基于系统调用分类的异常检测*

徐 明+, 陈 纯, 应 晶

(浙江大学 计算机科学与工程学系,浙江 杭州 310027)

Anomaly Detection Based on System Call Classification

XU Ming+, CHEN Chun, YING Jing

(Department of Computer Science and Engineering, Zhejiang University, Hangzhou 310027, China)

+ Corresponding author: Phn: +86-571-87932435, E-mail: xuming_903@tom.com, http://www.cs.zju.edu.cn/

Received 2002-10-10; Accepted 2003-07-02

Xu M, Chen C, Ying J. Anomaly detection based on system call classification. *Journal of Software*, 2004,15(3): 391~403.

http://www.jos.org.cn/1000-9825/15/391.htm

Abstract: The aim of this paper is to create a new anomaly detection model based on rules. A detailed classification of the LINUX system calls according to their function and level of threat is presented. The detection model only aims at critical calls (i.e. the threat level 1 calls). In the learning process, the detection model dynamically processes every critical call, but does not use data mining or statistics from static data. Therefore, the increment learning could be implemented. Based on some simple predefined rules and refining, the number of rules in the rule database could be reduced dramatically, so that the rule match time can be reduced effectively during detection processing. The experimental results clearly demonstrate that the detection model can effectively detect R2L, R2R and L2R attacks. Moreover the detected anomaly is limited in the corresponding requests, but not in the entire trace. The detection model is fit for the privileged processes, especially for those based on request-responses.

Key words: intrusion detection; system call; anomaly detection

摘 要: 提出了一种新的基于规则的异常检测模型.把系统调用按照功能和危险程度进行了分类,该模型只是针对每类中关键调用(即危险级别为1的系统调用).在学习过程中,动态地处理每个关键调用,而不是对静态的数据进行数据挖掘或统计,从而可以实现增量学习.同时通过预定义,精炼规则,有效地减少了规则数据库中的规则数目,缩减了检测过程中规则的匹配时间.实验结果清楚地表明,检测模型可以有效侦测出R2L,R2R和L2R型攻击,而且检测出的异常行为将被限制在相应的请求内而不是整个系统调用迹.检测模型适合于针对特权进程(特别是基于请求--反应型的特权进程)的异常入侵检测.

关键词: 入侵检测;系统调用;异常检测

中图法分类号: TP393 **文献标识码**: A

---

* **XU Ming** was born in 1970. He is a Ph.D. candidate at the Department of Computer Science and Engineering, Zhejiang University. His research interests are intrusion detection, network security and cryptography. **CHEN Chun** was born in 1955. He is a professor and doctoral supervisor at the Department of Computer Science and Engineering, Zhejiang University. His research areas are CAD&CG, CAM, AI and CSCW. **YING Jing** was born in 1971. He is a professor and doctoral supervisor at the Department of Computer Science and Engineering, Zhejiang University. His research interests include software engineering and CASE.

Intrusion Detection Systems (IDSs) have become an important part of information security systems. There are two general methods of detecting intrusions: knowledge-based (misuse detection or pattern recognition approach in some literatures) and behavior-based (anomaly detection in some literatures)[1]. Misuse detection techniques recognize the signatures of known attacks, match the observed behavior with those known signatures, and signal intrusions when there is a match. Misuse detection techniques are efficient and accurate in detecting known intrusions, but cannot detect novel intrusions whose signature patterns are unknown. Anomaly detection techniques establish a profile of the subject's normal behavior (norm profile), compare the observed behavior of the subject with its norm profile, and signal intrusions when the subject's observed behavior differs significantly from its norm profile. Anomaly detection techniques can detect both novel and known attacks. Since anomaly detection techniques signal all anomalies as intrusions, false alarms are expected when anomalies are caused by behavioral irregularities instead of intrusions[2].

System call trace is a common type of audit used for performing intrusion detection. A system call trace is an ordered sequence of system calls that a process performs during execution. System call traces of the privileged processes are useful for detecting R2L (Remote to Local), R2R (Remote to Root) and L2R (local to Root) exploits. Privileged processes are programs that perform services (such as ftp, http and mail), which require access to system resources that are inaccessible to the ordinary user. To enable these processes to perform their jobs, they are given privileges over and above those of an ordinary user (even though they can be invoked by ordinary users). In UNIX, processes usually run with the privileges of the user that invoked them. However, the privileged processes can change their privileges to that of the root by means of the setuid mechanism. One of the security problems with the privileged processes in UNIX is that the granularity of permissions is too coarse: Privileged processes need root status to access system resources, but granting them such a status gives them more permissions than necessary to perform their specific tasks [3]. Typically, intruder exploits a bug or a configure error in a privileged process by using a buffer overflow or race conditions etc. to create a root shell. Under these attacks, the call trace of an exploited program process is drastically different from that of the program process under normal conditions. The main difference is when changing user identity (from remote user to local user, remote user to root, and local user to root), files, executing program, and network etc. Examining these information from call traces can detect attacks when a process is being exploited. This paper aims to create a new anomaly detection model to detect R2L, R2R and L2R attacks, for the privileged programs, especially for these based on request-responses. The model can effectively detect attacks under lower false alarm by examining critical calls. The normal profile of the protected program is a rule database. The rule database can be automatically learned from normal system call traces under some simply predefined rules; the rule database can also be refined and implemented by an increment learning.

