Intrusion Detection with Symbolic Model Verifier

Ines Ben Tekaya
PRINCE Laboratory
4011 Hammam Sousse, Tunisia
bentekaya.ines@voila.fr

Mohamed Graiet
MIRACL, ISIMS
BP 1030, Sfax 3018, TUNISIA
mohamed.graiet@imag.fr

Bechir Ayeb
PRINCE Laboratory
4011 Hammam Sousse, Tunisia
ayeb_b@yahoo.com

Abstract— **Many intrusions came from internal users. This behavior can cause damage without human intervention: viruses, worms, trojan horses, etc. This paper describes our intrusion detection method in Linux/Unix commands using 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 transform Linux code to Symbolic Model Verifier.**

Keywords-attacks; intrusion; security; scenarios; 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 behavior is a difficult problem because a user can have much dynamic behavior and it will be almost impossible to create user profiles that determines the normal behavior. 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].

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 behavior and an intrusive behavior. From an abstract view point, we organize these work into one main approach: empirical approach. This classification included methods based on aggregative, training or experimental past data. The present work falls mainly within the model approach. The data are not based in the past event but they compose a model. It is a theoretical representation of a system which is composed of elements and relation.

The reminder of the paper is organized as follow. Section 2 deals with intrusion background. Section 3 describes our method. Section 4 proposes practical tool and experimental results for intrusion scenarios. Section 5 summarizes the paper, with concluding remarks.

II. Intrusion Background

The next subsections summarize attacks topology, some dataset used in the literature for intrusion detection and show detection methods using Unix commands

A. Attacks topology

Attacks take several forms to break one or more of the security properties. They can be grouped according to their functionality as described in the following subsections [7]:

- Gathering Security-relevant Information: Before experiencing an attack, a hacker tries to obtain necessary information that is probably sensible about the targeted system, which can be employed later to obtain access to this system. Useful information can be obtained by different ways such as network scanning and vulnerability scanning or even by using public search engines such as Google or social engineering methods.
- Access Gain Attacks: With information gathered by the above methods, attackers try to obtain a privileged access on a system by exploiting vulnerabilities in the services or the applications installed on this system or a bad configuration of the network. This kind of attacks primarily grants unauthorized access to the targeted system. For example, one of the configuration problems is the use of weak passwords in systems where a bad policy of password definition allows users to choose simple and easy guessable passwords. Otherwise, an attacker can use cracking tools such as “john the ripper” [8] to obtain passwords by brute-force. Buffer-overflow attacks are another example that allows attackers to execute arbitrary code on the targeted hosts.

- Denial of service (DoS): DOS attacks are designed to overload or disable the capabilities of a machine or a network, and thereby render it unusable or inaccessible. An example of denial of service is a fork bomb. 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.
- Malware Attacks: This category of attacks can result in damages as simple as displaying a simple flicker to catastrophic damages such as completely formatting hard disks. It groups virus, worm, Trojan horse, spyware, rootkit [9] and spam.

B. 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 classified into two classes: the class of intrusion detection and the class of masquerade detection.

Ilgun, et al. present the state transition analysis method [10][11]. 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 stat 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.

This 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 [12]. 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 [13].

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 [12][14].

Another method is called Bayes 1-Step Markov Model. It is proposed by Schonlau, et al. The authors use the information of 1-step command transition probabilities. They build transition matrices for each user’s training and testing data. The detector triggers the alarm when there is a considerable difference between the training data transition matrix and the testing data matrix. This technique was the best performer in terms of correct detections, but failed to get close to the desired false alarm rate [12].

Maxion use Naive Bayes classifiers and detect masqueraders by looking at the classifiers misclassification behavior [15]. 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.

C. 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 statical 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 statical 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 our method. It combines tests on the direct and indirect ways to detect the intrusions. It focuses on global analysis. The following proposition plays a central role here.

Proposition 1. *A global analysis can not be realized in k local analysis.*

Example 1. Let *GA* is a global analysis and *LA={u_1, u_2, …, u_k}* a k local analysis. Suppose that *GA* can be realized in k local analysis. In this case, if GA is false, we must have one or more u_i is false.

