AN SNMP AGENT FOR STATEFUL INTRUSION INSPECTION

Luciano Paschoal Gaspary
Universidade do Vale do Rio dos Sinos – Centro de Ciências Exatas e Tecnológicas
Av. Unisinos 950 – CEP 93.022-000 – São Leopoldo, Brazil
paschoal@exatas.unisinos.br

Edgar Meneghetti
Liane Rockenbach Tarouco
Universidade Federal do Rio Grande do Sul – Instituto de Informática
Av. Bento Gonçalves 9500 – CEP 91.591-970 – Porto Alegre, Brazil
edgar@cesup.ufrgs.br, liane@penta.ufrgs.br

Abstract: Intrusion Detection Systems (IDSs) have been increasingly used in organizations, in addition to other security mechanisms, to detect intrusions to systems and networks. In the recent years several IDSs have been released, but (a) the high number of false alarms generated, (b) the lack of a high-level notation for attack signature specification, and (c) the difficulty to integrate IDSs with existing network management infrastructure hinder their wide-spread and efficient use. In this paper we address these problems by presenting an SNMP agent for stateful intrusion inspection. By using a state machine-based language called PTSL (Protocol Trace Specification Language), the network manager can describe attack signatures that should be monitored. The signatures to be used by the agent are configured by the network manager through the IETF Script MIB. Once programmed, the agent starts monitoring the occurrence of the signatures on the network traffic and stores statistics, according to their occurrence, in an extended RMON2 MIB. These statistics may be retrieved from any SNMP-based management application and can be used to accomplish signature-based analysis. The paper also describes two experiments that have been carried out with the agent to assess its performance and to demonstrate its effectiveness in terms of false alarm generation rates.

Keywords: Network security management, intrusion detection, misuse detection, stateful inspection, intrusion detection SNMP agent, RMON2.

1. INTRODUCTION

Due mainly to the weaknesses of firewall technologies in blocking some malicious incoming network traffic and to the growing number of attacks being initiated from hosts located in the same network as the victim host, Intrusion Detection Systems (IDSs) have been incorporated into the organizations security infrastructure. In order to satisfy the demand for such systems, several IDSs as Snort [1], NFR [2], Bro [3] and Stat [4] (to mention just a few) have been released to the market in the recent years. Despite the intensive research on intrusion detection, (a) reducing the false alarm (false positive) rates generated by IDSs, (b) providing the network manager with a high-level notation for attack signature specification, and (c) integrating security mechanisms

with the already existing network management infrastructure are some of the current challenges in the field.

Intrusion Detection Systems employ either *anomaly* or *signature analysis* (*misuse*) to detect attacks. Anomaly-based analysis uses statistical methods to distinguish normal from unexpected behavior, while signature-based analysis tries to match the content of data sources (e.g. network packets and system logs) with the specification of known attacks (attack signatures). Regardless of the technique used, around 90% of the alarms generated by an IDS are false positives [5].

In IDSs that employ anomaly analysis, it is difficult to determine what should be considered normal and abnormal. While attacks such as distributed denial of service, which generate considerable network traffic changes, may be detected at close to zero percent false positive rates (good examples may be found in [6, 7]), the same does not happen to many attacks that do not produce such substantial changes in the network traffic. In the case of these more subtle attacks, depending on the thresholds defined, either they may not be detected or many false alarms may be generated.

With IDSs that use signature analysis, the problem of high false alarm rates also occurs, mainly due to the limited expressive capability of the languages available to model attack signatures and to incomplete representation of signatures. In general, the signature concern is to observe packet fields and not protocol interactions (stateless inspection). An example of the effect of this limitation is the behavior of some IDSs when configured to detect the TCP SYN/TCP RST port scanning technique. The signature used by them consists of the observation of TCP packets with the RST flag on. The problem of this approach is that a TCP RST packet is not generated only by a station that does not have a certain kind of service available (when it receives a connection request). A station also uses this type of packet in order to restart an ongoing connection. As those IDSs are not able to correlate packets or the signature is not precise enough they cannot distinguish between TCP RSTs that represent port scanning from those used during a conventional connection, triggering alarms in both cases. The problem of false alarm generation is also related to the fact that we are not always able to capture the essence of the potential threat. The techniques used by the intruders and the threats posed by them to the system evolve over time and become more sophisticated, while the signatures lag behind. In some cases, there is also a motivation to specify a signature that will generate a large false alarm rate because the intent is to capture and analyse other but similar hypothetical scenarios.

With reference to the integration of security mechanisms and current Network Management Systems (NMSs), there is still a wide gap between security and network management, despite some initiatives (as the ones proposed in [6, 7, 8]). There is no Management Information Base (MIB) related to intrusion detection available. Several Network Management Systems offer an interface to configure Intrusion Detection Systems and are able to receive events generated by them. However, as stated by Qin et al., these systems lack efficient and effective capabilities of analyzing and managing the alarms sent by the IDS.