The structure of this paper is as follows. In Section 1, we classify the system calls according to their function and level of threat. We present a complete classification of the LINUX system calls. Section 2 describes how to automatically build up the profile of normal program behavior under simple predefined rules and how to detect anomaly. In Section 3, we present a prototype of the detection model and some experiment results. Discussion is given in Section 4. In Section 5, we review the related approaches presented in literatures. Section 6 concludes the paper and discusses future activities.

1 Call Classification

System call is an event, which happens at the user-kernel interface. A close examination of this boundary is very useful for isolating bugs, checking sanities, capturing race conditions and detecting overflow attacks. The call classification presented in this section identifies the system calls that may jeopardize the system security. The system calls in LINUX 2.4 have been classified in categories according to their functionality and threat level as

reported in Table 1. The calls are classified as six categories by their functionality, i.e. file system, process, network, module, signal and other. Each call category is further classified into four groups according to their threat level, i.e. 1(Allows full control of the system), 2(Used for a denial of service attack), 3(Used for subverting the invoking process), and 4(Harmless). By convention, less dangerous system calls are assigned a larger threat level number. This classification corresponds to a threat hierarchy, since a system call classified at threat level n may also be employed to carry on an attack at threat level m if $m \geq n$ [4]. In fact, the network category call is only one system "socketcall". The "socketcall" is expanded to 17 socket calls by its sub-call. Therefore, there are 241 calls (225+17−1) in total in Table 1. Threat level 1 calls are named as critical calls.

Table 1 System call categories

Call group	Threat level	System calls	Number of calls
File system	1	chmod, chown, chown32, fchmod, fchown, fchown32, lchown, lchown32, link, mknod, mount, open, rename, symlink, unlink	15
	2	close, creat, dup2, flock, ftruncate, ftruncate64, ioctl, mkdir, nfsservctl, quotactl, rmdir, truncate, truncate64, umount, umount2	15
	3	chdir, chroot, dup, fchdir, fcntl, fcntl64, fsync, llseek, lseek, newselect, poll, pread, putpmsg, pwrite, read, readv, select, sendfile, umask, utime, afs_syscall, write, writev	23
	4	access, bdflush, fdatasync, fstat, fstat64, fstatfs, getcwd, getdents, getdents64, getpmsg, lstat, lstat64, oldfstat, oldlstat, oldolduname, oldstat, olduname, pipe, readahead, readdir, readlink, stat, stat64, statfs, sync, sysfs, ustat	27
Process	1	execve, setfsgid, setfsgid32, setfsuid, setfsuid32, setgid, setgid32, setgroups, setgroups32, setregid, setregid32, setresgid, setresgid32, setresuid, setresuid32, setreuid, setreuid32, setuid, setuid32	19
	2	vfork, adjtimex, brk, clone, exit, fork, ioperm, iopl, kill, modifyldt, nice, ptrace, reboot, sched_setparam, sched_setscheduler, sched_yield, setpriority, setrlimit, vhangup, vm86, vm86old	21
	3	capset, personality, prctl, setpgid, setsid, uselib, wait4, waitpid	8
	4	acct, capget, getegid, getegid32, geteuid, geteuid32, getgid, getgid32, getgroups, getgroups32, getpgid, getpgrp, getpid, getppid, getpriority, getresgid, getresgid32, getresuid, getresuid32, getrlimit, getrusage, getsid, getuid, getuid32, sched_get_priority_max, sched_get_priority_min, sched_getparam, sched_getscheduler, sched_rr_get_interval	29
Network	1	accept, bind, connect, listen, socket, socketpair	6
	2	recv, recvfrom, recvmsg, sedmsg, send, sendto, setsocketopt, shutdown	8
	4	getpeername, getsocketname, getsocketopt	3
Module	1	init_module, create_module	2
	2	deletemodule	1
	4	get_kernel_syms, query_module	2
Signal	2	rt_sigaction, rt_sigpending, rt_sigprocmask, rt_sigqueueinfo, rt_sigreturn, rt_sigsuspend, rt_sigtimedwait, sgetmask, sigaction, sigaltstack, signal, sigpending, sigreturn, sigsuspend, sigprocmask, ssetmask	16
Other	2	alarm, madvise, madvise1, mlock, mlockall, pivot_root, setdomainname, sethostname, setitimer, settimeofday, stime, swapoff, swapon, sysctl, syslog, ugetrlimit	16
	3	mincore, mmap, mmap2, modify_ldt, mprotect, mremap, munlock, munlockall, munmap, nanosleep, security	11
	4	break, ftime, getitimer, gettid, gettimeofday, gtty, idle, lock, mpx, msync, pause, prof, profil, stty, sysi, time, times, ulimit, uname	19

File system is particularly important for security because it provides a simple and consistent interface to operating system services and to devices. The critical calls of the file system category can change the file content, file attribute or file system attribute. The calls, which can change file content, file attribute or file system attribute and take file names as parameters, are all in threat level 1. In accordance with Goldberg *et al*.[5], the operations of read or write to file are only necessary to monitor the open calls. Because these operations need a file descriptor returned by open, only open in these calls, which can change the file content and take the file descriptors as parameters, is in threat level 1. However, these calls which can change file attribute are still being classified into threat level 1, although they use file descriptor but file name as parameters, because the open call can't limit them to change file attribute.

In the process management category, there are 19 calls which reach the highest level of danger for system

security. The execve can be used to start a root shell or other applications. The other 18 critical calls can be used to change user or group identifiers.