This supposition is false because we can find GA is false while LA is true. The example is here: We have two users X and Y. User X can execute the following actions : modify all executable files, named F and that he have write permission, owned by user Y. X append some code to files F. When any users, that have write permission in these file, execute F, all F files will be infected. These actions can be:

1. X search all Y executables files, that X have write permission,
2. X append some legal code to infect files *F*
3. Any authorised users execute one of *F* files
4. All *F* files will be infected

The local analysis for actions 1, 2 and 3 are legal. They have a true value, but the global analysis gives a false value: all F files will be infected.

To perform a global analysis we should specify what are the anti-properties that characterize an attack script.

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 the anti-properties AP to properties (P) using temporal logic.

AP1: Execute some illegal commands

The AP1 consider that user can execute some commands. For example, if the user is an administrator, he can execute commands like adduser, userdel, etc.

P1: Do not execute some illegal commands

P1 = {(Ui,,Cj)/Ui ∈ U et Cj ∈ C}

where: U: set of users

C: set of illegal commands

(Ui, Cj): Ui can use Cj

Use(Ui, Cj) → (Ui, Cj) ∉ P1

AP2: Change source or command destination

The AP2 consider that the command path was modified.

P2: Do not change source or command destination

P2 = {(Ui,,Fj)/Ui ∈ U et Fj ∈ F}

where: U: set of users

F: set of illegal folder

(Ui, Cj): Ui can't write on Fj

Write(Ui, Fj) → (Ui, Fj) ∉ P2

An example is: write(user1, /bin/cp)

AP3: Execute illegal actions (parameters, etc.),

The AP3 consider that some user can use or modify objects of other users that he don't have a permission.

P3: Do not execute illegal actions (parameters , etc.)

P3 = {(Ui,Oj)/Ui ∈ U et Oj ∈ O}

where: U: set of users

O: set of illegal objects.

(Ui,Oj) : Ui can read Oj

Read (Ui,Oj) → (Ui,Oj) ∉ P3

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.

The user observed behavior is the possible behavior. It is deduced from Linux/Unix terminal. We are interested about a script not about a line of commands.

In this paper, we concentrate on formal verification technique that is based on temporal logic, because that allows in general less involvement of the user in the verification process: model checking.

Our basic idea is to exploit model checking. This model use algorithms, executed by computer tools, to verify the correctness of our system. The user inputs a description of a model of the system (the possible behavior) and a description of the requirements specification (the desirable behavior) and leaves the verification up to the machine. If an error is recognized the tool provides a counter-example showing under which circumstances the error can be generated. The counterexample consists of a scenario in which the model behaves in an undesired way.

In the rest of this paper, we use the term Linux, which can be interchanged with Unix. Our method is based in the user's observed behavior and in the system specification. The user's observed behavior is modeled by a Linux script. It will be transformed into SMV code. However Linux script differs from SMV code. We propose LSc2SMV (Linux Script to Symbolic Model Verifier) tool to do the transformation.

--The user observed behavior is transformed by our proposed tool, named LSc2SMV (Linux Script to Symbolic Model Verifier), to SMV code.

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

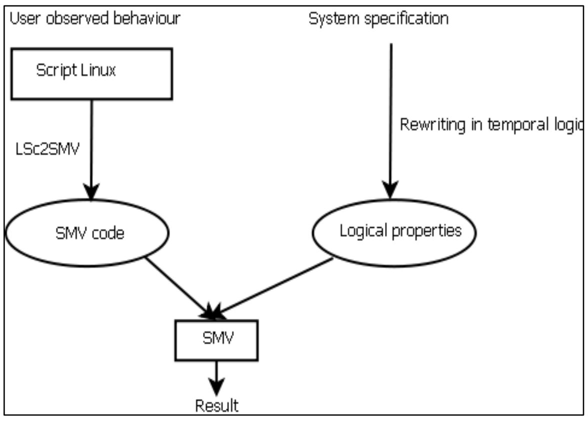


Figure 1. A diagram tracing our method.

The LSc2SMV tool will convert Linux script to an SMV code. It will be in the form of main module (). We show the transformation in constant, in variables, in arrays, in expressions, in functions, and in loops and conditions. Tables I, II, III, IV and V give this direct transformation.

Table I shows the transformation in constants and variables.