This paper addresses the false alarm rate, the lack of a high-level notation for attack signature specification and the lack of integration of IDSs and NMSs problems, by presenting an SNMP agent for intrusion detection that makes stateful inspection of data (packets) collected directly from the network. By using a state machine-based language called PTSL (Protocol Trace Specification Language) [9], the network manager can describe attack signatures that should be monitored. The signatures to be

used by the agent are configured by the network manager through the IETF Script MIB [10]. Once programmed, the agent starts monitoring the occurrence of the signatures on the network traffic and stores statistics, according to their occurrence, in an extended RMON2 MIB [11]. These statistics may be retrieved from any ordinary SNMP-based management application and can be used to accomplish signature-based analysis.

The main contribution of our work is the development of an agent that is able to do stateful inspection. As it will be shown along the paper, the agent provides accurate detection of both brute force and subtle attacks (concerning network traffic pattern changes). We have also developed a high-level and easy-to-learn language to specify attack signatures. The agent is fully integrated to the SNMP architecture. The configuration of the agent and the retrieval of results may be done using the SNMPv3 protocol.

The paper is organized as follows. Section 2 presents some related work. Section 3 presents PTSL language and some attack signatures described using this language. Section 4 approaches the internal architecture of the agent. In section 5 experiments that have been carried out with the agent are described. Section 6 closes the paper by presenting some final remarks and future work perspectives.

2. RELATED WORK

The problem of false alarms originates from the lack of accuracy in the process of detecting intrusions. As mentioned in the introduction, in the case of signature-based IDSs this imprecision is closely related (a) to the capabilities of the attack signature specification language provided and the respective intrusion detection engine or (b) to inprecise signature representation. Snort [1], for instance, uses a pattern matching model for detection of network attack signatures using identifiers such as TCP fields, IP addresses, TCP/UDP port numbers, ICMP type/code, and strings contained in the packet payload. Filtering rules are applied to each packet and stateful analysis is only partially provided (limited to TCP stream reassembly and inspection, and detection of some portscan e fingerprinting attacks), leading to a high number of false alarms. NFR [2] and Bro [3] suffer from the same problem. Statl [4], on the other hand, is an extensible state/transition-based attack description language used by Stat intrusion detection suite. This language allows one to describe computer penetrations as sequences of actions that an attacker performs to compromise a computer system. The detailed description of the signatures specified in Statl results in a lower number of false alarms (if compared to Snort, NFR and Bro).

Julisch et al propose in [12] some techniques to process alarm logs and filter false positives. This approach is based on the identification of alarm patterns, on the understanding of their root cause and, if non-malicious, on the usage of these alarm patterns for filtering. Finding filtering rules and the risk of filtering out true positives are some of the difficulties to implementing this approach.

None of the Intrusion Detection Systems listed so far offer mechanisms to make their integration with Network Management Systems easier. In the recent years, however, some efforts have been made to bridge this gap [6, 7, 8]. Qin et al. have proposed in [6, 7] the use of MIB II variables for network intrusion detection. This detection technique is clearly anomaly-based and, therefore, tend to be more efficient to detect attacks that generate considerable changes in the network traffic. In [8], Qin et. al

extend their previous work by proposing (a) an approach to integrate NMSs and IDSs and (b) a hierarchical correlation architecture for improving the detection accuracy and identifying coordinated intrusions.

Our work should be regarded as a complement to the efforts just mentioned. By proposing an SNMP agent for stateful intrusion inspection we provide an alternate approach, based on signature analysis, that is able to cope with both traffic-based intrusions (e.g DDoS) and slower traffic and stealth attacks, generating few false alarms. In the next section we introduce PTSL (Protocol Trace Specification Language) that is the language to be used by the network manager to specify attack signatures.

3. REPRESENTATION OF ATTACK SIGNATURES USING PTSL

PTSL (Protocol Trace Specification Language) is a language developed to allow the representation of protocol traces based on the concept of finite state machines (FSM). It is part of Trace, an architecture that supports high-layer protocol, service and application management through passive observation of protocol interactions (traces) in the network traffic. A full description of the language can be found in [9]. In this paper, we focus on how PTSL can be used to describe stateful network-based attack signatures.

The language is composed of graphical (Graphical PTSL) and textual (Textual PTSL) notations. These notations are not equivalent. The textual notation allows the complete representation of a trace (attack signature), including the specification of the FSM and the events that trigger transitions. In turn, the graphical notation covers only a subset of the textual notation, offering the possibility of graphically representing the FSM and only labelling the events that trigger transitions.