In the network category, there are 6 calls in threat level 1. Socket and socketpair initialize the socket and prepare to communicate. Accept, bind and listen are used to create socket server. Connect is used for client to initiate a connection on a socket. Only by these 6 calls can the socket really read or write.

As for the system calls in the module Group, a subverted process may use them for loading a malicious module. Init_module and create_module are regarded as reaching threat level 1 since no module can be activated without invoking the init_module or create_module first.

In the signal and other groups, no calls are in threat level 1 because they do not relate to the pivotal security resource.

2 Detection Model

The simplest detection anomaly method is to monitor all of the calls. Unfortunately, this monitoring is expensive, because it is constantly regardless of the functionality and threat level of the system calls. Furthermore, it may bring about false alarm. The basic idea of our detection model is to examine the critical calls to detect anomaly by observing the deviations from the normal profile. Only a critical call can gain the immediate and full control of the target system or privileged system resource. In other words, executing the unprotected system calls cannot really compromise and penetrate the system. Each call group is to create a rule set, but two rule sets to the process group: execve rule set and user rule set. All of the rule sets make up of the rule database, i.e. normal profile. The anomaly detection method is to examine whether the rule generated from the current critical call of the protected application matches the rule database, and to signal anomaly if mismatches appear. The rule generating method and the match function are introduced in Section 2.2.

2.1 Audit data

The audit data are system call traces generated by the protected application. System call is an event which happens at the user-kernel interface. A close examination of this boundary is very useful to isolate bugs, check sanities, capture race conditions, and detect overflow attacks.

2.2 Profiling normal behaviors

The process of profiling normal behaviors to the protected application is described in Fig.1(a). The normal profiles (rule database) are composed of file rule set, execve rule set, user rule set, network rule set, and module rule set. These rule sets will be learned from the normal system call traces of the protected application. Every critical call will be used to generate a rule.

2.2.1 Normal rule database

The form of rule in a normal rule database is $h->\{t_1,t_2,...,t_n\}$, where h is the rule head and $\{t_1,t_2,...,t_n\}$ is the rule tail. There are two operations between rules $r=h->t$ and rule set R.

(1) Unite:

$$unite(r,R)=\begin{cases}\{r\}\cup R, & \neg\exists r'\in R, r'=h'->t' \text{ and } h=\sim h'\\ \{h'->t\cup t'\}\cup(R-\{r'\}), & \exists r'\in R, r'=h'->t', h=\sim h' \text{ and } t'\neq\{*\}\\ R, & \exists r'\in R, r'=h'->t', h=\sim h' \text{ and } t'=\{*\}\end{cases}.$$

(2) Match:

$$match(r,R)=\begin{cases}\text{true}, & \exists r'\in R, r'=h'->t', t\subseteq t' \text{ and } h=\sim h'\\ \text{false}, & \text{other}\end{cases},$$

where function "=~" denotes the string match with "*" which is considered as an arbitrary string. The unite function can be used to add a new rule into the rule set. It can also be used to implement an increment learning. This is not easy to implement in other statistics-based methods. The match function can be used to judge whether the rule r is normal based on normal rule set R.

2.2.2 Predefining rules

In order to quicken the process of learning and reduce the number of rules in the normal rule database, some rules for file system need to be predefined by hand, based on the application specification and current configure file. The predefined rules are often simple and obvious. The form of the predefined rule is (uid, gid, request, file or directory)–>{operation}, such as (504,504,*, /home/tom/*)–>{O_RDONLY}, which means the user id 504 and group id 504 can read any file in the directory /home/tom/ and its subdirectorys at any request. Of course, even without the predefined rules, all rules can also be learned from the system call traces, but the learning time will be prolonged because more call traces need to be leaned in order to get a robust rules database. Without predefined rules, moreover, the total number of rules in the normal rule database will be larger than that with the predefined rules.

2.2.3 Auto learning rules

First, the current user information and request name must be parsed. If the current call is to set a user or a group, then the user information must be parsed; if the current call includes a request from the user, a request name must be parsed. The user information includes ruid, euid, suid, fsuid, rgid, egid, sgid, and fsgid. Here ruid is the real user ID; euid is the effective user ID; suid is the saved user ID; fsuid is the user ID that the kernel uses to check for all accesses to the file system; rgid is the real group ID of the current process; egid is the effective group ID; sgid is the saved group ID; and fsgid is the group ID that the kernel uses to check for all accesses to the file system. The current user information must be parsed trustily from calls of the set user or group. The request can be the request-response application's supporting interactive requests. It also can be the command line parameters of the simple application which doesn't support the interactive request. The key is that each request provides an independent function. The request determines the application, the current function, and the call sequences.

Second, if the current call is a critical call, the call is to be used to learn a rule and then unite this rule into its group rule set.

(1) When critical call belongs to the file group

- Generating rule: If the critical call is "open(file name, flags,…)", the rule is *r*=(fsuid, fsgid, request name, file name) –>{flags}. If the critical call is not open, the rule is *r*=(fsuid, fsgid, request name, file name) –>{call status}.

$$call\ status=\begin{cases}call\ name+\text{'}-1\text{'}, & \text{call return value}=-1\\ call\ name, & \text{other}\end{cases}$$

where, holding a current opened file descriptor table is necessary because some critical calls take the file descriptor rather than the file name as parameter. Holding a current work directory variable is also necessary because the path of the file name may use a relatively path. This can use a "chdir" or "chroot" call to get the current work directory. Using the current work directory variable changes the relative path to the absolute path. For example, a file rule is (504, 504, RETR,/home/ftp/readme.txt) –> {O_RDONLY, O_WRONLY}.