TABLE I. VARIABLES AND CONSTANTS CASES

Type	LSc	SMV
Integer variable	varname = valeur	VAR <signal> : number ;
Variable of an interval	for i in 0 1 2 3 4	VAR <signal> : 0..4 ;
Constant	SIZE=32	#define SIZE 32
Initialisation	signal = ready	init(signal) := ready ;
Modification	signal = busy	next(signal) := busy ;

Table II shows the transformation in arrays cases.

TABLE II. ARRAYS CASES

Type	LSc	SMV
Array	declare -a nametab	<nametab> : array <x>..<y> of <type> ;
Matrix	char mat[2][2]	mat : array 0..1 of array 2..0 of boolean ;

Table III shows the transformation in expressions cases.

TABLE III. EXPRESSIONS CASES

Type	LSc	SMV
Boolean operators	-a (and) -o (or) !(not)	(“and”,“or”,“not”)
Condition operators	if-then-else case switch	if-then-else case switch
Arithmetical operators	+, -,* , /, %	+, -,* , /,mod
Comparison operators	-eq , -ne, -lt, -gt, -le, -ge	“=”,“<”, “>”, “>=”, “<=”)

Table IV shows the transformation in the function case form.

TABLE IV. FUNCTION CASE

Type	LSc	SMV
function	function name() {... } ;	MODULE name(input, output) {... }

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

TABLE V. 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	switch(<expr>){ <case1> : <stmt1> ... <casen> : <stmtn> [default : <dftlstmt>]}

	endsw	

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 and in Linux script.

Some other conversion in the file name or in the folder name can be made. This is because SMV cannot support some character like . or / in the variable name. The Table VI gives some conversion.

TABLE VI. NAME TRANSFORMATION

Type	LSc	SMV
File name	/etc/passwd,/etc/inittab, /etc/ld.so.conf, etc/lilo.conf,etc/group	etcpasswd,etcinittab, etcldsoconf,etcliloconf, etcgroups
Folder name	/var,/usr/bin,/dev, /etc/security, /var/spool,/etc, /usr/etc,/usr,/usr/lib,/	var,usrbin,dev,etcsecurity, varspool,etc,usretc,usr, usrlib,slash

IV. TLID: TOOL FOR LINUX INTRUSION DETECTION

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 which are bashhist.c, histexpand.c, histfile.c, history.h and history.c (to obtain the patch e-mail : bentekaya.ines@voila.fr). When a user writes anything in the console, it will be saved in a file using his name. This patch can be used in every system to survey a command user.

Figure 2 gives some functionality of TLID. You can choose a user, a day and we obtain the behavior. It is composed by time, PID and commands.

After that you can choose a property to verify. In this example, we choose to verify the use of illegal parameters. The button LSc2SMV became enabling. When we click below, we obtain the SMV file. This file contains the verification of action 1: cd /tmp and action 2: cp /etc/ld.so.conf /tmp. It consists to verify the permission of using folder /tmp and /etc/ld.so.conf file. This is given by SMV file in Figure 4. The two properties we specified are file confidentiality (conf) and folder confidentiality (confo). We choose ``Prop|Verify all" to verify if the properties we specified in fact hold true or false for all time. The result is given by Figure 5. The conf property should be false, and a counterexample appears in the trace page. This because ines user use a file that he don't have a permission.

TLID can do a local analysis a global analysis between users.

Intrusion scenario Sc between users can be defined as:

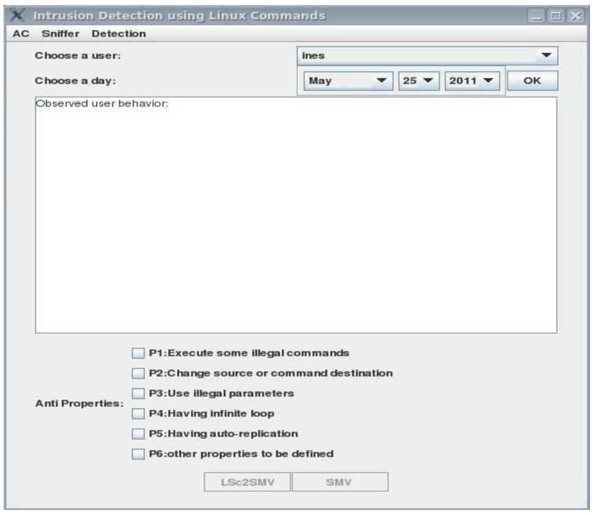
Sc = {A, V, S} with:

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.