3.1 Graphical PTSL Notation

Several attack signatures have been modelled using PTSL. Figures 1 and 2 illustrate some of these specifications, described using the graphical notation of the language. Figure 1 shows a signature that can be used to detect the TCP SYN/TCP RST port scanning technique.

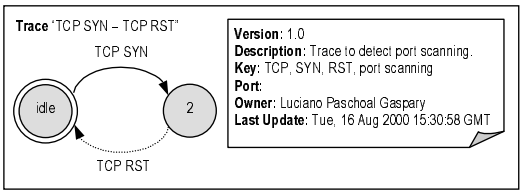

Figure 1. Signature to detect TCP SYN/TCP RST port scanning

Figure 2 shows other examples. In (a) one can see the signature to detect the `rpcinfo` command (available in Unix environments). This command returns a list of server processes that accept RPCs (Remote Procedure Calls), which is a useful information for the intruder. Similarly, in (b) it is shown the signature to detect the `showmount` command. Although (a) and (b) may appear in legitimate traffic, the oc-

currence of these signatures during unusual time periods or with high frequency can be regarded as attack evidence (e.g. someone scanning stations running `portmapper`). The signature described in (c) is composed of an HTTP request where the attacker uses the string /scripts/..\%C0\%AF../winnt/system32/cmd.exe?/c+dir+c:\ as argument. An URL like this indicates that he intends to execute some script or CGI at the HTTP server to obtain a list of the files located in the server. The signature to detect the SYN flood attack is depicted in (d). This attack consists of sending a huge number of connection setup packets (TCP packet with the SYN flag on with a fake source address) to a target host. This fake address must be unreachable or non-existent (usually a reserved value). When the target host receives these SYN packets, it creates a new entry on its connection table and sends a SYN/ACK packet back to the possible client. After sending the reply packet, the target host waits for acknowledgement from the client to establish the connection. As the source address is fake, the server will wait a long time for this reply. In a given time, the connection queue of the server will be full and all new connection requests will be discarded, creating a denial of service. Unlike other examples presented, this attack is identified by observing unsuccessful occurrences of the trace.

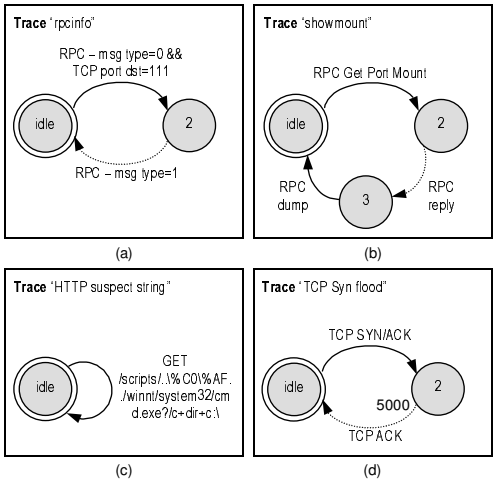

Figure 2. Graphical representation of attack signatures

Representation of states and transitions. As for PTSL graphical notation, one can observe from the previous examples that states are represented by circles. From the initial state (`idle`) other n states can be created, but they must always be reachable. The final state is identified by two concentric circles. In all examples presented, the initial and final states are the same. State transitions are represented by unidirectional arrows. The continuous arrow indicates that the transition is triggered by the client host, while the dotted arrow determines that the transition is triggered by an event coming from the server host. The text associated with a transition is merely a label to the event that triggers it; the full specification can only be made via textual notation.

Representation of timeouts. Transitions, by default, do not have a time limit to be triggered. To associate a timeout with a transition, an explicit value (in milliseconds) must be set. In the example shown in figure 2d, the value `5000` associated to transition `TCP ACK` indicates that the transition from state `2` to the initial state has up to five seconds to be triggered.

Representation of information for cataloging and version control. The graphical notation also offers a constructor where information about the signature, which are relevant to cataloguing and version control of specifications, are included. The data stored for a signature are: `Version`, `Description`, `Key`, `Owner` and `Last Update`. Besides these data, there is also a `Port` field, used to indicate the TCP or UDP port of the monitored protocol.

3.2 Textual PTSL Notation

Figure 3 presents the textual specification of the signature previously shown in figure 1. All specifications written in Textual PTSL start with the `Trace` keyword and end with the `EndTrace` keyword (lines 1 and 36). Catalog and version control information come right after the `Trace` keyword (lines 2–7). Forthwith, the specification is split into three sections: `MessagesSection` (lines 9–21), `GroupsSection` (not used in this example and not detailed in the paper) and `StatesSection` (lines 23–34). In `MessagesSection` and `GroupsSection` the events that trigger transitions are defined. The FSM that specifies the trace is defined in `StatesSection`.

