International Conference on Control, Engineering & Information Technology (CEIT'13)
Proceedings Engineering & Technology - Vol.3, pp. 45-50, 2013

TLID: A Tool for Linux Intrusion Detection

Ines Ben Tekaya[#1], Bechir Ayeb[#1], Mohamed Graiet*[2]

[#] *PRINCE Laboratory*
4011 Hammam Sousse, Tunisia
[1] bentekaya.ines@voila.fr
ayeb_b@yahoo.com
[*] *MIRACL, ISIMS*
BP 1030, Sfax 3018, Tunisia
[2] mohamed.graiet@imag.fr

***Abstract*— In this paper we describe architecture and implementation of a Tool for Linux Intrusion Detection (TLID). The TLID is based on formal verification. The main features of this work are twofold. It exploits formal method in the intrusion detection field. It presents our tool TLID which can automatically transform Linux code to Symbolic Model Verifier.**

***Keywords*— Computer attacks, intrusion detection, computer security, Linux commands, model verifier**

I. INTRODUCTION

The intrusion field was introduced by Anderson. It was defined as an attempt or a threat to be the potential possibility of a deliberate unauthorized attempt to access information, manipulate information, or render a system unreliable or unusable [1]. The difference between intrusion and attack consists of the fact that intrusion is a malicious, externally or internally induced fault resulting from an attack that has succeeded in exploiting vulnerability, while a fault is the adjudged or hypothesized cause of an error, the cause of which is intended to be avoided or tolerated. An attack is a malicious technical interaction fault aiming to exploit vulnerability as a step towards achieving the final aim of the attacker [2].

A statistical study shows that 98% of enterprises have a firewall to be protected from external attacks; however, 80% of attacks came from internal users [3]. Detecting internal normal user behaviour is a difficult problem because a user can have much dynamic behaviour and it will be almost difficult to create user profiles that determine the normal behaviour. Using a system to distinct normal user from intruders is necessary. This system is called Intrusion Detection System (IDS). It is defined as a security technology attempting to identify and isolate computer systems intrusions [4].

During the last two decades, many strategies and methods for intrusion detection have been developed. We choose to work with Unix/Linux operating system because in people's minds, if it is non-Windows, it is secure [5]. This hypothesis will be countered here. More details for Unix/Linux system can be found in [6]. The literature on detection using Linux/Unix commands offers a variety of methods. Despite their diversity, their common objective is: to distinguish between a normal behaviour and an intrusive behaviour. They use aggregative, training or experimental past data, such as CPU usage, memory usage, session times and resources access. However, in intrusion detection, we often find that the user can have a dynamic behaviour that is different from past data. So, in this work we don't use past data but we extract the relation between current data to determine the user behaviour class.

The reminder of the paper is organized as follow. Section 2 deals with intrusion background. In section 3, we describe our method. In section 4 we propose the TLID tool, and we show some experimental results for intrusion scenarios. In section 5 we will draw our conclusions and plan for future work.

II. INTRUSION BACKGROUND

The next subsections summarize detection methods using UNIX commands and show their limitations.

A. Detection Using UNIX Commands

The object of intrusion can be files, data bases, network connection, Input/output systems or commands Linux/Unix.

In this paper we are interested about intrusion using Linux/Unix commands because it can characterize user behaviour more efficiently than other object. The followings paragraphs present some works about methods using Unix commands. These works are interested on intrusion detection or on a specific intrusion like masquerade detection.

Ilgun, et al. present the state transition analysis method [7][8]. They used the known Unix intrusion to create a penetration scenario. A penetration is viewed as a sequence of actions performed by an attacker that leads from some initial state on a system to a target compromised state, where a state is a snapshot of the system representing the values of all volatile, semi-permanent and permanent memory locations on the system. The initial state corresponds to the state of the system just prior to the execution of the penetration. The compromised state corresponds to the state resulting from the completion of the penetration. Between the initial and compromised states are one or more intermediate state transitions that an attacker performs to achieve the compromise.

