บรรณานุกรม

[1] N. Nuansri, S. Singh, T. S. Dillon ,"A Process State-Transition Analysis and its Application to Intrusion Detection", Dept. Computer Science and Computer Engineering, and Applied Computing Research Institue (ACRI), Latrobe University, Bundoora VIC 3083 Melbourne, Australia, 1999.

[2] C. Dowell and P. Ramstedt, "The COMPUTER WATCH data reduction tool," in *Proceedings of the 13th National Computer Security Conference*, 1990.

[3] W.T. Tener, "Discovery: an expert system in the commercial data security environment," in *Proceedings of the 4th IFIP TC11 International Conference on Computer Security*, 1986.3.

[4] S. E. Smaha, "Haystack: An Intrusion Detection System," in *Proceedings of the 4th Aerospace Computer Security Applications Conference*, florida (December 12-16, 1998) pp. 37-44.

[5] D. Anderson, T. Lunt, and H. Javitz, "A. Tamaru, and A. Valdes, Next-generation Intrusion Detection Expert System (NIDES)," A Summary, SRI-CSL-95-07, SRI International, 1995.

[6] J. R. Winkler and J. C. Landry, "Intrusion and Anomaly Detection: ISOA Update," in *Proceedings. 15th National Computer Security Conf.*, Baltimore, MD, October 1992, pp. 272-281.

[7] Sebring, M. M.; Sellhouse, E.; Hanna, M. E.; Whitehurst, R. A.: "Expert system in intrusion detection: A case study," in *Proceedings of the 11th National Computer Security Conference*, Baltimore, MD, Oct. 1988, pp. 74 – 81.

[8] Vaccarro, H. S. and Liepins, G. E. "Detection of Anomalous Computer Session Activity," in *Procedding of the IEEE Symposium on Research in Security and Privacy*, pp. 208-209, Oakland, California, May 1-3, 1989. Washington, DC: IEEE Computer Society Press, 1989.

[9] T. Escamilla, "Intrusion Detection Network Security Beyond the Firewall," USA: John Wiley & Sons, Inc, 1998.

[10] CERT and CERT Coordination Center are registered U.S. Patent and Trademark Office. 2004. CERT/CC Advisor. http://www.cert.org/advisors/.

[11] ร.อ. วิวัฒ เรืองมี. 2544. **Basic Intrusion Detection FAQ.** http://www.thaicert.nectec.or.th/paper/ids/idsfaq1.php.

[12] Paul Festa. 1999. **Study says "buffer overflow" is most common security bug** http://community.core-sdi.com/~juliano/0-1003-200-1462855.html.

[13] B. Hatch, J. Lee, G. Kurtz, "Hacking Linux Exposed: Linux Security Secrets & Solutions," 2001.

[14] J.P., Anderson, "Computer Security Threat Monitoring and Surveillance," Technical Report James P. Anderson Company, Fort Washington, Pennsylvania (February 1980).

[15] T. F. Lunt et al., "A Real-Time Intrusion Detection Expert System(IDES)," Interim Progress Report, Project 6784, SRI International, May 1990.

[16] M. Crosbie and G. Spafford, "Active Defense of a Computer System using Autonomous Agents," technical Report, Purdue University, Department of Computer Sciences, February 1995.

[17] G. Viga and R. A. Kermmerer, Netstat, "A Network-based intrusion detection approach," In *Proceeding of the 1998 Annual Computer Security Applications Conference (ACSAC98)*, pages 25-34, Los Alamitos, CA, December 1998, IEEE Computer Society, IEEE Computer Society Press. Scottsdale, AZ.

[18] P. Porras and R. Kemmerer, "Penetration State Transition Analysis: A Ruled based Intrusion Detection Approach," *Eighth Annual Computer Security Application Conference*, 1992.

[19] S. Kumar and E. Spafford, "A Pattern-Matching Model for Intrusion Detection," *Nat'l Computer Security Conference*, 1994.

[20] S.T. Eckmann, G. Vigna, and R.A. Kemmerer, "STATL: An Attack Language for State-based Intrusion Detection," *Journal of Computer Security*, 10(1/2):71-104, 2002.