Representation of messages. Whenever the fields of a captured packet match the ones specified at a `Message` for the current state, a transition is triggered in the FSM. The way those fields are specified depends on the type of protocol to be monitored. In the case of binary protocols (e.g. IP, TCP and UDP), known by their fixed length fields, the identification of a field is determined by a bit offset starting from the beginning of the protocol header; it is also needed to specify the size of the field, in bits (this is the `BitCounter` strategy). On the other hand, in the case of variable-length character-based protocols, where fields are usually split by white space characters (e.g. HTTP), the identification of a field is made by its position inside the message (`FieldCounter` strategy). In GET /scripts/..\%C0\%AF../winnt/system32/cmd.exe?/c+dir+c:\, for instance, GET is at position 0 and /scripts/..\%C0\%AF../winnt/system32/cmd.exe?/c+dir+c:\ is at position 1.

The signature shown in figure 1 is for a binary protocol. The `TCP SYN` message specification is shown in figure 3 (lines 11–14). In line 12 the message is defined as being of type `client`, meaning that the state transition associated with the message will be triggered by the client host. In line 13 the only field that is supposed to be analyzed is specified. All information necessary to identify it are: fetch strategy (`BitCounter`), protocol encapsulation (`Ethernet/IP`), field position (`110`), field length (`1`), expected value (`1`), comparison operator (`=`), and, optionally, a field description. The reply message `TCP RST` is shown in lines 16–19. The message type is defined in line 17 as `server`, i.e., the state transition will be triggered by the server host. Finally, the field to be analyzed is defined in line 18. Since the messages of the signatures illustrated in figure 2a, b and d are composed of binary protocol fields, they should be specified in a similar way.

```
1  Trace "TCP SYN - TCP RST"
2  Version: 1.0
3  Description: Trace to detect port scanning.
4  Key: TCP, SYN, RST, port scanning
5  Port:
6  Owner: Luciano Paschoal Gaspary
7  Last Update: Tue, 16 Aug 2000 15:30:58 GMT
8
9  MessagesSection
10
11 Message "TCP SYN"
12 MessageType: client
13 BitCounter Ethernet/IP 110 1 1 = "Field SYN  - 1 means TCP Connect"
14 EndMessage
15
16 Message "TCP RST"
17 MessageType: server
18 BitCounter Ethernet/IP 109 1 1 = "Field RST"
19 EndMessage
20
21 EndMessagesSection
22
23 StatesSection
24 FinalState idle
25
26 State idle
27 "TCP SYN" GotoState 2
28 EndState
29
30 State 2
31 "TCP RST" GotoState idle
32 EndState
33
34 EndStatesSection
35
36 EndTrace
```

Figure 3. Representation of a signature using Textual PTSL

As opposed to the example mentioned above, the signature specified in figure 2c is for a character-based protocol (HTTP). The attack is composed of a single transition and is recognized whenever an HTTP GET request packet with the argument /scripts/..\%C0\%AF../winnt/ system32/cmd.exe?/c+dir+c:\ is observed. Figure 4 presents part (the `MessagesSection`) of the textual specification for the trace shown in figure 2c. Lines 3–8 describe the HTTP request. In line 5 the `GET` field is defined. The information needed to identify a character-based protocol field are: fetch strategy (`FieldCounter`), protocol encapsulation (`Ethernet/IP/TCP`), field position (`0`), expected value (`GET`), comparison operator (`=`), and, optionally, a field description.

```
1  MessagesSection
2
3  Message "GET /scripts/..\%C0\%AF../winnt/system32/cmd.exe?/c+dir+c:\"
4  MessageType: client
5  FieldCounter Ethernet/IP/TCP 0 GET =
6  FieldCounter Ethernet/IP/TCP 1 /scripts/..\%C0\%AF../winnt/system32/cmd.exe?
7  /c+dir+c:\ =
8  EndMessage
9
10 EndMessagesSection
```

Figure 4. Field identification in character-based protocols

Representation of the FSM. Lines 23–34 in figure 3 define the textual specification of the state machine shown in figure 1. The final state is identified just after `StatesSection` (line 24). The states `idle` and 2 are defined in lines 26–28 and 30–32, respectively. The state specification only lists the events (messages and groups) that may trigger transitions, indicating, for each of them, which is the next state (lines 27 and 31).

