25

ADELE: AN ATTACK DESCRIPTION LANGUAGE FOR KNOWLEDGE-BASED INTRUSION DETECTION

CÉDRIC MICHEL, LUDOVIC MÉ
Supelec, BP 28, 35511 Cesson Sévigné Cedex - France
{cmichel,lme}@supelec-rennes.fr

Keywords: Intrusion detection, attack description language

Abstract ADeLe is an attack description language designed to model a database of known attack scenarios. As the descriptions might contain executable attack code, it allows one to test the efficiency of given Intrusion Detection Systems (IDS). Signatures can also be extracted from the descriptions to configure a particular IDS.

1. INTRODUCTION

In this article, we introduce an attack description language, ADeLe, designed for knowledge-based intrusion detection (also called misuse detection).

The primary goal of ADeLe is to combine all the knowledge available for a given attack in one and only one readable and high-level description.

ADeLe is then to be used to model known attack scenarios in order to build an attack database. This database should then be used itself: to configure the probes and detection engines of a given intrusion detection system (IDS) or to test the detection capabilities of a given IDS (by means of attack replay).

The ADeLe language has been developed simultaneously with the Lambda [2] language within the Mirador[1] project, which may account for significant overlaps between the two languages. Both languages allow the expression of the attack code as well as rules for detection and correlation. However, Lambda uses a declarative approach while ADeLe uses a more procedural approach.

[1]Project funded by the DGA/CELAR/CASSI which is part of the French Ministry of Defense.

This article is organized as follows: section 2 describes some related work and presents the main characteristics of an attack description in ADeLe. Section 3 details the language itself. Section 4 summarizes and outlines future work. Finally, an example of attack description in ADeLe is given in the appendix.

2. ADELE AND RELATED WORK

As attacks against computers and networks are becoming more frequent and more sophisticated, there is a need for representing and sharing information about these attacks [5]. A simple intrusion detection signature is far from being sufficient to describe an attack. We also need to know what the exploited vulnerability is, how the attack was performed, what are its consequences, and how to react (automatically or not) in order to stop it.

To describe an attack completely, i.e. to describe every aspect, we need to use several specific languages, with different purposes. References [17, 5] propose six different classes of such "attack languages": exploit, event, detection, correlation, reporting, and response. Exploit languages [14, 4, 2] are used to describe the stages to be followed to perform an intrusion. Event languages [15, 8, 1] describe the format of events used during the detection process. Detection languages [10, 12, 13, 5, 11, 2] allow the expression of the manifestation of an attack in terms of occurences of events. Correlation languages [2] permit analysis of alerts provided by several IDS in order to generate meta-alerts. Reporting languages [6, 3] describe the format of alerts produced by the IDS. Finally, response languages are used to express countermeasures to be taken after detection of an attack.

In table 1, ADeLe and the previously referenced attack languages are organized into these classes. The majority of these languages specifically address one aspect of the attack.

As we could not find any language allowing the description of all aspects of an attack, we decided to create ADeLe while Cuppens and Ortalo were creating Lambda [2]. So, ADeLe is directly concerned with four classes of attack languages: exploit, detection, correlation and response. A description in ADeLe thus contains three parts (as detection and correlation are merged). We propose neither a report language, as we made the choice to use IDMEF [3] in order to facilitate inter-operability between IDS, nor an event language, as the notation we chose to designate fields within events (cf 3.2.1) is an intermediary representation independant of the raw format.

ADeLe relies on a largely accepted intrusion detection framework, in which we assume that there are probes and detection engines. Probes are located in various places of the monitored environment delivering events. Detection engines from particular IDS (host-based or network-based) deliver alerts. We also assume that these alerts use the IDMEF format [3].