[21] U. Lindqvist and P.A. Porras, "Detecting Computer and Network Misuse with the production Based Expert System Toolset (P-BEST)," in *IEEE Symposium on Security and Privacy*, pages 146-161, Oakland, California, May 1999.

[22] V. Paxon. Bro, "A system for Detecting Network Intruders in Real-time," in *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, january 1998.

[23] M. Christodorescu & S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," *USENIX Security Symp.*, 2003

[24] A. Habib, M. Hefeeda & B. Bhargava, "Detecting Service Violations and DoS Attacks," NDSS, 2003.

[25] V. Paxson, "An Analysis of Using Reflectors for Distributed Denial-of-Service Attacks," *Computer Communication Review* 31(3), 2001.

[26] S. Forrest, S.A. Horfmeyr and A. Somayaji, "Intrusion Detection using Sequences of System Calls," *Journal of Computer Security* Col.6 (1998) pages 151-180.

[27] A. Kosoresow and S. Hofmeyr, "Intrusion Detection via system call traces," *IEEE Software* ,1997.

[28] 19 W. Lee and S. Stolfo, Data Mining Approaches for Intrusion detection, USENIX Security Symposium, 1998.

[29] S. Snap, J. Bretano, G. Dias, T. Goan, L.Heberlein, C. Ho, K. Levitt, B. Mukherjee, S. Smaha, T. Grance, D. Teal, D. Mansur, "DIDS : Motivation, Architecture and an Early Prototype," in *Proceedings of the 14th National Computer Security Conferencem,* 1991.

[30] L. Heberlein, G. Dias, K. Levit, B. Mukherjee, J. Wood, D. Wolber, "A Network Security Monitor," *Proceedings of the IEEE Computer Society Symposium*, Research in Security and Privacy, pp. 296-303, May 1990.

[31] M.M. Sebring, E. Shellhouse, M. Hanna and R. Whitehurst, "Expert Systems in Intrusion Detection: A Case Study," in *Proceedings of the 11th National Computer Security Conference*, October 1988.

[32] S. Freeman, A. Bivens, J. Branch, B. Szymanski, "Host-Based Intrusion Detection Using User Signatures".

[33] Hochberg, et al., "NADIR: An Automated System for Detecting Network Intrusion and Misuse," *Computers & Security, Elsevier Science Publishers*, pp. 235-248, 1993.

[34] W. Richard Stevens, "Advanced Programming in the UNIX Environment," March 1996

[35] Allan Cruse. 2004. **The System Call Interface.** http://www.cs.usfca.edu/~crouse/cs326/lesson02.ppt.

[36] N. Nuansri, "Activity Tracing Techniques for Fault and Security Intrusion Detection in Computer Systems," A Thesis submitted in total fulfillment of the requirements for the degree of Doctor of Philosophy, Department of Computer Science and Computer Engineering, Faculty of Science, Technology and Engineering, Latrobe University, La Trobe University, Australia May 1999

[37] CERT Coordination Center, Carnegie Mellon University. 2003. **CERT/CC Overview Incident and Vulnerability Trends.** http://www.cert.org/present/cert-overview-trends/module-4.pdf/

ผลงานตีพิมพ์เผยแพร่จากวิทยานิพนธ์

Intrusion Detection System : System Call Tracing Technique

Pattanawadee Siwatintuko, Nittida Nuansri[1]
Computer System Design Lab, Dept. of Computer Engineering,
Faculty of Engineering, Prince of Songkla University,
Had Yai, Songkla 90112 Thailand
Email: spattana@unicorn.eng.psu.ac.th

Abstract

This paper presents a result of an intrusion detection system implemented using a system call tracing and a state transition analysis technique. The system consists of three modules, a process monitoring, a state transition analysis, and a detecting module. The process monitoring module monitors system call usage and user credentials of a process. The state transition technique takes its input from the monitoring module to analyze and define process state at a particular time. If the underlying process changes its state into a forbidden state, then it is flagged as a suspicious activity and the detection module is called. This technique yields a satisfactory result.

Keywords

Intrusion Detection System (IDS), System Calls, State Transition Analysis.