- Uniting rule r into file rule set FS: $FS = unite(r, FS)$.

(2) When critical call belongs to the process group

- If the call is "execve(filename,…)=result", the generating rule is r=(euid, eqid, file name, other parameters) –> {result}, then unite the rule r into execve rule set ES: $ES = unite(r, ES)$. For example, an execve rule is $(0, 0, \text{START}, \text{/usr/sbin/in.ftpd}, \text{"–l,–a"}) -> \{0\}$.
- If the call is to set a user or a group and the call result is successful, the generating rule is

$$r = (\text{old\_ruid}, \text{old\_euid}, \text{old\_suid}, \text{old\_fsuid}, \text{oldrgid}, \text{oldegid}, \text{oldsgid}, \text{old\_fsgid}, \text{request name}) -> (\text{new\_ruid}, \text{new\_euid}, \text{new\_suid}, \text{new\_fsuid}, \text{new\_rgid}, \text{new\_egid}, \text{new\_sgid}, \text{new\_fsgid})$$

Then unite the rule r into the user rule set SS: $SS = unite(r, SS)$. For example, a possible rule is

$(0,0,0,0,0,0,0,0, \text{START}) -> \{(4294967295,0,4294967295,0,0,0,0,0), (4294967295, 504, 4294967295, 504, 0,0,0,0)\}$

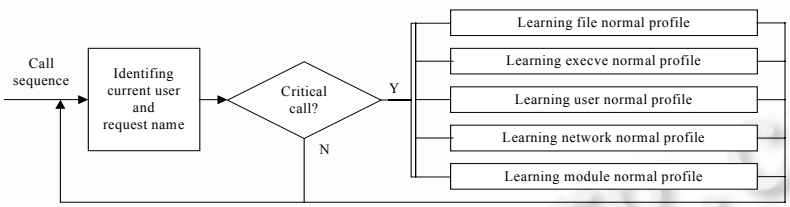
(3) When critical call belongs to the network group

- Generating rule: if the network family of the call is "PF_INET", the learning rule is $r = (\text{euid}, \text{egid}, \text{request name}) -> \{\text{call status}\}$. The call status is the same as above. If the network family of the call is "PF_UNIX"(i.e. the socket operation is to file), the learning rule is $\text{l r} = (\text{euid}, \text{egid}, \text{rquest name}, \text{file name}) -> \{\text{callStatus}\}$. Unite the rule r into the network rule set NS: $NS = unite(r, NS)$. Two possible network rules are

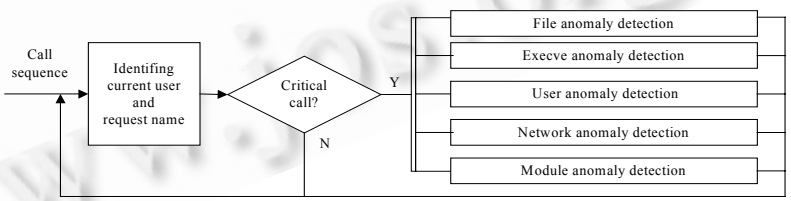
$(0,0, \text{LIST}) -> \{\text{socket}, \text{bind}\}$ and $(0,0, \text{START}, \text{/var/run/.nscd\_socket}) -> \{\text{socket}, \text{connect} -1\}$.

(4) When critical call belongs to the module group

- Generating rule: the rule is $r = (\text{euid}, \text{egid}, \text{request name}, \text{call name}, \text{module name}) -> \{\text{call result}\}$.
- Uniting rule r into module rules set MS: $MS = unite(r, MS)$. A possible module rule is: $(0,0, \text{START}, \text{create\_module}, \text{msdos}) -> \{0\}$.



(a) Learning the normal profile of the application from system call traces



(b) Detecting normaly from system call traces

Fig.1 The learning of the normal profiles and detection of the anomaly processes

2.2.4 Refining rule database

After automatically learning the rule sets, the rule database msut be refined in order to reduce the size of the rule database and make the rule database more generalizable. Generalization of the rules can enhance the robustness of the rule database. The refining of the rule uses the key words euid, egid and the request name in the rule head to implement.

Let the rule set be R; the form of the rule in R be $(h_1, h_2, ..., h_i, ..., h_n) -> t$; the key word of the refining be h_i; $H_i = \{h_i \mid r = (h_1, h_2, ..., h_i, ..., h_n) -> t, \forall r \in R$; after refining, the new rule set is R'.

(1) I=0; $R'=\varnothing$;

(2) Get a rule $(h_1,h_2,...,h_{i-1},h_i,h_{i+1},...,h_n)->t$ from R. $H'_i=\{h_i\}$;I++;

(3) Examine every rule $(h'_1,h'_2,...,h'_{i-1},h'_i,h'_{i+1},...,h'_n)->t'$ in R. If $(h_1,h_2,...,h_{i-1})=(h'_1,h'_2,...,h'_{i-1})$, $(h_{i+1},h_{i+2},...,h_n)=(h'_{i+1},h'_{i+2},...,h'_n)$ and $t\subseteq t'$, then $H'_i=H'\cup\{h'_i\}$;

(4) If $H'_i=H_i$, then $R'=R'\cup(h_1,h_2,...,h_{i-1},*,h_{i+1},...,h_n)->t$, else $R'=R'\cup(h_1,h_2,...,h_{i-1},h_i,h_{i+1},...,h_n)->t$;

(5) If I<|R|, then Goto (2), else end.

