBM Zurich for their warm welcome

10. Bibliography

[1] *Network Packet Capture Faq*, Internet Security Systems, http://www.iss.net/vd/packcapt.html

[2] *CERT Advisory CA-94:01*

[3] Brecht Claerhout, *IP-spoof.txt, a short overview of IP spoofing*, http://www.rootshell.com

[4] Laurent Joncheray, *A Simple Active Attack Against TCP*, April 1995, http://rootshell.connectnet.com/docs/tcp_attack.ps.gz

[5] *tcpdump*, LBNL's Network Research Group, http://www-nrg.ee.lbl.gov/nrg.html

[6] *libpcap*, LBNL's Network Research Group, http://www-nrg.ee.lbl.gov/nrg.html

[7] Steven McCanne, Van Jacobson, *The BSD Packet Filter*, Lawrence Berkeley Laboratory

[8] *NFR, a Network Flight Recorde*r, http://www.nfr.com

[9] Vern Paxson, *Bro, a system for detecting network intruders in real-time*, NRG-LBNL http://www-nrg.ee.lbl.gov/nrg-papers.html

[10] *IP aliasing Linux mini-HOWTO*, http://sunsite.unc.edu/LDP/HOWTO/mini/IP-Alias.html

[11] *The Expect Home Page*, http://expect.nist.gov/

[12] Don Libes, *Exploring Expect*, O'Reilly and Associates, ISBN 1-56592-090-2.

[13] *SSH (Secure Shell) Remote Login Program*, http://www.cs.hut.fi/ssh/

[14] Ulrich Flegel, *The Interaction between SSH and X11*, 1997, http://rootshell.connectnet.com/docs/ssh-x11.ps.gz

[15] Thomas H. Ptacek, Timothy N. Newsham, *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*, Secure Networks Inc., January 1998 http://www.securenetworks.com/papers/index.html

[16] Wietse Zweitze Venema, *Tcp_wrapper*, ftp://ftp.win.tue.nl/pub/security/index.html

[17] S. Staniford-Chen, *Common Intrusion Detection Framework*,
http://seclab.cs.ucdavis.edu/cidf/

[18] W. Richard Stevens, *TCP/IP Illustrated Volume 1*, Addison-Wesley, ISBN 0-201-63346-9

[19] Bjarne Stroustrup, *The C++ Programming Language 2nd ed*, Addison-Wesley,
ISBN 0-201-53992-6

[20] RFC 959, *File Transfert Protocol (FTP)*
ftp://sunsite.cnlab-switch.ch/doc/standard/rfc/9xx/959

[21] RFC 854, *Telnet Protocol Specification*
ftp://sunsite.cnlab-switch.ch/doc/standard/rfc/8xx/854

[22] Steve M. Bellovin, *There be dragons*, August 1992, AT&T Bell Laboratories
http://rootshell.connectnet.com/docs/dragons_bellovin.ps.gz

Annex I - Overview & User Guide

1. Introduction

This document includes the description of the file hierarchy of the Sniffer Detector and a user guide. In the first part, it describes the organization of the directories, as they are on a system in operation. The second part takes each component and describes the purpose of each file. Finally, the user guide guides you through the installation and utilization of the system.

As a reminder, Figure 1 shows the organization of the system[1].