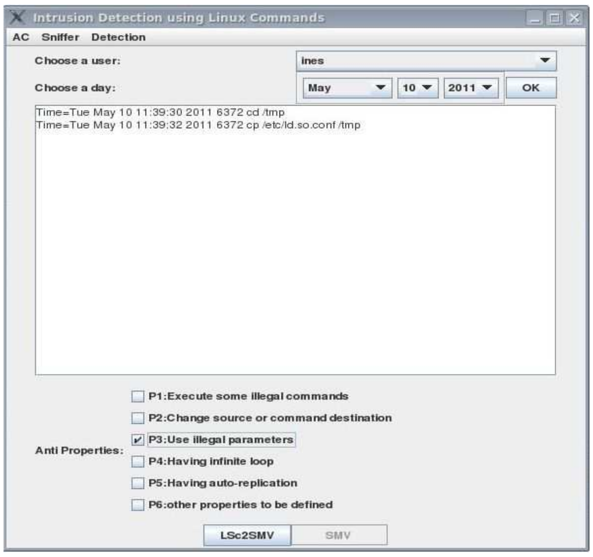
Choose a user: ines

Choose a day: May 25 2011

Observed user behavior:

Anti Properties:
- [ ] P1:Execute some illegal commands
- [ ] P2:Change source or command destination
- [ ] P3:Use illegal parameters
- [ ] P4:Having infinite loop
- [ ] P5:Having auto-replication
- [ ] P6:other properties to be defined

Figure 2. TLID



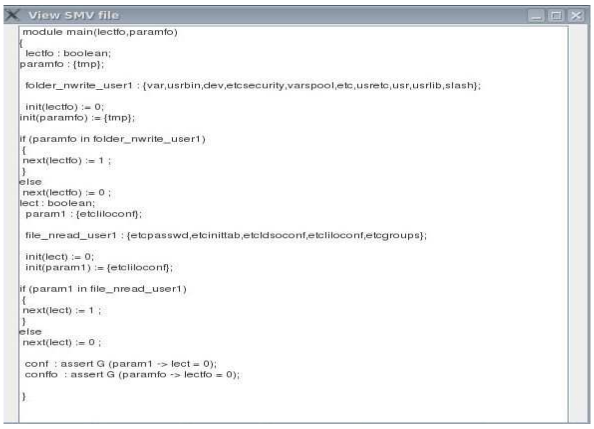
Choose a user: ines

Choose a day: May 10 2011

```
Time=Tue May 10 11:39:30 2011 6372 cd /tmp
Time=Tue May 10 11:39:32 2011 6372 cp /etc/ld.so.conf /tmp
```

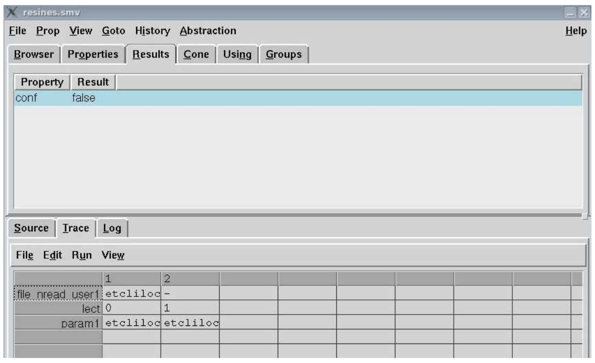
Anti Properties:
- [ ] P1:Execute some illegal commands
- [ ] P2:Change source or command destination
- [✓ P3:Use illegal parameters]
- [ ] P4:Having infinite loop
- [ ] P5:Having auto-replication
- [ ] P6:other properties to be defined