Clearly, the computation complexity of the refining is $O(|R|^2)$. The essential idea of the refining is an attempt to add an arbitrary match string "*" into the rule head. Without refining, of course, the rule set can also be used to detect anomaly. Refining, however, can reduce the size of the rule database and the rule match time during detection.

2.3 Detection of anomaly

The detection process is similar to the learning process (Fig.1(b)). The detection method is to examine whether the rule generated by the current critical call matches to its group rule set (i.e. its group normal profile). A match means this call behavior is normal, while a mismatch is regarded as an anomaly.

3 Prototype and Experimental Result

On Red Hat LINUX 7.2, by using PERL a prototype is implemented. We select LINUX because its source code is freely available under the GNU General Public License. The kernel version is 2.4.16. The wu-ftp is the privileged application used in our experiment. We use wu-ftp2.6.0 as anomaly detection application because it is a widely deployed software package to provide File Transfer Protocol (FTP) services on UNIX and LINUX systems, and exists many intrusion scripts on the Internet.

In this experiment, the wu-ftp has 4 users: root (uid=0, gid=0), anonymous/ftp (uid=14, gid=50), user (uid=504, gid=0) and abc (uid=505, gid=505). In our prototype, the wu-ftp is managed by the inetd daemon. Wu-ftpd2.6.0 has 47 directional support requests, but in which 10 requests are not implemented. START and OTHER requests are added into the request set. The START denotes the ftpd start process, and OTHER denotes the unidentifiable request. The request can simply use "read (0, request name,...)" to identify. But in the living traces, one request can be detached from multi "read (0,…)" call. For example, the FTPD request "PWD" can be: read (0, "P", 1)=1, read (0, "W", 1)=1, read (0, "D", 1)=1,read (0, "\n", 1)=1. On this condition, parsing request name needs to join multi calls. The predefined rules define that a normal ftp user can read and write any file in his home directory and anonymous can read any file in /home/ftp/public). The predefined rules are: (1) (504, 0, *, /home /user/*)->{*} ; (2) (14, 50, *, /home/ftp/public/*)->{O_RDONLY}; (3) (505,505, *, /home/abc/*) ->{*} .

3.1 Trace data

We use command "strace" ("strace –p pid –f –o output.file") to get the trace data. All of the training data are normal traces, while the testing data include both the normal traces and intrusion traces. The data used in this study are described in table 2. The normal traces data come from a living wu-ftp daemon, with being carefully checked by Snort[6] and experts to keep no anomaly activity in the normal training traces. The test normal traces come from the living traces in two days. The intrusion traces come from the intrusion scripts on Internet. On the intrusion trace data, we have two wu-ftpd SITE EXEC vulnerability traces created from two different intrusion scripts (CERT Advisory CA-2000-13 http://www.cert.org/advisories/CA-2000-13.html): one trace of the wu-ftpd file name glob heap corruption vulnerability (CERT Advisory CA-2001-33 http://www.cert.org/advisories/CA-2001-33.html), and the other trace of the Passive ftp Vulnerability (http://www.checkpoint.com/techsupport/alerts/pasvftp.html). The two site exec scripts can triumphantly intrude systems and get a root shell, but the other two scripts cannot

triumphantly intrude systems. In order to attain a real live test effect, the intrusion traces are crude trace data. So the intrusion test data may include normal call sequence data.

Table 2 The training and test trace data

	Number of traces	Number of calls	Number of requests
Training traces	32	727 393	17 787
Test traces (normal trace)	59	842 295	13 084
Site exec 1 trace (intrusion trace)	1	3 756	220
Site exec 2 trace (intrusion trace)	1	3 164	132
Globing trace (intrusion trace)	1	1 916	11
Passive (intrusion trace)	1	3 010	60

3.2 Building robust rule sets

The rule database is described as robust if the rule database includes a sufficient portion of the legitimate rules so that the false positive rate remains tolerable[7]. Figure 2 shows the number of the changed and added rules for the rule database, with the chronological increasing of number of the learned calls for the wu-ftp sever under the predefined rules. To build the rule database, the wu-ftp is exercised extensively, i.e. in a carefully controlled way in order to exercise as much of code as possible. Rules are added into the rule sets or changed as encountered. Gradually, the number of the new added and changed rules drops off. Defining the rules database is sufficiently robust when the slope of growth flattens. The graph shows that a wide variety of rules for normal behavior are seen in the early part of the traces (about 200 000 system calls). After that, actually no changed or added rule is encountered under normal wu-ftp conditions. However, the curve is growth not smooth. This denotes that the normal rule database generated by automatic learning may not include all of the normal rules.