4. THE INTRUSION DETECTION SNMP AGENT

The intrusion detection agent requires as input attack signatures specified in PTSL. The configuration of which signatures should be monitored at a given moment is made by the network manager through the Script MIB. Once programmed, the agent starts monitoring the occurrence of the signatures on the network traffic and stores statistics, according to their occurrence, in an extended RMON2 MIB. These statistics may be retrieved from any SNMP-based management application by periodically polling the agent. In order to reduce management traffic, it is possible to use the `alarm` and `event` RMON MIB groups instead. In this case, the network manager must configure thresholds to certain RMON2 MIB variables and define notifications that will be sent to the management station when these thresholds are reached. Figure 5a and b illustrates the communication flow between manager and agent considering both approaches. Expression [13] and Event [14] MIBs could also be used. The former provides the manager with a flexible mechanism to define thresholds (based on expressions), while the latter extends the capabilities of the RMON `alarm` and `event` groups by allowing alarms to be generated for MIB objects that are on another network element.

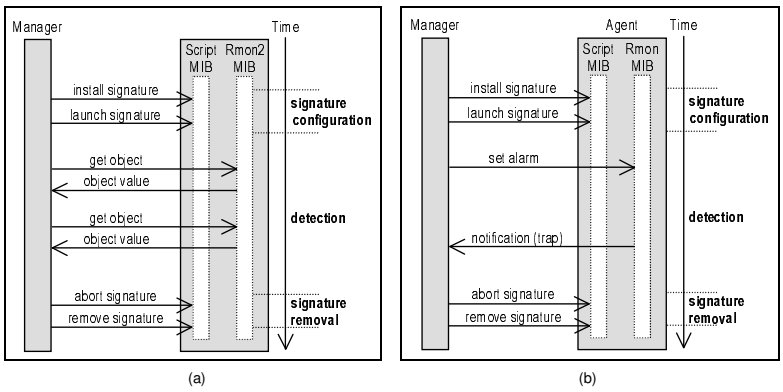

Figure 5. Communication flow between manager and agent

4.1 Architecture

The intrusion detection SNMP agent runs on Linux stations and was implemented using the C language, the POSIX thread library, the NET-SNMP framework [15] and Jasmin [16]. Figure 6 shows the agent architecture. The PTSL manager thread is responsible for the integration between the Script MIB and the PTSL core. It updates both the data structures used by the PTSL core and the RMON2 `protocolDir` table whenever a new signature is configured to be monitored or an existing signature is

requested to be removed from the agent. Three more threads – queue, PTSL engine and RMON2 – operate in a producer/consumer fashion. The first thread captures all the packets arriving at the network interface card using the libpcap library and inserts them in a circular queue. The second thread processes every packet in the queue, without removing them from it, to identify if they have the characteristics expected to allow one or more signatures to evolve in the state machine. If so, special marks are attached to the packets. Finally, the RMON2 thread removes every packet from the queue and, according to the markings, updates the RMON2 tables in the mySQL database.

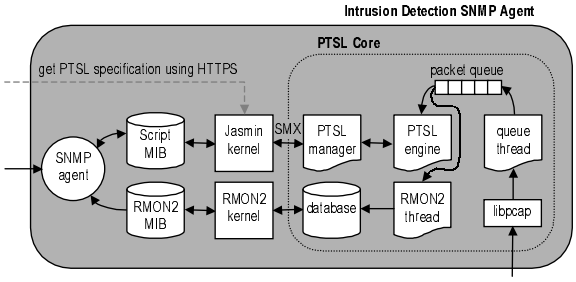

Figure 6. Internal organization of the agent

4.2 The Management Information Base

Every time that a signature is observed between any pair of hosts, data is stored in the mySQL database. This database is source of information for the SNMP sub-agent that implements an extended version of the RMON2 MIB [11]. One of the differences between our MIB and RMON2 is that the `protocolDir` group, which indicates the protocol encapsulations that the agent is able to monitor, now allows protocol traces (attack signatures) to be indexed. Therefore, monitoring granularity is considerably increased. Besides gathering statistics about the traffic generated by hosts using certain protocols, the agent also stores information related to the occurrence of attack signatures. Table 1 shows a set of entries that could appear in an agent protocol directory.

Table 1. RMON2 protocolDir table

ID	*LocalIndex*	*Description*
00-00-00-01-00-00-08-00	1	ether2.ip
00-00-00-01-00-00-08-00-00-00-00-17	2	ether2.ip.tcp
00-00-00-01-00-00-08-00-00-00-00-17-00-00-00-50	3	ether2.ip.tcp.http
00-00-00-01-00-00-08-00-00-00-00-17-00-01-00-04	4	ether2.ip.tcp.rpcinfo
00-00-00-01-00-00-08-00-00-00-00-17-00-01-00-05	5	ether2.ip.tcp.showmount
00-00-00-01-00-00-08-00-00-00-00-17-00-01-00-06	6	ether2.ip.tcp.http suspect string

As it was mentioned in the previous sub-section, the `protocolDir` table is automatically updated when a new signature is configured to be monitored or an existing signature is requested to be removed from the agent. The ID (`protocolDirID`) is

composed of $n \times 4$ bytes, where n is the number of protocols that comprise the encapsulation [17]. The number used to identify ethernet (00-00-00-01), IP (00-00-08-00) and high-layer protocols is never longer than 16 bits (2 bytes). Hence, to avoid conflicts with the identification of existing protocols, IDs above 65.535 are assigned to attack signatures (see table 1 above).