Another method is based on sequence matching. The incoming stream event is segmented into overlapping fixed length sequences. The choice of the sequence length, l, depends on the profiled user. In practical, it's fixed to the value l = 10 in the SEA dataset [9]. Each sequence is then treated as an instance in an l-dimensional space and is compared to the known profile. The profile is a set, {T}, of

previously stored instances and comparison is performed between all $y \in \{T\}$ and the test sequence via a similarity measure. Similarity is defined by a measure, Sim(x, y), which makes a point-by-point comparison of two sequences, x and y, counting matches and assigning greater weight to adjacent matches.

The maximum of all similarity values computed forms the score for the test command sequence. Since these scores are very noisy, the most recent 100 scores are averaged. If the average score is below a threshold an alarm is raised. The threshold is determined based on the quantiles of the empirical distribution of average scores [10].

Another method, used statistical method, is called uniqueness. It is based on the idea that commands not previously seen in the training data may indicate an attempted masquerade. Uniquely used commands account for 3% of the data. A command has popularity i if exactly i users use that command. They group the commands such that each group contains only commands with the same popularity. They define a test statistic that builds on the notion of unpopular and uniquely used commands. They assign the same threshold to all users. This threshold is estimated via cross validation: They split the original training data in the SEA dataset into two data sets of 4000 and 1000 commands. Using the larger data set as training data, they assign scores for the smaller one. This is repeated five times, each time assigning scores to a distinct set of 1000 commands. They set the threshold to the 99th percentile of the combined scores across all users and all five cross validations. For their data, the resulting threshold is 0.2319 [9][11].

Maxion use Naive Bayes classifiers and detect masqueraders by looking at the classifiers misclassification behavior [12]. This method use command occurrence probability distribution modeling the UNIX sequence. The goal of the training procedure is to establish profiles of self and nonself, and to determine a decision threshold for discriminating between examples of self and nonself. For each User X in the SEA dataset, a model of Not X can also be built using training data from all other victims. The probability of the test sequence having been generated by Not X can then be assessed in the same way as the probability of its having been generated by User X. The larger the ratio of the probability of originating with X to the probability of originating with Not X, the greater the evidence in favor of assigning the test sequence to X. The exact cut-off for classification as X, that is the ratio of probabilities below which the likelihood that the sequence was generated by X is deemed too low, can be determined by a cross-validation experiment during which probability ratios for sequences which are known to have been generated by self are calculated, and the range of values these legitimate sequences cover is examined.

B. Limitations in existing methods

The intrusion detection method in Linux/Unix commands using formal verification seeks to improve on some of limitations that the authors observed in the existing methods. This section briefly identifies some of their characteristics. The major weakness of these methods is that they depend on aggregative, training or experimental past data. The results of statistical methods are closed to the training data while the result of state transition analysis method is depend with the defined penetrations attacks which are non valuable now.

Another limitation is they are based on analysing command by command (line per line). This local analysis can not be equivalent to a global analysis (all of lines).

Lastly, they cannot make difference between the orders of commands in the sequence used. The statistical methods are based on the command frequency while a state transition analysis method can't detect the attacks based in frequency such as deny of service.

In the following, we focus in these limitations to present our method based on model using formal verification with Symbolic Model Verifier (SMV).

III. INTRUSION DETECTION IN LINUX/UNIX COMMANDS WITH SMV

This section presents an overview about our intrusion detection method by user specification. It's based on temporal logic and formal verification. It focuses on global analysis of user behaviour.

The user's observed behaviour is deduced from Linux terminal. In the rest of this paper, we use the term Linux, which can be interchanged with Unix. We are interested about a Linux script not about a line of commands. To perform a global analysis we should specify what are the anti-properties that characterize an attack script?

These properties formed the requirements specification (the desirable behaviour). We choose to transform them into temporal logic.

In this paper, we concentrate on formal verification technique, especially model checking, because that allows, in general, less user involvement in the verification process. We exploit model-checking to automatically verify if a given user behaviour (user's observed behaviour) satisfied the requirement specification (the desirable behaviour). If the verification tool provides a counter-example, we deduce that the user's observed behaviour don't satisfy the requirement specification. We choose the SMV tool for verification.

The user's observed behaviour will be transformed into SMV code. However Linux script differs from SMV code. We propose our tool, named LSc2SMV (Linux Script to Symbolic Model Verifier) to do the transformation.

We obtain a SMV program containing logical properties which we verify by SMV tool. The result will be verified properties if the behaviour is normal or violated properties if the behaviour is intrusive. Figure 1 illustrates this schema.