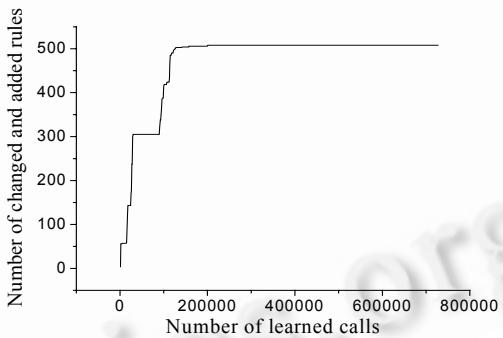


Fig.2 Growth curve of number of the changed and added rules

The number of rules in the rule database is described in Table 3. From it, the three simple predefined rules are effective to reduce the number of rules in the file rule set. The refining process can also effectively reduce the number of rules in the file rule set, execve rule set and network rule set.

Table 3 Number of rules in normal rule database

Number of rules	File rule set	Execve rule set	User rule set	Network rule set	Module rule set	Total
Without predefined rules	6 223	10	35	69	0	6 250
With predefined rules	344	10	35	69	0	372
After refining	236	6	35	55	0	252

3.3 Detection of performance

In IDS, there are two types of errors: false positives and false negatives. These errors are defined as: a false positive occurs when a trace generated by legitimate behavior is classified as anomalous; and a false negative

occurs when a trace generated by an intrusion is classified as normal. How do we classify a trace as anomalous or normal? Simply, if at least one anomalous call (i.e. generating rule by a critical call in trace does not match to its rule set) is found in a trace, this trace will be regarded as an anomalous trace; and if none of the call anomalies is found, this trace will be regarded as a normal trace. This method is simple and easy to implement. But in fact, the normal rule database cannot include all of the normal rules. If only one normal rule is not in the normal rule database, a normal trace, in which a call can generate this rule, will be classified as an anomalous trace (i.e. false positive). Although we would like to minimize both kinds of errors, we are more willing to tolerate false negatives than false positives[8].

To limit false positives, first, the local strength of the anomaly in trace t is measured by

$$Ar_i(t_i) = \text{number of the call anomalies in } t(i) \text{ / total number of critical calls in } t(i)$$

where $t(i)$ denotes the i^{th} request segment in trace t; the request segment is a segment of trace t in which a request is complete. Using the changing of the request name in trace t can identify request segment, i.e. $t(1)$ is the section from the beginning to the first call recorder, in which the request name is changed, and $t(2)$ is the section from the last call recorder of $t(1)$ to the first call recorder, in which the request name is changed etc. Therefore, the trace t can be regarded as $t(1),t(2),\ldots,t(n)$ based on the changing of the request names. Defining the local strength of the anomaly is due to the fact that the real anomaly is often at local and each request provides an independent and simple function. Figure 3 describes the local anomaly strength of normal test traces and 4 intrusion traces. From Fig.3, we see the anomaly is local and outburst because there are many normal operations even in the intrusion traces. The figure also shows there are obvious differences at the maximum of local anomaly strength between normal traces and intrusion traces.

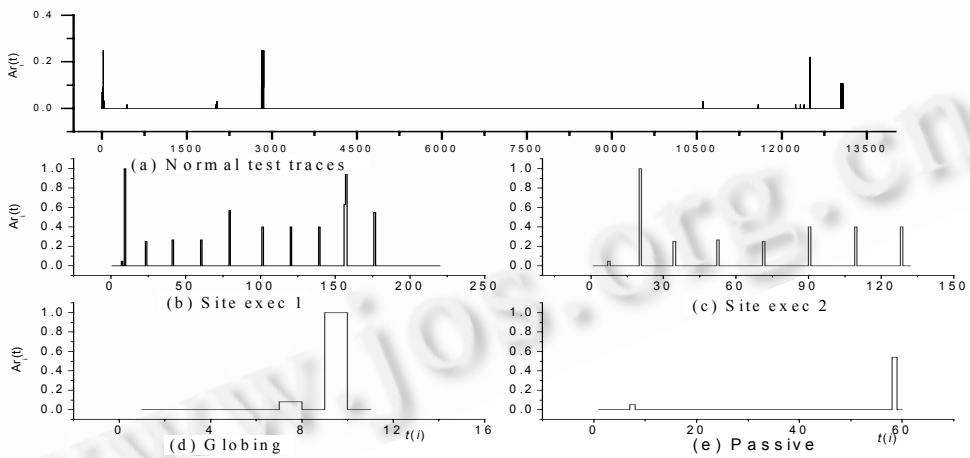


Fig.3 The local anomaly strength of traces

Second, define the strength signal of the anomaly of trace t as $Sa(t) = \max\{Ar_i(t), \forall t(i) \in t\}$. The higher the $Sa(t)$, the more likely the trace t is an intrusion trace. In practice, computing the current $Sa(t)$ for the current detected calls of trace t can detect anomaly online. Trace t is regarded as anomalous if the current $Sa(t) > C$, where C is the predefined threshold of anomaly. Table 4 describes the anomaly signal strength of 4 intrusion traces and request names in which the local anomaly strength achieves its maximum. From Table 4, we observe the $Sa(t)$ of 4 intrusion traces at least achieve 0.58, and the normal test trace's only achieves 0.25. The normal and anomalous is easy to distinguish. In our example, as long as the threshold C holds within $0.25<C<0.58$, the normal and anomalous can be exactly distinguished. Moreover, all requests in which the local anomaly strength achieves its maximum are

the key requests in their intrusion activities respectively.