We have implemented most of the RMON2 groups, including `nlHost`, `alHost`, `nlMatrix`, and `alMatrix`. The later stores statistical data about the signature when it is observed between each pair of hosts at the granularity of attack signatures. Table 2 illustrates the contents of the `alMatrixSD` table. The semantic of the MIB has not been changed, since it still stores packet and octet rates. To determine the number of occurrences of a signature between two hosts it is necessary to divide the number of packets (stored in the MIB) by the number of transitions that form the signature. From the third line of the table, for example, one can infer that the signature TCP SYN – TCP RST has been observed 127 times (254 packets divided by 2 transitions).

Table 2. Information from the alMatrixSD table

Source Address	*Destination Address*	*Protocol*	*Packets*	*Octets*
172.16.108.1	172.16.108.2	ether2.ip.tcp.http suspect string	4	4.350
172.16.108.32	172.16.108.2	ether2.ip.tcp.rpcinfo	8	7.300
172.16.108.1	172.16.108.254	ether2.ip.tcp syn-tcp rst	254	1.202.126
125.120.10.100	172.16.108.254	ether2.ip.tcp.showmount	20	3.204

4.3 Signature-based Intrusion Detection

The agent can be used to accomplish signature-based intrusion detection. Figures 1 and 2c and d show examples of signatures that, regardless of when they are observed, indicate the occurrence of a scanning (figure 1) or attack (figure 2c and d). To accurately detect them it is necessary to define how many occurrences of the signature should be observed within a time interval in order to be considered an attack. Figure 7 presents a sample Tcl script that could be used to install and monitor the occurrence of these signatures.

The programming of the Script MIB on the agent is made with the aid of a specially developed package (line 2). In lines 10–13 the PTSL signature is installed. In line 14 the agent is asked to start observing the network for the occurrence of the signature just installed. Then, the script polls the agent every 120 second (line 26) to get information (line 18) and checks whether the signature has been counted or not (line 20). If the signature has been observed three times within an interval an alarm is generated.

The examples presented in figure 2a and b, on the other hand, can be part of legitimate traffic. Therefore, the identification of these network patterns as part of an attack is not straightforward. In this case, the network manager must have a precise characterization of the network (baseline) to be able to create rules to efficiently detect when such traffic can be regarded an attack evidence.

5. EVALUATION OF THE AGENT

This section describes two experiments that were accomplished to assess the performance of the agent and its behavior regarding false positive generation rates.

```
package require Tnm 3.0
package require ScriptMib 1.0

set oid "protocolDist.protocolDistStatsEntry.protocolDistStatsPkts.1.10"
set prev 0

if { [catch {::Tnm::snmp generator -address $agent} s] } {
   ::Tnm::log exit -code runtimeError "Error creating SNMP session: $s"
   }
if { [catch {ScriptMib::InstallScript $ma $m_owner $m_name $m_lang $m_src \
   $m_descr $m_args $m_ltime $m_etime $m_mrun $m_mcomp} e]} {
   ::Tnm::log exit -code runtimeError "Error installing script: $e"
   }
::ScriptMib::RunScript $ma $m_owner $m_name 0

proc monitor {} {
   global s oid prev
   set val [$s get $oid]
   set val [lindex [lindex $val 0] 2];
   if {[expr $val - $prev] > 3} {
      ::Tnm::log "Intrusion alarm: $m_name, $m_descr"
   }
   set prev $val
}
::Tnm::job create \
   -interval 120000 -error {::Tnm::log exit -code runtimeError $errorInfo} \
   -exit {::Tnm::log exit} -command {monitor}
vwait forever
```

Figure 7. Sample script to install and monitor the occurrence of an attack

5.1 Performance Analysis

In order to identify the network load supported by the agent (without dropping packets) some experiments have been carried out. The test environment was composed of three hosts connected through a 10 Mbps IEEE 802.3 network segment. The first host was used to generate network traffic, while the second one was supposed to receive it. The third host, a 450 MHz K6II PC with 64 MB RAM, was used to run the agent. The results obtained were the following:

- The sustained agent capacity (without packet loss) is around 235 datagrams per second when one signature is monitored. This rate was obtained by the consecutive generation of UDP datagram sequences that matched exactly the signature configured;
- The increase in the number of signatures monitored causes performance degradation of the agent. The agent capacity was reduced to 172 datagrams/second when it was configured to monitor five signatures simultaneously and traffic that matched exactly these signatures was generated;
- When generating traffic at 10 Mbps (around 5000 datagrams/second), with all datagrams being part of the signature, the agent discards packets.