1. Introduction

Many intrusion detection reports have stated that Unix is one of the favorite target operating system for intruders because of its characteristics that there is a privileged user called *"root"* who has the control of all jobs and resources in a system. Thus, gaining the root privileged access is the main objective of most intruders. Literature survey also revealed that almost successful break-ins targeted to, and later used, a set of Unix special commands called *"setuid/setgid"* programs. These commands, as designed, are mostly executed with root privileged in order to perform their jobs. Thus if they are any flaws, either implementation or programming ones, they are vulnerable to being attacked.

According to the study of [1], *setuid/setgid* commands were investigated into more details and reported that most intrusive activities can be detected at the instance of the success attempt by monitoring the ownership changing states of any suspicious processes. From this study, a state transition analysis was used to model all possible states of Unix processes according to the changing of their user credential values. Supporting rules for intrusive activity detection were also proposed. Based on this study as well as our further investigations, we have designed and implemented an intrusion detection system using system call tracing method which is capable of detecting suspicious intrusive activities. Essential information related to the implementation of our IDS is also provided in the section 2. The details of this intrusion detecion system (IDS) is given in section 3. Then section 4 shows the result of this system followed by conclusion in section 5.

2. Related information used in detecting intrusive activities

In this section, we provide basic information used to detect intrusive activities on a Unix-based operating system. These are, user credentials and their characteristics in Unix processes which are described in section 2.1, and system call tracing technique used to trace processes activities is then given in section 2.2. Moreover, the state transition analysis model used to determine process states, especially the suspicious state, is given in section 2.3.

---

[1] Lecture: Dept of Computer Science, Faculty of Science, Prince of Songkla University

2.1 Identifiers and Credentials in Unix

In a Unix system, each user is identified by set of identifier numbers called a *"user-identifier"* or *"uid"* and a *"group-identifier"* or *"gid"*. These identifiers are represented by integer values, fixed for each user, obtained when a user is created and are used every time a user invokes or creates a process. In fact, the user and group identifiers consist of sub-identifiers called a *"real"* and an *"effective"* user/group ids. Each process associated with real user-id and real group-id for in order to know who invoked the process, the effective user-id and the effective group-id are used to determine whether the process can access a resource. Normally the effective user and group ids will be the same as the real user and group ids, except in the *setuid/setgid* command when the process run these commands the effective user-id will be set to the same value of the owner of the command. In *setgid* commands, the effective group-id will be set to the command owner group as well. Almost every *setuid/setgid* commands are *root setuid/setgid* that is while the process related to these commands are executing, they hold the privilege of *root* at one instance of time.

2.2 System call tracing analysis

Every Unix user process interfaces with the kernel in order to perform its tasks via system calls provided by the operating system [2]. In addition to this fact, process information such as process user identifiers, process identifiers, and many other information related to the current state of each process while executing is kept in a process table [3] which can be extracted (with the right permission) for investigation. Thus, if we can trace all system calls used by a process and obtain process information required to reveal the process activity at the time, we should be able to determine the process intention easily. In order to monitor this information we used the system call named *ktruss()*, more details in section 3, to trace process system calls accessing and process information which is used in the detecting section.

2.3 State Transition Analysis

Information obtained from the tracing method stated in the previous section is then used to study the various states of a process during its process life time. According to the study of [1], it is stated that processes can be categorized into several states while performing its jobs. These states is described by a 4-tuple instance which is *(real user ID, effective user ID, real group ID, effective group ID)* or *(UID, EUID, GID, EGID)*. Each of these components of the tuple are described below.

Real user ID and effective user ID

At a particular time, when a process is running., the value of *UID* and *EUID* can be denoted by one of the follwing values:
uid – (user's id) the user identifier number assigned by the system administrator during the user creation process to the user whose process is being traced.
sid – (special id) a user identifier number of high privileged user.
oid – (other's id) a user identifier number that does not to be into *uid* or *sid*.

Real group ID and effective group ID

Similar to the real and effective user ID, the value of *GID* and *EGID* can be one of the following values:
gid – (group's id) a normal group identifier number for the user who run the process.
sgid – (special group id) a group identifier number for special system or high privileged group.
ogid – (other group id) a group identifier number that does not to be into *gid* or *sgid*.