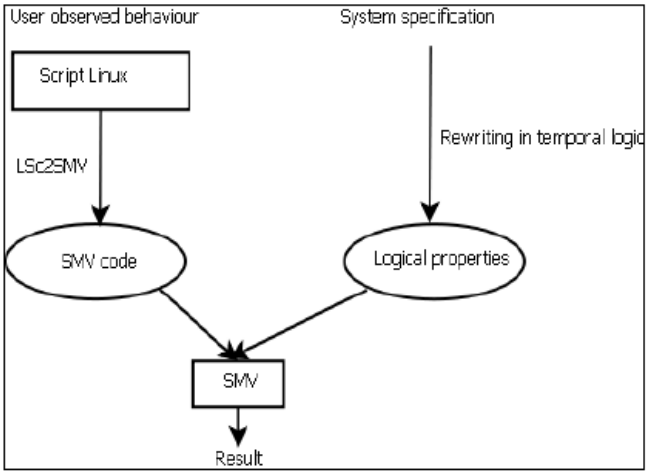

Fig.1 A diagram tracing our method.

A. From anti-proprieties to temporal logic

The anti-properties (AP) are unwanted properties that can cause damage in our system. They can be:

- AP1: Execute some illegal commands,
- AP2: Change source or command destination,
- AP3: Execute illegal actions (parameters, etc.),
- AP4: Having infinite loop,
- AP5: Having auto-replication,
- AP6: Detain a resource infinitely …

The system specification are formalizes using the AP. They can be expressed in proportional logic or temporal logic.

Propositional logic is the branch of logic that studies ways of joining and/or modifying entire propositions, statements or sentences to form more complicated propositions, statements or sentences, as well as the logical relationships and properties that are derived from these methods of combining or altering statements.

The temporal logic is used within the framework of the reagent systems, which where the software is supposed to maintain a relation of coherence between the input flows and the output flows. The temporal logic allows expressing the state evolution of a system.

We choose the temporal logic because temporal logic is an extension of propositional logic. Either in temporal logic, propositions are qualified in terms of time.

The following paragraph explains how to write some of the anti-properties AP and properties (P) using temporal logic.

AP4: Having infinite loop

The AP4 consider that user can modify the system performance. So they consume memory to overload the system.

P4: Do not have infinite loop ; AP4 = G ^¬(ai ^aj)

let:G: always

^: and operator

¬: not operator

ai : loop and aj: loop condition

An example is: while(true), while(i :=i+1), etc.

Some others anti-properties can be formalized such as having auto-replication, detain a resource infinitely, etc. Due to space limitation, others properties can be found in [13].

B. SMV

SMV is a formal verification tool, which means that when you write a specification for a given system, it verifies that every possible behavior of the system satisfies the specification.

A specification for SMV is a collection of properties. Properties are specified in a notation called temporal logic. Temporal logic specifications can be automatically formally verified by a technique called model checking.

SMV is quite effective in automatically verifying properties. Sometimes, when checking properties, the verifier will produce a counterexample. This is a behavioral trace that violates the specified property. This makes SMV a very effective debugging tool, as well as a formal verification system, so that is why we choose SMV for verification.

C. From Linux script to SMV

The LSc2SMV tool will convert Linux script (LSc) to an SMV code. It will be in the form of main module ().Table I shows the transformation in constants and variables.

TABLE I
VARIABLES AND CONSTANTS CASES

Type	LSc	SMV
Integer variable	varname = valeur	VAR <varname> : number ;
Variable of an interval	for i in 0 1 2 3 4	VAR <varname> : 0..4 ;
Constant	SIZE=32	#define SIZE 32

Table II shows the transformation in the condition and loop cases form.

TABLE II
CONDITIONS AND LOOP CASES

Type	LSc	SMV
Condition	if[<condition>] <stmt1> else <stmt2> fi	if(<condition>) <stmt1> else <stmt2>
Case	case $variable in val1) stmt1> ; ; ..... *) <stmtn> ; ; esac	case{<cond1> : <stmt1> ... <condn> : <stmtn> [default : <dftlstmt>]}
Switch	switch(<expr>) <case1> : <stmt1> breaksw <casen> : <stmtn> breaksw default : <dftlstmt> breaksw endsw	switch(<expr>){ <case1> : <stmt1> ... <casen> : <stmtn> [default : <dftlstmt>]}
for	for var in $files ; do	for(var = init ; cond ; var = next) <stmt>
while	while condition ; do <stmt> done	-

The indirect transformation is based on properties to verify in Linux script.