	BSM	Tcpdump	Bishop	CASL	NASL	CISL	IDMEF	Kumar
	[15]	[8]	[1]	[14]	[4]	[6]	[3]	[10]
Event languages	x	x	x					
Exploit languages				x	x			
Reporting languages						x	x	
Detection languages								x
Correlation languages								
Response languages								

	BRO	Snort	SNP-L	STATL	GASSATA	LAMBDA	ADeLe
	[12]	[13]	[16]	[5]	[11]	[2]	
Event languages							
Exploit languages						x	x
Reporting languages							
Detection languages	x	x	x	x	x	x	x
Correlation languages						x	x
Response languages							x

Table 1 Examples of specific attack languages (non exhaustive list)

ADeLe is designed to allow attack descriptions which are: readable (XML-like tags are used to encapsulate each part), comprehensive (it is possible to represent every aspect of an attack, i.e. from the attacker's and the defender's points of view, in only one description), generic (operators allow the exploit part and the detection part to be generic to reduce the number of attacks present in the database by having only one description for multiple variants of the same attack), and modular (defining an attack composed of several attacks already described is allowed and defining a meta-alert also).

ADeLe was proposed with the dual purpose of being able to configure and also to test IDS. Indeed, an ADeLe description is not directly operational. It contains information about attacks, but this information needs to be extracted and transformed through several compilers[2] (or interpreters) performing the configuration. Because an ADeLe description contains source code of the

[2]One compiler extracts everything defining what the interesting events are for a particular attack and transform it to configure the concerned probes. Another compiler extracts the signature of the attack scenario in terms of occurrence of events or alerts and thus allows configuration of the detection engine for a particular IDS.

attack, we are also given the opportunity to constitute a database of executable attacks. Given the appropriate interpreter (or compiler), it is possible to run a complete database of attacks against intrusion detection systems. It allows testing of the detection efficiency of those IDS in terms of false negative rate.

3. ATTACK DESCRIPTION IN ADELE

An attack description written in ADeLe is organized as presented in the appendix. References to line numbers refer to the source code of the example in the appendix.

An ADeLe description looks like a function in C with name and parameters (cf line 1). The name is the one that will be reported in the alert when detection has occurred. Possible typed parameters can follow, representing input and output variables. They are used to make the exploit part re-usable. If one describes in ADeLe a global attack scenario made of several smaller attacks, also described in ADeLe, he/she just has to reference the name of these attacks with the appropriate parameters. In most cases, input parameters refer to information needed to launch the attack, and output parameters refer to information or an access level gained by the attacker.

The body of the description is made up of three parts. We describe the `EXPLOIT` part in 3.1. Then we detail the `DETECTION` part in 3.2. Finally, we present the `RESPONSE` part in 3.3.

3.1. THE EXPLOIT PART

The first part of the attack description represents the attacker's point of view. It links together three aspects of the attack: the requirements for launching it, its code (or its stage description), and the results gained by the attacker. Thus, the exploit part is composed of three sub-parts: pre-condition, code, and post-condition.

The <PRECOND> section. In this section, we can express the requirements for launching the attack. Most of the time, it is knowledge about the vulnerabilities that must be present on the target. It can be specific to the target (operating system, version of installed software…). It can also be something more general, like the level of privilege (as defined in [9]) needed by the attacker to launch a successful attack against the target[3] (cf line 4).

The <ATTACK> section. This is the location of the source code of the attack. We allow this code to be expressed in any language. We use a specific tag

[3]However, the attacker can launch a blind attack without ensuring that he has the required privileges. In this case, the attack will probably fail.

(cf line 6) to identify the language used and to allow an appropriate interpreter to execute the code of the attack. It can be a general programming language such as "C," " C++," "Perl," or specific exploit languages such as "Casl"[14] or "Nasl"[4]. In the case where no source code is available, we can use a informal textual description of the attack ("Text").

We have begun to define a high-level scripting language adapted to exploit code writing: "EDL"[4]. It relies on classical scripting languages concepts: typed data, variables, boolean expressions, mathematic operators, a few keywords to ensure flow control, and libraries of functions. Those functions can be high-level functions (e.g., commands used during a FTP connection) as well as low-level functions (e.g., network packet generators).

To describe the actions defining an attack, we use these functions (cf line 13). The actions defining the attack are executed in the order where they appear, except if there is application of an action operator.

Actions operators. In order to represent the variants of the same attack, we introduce three operators: **Non_ordered, One_among,** and **Subset_of.** Use of these operators allows us to reduce the database size because we describe a single, but more generic, attack.