States

When the process running at a particular time, a process will be in one and only one state depending of the 4-tuple at that time. We differentiate four types of states: normal, *special privileged, superuser and system group, and another user.* To provide the definition of each state, *UID EUID GID and EGID* which are denoted by the meaning of *uid, suid, oid, gid, sgid and ogid* are used.

Definition 1: Normal State

A process will be in the normal state if and only if the process 4-tuple values are:

(uid, uid, gid, gid)

Definition 2: Special Privileged State

This special privileged state consists of several states corresponding to system call which cause the state transition. This includes

- **setuid**: a state induced by the *setuid()* system call. Its 4-tuple representation is
  (uid, sid, gid, gid).
- **setreuid**: a state induced by the *setreuid()* system call. Its 4-tuple value is
  (sid, uid, gid, gid).
- **setgid**: a state induced by the *setgid()* system call. Its 4-tuple value is
  (uid, uid, sgid, gid).
- **setregid**: a state induced by the *setregid()* system call. Its 4-tuple value is
  (uid, uid, gid, sgid).

Definition 3: Superuser and System group States

Superuser state

- A real user ID and an effective user ID are both of privileged user IDs : *(sid, sid, gid, gid)*

System Group State

- A group ID and an effective group ID are both to be in privileged group IDs: *(uid, uid, sgid, sgid)*

Definition 4: Another User State

A process is said to be in another user state when a process user ID or group ID attributes have their values changed in one or a combination of the following :

- A real user ID and an effective user ID are both changed to the other user ids: *(oid, oid, gid, gid)*
- A group ID and an effective group ID are both changed to other user group ids: *(uid, uid, ogid, ogid)*

Fig. 1 show some of important states and their transitions according to the definition above. The another user state is not shown.

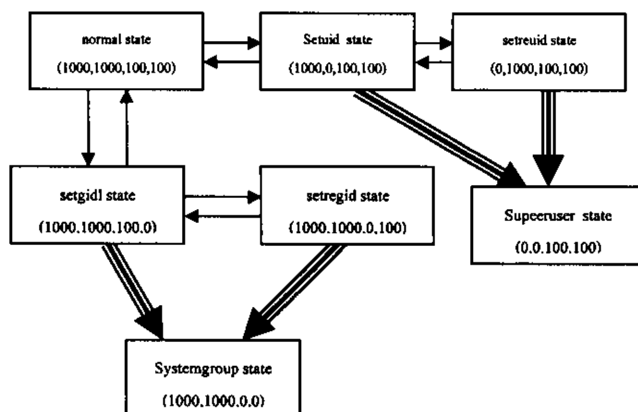

Figure 1. State transition diagram for some system states. The heavy arrows represent "forbidden" or intrusion activities.

From the Fig1. The intrusion activities are the process in *superuser* state and *systemgroup* state which transited from the *special privileged* state: *setuid, setreuid, setgid* and *setregid* state. These can be a system call, program, or other activities that cause the state transition. Next we will describe the supporting rules used to detect those activities.

3. Design and implementation details

The system consists of 3 modules as shown in figure 2. There are a process monitoring, a state transition analysis, and a detecting module.

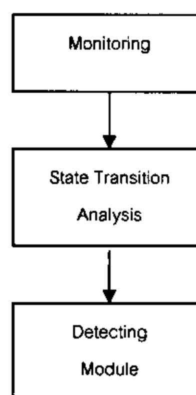

Figure 2. The system modules

The monitoring process monitors process activities by tracing all system calls used by that process. Then the selected process information such as user credentials, system call names, opened file names, are obtained

and sent to the state transition analysis which will analyze the process state all the time. If an underlying process has changed its state from normal states into a special privileged state, it will be closely monitor. Normal processes are allowed to switch to privileged process in order to complete its task, e.g. when it uses a *setuid/setgid* command, but it will always swap back to a normal state once it has finished the special privilege. However, if the process has gone further into a *superuser* state, which is forbidden for normal user processes, then the process is flagged as a suspicious state and is sent to the detecting module which investigates into more details of the process activities. This module makes a decision whether the suspicious process is an intrusive process and stops the process if it is confirmed to be an intrusion. In order to make a correct decision, the module uses all of the traced information, including resources, such as files opened by the process.