Table 4 The anomaly signal strength and corresponding request name

	Site exec 1	Site exec 2	Globing	Passive
Sa(*t*)	1	1	1	0.58
Request name	SITE	SITE	CWD	PASV

3.4 Increment learning

We add a new user "newuser" for wu-ftp to test the increment learning based on the former rule database. With the increasing of number of the new learned calls from the new user's traces, the number of new added and changed rules is described in Fig. 4. Figure 4 shows that almost all of the changing and adding are seen in the early part of the new learned traces (about 18000 system calls). After that, no changing or adding is encountered and the cure becomes flat under normal trace with user the "newuser". After the increment learning and being refined, the number of rules in the database is described in Table 5.

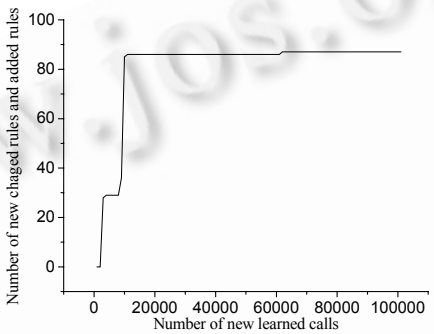


Fig.4 Increment growth curve of the number of new changed and added rules

Table 5 The number of rules in rule database after increment learning

Number of	File rules	Execve rules	User rules	Network rules	Module rules	Total
Former	236	6	35	55	0	252
After increment learning	249	6	40	68	0	283

4 Discussion

4.1 Impact on application performance

In our model, the learning and detection processes mainly use the critical calls (about 10% ~ 20% of all calls) in traces. Furthermore, learning require new rules to appear only once. Unlike other statistic models[2], data mining models[9] and neural network models[10], these models often need the features to appear time after time. The needed number of the learned calls is less than that of the models. The main performance loss in the current prototype comes from the "strace" command. An amelioration way could be the integration of the "strace" function into the operation system kernel. For LINUX, this is a feasible way.

4.2 Extensibilities

Following the three steps described below, the model can be easily extended to detect anomaly for other applications, especially for the privileged daemon processes based on the request-responses such as SENDMAIL, HTTPD, POPD, LPD and NAMED etc.

(1) Get the application request set. For most daemon applications, this is easy. The key of the requests is to provide

an independent function. For the popular HTTPD, the number of direct support request set is too small ("OPTIONS", "GET", "HEAD", "POST", "PUT", "DELETE", "TRACE", and extension-method), but the function is very complex. The request name can be reconstructed by:

$$request\ name=\begin{cases} request, & \text{to *.htm or *.html} \\ request+file\ name+file\ extention\ name, & \text{to CGI,*.jsp,*.php, or *.asp etc.} \\ request+file\ extention\ name, & \text{other} \end{cases}$$

(2) Find the way to identify request from the living system call trace. Usually the inclusion request system call record can use "read (fd,…)" to identify request, where the fd may be 0 (keyboard), file descriptor, or socket.

(3) Predefine the predefined rules based on application specification and current configure file. For example, we can predefine the user as that nobody can read all files at directory /home/http/html/ and its subdirectories for HTTPD server.

4.3 Mass user application

Some applications, such as POPD and SMTP, may have mass users. Mass users can lead to mass rules in a normal rule database. On this occasion, only using gid in rule head but not using uid can effectively reduce the number of rules.

4.4 Shortage

Monitoring critical calls can make it more difficult for intruders to avoid being detected. In order to successfully intrude and go undetected, intruders would have to use system calls which are not critical calls, or critical calls but the operations must match the normal rule database. Our model is not a complete intrusion detection system, because our model can efficiently detect R2L, R2R and L2R attacks but cannot detect all other kinds of intrusions such as SCAN and DoS (Denial of Service) attacks. Although using no critical calls cannot really compromise and penetrate a system, intruders can implement SCAN or DoS attacks and go undetected under our detection model. Therefore, our model needs to be used with other IDS such as NIDS (Network IDS) to create a multi-layer defense system.

Because some rules include user information and file operation, the rule database could not cover all of the normal rules. The anomalies appear in a normal trace just for it. Luckily, in a detection model, the local signal strength of anomaly instead of anomaly appearance is used to judge whether a trace is anomalous or not. If the rule database is sufficiently robust (i.e. the rules cover almost all of the normal rules), this method can effectively reduce the false alarms.

5 Related Work

Restricting program behavior based on externally specified rules has a very long history dating back to the reference monitors of operating systems several decades ago. This section highlights more recent mechanisms and compares them with our work.

Some groups in the recent past have proposed methods for addressing security issues by means of system call. In 1996, Forrest and others first introduced a simple intrusion detection method based on monitoring the system calls of the privileged processes[8,11]. Her work shows that the process normal behavior could be characterized by local patterns (fixed length call sequences) in its traces, and deviations from these patterns could be used to identify security violations of an executing process. Over the past several years, many statistical learning techniques based on system call have been developed. Several such methods have the potential for generating more accurate and (or) more compact models based on the system call trace data[7,8,12,13]. All these methods examine all calls in trace even

if the examined call does not relate to security at all, and only use call name but ignore call parameters and result. Examining every call will lower the IDS efficiency and can lead to an easily generated false alarm. Moreover call parameters and result can provide much useful information for IDS, such as user identity changing and file name etc. Therefore in our detection model, we examine critical calls instead of all in trace and use their call names, important call parameters and call results to detection attack. This can contribute to the more effective and accurate detection of intrusion.