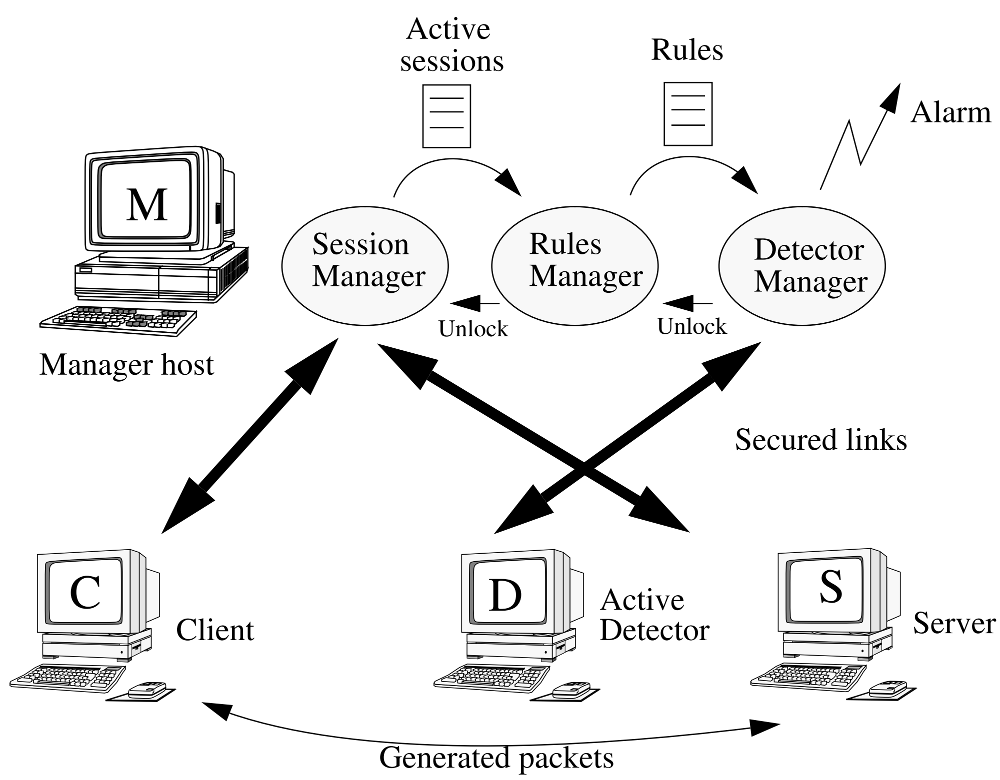

Figure 1. Architecture

The manager sends requests to the client and server to generate sessions. It updates the rule tables of the detector accordingly, and receives the alarms from the detectors. We find the same organization in the file hierarchy.

1.The whole project is described in details in the document "Sniffer detector - Industrial thesis", Stephane Grundschober, IBM & EURECOM, spring 1998

2. Directory structure

When one checks out the SnifferDetector project from the CVS repository[1], a new directory is created. From this directory, one has a view of the whole package:

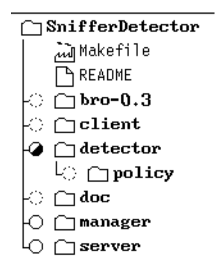

Figure 2. Root directory

On every host involved in the system, one must create a new account and group. For example, one could create a user "sniff" and the group "ids"[2]. The SnifferDetector directory must be copied to ~/sniffer_detector on the client/server/detector/manager hosts. The scripts and the manager have this path "hard coded".

In this directory, one finds:

- bro-0.3 : The source code for bro[3], the network monitoring tool
- client : Contains the client front-end and the sessions generation scripts
- server : Contains the server front-end
- manager : Contains the source code for the manager
- detector : Contains the detector front-end, and a copy of bro (setuided root)
  The policy subdirectory contains the scripts used by bro
- doc : Location of this document.

In this directory, one also finds a README file, explaining which packages are needed to compile the system, and how to install it.
The Makefile lets one configure, compile, install and clean the distribution.

1.cvs checkout -d /g/gsal/SOFT/master SnifferDetector
2.If you use the adduser command (Linux specific), the files /etc/passwd (and /etc/shadow if it exists) and /etc/groups are modified. A new directory is created, usually under /home. In this case, it would be /home/sniff. In the following, we will refer to this directory as ~/. (tilde).
3.ftp://ftp.ee.lbl.gov/papers/bro-usenix98-revised.ps.Z

3. Files

This section introduces each directory and gives details on the purpose of each file.

3.1 Client

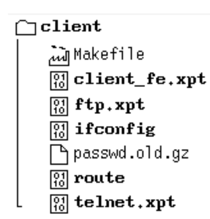

Figure 3. Client subdirectory

The client is essentially made of Expect[1] scripts:

- client_fe.xpt : Front end. It receives the START request, and launches either telnet.xpt or ftp.xpt.
- ftp.xpt : Generates a FTP session
- telnet.xpt : Generates a Telnet session
- ifconfig and route : a copy of the system commands with the setuid bit set to gain root privileges. This lets the front-end reconfigure the network interfaces to generate the sessions with different IP-addresses.
  These two files are copied by the "make install" procedure.
- passwd.old.gz : used by the FTP script to fake a transmission of something interesting.
- Makefile : contains only a "real_clean" rule deleting the local copy of ifconfig and route.