The implementation is based on a NetBSD operating system which provides two system calls called *ktrace()* and *ktruss()* capable of tracing process activities. These system calls provide information kept in a process table of every process including process credentials which are the essential information required for our detection technique. With a small modification of these system calls, all information necessary for process activity are obtained to be used as an input of the state transition analysis module. The *ktruss()*, a modified version of the original *ktrace()* supported by the NetBSD system was selected in the implementation. It is used to trace a target process and report process credentials and other information required for the analysis module. In order to determine suspicious activities for the detection module, several supporting rules detailed below were required.

Supporting Rules for Intrusion Detection and implementations.

Six rules have been stated and are used to support the state transition analysis described above. Each rule is used to detect different method of intrusive activities. From the study of [1] reported that most of intrusive activities can be grouped in to several groups and a particular rule set is applied to detect each of these activities. These rules and detection algorithms were implemented in our detection system. The proposed algorithms were tested against simulated activities according to each rule which yielded a positive result.

Rule 0: Only the special system calls *setreuid ()* and the *setregid()* are permitted to change the (real) UID or GID respectively.

If any system call or program other than these changes the real UID or GID of the process, it is flagged as an intrusive activity.

From Rule 0 we can describe that when the process in special privileged that is the *effective uid* or the *effective gid* belong to privileged user or privileged group which almost means *"root"* what we do is when we monitor the event that the real *uid* or real *gid* change to 0 (swap between *uid* and *euid* or *gid* and *egid*) the program will check the system call usage. If it makes change of process *uid* or *gid* an the system call is not the *setreuid()* nor the *setregid()* we flag that activity.

Rule 1: No *execve()* call is allowed in a *special privileged* state.

When a process in the *special privileged* state, it is not allowed to *execve* another program or command, whenever a privileged command requires to execute other commands, it must release the current privileges in the child process before executing a new program. If the new program also requires a system privileges to do its task, it can be as a *setuid* program. Implementation of this rule is done by monitoring a call of the system call *execve()* in the *special privileged* state of a process. That is if a process is in a *special privileged* state and tries to run another program by issuing the *execve()*, we then flag the process.

Rule 2: A process in a *special privileged* state is not allowed to create a *setuid/setgid* program.

It is not common that general users need to create a *setuid* program, we have to assume that users do know what they are doing, if they try to do this we detect that they're intruder.

For implementation this rule, firstly we check if the system call *open()* or system call *chmod ()* is called when a process is in a special privileged state. The mode parameter of these system calls are checked whether they are an *S_ISUID* or an *S_ISGID* (the S_ISUID flag is defined by 0400 while the S_ISGID flag is

0200). To test this rule, a simple program which tries to create a *setuid* file is created, then the intrusion detection programm is used to detect this activity.

Rule 3: A process is not allowed to modify system programs.

In special privileged state we not allowed the process modify the system program. Generally, all system commands, once installed, do not require modifications by any normal process. If it have to, general "root", the real user ID 0 will do. For implementation the detection first we check if target opened file is one of the the system files [4] then a flag value of the system call *open()* are checked. Those flags are :

O_RDONLY	*0x0000*
O_WRONLY	*0x0001*
O_RDWR	*0x0002*
O_APPEND	*0x0008*

The following text box demonstrates part of results and its test for this rule. We test its detection capability by feeding the IDS with a simple program which tries to create a file in one of the system directories (/usr/src/sys/kern/..).

```
system call open() is called
path_value in ktrsyscall /usr/lib/libc.so.12
arg[0] ----> 0x14  arg[1] -------> 0x8060000

/* following process try to open file in system
directory with flag write only so no
permission for this activity*/

system call open() is called
path_value in ktrsyscall
/usr/src/sys/kern/kern_exit.c
arg[0] ----> 0x1 arg [1]-----> 0x8049b84
-----------------------------------------
path value /usr/src/sys/kern/kern_exit.c
open with flag write only
No permission
```

Rule 4: For intrusion detection, we consider any process creating new user accounts as suspicious unless it has *superuser* credentials.