We have also run the agent on a small production network, characterized as follows: (a) IEEE 802.3 network running at 10 Mbps, (b) 10 hosts (connected to a hub) running Windows operating system and configured to share files and printers, (c) average traffic rate of 150 packets/second during prime hours, (d) 75% of the packets was between 65 and 256 bytes long and 21% of the packets was longer than 1024 bytes, and (e) application protocols composed of HTTP (45%), NetBIOS (27%) and SMTP (15%). In this scenario, packet discards have not been observed.

5.2 Alarm Generation Analysis

To demonstrate that our approach tend to generate few false positives we have carried out an experiment based on previous work done at Lincoln Laboratory [18]. The main idea is to create background traffic that is similar to the traffic observed on the production network. In the next step packets corresponding to attacks are merged to the background traffic. By reproducing the resulting traffic, one or more IDSs can be evaluated regarding false positive generation rates. In this experiment we have assessed our agent.

We have used the packet trace file available at Lincoln Laboratory named DDos 1.0. It is a distributed denial of service attack that explores a buffer overflow technique in a `sadmind` server running on Solaris operating system. The attack has five phases:

- *Phase 1 (host scanning)*: the purpose of this phase is to identify which hosts are up and running by sending ICMP echo request packets to all hosts of the network. This phase was cut from the original packet trace and not considered in the evaluation, because ICMP echos and replies appear a lot in legitimate traffic (would generate many false positives);
- *Phase 2 (look for sadmind running on a target)*: in this phase the `portmapper` of the hosts (that replied the ICMP echo request packets in the previous phase) are queried in order to identify which port the `sadmind` daemon is listening. Next, a TCP connection to this port is opened;
- *Phase 3 (try to compromise the target)*: once the TCP connection is established, a buffer overflow technique is used to edit the password file and create a new user account;
- *Phase 4 (identify which target has been compromised)*: in this phase one must open a telnet connection using the user account created in the previous step;
- *Phase 5 (launch a distributed denial of service attack)*: this phase was also cut from the original packet trace and not considered in our evaluation. We have focused on the detection of the earlier stages of the attack, because we believe that detecting the distributed denial of service when it is occurring is not very useful, since very little can be done against it.

The packet trace just described was merged to background traffic, collected from an IEEE 802.3 network segment composed of 10 personal computers. The traffic is characterized as follows: the application protocols used were NetBios (48%), HTTP (22%), mail (11%), FTP (8%), SSH (3%) and other (8%). The packet size distribution was: 38% (<64 bytes), 51% ($\geq$ 65 and <128 bytes), 1% ($\geq$ 128 and <256 bytes), 1% ($\geq$ 256 and <512 bytes), 8% ($\geq$ 512 and <1024 bytes) and 1% ($\geq$ 1024 bytes). It is important to highlight that the background traffic has some legitimate `sadmind` traffic.

The test platform consisted of three hosts connected to a hub. The first host was used as the traffic generator. The second host ran the agent prototype. The agent was configured to monitor several attack signatures (including the ones presented in the paper and the signatures to detect phases 2, 3 and 4 of the attack). The third host executed a MIB browser, which was configured to poll the agent once a second (to get the value associated to the `protocolDistStatsPkts` variable).

We have then used `tcpreplay` [19] to reproduce the traffic. As we did not want the agent to discard packets (to be able to focus the evaluation on the accuracy of the detection process), we have replayed the traffic at low speed.

Our agent has triggered three alarms related, respectively, to phases 2, 3 and 4 of the attack. No false positives have been generated. We believe this has occurred due to the mechanism adopted by the agent, which is able to analyze packet sequences (stateful analysis). Packet correlation (intrinsic characteristic of the agent) helps on distinguishing legitim and attack traffic.

6. CONCLUSIONS AND FUTURE WORK

This paper presented an SNMP agent for stateful intrusion inspection. We have also presented PTSL, a language for the representation of protocol traces that, in this paper, was used to model attack signatures. Then we have presented some experiment results related to the agent performance and alarm generation rates. Providing the network manager with a high-level notation for attack signature specification, reducing the false positive rates generated, and integrating security mechanisms with the already existing network management infrastructure were the general objectives of the work. It is important to highlight that the absence of false alarms in our experiment is due to the programming of the signatures, and not due to the network management portion of our agent.

PTSL language has shown to be very adequate for the specification of attack signatures because of its simplicity. The expression power of PTSL is another point to be highlighted. The possibility of correlating packets, whether from the same flow or not, enables the identification of attacks with a low error rate, considerably reducing the number of false alarms. Besides, while most IDSs allow the selection of packets based on a few predetermined header fields only up to the transport layer, PTSL goes beyond, allowing the use of filters based on any protocol, all the way up to the application layer.