1.http://expect.nist.gov/

3.2 Server

Figure 4. Server subdirectory

- server_fe.xpt : Front-end, receives the START and STOP requests. It reconfigures the interfaces in order to be able to generate sessions with different IP addresses. It also creates accounts for the duration of the session.
- server_wrapper.c : The front-end need root privileges to manipulate the passwords files and the network interfaces. Unfortunately, a security feature of the system does not allow setuid scripts to be executed. Therefore, this wrapper (which is setuid root) simply calls the expect script to gives it superuser privileges.
- Makefile : "build" rule for the server_wrapper, and "real_clean" rule deleting the server_wrapper.

3.3 Detector

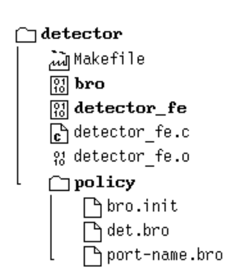

Figure 5. Detector subdirectory

- bro : a copy of bro, but setuid root in order to have access to the interface in promiscuous mode.
- detector_fe.c : source code for the detector front-end. It launches bro, and transmits the Look_xxx requests. It also deals with RESET and QUIT requests. It also monitors Bro memory size, and restarts Bro if it becomes too big. It also restarts Bro if it receives a SIGCHLD signal.
- Makefile : rule to build the detector front-end and to clean the directory.
- policy : directory containing the scripts telling bro what to do.

3.4 Manager

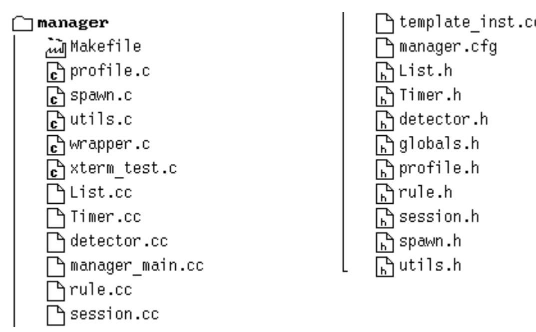

Figure 6. Manager subdirectory

- manager : the manager executable, include the three managers (session, rule and detector manager)
- manager.cfg : configuration file, containing the list of the clients, servers, detectors, and protocols
- manager_main.cc : implements the user interface
- session.cc : implements the session manager. This class reads the configuration file (manager.cfg), and controls the servers and clients.
- rule.cc : implements the rule manager. This class receives from the session manager a session list, and has the intelligence to derive the rules from it. It then transmits these two lists[1] to the detector manager.
- detector.cc : implements the detector manager. This class is in charge of managing the detectors. This includes starting them, transmitting the new rules tables, and receiving the alarm messages. It also distinguishes between session-generated alarms and real alarms. Real alarms are written in an alarm file. The name and path of this file is defined in the configuration file (manager.cfg) under the *Alarm_file* entry.
- List.cc, List.h, template_inst.cc : This class is used by the timer manager. It implements a list template. Explicit instantiation is done in the template_inst.cc file in order to be able to do separate compilation.
- Timer.cc, Timer.h : implements a timer manager. This class lets one schedule asynchronous events and manage simple wait calls (like a sleep).
- global.h : some global definitions, included by all the manager classes. It is essentially C structures holding active_sessions, rules, ... data.

1.The data structures used are defined in the file globals.h.

- spawn.c : implements a "fork" with redirection of stdin/out to file descriptors.
- utils.c : implements a random function
- profile.c : implements a set of functions used to parse the configuration file
- wrapper, wrapper.c : this wrapper forks the command line arguments and copy data from the parent to the child, and vice-versa. In the latter case, it sends the parent a SIG_USR1 signal.

4. Chain of processes

When running, the system is a chain of processes communicating via pipes (redirection of stdin/out). Figure 7 shows a typical situation when generating a session (shells have been omitted). A bold typeface identifies processes running with UID 0.