Actions (or groups of actions) can take place within the *Non_ordered* operator (cf lines 16-29): they all have to be executed in any order. If, for a stage of the attack, a choice can be made among actions (or groups of actions) equivalent in terms of consequences, the *One_among* operator can be used (cf lines 41-50). That represents the execution of only one of the actions of the set. To express the execution of a subset of actions of cardinality C among a set of N actions ($1 \leq C \leq N$), the *Subset_of* operator can be used. The execution order is unspecified.

For execution, various strategies will have to be considered for the choice of the variants of the attack (e.g., random choice of action, always the first possibility…).

Types and operators. We allow the declaration of variables (of type String, Integer, Boolean IPaddr…). We also define classical boolean operators (logical operators, comparisons…) and pattern matching in the character strings (keyword `IN`).

Conditional blocks. The actions defined in the attack scenario produce intermediate results which must first be tested to decide on the continuation of the attack. It is defined as follows:

`IF` (<boolean expression>){ <actions> } `ELSE` { <actions> }

[4]Exploit Description Language.

Iteration. To describe the repetition of actions, the "WHILE" iteration can be used: `WHILE` (<boolean expression>){ <actions> }

A link between attack and detection.
The role of the <DETECT> part is to give indications about the way the attack can be detected. As the events (or alerts) correspond to stages of the attack, it is useful to establish a link between the actions and their detection. This is why, at the time of the expression of the attack scenario, we associate events/alerts with the actions or groups of actions observable by an IDS. In concrete terms, we delimit the portion of the attack scenario which corresponds to a type of event/alert in order to be able to name it for later reference in the <DETECT> part (cf lines 12-14). If we are unable, for a given attack, to observe its intermediate stages, we will define only one event that includes all the stages of the attack. In certain cases, an action is observable (detectable) neither on the target network nor on the target system (e.g., the attacker is cracking passwords on his/her local machine). It will then be associated with no event.

The <POSTCOND> section. In this section, what has been obtained by the attacker is expressed. Most of the time, it is an increased level of privilege (cf line 62). It could also be disclosure of information, corruption of information, denial of service, or theft of resources [7]. Moreover, as proposed by a reviewer, these last goals could be refined in order to express more specific gains, which could be subgoals in chain of attacks. For example, we could add changes in configuration, impersonation, injection of false data, creation of a backdoor, etc.

3.2. THE DETECTION PART

In this part, we propose a new high-level language to express the detection of attacks. This language allow us to write basic signatures as well as complex scenarios involving combination of known attacks. It is made possible because events coming from the probes are handled in the same way as alerts from IDS.

There are three sections related to the detection. Section 3.2.1 details how the detection itself is expressed. Section 3.2.2 deals with the confirmation of the diagnosis of detection. And finally, Section 3.2.3 describes the emission of an alert using the IDMEF format.

3.2.1 The <DETECT> section. This section is itself divided into three subsections. The first one is used to name the events/alerts expected to appear during the attack and to define their general types. In the second one, we express the temporal correlation between those named events/alerts. In the last one, we refine the characteristics of each event/alert and also the contextual correlation between events/alerts belonging to the same attack.

The <EVENTS> subsection. A name is given to each event/alert that should be observable during the attack. At this stage of the description, we only define the general types of those events/alerts. If possible, the names should already be defined in the <ATTACK> section (cf 3.1).

We decided to use the IDMEF format [3] **notation** not only for alerts returned by IDS, but also for events coming from the various probes. The prefix for alerts and events is then one of the following: "`Alert.`," "`Network.`," "`System.`," or "`Appli.`" (cf lines 69-75).

However, as the IDMEF data model is designed only for alerts, we use a similar hierarchical notation to allow access to the fields of events from the probes, as explained in the <CONTEXT> subsection.

The <ENCHAIN> subsection. Once the events[5] are named, we express their temporal relationships (except when the detection of the attack involved a single event/alert). In this section, we define the **global scenario** of detection. It can be seen as a temporal automaton in which transitions are fired when expected events occur. The detection is achieved when the last event specified in the scenario has occurred.