Again, only the *superuser* should be allowed to create new accounts.

Before implementing this rule, we had studied several system utilities and commands used to manipulate user accounts, in order not to break the normal system services, such as the ability for a user to change their own password. It has been observed that there is a difference between normal password changing process (by the legitimated user) and a process to create a new user. The significant difference is that the process of a user creation will not be completed unless a new entry of the password file is added. In addition, a user home directory is also created. However, the process of changing password does not require these entries.

Thus, the implementation of this rule also monitor on the occurrence of a new entry of the password file if the *open()* system call is used to open the system password file. Further more, we have to monitor whether the related process tries to create a home directory of that newly entry.

Rule 5: Some system call functions are strictly limited to *superuser (root)*.

These system calls are *mount(), umount(), nfssvc(), quotactl(), reboot(), settimeofday(), swapon()*.

Generally, novice users might want to try or learn how several commands or system calls work. For example, a user tries on a *mount (1)* command or a *reboot(1)* command. However, these commands are already protected by the system itself that in order to perform some of the system commands, only the super user (root) can invoke the commands. Thus, normal users can only try the commands (as well as the related system calls) without any significant effect. However, if a user illegally obtained a system privileged then invoked these commands or system calls, then the whole system is said to be in a unsafe state, thus we can flag the activity as a suspicious or forbidden one.

4. Results

As mentioned in each the implementation section, especially in each rule section, an appropriate *bad* intention programs were written to simulate varieties of activities known to be suspicious or attacking ones.

These simulated programs were then tested on the system running our intrusion detection modules. All of the suspicious simulated activities were detected by the IDS. However, the detection system has not been tested with a real intrusion programs yet, since it is not easy to find the right intrusive program as a test tool. An operating system, especially the current NetBSD version, has been patched for all known vulnerabilities, thus previous attacking programs exploiting them do not work any more.

However, since our method does not rely on *known attacking patterns*, rather it is based on the idea that attacking activities always aim for *root* privilege and the information of the underlying activity is obtainable while a process is running. Since we created a module to extract this information from a process as a real time program, we are able to detect the attacking instance just only after the instruction is executed and the activity can be stopped even before its next instruction is proceeded. Once the detection system is truly tested, it will be implemented as a daemon process on a real environment.

5. Conclusions

We have described another intrusion detection system implemented on a NetBSD version 1.6 N. It is based on a state transition analysis of system call tracing activities and six supporting rules to protect an underlying system and to detect intrusive activities. The IDS has been tested using simulated intrusive activities and provided a satisfactory result. The intrusion system is currently being fine tuning so that it can be installed as a Unix daemon process for intrusion detection purpose. It will also be ported to other Unix-liked operating systems such as a FreeBSD unix, or Linux systems in the near future.

References

[1] Nuansri, N., Singh, S., Dillon, T.S. "A Process State-Transition Analysis and its Application to Intrusion", Department of Computer Science and Computer Engineering, and Applied Computing Research Institue (ACRI), Latrobe University, Bundoora VIC 3083 Melbourne, Australia, 1999.

[2] Leffler, Samuel J., Mckvsich, Marshall kirk, Karels, Michael J., Quarterman, John S., "The Design and Implementation of The 4.3 BSD UNIX Operating System,", Addsion-Wesley 1988.

[3] Bach, Maurice J., "The Sesign of the UNIX Operating System", Prentice-Hall 1986.

[4] R. A. Kemmerer, "NSTAT: A Model-based Real-time Network Intrusion Detection System", Technical Report TRCS97-18, Department of Computer Science, University of California, Santa Barbara (1997), http://www.cs.uscsb.edu/.

[5] Nuansri, N. "Activity tracing Technique for Fault and Security Intrusion Detection in Computer Systems", Thesis, Department of Computer Science and Computer Engineering, Faculty of Science, Technology and Engineering, La Trobe University Bundoora, Victoria 3083 Australia, May 1999.

[6] , Keith Haviland and Dina Gray, "Unix System Programming, A programmer's guide to software development", Second editionAddison-Wesley 1998.

[7] S.A. Hofmeryr,