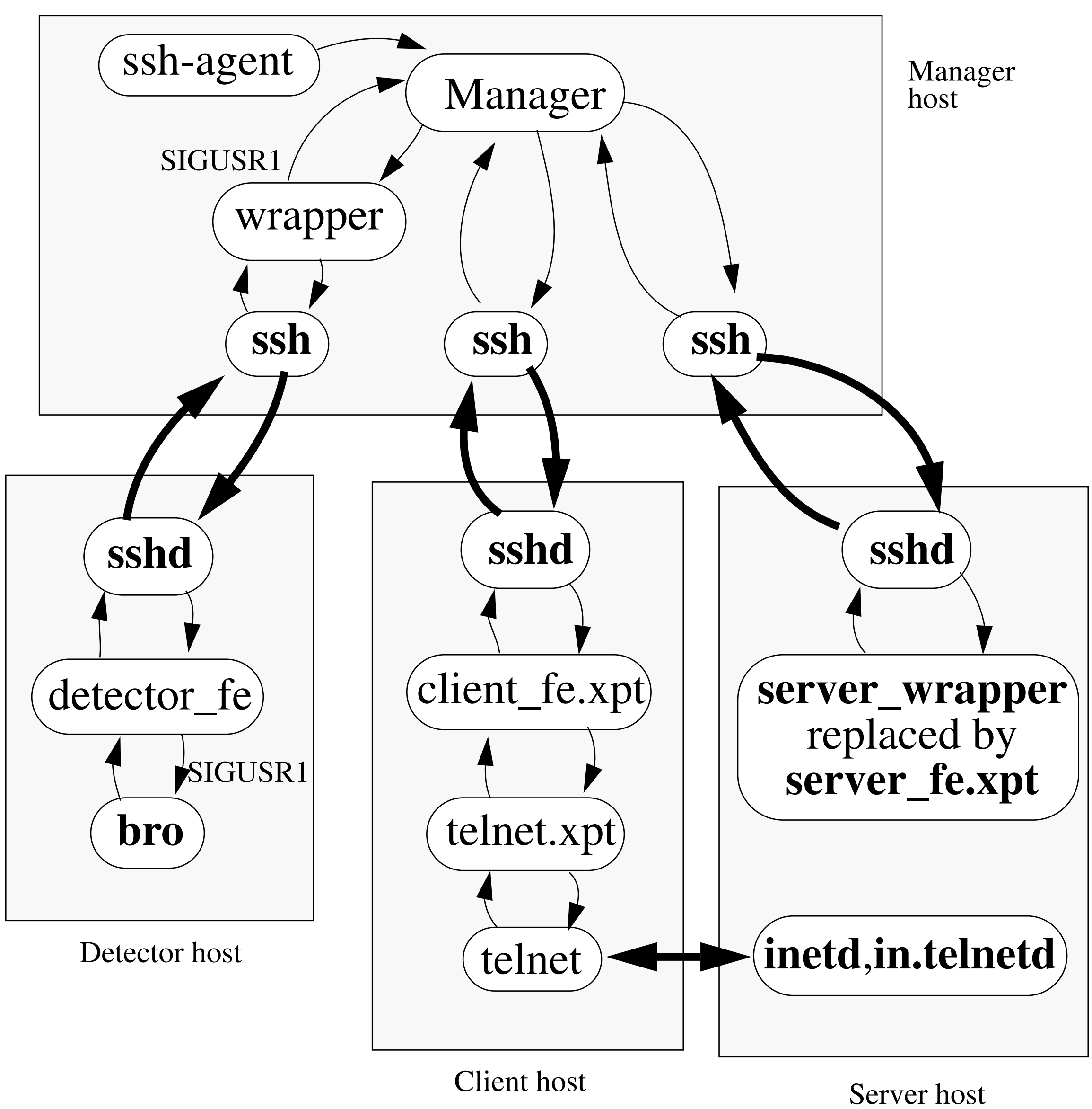

Figure 7. Chain of processes

5. User Guide

5.1 Requirements

- The current version runs on Linux, with **IP-aliasing**[1] enabled: the clients and servers reconfigure one of their IP-addresses to generate the sessions.

- The communications between the various components is done via **ssh**[2] (v1.2.25).

- The clients and servers scripts use **expect** (v5.25) and **tcl** (v8.0)[3].

- To compile the detector (bro), **libpcap**[4] (v0.4a5) must be installed.

- To compile the whole system, **g++** (v2.7.2.1) is required.

The version numbers indicate which versions have been used for the tests. Other versions may also work, but this has not been tested.

5.2 Compilation

In the sniffer_detector directory, type *make configure*. It will launch the configure script in the bro directory. If everything went well, you can do a *make* to compile the whole system.

If the compilation went well, you still DO NOT HAVE a working system. You must first go through the complete installation procedure, as described in the following paragraph.