To describe the global scenario of detection (in only one expression), we use any combination of the following operators (cf line 78):

- **sequence:** two events that should appear in sequence are separated by a semicolon (whatever may be the number of events of different types occurring in-between). For instance, the condition (`E0 ; E1`) is verified for the following events flow: (`E0 A B E1`).
- **unspecified order:** when several events (or groups of events) must all occur in the attack in an unspecified order, the operator `Non_ordered{<events>}` can be used. It defines a temporal interval for which the boundaries are the occurrences of the first and the last events.
- **exclusive choice:** to express that one among several events can be used for one stage of the attack, we use the operator `One_among{<events>}`. The condition is verified as soon as one (and only one) event of the set has occurred.
- **subset:** to express that one or more events from a set should occur (in no particular order), we use the operator `Subset_of{<events>}`.
- **repetition:** to express a series of *n* events of the same type, we use the notation: `E1^`*n* .
- **aliases:** in the <EVENTS> subsection, we manipulate only occurrences of general events. As soon as an event appears several times in the scenario,

[5] In this subsection, we use the term events to designate both events and alerts.

we cannot isolate a particular occurrence of this event. For instance, `E0 ; E1 ; E0` represents two different occurrences of `E0`. In order to isolate a particular occurrence, we allow the declaration and assignment of a variable. For example, to isolate the 7th occurrence of 10 similar events, we write: `(E1)^6 ; EVT1 :=E1; (E1) ^ 3`. This expression can be considered as an alias definition for a particular event.

- **non occurrence:** to allow an IDS to invalidate the hypothesis that the attack is in progress, we define an operator to express the fact that a particular event should not occur during a specified interval of events. Example: `{E0 ; E1 ; E2} WITHOUT F`. This example means that the event `F` should not occur between events `E0` and `E2`.
- **time constraints:** in addition to the global scenario, we can express time constraints[6] between events in order to invalidate the hypothesis that the attack is still in progress. If we define multiple constraints, they simultaneously apply to the detection scenario.
  Example: `(E2. Time. time - E1. Time. time) <= "2mn"`

The <CONTEXT> subsection.

For readability's sake, we chose to separate the temporal (<ENCHAIN>) and the contextual (<CONTEXT>) aspects of a scenario.

A context rule is a constraint applied to **occurrences** of one or more previously defined events/alerts (or aliases). All the constraints must be satisfied simultaneously [7].

Constraints can be expressed on any field to filter only events/alerts that should appear for a given attack. Some constraints apply on only one event; they are used for configuration of the concerned probes. Other constraints link up several events/alerts which belong to the same attack[8]; they are used to configure the detection engines for a given IDS.

When alerts are linked by constraints, it is called **alert correlation.**

Notation of fields within events.

As said before, we designate the fields of events using the IDMEF notation whenever it is possible. Each kind of event (prefixed with "`Network`," "`System`," or "`Appli`")[9] should have **at least** the following fields:

- **alertid**: serial number for the event
- **Time.***: time of the generation of the event

[6] Most of the time, it is an explicit timeout.
[7] Agreed that a constraint is only applicable when the concerned event/alert has occurred.
[8] For instance, all events refer to the same target.
[9] Fields within alerts are prefixed with "Alert". Their notation is fully defined in [3].

- **Analyzer.ident**: identifier for the probe which has produced the event
- **Classification[0].name**: name used to refer to that kind of event
- **Target[0].***: fields containing information about the target of the event
- **Source[0].***: fields containing information about the source of the event

However, because IDMEF was designed for alert purposes, we needed to add specific fields for each kind of information source. Those additional fields allow us to access low-level information contained in events.

The notation we use to access the fields of instantiated events is easily extensible if necessary. It is a high-level view of the events. It allows us to hide the details of the raw format.

For example, to express interesting network events, we propose the following hierarchical object notation for the fields (non exhaustive):

```
Network.Ip.Header.(version|tos|ttl|src|dst|…)
Network.Ip.Tcp.Header.(src|dst|seqnum|flags|…)
```

As system events are more platform-dependent, we define fields such as:

```
System.Bsm.(event|program|args|env|…)
System.Nt.(…)
```

Events from application logs are defined in the same way:

```
Appli.Apache.(…)
Appli.Wuftp.(…)
```

Following this notation for alerts and events, here are some examples of constraints:

- comparison (==,<,>,<=,>=,!=) between a numeric field and a constant (cf lines 81-82)
- one or more fields unified with a variable so that any new occurrence of this variable represents the same value (cf lines 83-89)
- numeric field value included in a list of values or intervals (keyword `IN`)
- locating substrings expressed by regular expressions in a given field (keyword `MATCHES`)
- alternatives in a constraint (logical OR)

3.2.2 The <CONFIRM> section. In this section, we express what to do to verify that the attack, if successful, has produced the expected consequences. This is a way to eliminate some false positives. To achieve this goal, we define a few boolean functions[10]:

[10] This list is not exhaustive.