Some other conversion in the file name or in the folder name, in arrays, in expressions cases, in functions … can be given. More details can be found in [14].

IV. TLID: TOOL FOR LINUX INTRUSION DETECTION

The TLID architecture can survey a user and analyze his behaviour.

A. Survey a user

There are two solutions to survey a user:

- The first solution consists in using the file .bash_history. But this file cannot give a strengthened and real-time history because when you use other shell, like csh,, this method cannot save the history. Either when you tape kill -9.
- The second solution is to develop a patch. It consists to modify file system in Linux, which are bashhist.c, histexpand.c, histfile.c, history.h and history.c. We do this because Linux is an open source (to obtain the patch e-mail : bentekaya.ines@voila.fr).

You can choose a user and we obtain the user's observed behaviour. You can either choose a user and a day, shown in figure 2, and we obtain the user's observed behaviour in this day. It is composed by time, process identifier (PID) and commands.

B. Analyse user behaviour

After survey a user, you can choose a property to verify. In this example, we choose to verify the service deny in figure 3. The button LSc2SMV became enabling. When we click below, we obtain the SMV file. This file contains the verification of every actions do by selected user in the chosen day. It consists to verify the specified properties. We choose ``Prop|Verify all'' to verify if the properties we specified in fact hold true or false for all time. If the property should be false, a counterexample appears in the trace page.

C. Intrusion scenario example

Intrusion scenario Sc between users can be defined as:

Sc = {A, V, S} with:

A: an attacker

V: a victim

S = {s1, s2… sn}: a set of steps

Every step is a sequence of commands with their parameters. The next paragraph shows an example of scenario. It have been developed and tested in Linux Red Hat Enterprise version 5 and we use TLID and SMV for verification.

We develop an example of denial of service which is a fork bomb. The code in figure 4 is the following:

```
[ines@localhost tmp]$ function testb()
{
testb|testb &
} ;testb
```

It works by creating a large number of processes very quickly in order to saturate the available space in the list of processes kept by the computer's operating system. If the process table becomes saturated, no new programs may start until another process terminates.

The generated SMV code is given by figure 4. The properties to verify is called deny. We choose ``Prop|Verify all'' to verify deny. The result is given by figure 5. We have a violated property (false value) because the behaviour is intrusive.

V. CONCLUSIONS

In this paper, we are interested by attacks using Linux commands. We have proposed a method that exploits model checking. This model use algorithms, executed by computer tools, to verify the correctness of our system. It combines security field with formal verification.

The user's observed behaviour is deduced from Linux terminal. We are interested about a Linux script not about a line of commands. To perform a global analysis we should specify the anti-properties that characterize an attack script.

These properties formed the requirements specification (the desirable behaviour). We choose to transform them into temporal logic. We exploit model-checking to automatically verify if a given user behaviour satisfied the requirement specification. If the verification tool provides a counter-example, we deduce that the user's observed behaviour don't satisfy the requirement specification.

This method is applied to distinct normal user behavior from intruders' behavior. It has lead to the TLID tool development. We give some experimental results to show how the TLID works under some attacks.

There is another attacks group which can be named unknown attacks. In this new group, attacks could cause the intrusion detection systems crash and thus incomplete testing. It becomes clear that present approaches to evaluate intrusion detection system are limited to some known attacks.

We divide our future work into two main parts: refine and improve attacker competence and extend scenario to include multi-attacks and equivalent attacks.

REFERENCES

[1] J. P. Anderson, "Computer Security Threat Monitoring and Surveillance," Technical report, Washing, PA, James P. Anderson Co., 1980.

[2] D. Powell and R. Stroud, "Conceptual Model and Architecture of MAFTIA", Eds., MAFTIA (Malicious and Accidental Fault Tolerance for Internet Applications) project deliverable D21, LAAS-CNRS Report 03011, 2003.