5.3 Installation

On every host involved in the system, you must create a new account and group. For example, you could create a user "sniff" and the group "ids"[5]. You must copy the distribution in a directory named *~/sniffer_detector*. You may edit the *Makefile* if you do not want to compile the whole system on each host (some may be only clients/servers, others only detector). See annex 6.1 on page XVII for an example of a modified makefile.

A sshd deamon must be running on each host (except the manager host if it is its only functionality). The public key of the manager must be copied to each system, in the *~/.ssh/authorized_keys*. The configuration file (*/etc/sshd_config*) must authorize RSA

1. http://sunsite.unc.edu/LDP/HOWTO/mini/IP-Alias.html
2.http://www.cs.hut.fi/ssh/#ftp-sites
3.http://expect.nist.gov/
4.ftp://ftp.ee.lbl.gov/libpcap.tar.Z
5.If you use the adduser command (Linux specific), the files /etc/passwd (and /etc/shadow if it exists) and /etc/groups are modified. A new directory is created, usually under /home. In this case, it would be /home/sniff. In the following, we will refer to this directory as ~/. (tilde).

authentication and avoid password or rhost-like authentication. See annex 6.2 on page XVIII.

5.3.1 On a client, server host

First, make sure that IP-aliasing is enabled[1], and expect/tcl is installed.

To be able to generate some "interesting" informations for a sniffer, we would like to generate for example a telnet session with a *root* login. We need to create an account with that login, but without giving it the privileges associated with the userid 0. Therefore, you should rename the login name 'root' to 'root2' for example, and do the same with any other login you are planning to generate (operator, ...). This modification concerns the */etc/passwd* file, */etc/shadow* if it exists and */etc/group*. Unfortunately, many configuration files use the 'root' login name. The first that comes to mind is */etc/inetd.conf*, where each service is configured to run under the root ID. You must therefore change all references of 'root' to 'root2' and restart inetd. There are many configuration files affected by these modifications. cron-related files may need an update. Which files and how many need to be updated depends on your installation. See annex 6.3 on page XIX for a list of files which may need to be updated.

Do not forget to authorize this new 'root' user to access to the ftp service (*/etc/ftpuser*), and let it also telnet in (*/etc/securetty*). See annex 6.3 on page XIX for the modifications to do.

You must then edit the files *server_fe.xpt* and *client_fe.xpt* to configure the interface and gateway network address. You must also define in *server_fe.xpt* what UID[2] and GID can be used for the **temporary** accounts:

```
################# configuration ###########
set interface tr0:0
set gateway 192.168.2.1
set netmask 255.255.255.0
set user_id 25000
set user_group 100
###########################################
```

Edit the Makefile in the sniffer_detector directory. There are two lines at the beginning that define the login name of the user with ID 0 (this may not be 'root' if you modified your configuration files) and the group name of the user under which the system will be running (this is usually the name of the account where you installed the distribution, and was *ids* in the example given at the beginning of this section):

1.If the command 'ls /proc/net/alias*' prints something, then IP-aliasing is enabled. If not, you must compile the kernel to enable addresses aliases, compile the ip-alias module, and start kerneld to load dynamically the module into the kernel. See http://sunsite.unc.edu/LDP/HOWTO/mini/IP-Alias.html
2.These IDs must be unused (free) ones.

```
$ more Makefile
RID=root2
GID=ids
...
```

Edit these two lines corresponding to your local configuration.

You may now *su* to the super-user account (*su - root2*) and do a *make install*. This operation will copy some system files (ifconfig, route) in local directories. It will then set the *suid* bit and change the owner to 'root2'. This will let the scripts manipulate the interface configuration and route table without running under root identity.

5.3.2 On a detector host

You just need to edit the Makefile to update the RID and GID variables corresponding to the local setup. (You may also delete some rules[1] if this machine hosts only the detector). A *make install* under the super-user identity will copy bro under the *~/sniffer_detector/detector* subdirectory, and set his *suid* bit in order to let it access the ethernet interface in raw mode.

NOTE: libpcap, a library implementing a host and network independent packet capture interface, used by bro does not support tokenring. The detector must therefore be placed on an ethernet segment, or fddi (not tested).

5.3.3 On the manager host

There is no need to do a *make install*; the manager does not run under root privileges.

There are two steps to configure the manager:

1) edit the configuration file ~/sniffer_detector/manager/manager.cfg. The manager will always look in the current directory (at execution time) for this file.