`Wrong_Hashes (<file name>)`: evaluates to `true` when the file has been tampered with. It can be used for a Trojan Horse attack, in order to compare the digests of the binaries (which are suspected to be corrupted) with the last safe version.

`File_Contains (<file name>, <regular expression>)`: evaluates to `true` when the contents of the file match the given expression. It can be used to test the contents of a file (possibly modified).

`Unreachable_Machine(<IP address>)`: evaluates to `true` when the machine does not respond. It can be used when an attack is intended to crash the target machine.

Depending on the results of the active checking for a given attack, we can consider two cases:

- all the checks are positive, i.e. the attack was successful. An alert will be reported.
- some checks are negative, i.e. it was just an attempt and the attack failed. An alert with the appropriate `Alert.impact` field will be reported.

If, for a given attack, the proportion of failed (but detected) attacks over the successful ones becomes significant (and the target is found vulnerable to the attack), we would then suspect that the signature is leading to many false positives. The signature should then be reconsidered and improved.

3.2.3 The <REPORT> section. When the attack described in ADeLe is detected, an alert has to be sent. For that, the appropriate fields of an alert in the IDMEF format have to be filled. There are three kinds of values given to the fields of the alert: It can be:

- constant values (cf lines 103-104)
- values generated at runtime (cf lines 99,101)
- values coming from the events/alerts which led to the detection (cf lines 107,109)

3.3. THE RESPONSE PART

To our knowledge, there is no publicly known language designed to express automatic response in reaction to the detection of an attack. Automatic response can be a dangerous feature and sometimes the cure is worse than the disease. Indeed, as IP spoofing is common, the IP address of the supposed source of the attack may differ from the actual source of the attack. In this case, automatic reaction can lead to denial of services (DoS) because legitimate users can be denied access to data, services, or machines. However, we think that it would be useful to allow (even for a small number of attacks) expression of automatic response. A few precautions can be taken to limit the risk of DOS. For instance,

restrictions could apply only to external [11] IP addresses or to normal users, rather than privileged users. We propose a (non-exhaustive) list of functions to be used for automatic response.

A first method to prevent further attacks is to deny temporarily access to a resource:

`Close_Port(target_ip, "TCP" | "UDP" ,port_number)` closes a port used for an attack[12].

`Black_List (source_ip)` adds an IP address to a black list[13].

In order to stop an attack in progress, more protective measures can be taken:

`Reset_TCP_connection(source_ip,source_port,target_ip, target_port)` terminates an offending TCP communication.

`Kill_Process(target_ip,user_name,process_id|"ALL")` kills one process (or all processes) from a particular user.

`Shutdown_Machine(target_ip)` shutdowns a machine remotely to ensure data integrity.

A more general function executes any script in reaction to the attack: `Script_Exec(script_name)`.

Such functions can be used in an ADeLe description in the <RESPONSE> part. Their parameters will then be either constants or values extracted from events/alerts which led to the detection.

4. CONCLUSIONS & FUTURE WORK

We presented in this article an attack description language. It has been designed to be easily readable. It can express both the attacker's and the defender's points of view. As the description of the attack contains (possibly executable) exploit code, a database of attack descriptions written in ADeLe makes it possible to test the efficiency of some IDS. The signature of the attack can be extracted from the description to allow configuration of probes and detection engines from a given IDS. Automatic response after the detection of the attack can be included in the description.

A preliminary version of a compiler from ADeLe to G[A]SSA TA [11] exists and allows us to write signatures directly in ADeLe. In the near future, we plan to write compilers from ADeLe to other IDS.

[11] Outside the monitored network.
[12] Until the vulnerability has been corrected.
[13] Used by a firewall.