The use of an extended RMON2 MIB to store information related to the occurrence of attack signatures is a significant step towards integration of security mechanisms with the current existing SNMP-based management applications and platforms. Our agent should not be used isolated from other approaches. While the work published by Qin et al. in [6, 7, 8] tend to be more efficient to detect attacks that generate considerable changes in the network traffic (anomaly-based detection), our agent is able to cope with both traffic-based intrusions and slower traffic and stealth attacks (since it is signature-based).

Regarding security, the agent supports all facilities provided by SNMPv3, including the User-based Security Model (USM) [20] and the View-based Access Control Model (VACM) [21]. Using these facilities it is assured that the agent cannot be "reprogrammed" by a person who is not allowed to do this.

According to the results presented in sub-section 5.1 one can observe that the passive network traffic monitoring technique is computationally onerous. To achieve better results and not to compromise the intrusion detection process we have considered the following alternatives: use of hosts with more than one processor, distribution of signatures to more than one host, filtering out packets that are not useful for the signatures being monitored (using BPF filters), and replacement of the mySQL database by a more efficient alternative.

References

[1] Snort The Open Source Network Intrusion Detection System. `http://www.snort.org/`.

[2] NFR Security. `http://www.nfr.net/`.

[3] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24), Dec. 1999, p. 2435–2463.

[4] G. Vigna, S. T. Eckmann, and R. A. Kemmerer. The STAT Tool Suite. In *Proceedings of DARPA Information Survivability Conference & Exposition (DISCEX 2000)*, 2000.

[5] D. Alessandri. Using Rule-Based Activity Descriptions to Evaluate Intrusion-Detection Systems. In *Proceedings of International Workshop on the Recent Advances on Intrusion Detection (RAID 2000)*, 2000.

[6] J. B. D. Cabrera, L. Lewis, X. Qin, W. Lee, R. K. Prasanth, B. Ravichandran, and R. K. Mehra. Proactive Detection of Distributed Denial of Service Attacks using MIB Traffic Variables – a Feasibility Study. In *Proceedings of IFIP/IEEE International Symposium on Integrated Management (IM 2001)*, 2001.

[7] X. Qin, W. Lee, L. Lewis, and J. B. D. Cabrera. Using MIB II Variables for Network Intrusion Detection. *Data Mining for Security Applications, Advances in Computer Security*. Kluwer Academic Press, March 2002.

[8] X. Qin, W. Lee, L. Lewis, and J. B. D. Cabrera. Integrating Intrusion Detection and Network Management. In *Proceedings of IFIP/IEEE Network Operations and Management Symposium (NOMS 2002)*, 2002, p. 329–344.

[9] L. P. Gaspary, L. F. Balbinot, and L. R. Tarouco. Monitoring High-Layer Protocol Behavior Using the Trace Architecture. In *Proceedings of Latin American Network Operation and Management Symposium (LANOMS 2001)*, 2001, p. 99–110.

[10] D. Levi and J. Schönwälder. Definitions of Managed Objects for the Delegation of Management Scripts. *RFC 3165*, Aug. 2001.

[11] S. Waldbusser. Remote Network Monitoring Management Information Base Version 2 using SMIv2. *RFC 2021*, Jan. 1997.

[12] K. Julisch. Dealing with False Positives in Intrusion Detection. In *Proceedings of International Workshop on the Recent Advances on Intrusion Detection (RAID 2000)*, 2000.

[13] R. Kavasseri and B. Stewart. Distributed Management Expression MIB. *RFC 2982*, Oct. 2000.

[14] R. Kavasseri and B. Stewart. Event MIB. *RFC 2981*, Oct. 2000.

[15] NET-SNMP. `http://net-snmp.sourceforge.net/`.

[16] Jasmin - A Script MIB Implementation. `http://www.ibr.cu.tu-bs.de/projects/jasmin`.

[17] A. Bierman, C. Bucci, and R. Iddon. Remote Network Monitoring MIB Protocol Identifier Reference. *RFC 2895*, Aug. 2000.

[18] R. Lippmann et al. Evaluating Intrusion Detection Systems: the 1998 DARPA Off-line Intrusion Detection Evaluation. In *Proceedings of DARPA Information Survivability Conference & Exposition (DISCEX 2000)*, 2000.

[19] Tcpreplay. `http://sourceforge.net/projects/tcpreplay/`.

[20] U. Blumenthal and B. Wijnen. User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3). *RFC 2574*, Apr. 1999.

[21] B. Wijnen, R. Presuhn, and K. McCloghrie. View-based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP). *RFC 2575*, Apr. 1999.