The first section defines the client hosts. You must define how many clients are available (Clients = xx), then for each client its name or IP address (Client_x = clientname). For each client, you must also provide a list of free IP addresses it can use to generate the sessions (Client_x_IP_addrs = yy and Client_x_IP_addr_y = IPADDR fields). These addresses must be routable between all the client/server hosts, and from the outside world. Here is an example (with dummy IP addr...):

```
/* clients */
Clients = 2
/* peccia */
Client_1 = peccia
/* fake IP addresses to use */
Client_1_IP_addrs = 2
Client_1_IP_addr_1 = 192.168.2.10
Client_1_IP_addr_2 = 192.168.2.11
```

1.See annex 6.1 on page XVII for an example of a modified Makefile

```
/* responder */
Client_2 = responder
/* fake IP addresses to use */
Client_2_IP_addrs = 2
Client_2_IP_addr_1 = 192.168.128.20
Client_2_IP_addr_2 = 192.168.128.21
```

The second section defines the server hosts, in the same way as the clients do.

The third section defines the protocols.

Currently, only two protocol types are defined:

Protocol	type	# of options	options
Telnet	23	3	1: login 2: passwd 3: (root|user)
FTP	21	2	1: login 2: passwd

The third option of the telnet protocol defines whether the shell prompt should be a ’#’ symbol (root) or a normal ’$’ symbol (user). This is to fake the common behavior of shells presenting a ’#’ as prompt when the user has an uid=0.

```
/* protocols */
Protocols = 1

/* telnet */
Prot_1_type = 23
Prot_1_options = 3
Prot_1_opt_1 = root
Prot_1_opt_2 = zklc5FG
Prot_1_opt_3 = root
```

The fourth section defines the number of detectors and their names.

```
/* detectors */
Detectors = 1
/* peccia */
Detector_1 = peccia
```

The last section (misc) defines miscellaneous parameters:

- Alarm_file: the complete path and name of the file where the manager will write the alarms and warnings messages.

- Session_file: the complete path and name of the file where the manager will write a summary of which session has been generated and when.

- Session_timeout: defines how many seconds the manager must wait between the sending of a request and the acknowledgment from the client or the server.

```
/* misc */
Alarm_file = /home/stg/sniffer_detector/manager/alarm.out
Session_file = /home/stg/sniffer_detector/manager/session.out

Session_timeout = 360
```

2) ssh configuration:

Once you have generated your key pair and copied the public version on each subordinate host (in *~/.ssh/authorized_keys*), check whether the connection works:

- first run the ssh-agent: `ssh-agent $SHELL`

- add your key: `ssh-add`
  you must enter your passphrase to add the key

- try to connect with `'ssh hostname'` to every host involved in the system from the manager. You must use the same names or IP addresses defined under the Client_x..., Server_x... and Detector_x... entries in the configuration file (*manager.cfg*). SSH will add the host keys to the known hosts file[1]. You may also do it by hand.

The connection should be established without prompting for passphrase, password, etc... See the ssh faq[2] or man pages if something does not work as expected.

```
peccia:~$ ssh peccia
Host key not found from the list of known hosts.
Are you sure you want to continue connecting (yes/no)? yes
Host 'peccia' added to the list of known hosts.
Last login: Mon May 18 11:56:43 1998 from internet-gateway
Linux 2.0.29.
No mail.

peccia:~$
```

1.~/.ssh/known_hosts
2.http://www.uni-karlsruhe.de/~ig25/ssh-faq/ and http://www.tac.nyc.ny.us/~kim/ssh/

5.4 Utilization

Once an ssh-agent is running with your manager key loaded in memory, you can run the manager. You will get a menu that looks like this:

```
 Sniffer Detector control panel

 1) Dump clients list
 2) Dump servers list
 3) Dump protocols list
 4) Dump rules defined by the rule manager

 5) Generate a new session
 6) Generate a random session
 7) Toggle automatic session generation. Status: off
 8) Rules garbage collection
 9) Beep is on
10) Print warning is on

 0) Quit


      Please enter your choice:
```

The first three choices let you display what you configured in the *manager.cfg* file. The choice #4 display the rules defined by the rule manager, based on the sessions that have been generated.