Acknowledgments

This work was funded by the DGA/CELAR/CASSI as part of the Mirador project. The authors would like to thank all the members of the project for many useful comments and ideas.

References

[1] M. Bishop. A standard audit trail format. Technical report, Department of Computer Science, University of California at Davis, 1995.

[2] F. Cuppens and R. Ortalo. Lambda: A language to model a database for detection of attacks. In *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID' 2000),* October 2000.

[3] D. Curry. Intrusion detection message exchange format, extensible markup language (xml) document type definition. `draft-ietf-idwg-idmef-xml-02.txt`, December 2000.

[4] R. Deraison. The nessus attack scripting language reference guide. http://www.nessus.org, September 1999.

[5] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. Statl: An attack language for state-based intrusion detection. In *Proceedings of the ACM Workshop on Intrusion Detection,* November 2000.

[6] R. Feiertag, C. Kahn, P. Porras, D. Schnackenberg, S. Staniford-Chen, and B. Tung. A common intrusion specification language (cisl). specification draft, http://www.gidos.org, June 1999.

[7] J. D. Howard and T. A. Longstaff. A common language for computer security incidents. Technical Report SAND98-8667, Sandia National Laboratories, October 1998.

[8] V. Jacobson, C. Leres, and S. McCanne. Tcpdump 3.5 documentation. http://www.tcpdump.org, 2000.

[9] K. Kendall. A database of computer attacks for the evaluation of intrusion detection systems. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[10] S. Kumar and E. H. Spafford. A software architecture to support misuse intrusion detection. Technical Report CSD-TR-95-009, The COAST Project Department of Computer Sciences, Purdue University, 1995.

[11] L. Mé. Gassata, a genetic algorithm as an alternative tool for security audit trails analysis. In *Proceedings of the first international workshop on the Recent Advances in Intrusion Detection (RAID'98),* 1998.

[12] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th Usenix Security Symposium,* January 1998.

[13] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the USENIX LISA '99 conference,* November 1999.

[14] Secure Networks. *Custom Attack Simulation Language (CASL),* January 1998.

[15] Sun Microsystems, Inc. Sunshield basic security module guide. Solaris Documentation.

[16] E. Turner and R. Zachary. Securenet pro software's snp-l scripting system. White paper, http://www.intrusion.com, July 2000.

[17] G. Vigna, S. T. Eckmann, and R. A. Kemmerer. Attack languages. In *Proceedings of the IEEE Information Survivability Workshop,* October 2000.

Appendix: An Example of Attack Description in ADeLe

The vulnerability exploited by the NFS_Mount attack is a bad configuration of the access rights of partitions remotely mounted via the NFS protocol. The attacker has initially a remote access level [9]. Thus the attacker can use several commands (`rpcinfo`, `showmount`, `finger`) to gather information about the potential target. He tries to find a home directory which is exported by the attacked system. If he finds such a home directory, he creates (on its local machine) an account with the same login name as the user owner of the remote exported home directory. Then, he mounts the remote partition via NFS and modifies/creates a ".rhosts" file containing the string "++", thus allowing access to anybody from anywhere under this login. Finally, he can initiate a connection to the remote machine with the "rlogin" command: he has obtained a user access level [9].