[3] C. Matheï,. (2004) "Ouverture des réseaux IP d'entreprise : risques ou opportunité ?" [Online]. Available: http://www.awt.be/contenu/tel/res/IPforum23-04_Réseau unifié et sécurisé.pdf.

[4] B. E. Cloete and L. M. Venter, "A comparison of Intrusion Detection systems" Computers & Security, vol 20, Issue 8, pp. 676-683, Dec. 2001.

[5] A. Patrizio. (2006) "Linux Malware On The Rise. " [Online]. Available: http://www.internetnews.com/devnews/article.php/3601946.

[6] M. Santana, "Chapter 6 - Linux and Unix Security, Computer and Information Security" Handbook 2009, pp. 79-92.

[7] Koral Ilgun , Richard A. Kemmerer , Phillip A. Porras. "State Transition Analysis: A Rule-Based Intrusion Detection Approach. " Journal IEEE TRANSACTIONS on Software Engineering, Vol. 21, No. 3, pp. 181-199, 1995.

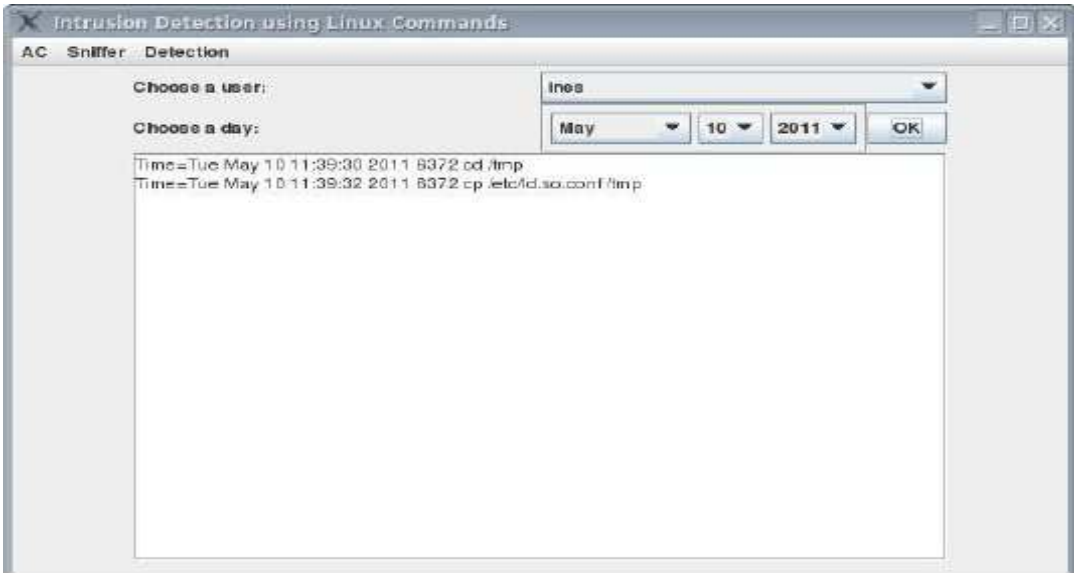

Fig.2 User's observed behaviour in a chosen day

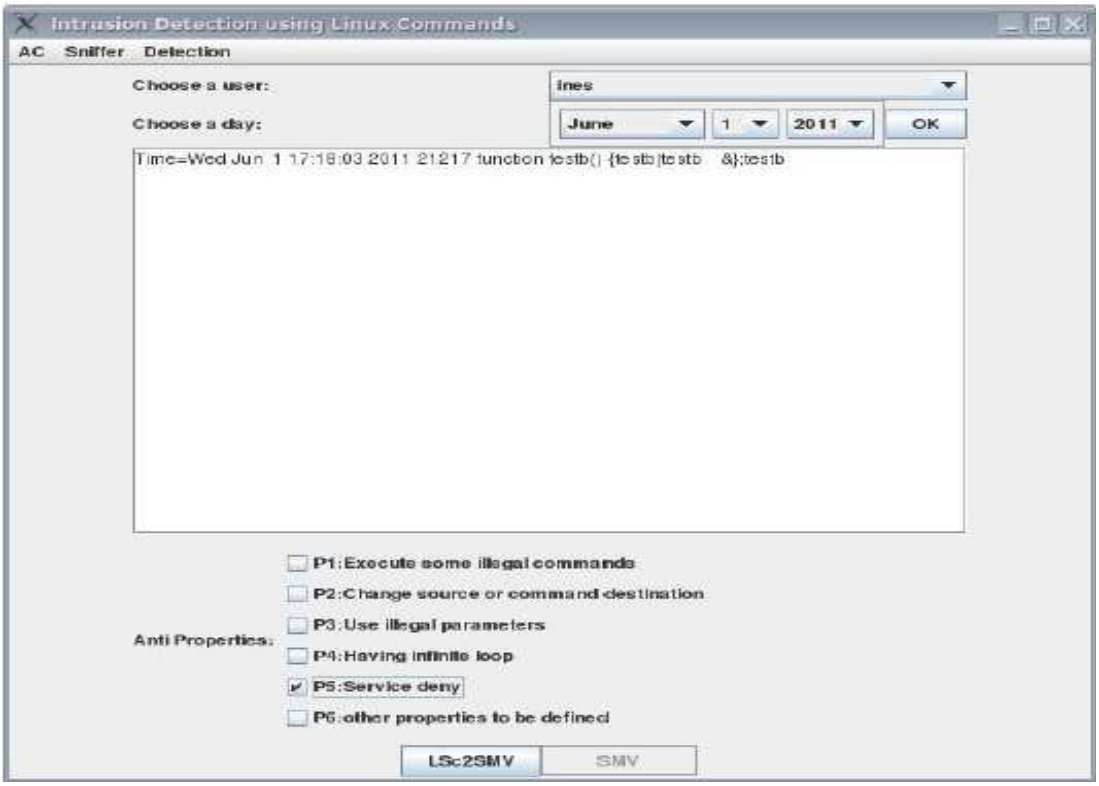

Fig.3 TLID

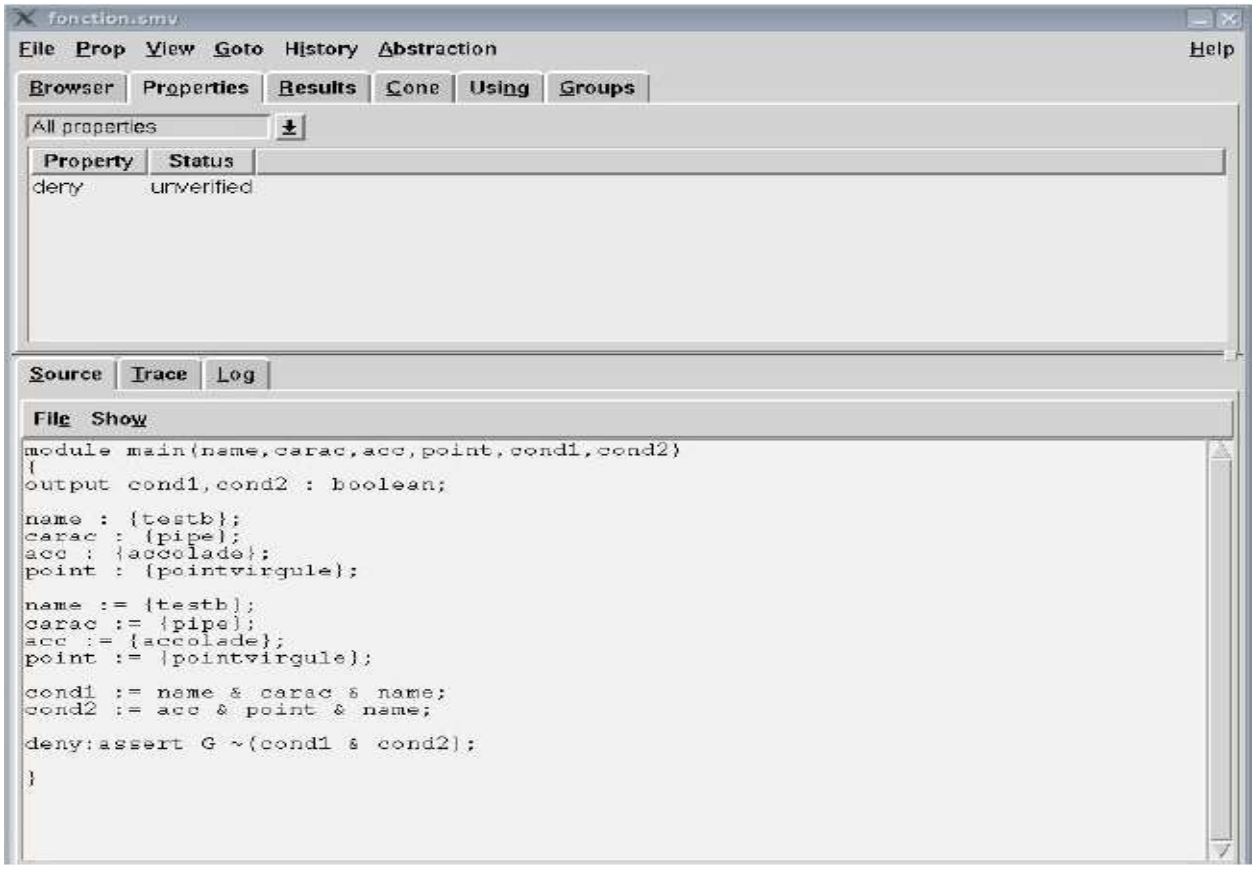

Fig.4 The generated SMV code

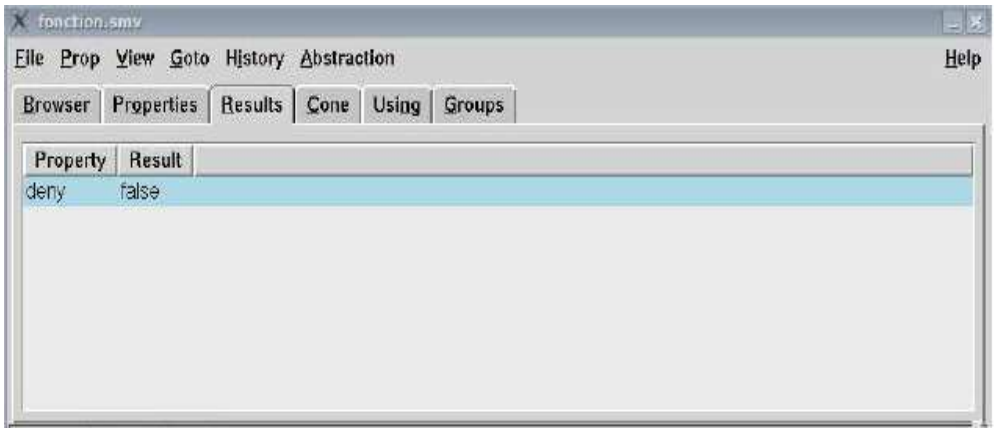

Fig.5 Verification result

[8] K. Ilgun. "USTAT - A Real-time Intrusion Detection System for UNIX," Master's Thesis, University of California at Santa Barbara, Nov. 1992.

[9] M. Schonlau, W. DuMouchel, W. H. Ju, A. F. Karr, M. Theus and Y. Vardi. "Computer Intrusion: DetectingMasquerades" Statistical Science, Vol. 16, No. 1,pp 1–17, 2001.

[10] T. Lane and C E. Brodley. "Sequence matching and learning in anomaly detection for computer security." In AAAI Workshop : AI Approaches to Fraud Detection and Risk Management, pp. 43–49. AAAI Press (1997).

[11] M. Theus and M. Schonlau. "Intrusion detection based on structural zeroes." Statistical Computing and Graphics Newsletter 9, pp. 12–17, 1998.

[12] M. Roy. "Masquerade detection using enriched command lines." In: Proceedings of international conference on Dependable Systems and Networks (DSN-03), pp. 5-14, June 2003.

[13] I.B. Tekaya, M. Graiet, and B. Ayeb. Intrusion detection with symbolic model verifier. In the Sixth International Conference on Software Engineering Advances, ISBN: 978-1-61208-165-6, pages 183–189 (2011).

[14] I.B. Tekaya, M. Graiet, and B. Ayeb. Intrusion detection in Linux/Unix commands using formal verification. In The Fourth IEEE International Symposium on Innovation in Information and Communication Technology (2011).