You can generate a session either manually (#5, you must pick manually each client, server, IP address and protocol), semi-manually (#6, the system picks a session at random and asks whether you want to generate it) and automatically. In the latter case (#7), you will be asked to enter a repetition time (the time between the end of a session and the beginning of the following). The system will then automatically generate random sessions. This is what the menu looks like when an automatic session is enabled:

```
Sniffer Detector control panel

 1) Dump clients list
 2) Dump servers list
 3) Dump protocols list
 4) Dump rules defined by the rule manager

 -) Generate a new session
 -) Generate a random session
 7) Toggle automatic session generation. Status: on, every 10 minutes
 8) Rules garbage collection
 9) Beep is off
10) Print warning is off

 0) Quit


      Please enter your choice:
```

Note that when automatic session generation is enabled, you cannot generate other ones manually (#5 and #6). Just re-enter choice #7 to disable the scheduling.

Choice #8 lets you remove old rules by specifying a threshold age.

Choice #9 toggles the beep when an alarm or warning is written to the alarm file.

Choice #10 toggles the writing of warning messages to the alarm file. If it is disabled, the beeps are also disabled for warnings.

When a session is generated, information is displayed that lets you follow what is going on:

```
17: Connecting responder (server side)
17: Start Server Request transmitted, waiting...
17: Answer received... OK!
17: Connecting ipg3 (client side)
17: Request transmitted, waiting...
17: Answer received... OK!
17: Stop Server Request transmitted, waiting...
17: Answer received... OK!
```

The first number is the session id. When something goes wrong, these messages may help to track down the cause of the problem.

After a while, the detector takes too much memory[1]. The detector front end automatically kills it and restarts a new detector process. It then asks the manager to redump the whole rule table. This is presented by the manager by these two lines:

```
Re-dumping the whole rule table upon request
Dumping done
```

This occurs about every 30 minutes on our network, with a load of about 36kbytes/s. The dumping time is about 30 seconds (rules dependant).

1.This is traffic dependant. Bro takes at startup about 1 Mbytes of memory. The detector_fe lets it grow until 2.8 Mbytes.

5.4.1 Alarm file

When suspicious traffic is detected by the detector, and when this traffic is not a current session being generated, the manager writes an alarm in the Alarm file[1]:

```
ALARM on Fri May 15 17:27:51 1998 trig. by a session detected 0h 0m 0s ago
        from peccia: 128.178.164.7[2087] -> 195.212.119.90[23]
        Rule id: 0 type: host generated for the last time 0h 0m 46s ago.
```

It tells when this session was detected by which detector, from which IP addresses and ports, and finally which rule(s) triggered this alarm. In this case, we see that 'someone' is trying to connect to one of the IP addresses we generated 46s ago. This IP no longer exists, so this connection will never get a prompt.

Another possibility is that the attacker tries another machine with the same login/password pair he sniffed:

```
ALARM on Fri May 15 17:30:14 1998 trig. by a session detected 0h 0m 0s ago
        from peccia: 195.212.119.253[3503] -> 128.178.164.7[23]
        Rule id: 2 type: protocol generated for the last time 0h 3m 9s ago.
```

In this case, the detector has seen a login/password pair (type: protocol) generated for the last time 3 minutes ago.

Unfortunately, generating a session may start other 'side-effect' sessions:

```
WARNING on Fri May 15 13:28:11 1998 trig. by a session detected 0h 0m 0s ago
        from peccia: 195.212.119.68[20] -> 194.196.13.245[31227]
        Rule id: 1 type: host generated for the last time 0h 0m 21s ago.
        Rule id: 0 type: host generated for the last time 0h 0m 21s ago.
```

This is the result of an ftp request. The server opens a connection from the port 20 back to the client. The same problem exists when the server uses *ident* before accepting a connection. The detector manager does not know if it is a side-effect or an attack occurring at the same time a session is generated. It sees on the other hand that both ends are IP addresses of the currently generated session. It tags the alarm as a *Warning* only. Because these warnings generate lots of output, you have the option of disabling it (#10).

The following line[2] is printed when a detector has been restarted after a memory exhaustion:

```
Detector ipg3 restarted on Tue Jun  9 10:47:56 1998.
```

1.Each alarm entry is followed by two blank lines ("\n\n" in a C printf) and flushed out.
2.This line is followed by one blank line and flushed out.

5.4.2 Sessions file

The sessions file records each session generated. This includes the date and time of the start of the session, the IP addresses used, and which protocol and options have been used. Once the session is over, it prints the date and time with the exit status[1]:

```
Tue Jun  9 11:00:56 1998 : Start 195.212.119.68->194.196.13.131 : 23 root zklc5FG root
Tue Jun  9 11:02:51 1998 : Stop ok
```

or if something has gone wrong:

```
Tue Jun  9 11:02:51 1998 : Stop with error...
```

1.This line is followed by one blank line and flushed out.