This attack is described in ADeLe as follows:

```
1  Alert NFS_Mount (IN IPaddr targetip, OUT String account, OUT Connection cnx) {
2  <EXPLOIT>
3    <PRECOND>
4      Accesslevel == "REMOTE" #initial access level required
5    </PRECOND>

6    <ATTACK> <LANG> "EDL" </LANG>
7      String  output;   #gets everything displayed in the console
8      Integer ret_val;  #exit code for the command
9      String rpc_services;
10     Integer ret_val0;
11     Connection shellhandler;

12     EVENT E0{
         #Exec_shell_cmd(<shell_command>,<console_output>,<return_value>)
13       Exec_shell_cmd("rpcinfo  -p  "+targetip,rpc_services,ret_val0);
14     }
15     IF (ret_val0==0)&&("portmapper" IN rpc_services)&&("mountd" IN rpc_services){
16       Non_ordered{ #unspecified order!
17           [ Integer ret_val1;
18               String exported_partitions;
19               EVENT E1{
20                 Exec_shell_cmd("showmount  -e  "+targetip,exported_partitions,ret_val1)
21               }
22           ]
23           [ Integer ret_val2;
24             String users_list;
25             EVENT E2{
26               Exec_shell_cmd("finger@"+targetip,users_list,ret_val2);
27             }
28           ]
29       }#Non_ordered
30       IF  (ret_val1==0)&&(ret_val2==0){
31         String partition_found;
32         String user;
             #Exists_exported_everyone(<input>,<partition_list>)
```

```
33          IF  Exists_exported_everyone(exported_partitions,partition_found)
              #Cross_part_users(<partition_list>,<users_list>,<matching_user>)
34          &&  Cross_partition_users(partition_found,users_list,user){
35            IF !Exists_local_user(user){
36              Add_local_user(user); #no observable event!
37            }
38            EVENT E3{
39              Exec_shell_cmd("mount -t nfs "+targetip+":/home/"+user+" /home/"+user,
                                output, ret_val)}
40            }
41            One_among{ #addition of "+ +" to the .rhosts file
42               [EVENT E4{
43                  Exec_ shell _cmd("echo '+ +' >>˜ '' +user+"/.rhosts",output,ret_val);
44                }
45               ]
46               [EVENT E5{
47                  Exec_ shell_ cmd("echo '+ +' >˜ " +user+" /.rhosts", output,ret_val);
48                }
49               ]
50            }#One_Among
51            EVENT E6{
52              shellhandler: =Exec_cmd_shell("rlogin "+targetip+" -1 "+account,
                                              output,ret_val);
53            }
54            #now we have "User" access level
55            account:=user;
56            CNX :=shellhandler;
57          }
58        }
59      }
60    </ATTACK>

61    <POSTCOND>
62      Accesslevel := "USER"
63    </POSTCOND>
64  </EXPLOIT>

65  <DETECTION>
66    <DETECT>
67      <EVENTS>
68        #list of events types occuring during this attack (IDMEF notation)
69        E0 : Network.Classification[0].name == "rpcinfo -p"
70        E1 : Network.Classification[0].name == "showmount -e"
71        E2 : Network.Classification[0].name == "finger o''
72        E3 : Network.Classification[0].name == "mount home_directory"
73        E4 : System.Classification[0].name == "file append"
74        E5 : System.Classification[0].name == "file create"
75        E6 : Network.Classification[0].name == "rlogin"
76      </EVENTS>

77      <ENCHAIN>
78        E0 ; Non_ordered{E1 E2} ; E3 ; One_among{E4 E5} ; E6
79      </ENCHAIN>

80      <CONTEXT>
81        E4.File_Modified.name == ".rhosts"
82        E5.File_Created.name == ".rhosts"
```

```
83        IPAddr X := E0.Target[0].Node.Address.address
84        E1.Target[0].Node.Address.address == X
85        E2.Target[0].Node.Address.address == X
86        E3.Target[0].Node.Address.address == X
87        E4.Target[0].Node.Address.address == X
88        E5.Target[0].Node.Address.address == X
89        E6.Target[0].Node.Address.address == X
90      </CONTEXT>
91    </DETECT>

92    <CONFIRM>
93      #File_Contains(<file>,<contents>)
94      File_Contains("/home/"+E6.Target[0]  .User.name+"/.rhosts"  ,  "+  +'');
95    </CONFIRM>

96    <REPORT>
97      #construction of the returned alert here
98      Alert.version                          := "1"
99      Alert.alertid                          := NewAlertid()
100     Alert.impact                           := ''9''
101     Alert.Time.time                        := NewTime()
102     Alert.Analyzer.ident                   := "121212"
103     Alert.Classification[0].origin         := "ADeLe''
104     Alert.Classification[0].name           := "NFS_Mount"
105     Alert.Classification[0].url            := ''''
106     Alert.Target[0].Node.Address.category := "2"
107     Alert.Target[0].Node.Address.address   := E0.Target[0].Node.Address.address
108     Alert.Source[0].Node.Address.category := "2"
109     Alert.Source[0].Node.Address.address   := E6.Source[0].Node.Address.address
110   </REPORT>
111 </DETECTION>

112 <RESPONSE>
113 </RESPONSE>
114 }
```