Figure 3. Observed ines behavior in May-10-2011



```
module main(lectfo,paramfo)
{
 lectfo : boolean;
paramfo : {tmp};

 folder_nwrite_user1 : {var,usrbin,dev,etcsecurity,varspool,etc,usretc,usr,usrlib,slash};

 init(lectfo) := 0;
init(paramfo) := {tmp};

if (paramfo in folder_nwrite_user1)
 {
 next(lectfo) := 1 ;
 }
else
 next(lectfo) := 0 ;
lect : boolean;
 param1 : {etcliloconf};

 file_nread_user1 : {etcpasswd,etcinittab,etcldsoconf,etcliloconf,etcgroups};

 init(lect) := 0;
 init(param1) := {etcliloconf};

if (param1 in file_nread_user1)
 {
 next(lect) := 1 ;
 }
else
 next(lect) := 0 ;

 conf  : assert G (param1 -> lect = 0);
 conffo  : assert G (paramfo -> lectfo = 0);

 }
```

Figure 4. SMV file


resines.smv
File Prop View Goto History Abstraction Help
Browser Properties Results Cone Using Groups
Source Trace Log
File Edit Run View

Property	Result
conf	false

	1	2
file_nread_user1	etcliloc	-
lect	0	1
param1	etcliloc	etcliloc

Figure 5. Verification with SMV

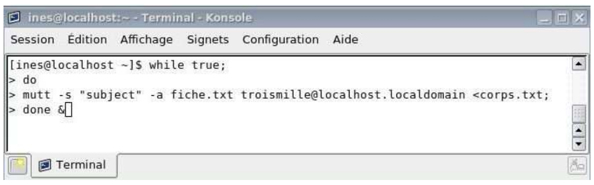
We have two users. The victim is named ‘troismille’ (user-id: 3000) and the attacker is named ‘ines’ (user-id: 5502).

[root@localhost ~]# cat /etc/passwd

Result:ines:x:5502:5502::/home/ ines:/bin/bash

troismille:x: 3000: 3000::/home/ troismille:/bin/bash

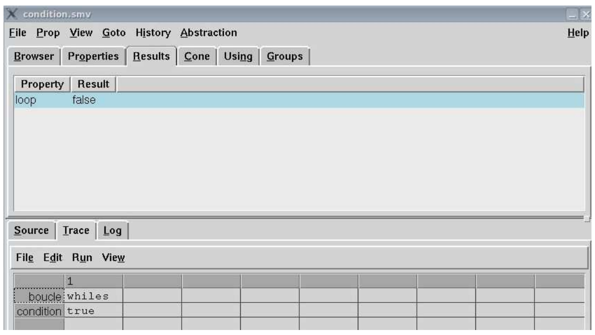
This scenario consists of sending many mail from user ines to user troismille to saturate his mail. In this case, the user troismille cannot access to his e-mail. The scenario is given by Figure 6.


ines@localhost:~ - Terminal - Konsole
Session Édition Affichage Signets Configuration Aide
Terminal

```
[ines@localhost ~]$ while true;
> do
> mutt -s "subject" -a fiche.txt troismille@localhost.localdomain <corps.txt;
> done &
```

Figure 6. An example of scenario

Using TLID, we choose to the anti property: Having infinite loop. If we don’t know how a property to choose, we can mark all checkbox. The result is given by Figure 7.


condition.smv
File Prop View Goto History Abstraction Help
Browser Properties Results Cone Using Groups
Source Trace Log
File Edit Run View

Property	Result
loop	false

	1
boucle	whiles
condition	true

Figure 7. The result

V. CONCLUSION

In this paper, we are interested by attacks using Linux commands. We have presented their topology. We have shown that their impact can be inoffensive or can destroy information system.

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 inputs a description of a model of the system (the possible behavior) and a description of the requirements specification (the desirable behavior) and leaves the verification up to the machine. If an error is recognized the tool provides a counter-example showing under which circumstances the error can be generated. The counterexample consists of a scenario in which the model behaves in an undesired way.

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/dev-news/article.php/3601946.

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

[7] M. E. S. Gadelrab, "Évaluation des Systèmes de Détection d'Intrusion," thèse, Université de Toulouse - Paul Sabatier, France, Dec. 2008.

[8] M. F. Krafft (2007) "John the Ripper password cracker:" [Online]. Available: http://www.openwall.com/john/.

[9] G. Hoglund, and J. Butler, "Rootkits: Subverting the Windows" Kernel, Addison-Wesley Professional, 2005.

[10] 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.

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

[12] 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.

[13] 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).

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

[15] 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.