6. Annexes

6.1 Makefile for a client/server host only

Each rule relating to the detector or the manager has been commented out.

```
RID=root2
GID=gsal

all:
#        cd bro-0.3;make
#        cd manager;make
#        cd detector;make
        cd server;make


install:
#        chmod 700 detector client server manager
        chmod 700  client server
#       cp bro-0.3/bro detector/.;cd detector;chown $(RID):$(GID) bro;chmod\
4750 bro
        cd client;cp /sbin/ifconfig .;cp /sbin/route .;chown $(RID):$(GID)\
ifconfig route;chmod 4750 ifconfig route
       cd server;chown $(RID):$(GID) server_wrapper;chmod 4750 server_wrapper

configure:
        cd bro-0.3;./configure
        @echo ""
        @echo "Don't forget to update the GID variable in this Makefile"
        @echo "before doing a make install..."

clean:
#        cd bro-0.3;make clean;rm -rf bind8
#        cd detector;make real_clean
        cd server; make real_clean
        cd client; make real_clean
#        cd manager;make real_clean
```

6.2 sshd_config

```
# This is ssh server systemwide configuration file.

Port 22
ListenAddress 0.0.0.0
HostKey /etc/ssh_host_key
RandomSeed /etc/ssh_random_seed
ServerKeyBits 768
LoginGraceTime 600
KeyRegenerationInterval 3600
PermitRootLogin yes
IgnoreRhosts no
StrictModes yes
QuietMode no
X11Forwarding yes
X11DisplayOffset 10
FascistLogging no
PrintMotd yes
KeepAlive yes
SyslogFacility DAEMON
RhostsAuthentication no
RhostsRSAAuthentication yes
RSAAuthentication yes
PasswordAuthentication no
PermitEmptyPasswords yes
UseLogin no
# PidFile /u/zappa/.ssh/pid
# AllowHosts *.our.com friend.other.com
# DenyHosts lowsecurity.theirs.com *.evil.org evil.org
# Umask 022
# SilentDeny on
```

6.3 List of configuration files referencing to 'root'

This list is NOT AN EXHAUSTIVE LIST of the configuration files you may modify after changing the name of the user with id 0 from 'root' to 'root2'.

/etc/passwd:
```
root2:x:0:0:root:/root:/bin/bash
...
```

/etc/shadow:
```
root2:61M3bn879wbDk:10245:0:::::
...
```

/etc/group:
```
root2::0:root2
bin::1:root2,bin,daemon
daemon::2:root2,bin,daemon
sys::3:root2,bin,adm
adm::4:root2,adm,daemon
tty::5:
disk::6:root2,adm
...
```

/etc/inetd.conf:
```
...
ftp     stream  tcp     nowait  root2   /usr/sbin/tcpd  wu.ftpd
telnet  stream  tcp     nowait  root2    /usr/sbin/tcpd in.telnetd
...
```

/etc/ftpusers:
```
#root  - before commenting this line out, 'root' was NOT allowed to log in
uucp
news
```

/etc/sudoers:
```
...
# User specification
root2   ALL=ALL
```

/etc/securetty: add these ttys:
```
ttyp0
ttyp1
ttyp2
ttyp3
```

/etc/crontab:
```
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root2
# Run any at jobs every minute
* * * * * root2 [ -x /usr/sbin/atrun ] && /usr/sbin/atrun
# run-parts
01 * * * * root2 run-parts /etc/cron.hourly
```

```
02 1 * * * root2 run-parts /etc/cron.daily
02 2 * * 0 root2 run-parts /etc/cron.weekly
02 3 1 * * root2 run-parts /etc/cron.monthly
# Remove /tmp, /var/tmp files not accessed in 10 days (240 hours)
41 02 * * * root2 /usr/sbin/tmpwatch 240 /tmp /var/tmp
# Remove formatted man pages not accessed in 10 days
39 02 * * * root2 /usr/sbin/tmpwatch 240 /var/catman/cat?
```

/etc/cron.(hourly | daily | weekly | montly)

/etc/logrotate.conf:

```
...
# send errors to root
errors root2
...
```

any other file you might think of...