Other groups aim to use interception and control call to enhance system security. In Goldberg *et al*.[5], a user-level tracing mechanism to restrict the execution environment of the untrusted helper applications was described. Our solution is based on a similar analysis of the potential problems associated with a subset of the system calls, but we control a different set of programs (i.e. privileged programs instead of helper applications). In Sekar *et al*.[14,15], a high-level specification language called Auditing Specification Language was introduced for specifying normal and abnormal behaviors of processes as logical assertions on the sequence of system calls and system calls argument values invoked by the processes. Although very elegant, this approach is less flexible than ours. Our starting point is an automatical rule learning instead of a manual rule generating by expert. In RESUM [4], the model is similar to ours. They also analyses the potential problems associated with a subset of system calls, but the classification result of calls has a little difference. Furthermore, their main aim is to give a loadable kernel module and a patched kernel to intercept and control call, but ours is to automatically implement a generation rule and to detect anomaly. The BlueBox[16] is very similar to RESUM and our model. BlueBox creates an infrastructure for defining and enforcing the very fine-grained process capabilities in the kernel. But its rule must be generated by hand.

The LINUX Intrusion Detection system (LIDS)[17] aims to extend the concept of capabilities present in the basic LINUX system by defining fine-grained file access capabilities for each process. Our rules for file system objects are very similar to this. But the LIDS is only to protect file system. The Domain-and-Type-Enforcement (DTE), based on system by Walker etc [18], groups file system objects into sets called types and puts a subject (an executable) into a domain which has specific access rights to types. It does not provide protection on non–file–system–object resources and incur more complexity when providing fine–granularity control than ours.

Compared our model with the above methods, the merits are listed below. Firstly, the learning and detecting of our model is more effective than the above methods which need leaning, because in our model learning and detecting only dynamically processes the critical calls but not all of the calls; furthermore, learning requires new rules to appear only once, but in other statistic models[2], data mining models[9] and neural network models[10], the features often need to appear time after time. Secondly, our model can automatically learn normal profile, but some of the above methods cannot. Thirdly, the size of rule database is smaller than that of the above methods because our model can use the predefined rules and refining to effectively reduce the number of rules, so that the rule match time is less during detection. Fourthly, the learning is automatic, but some of the above methods must create rule by hand. Finally, our model protects file system, network, process and user, but some of the above methods only protect file system.

6 Conclusions

In this paper, how to create a new anomaly detection model based on rules for the privileged Programs, especially for these based on request-response, has been described. The model is supported by a detailed system call classification, which identifies the critical calls that need to be monitored. Based on the classifications, a method for automatically generating, refining, and learning the rule database under some simple predefined rules is provided. The size of the generated rule database is small and the rule is easy to understand. The experimental results clearly

demonstrate that the detection model can effectively detect R2L, R2R and L2R attacks. Moreover the detected anomaly will be limited in the corresponding requests, but not in an entire trace.

Future work involves the use of more trace data and the extention of prototype to other server applications to test our model. We also plan to implement our model in a system kernel so that we can use the detected results to directly stop the hostile calls before the intruder penetrates into the system. We also plan to extend our model to detect other kinds of attacks, such as SCAN and DoS attacks.

References:

[1] Debar H, Dacier M, Wespi A. Toward a taxonomy of intrusion-detection systems. Computer Networks, 1999,31(8):805~822.

[2] Ye N, Li XY, Chen Q, Emran SM, Xu MM. Probabilistic techniques for intrusion detection based on computer audit data. IEEE Trans. on Systems, Man, and Cybernetics—Part A: Systems and Humans, 2001,31(4):266~274.

[3] Ko C, Fink G, Levitt K. Automated detection of vulnerabilities in privileged programs by execution monitoring. In: Proc. of the 10th Annual Computer Security Applications Conf. Orlando: IEEE Computer Society Press, 1994. 134~144.

[4] Bernaschi M, Gabrielli E, Mancini LV. REMUS: A security-enhanced operating system. ACM Trans. on Information and System Security, 2002,5(1):36~61.

[5] Goldberg I, Waqner D, Thomas R, Brewer EA. A secure environment for untrusted helper applications. In: Proc. of the 6th USENIX UNIX Security Symp. San Jose: USENIX, 1996. 1~13.

[6] Marty R. Snort-Lightweight intrusion detection for networks. In: Proc. of the 13th Conf. on Systems Administration. Washington: USENIX, 1999. 229~238.

[7] Warrender C, Forrest S, Pearlmutter B. Detecting intrusions using system calls: alternative data models. In: Proc. of the 1999 IEEE Symp. on Security and Privacy. Oakland: IEEE Computer Society Press, 1999. 133~145.

[8] Hofmeyr SA, Forrest S, Somayaji A. Intrusion detection using sequences of system calls. Journal of Computer Security, 1998,6(3): 151~180.

[9] Lee W, Stolfo SJ, Chan PK, Eskin E, Fan W, Miller M, Hershkop S, Zhang J. Real time data mining-based intrusion detection. In: Proc. of the 2nd DARPA Information Survivability Conf. & Exposition II. Anaheim: IEEE Computer Society Press, 2001. 89 ~100.

[10] Lee SC, Heinbuch DV. Training a neural-network based intrusion detector to recognize novel attacks. IEEE Trans. on Systems, Man, and Cybernetics—Part A: Systems and Humans, 2001,31(4):294~299.

[11] Forrest S, Hofmeyr SA, Somayaji A, Longstaff TA. A sense of self for UNIX processes. In: Proc. of the 1996 IEEE Symp. on Security and Privacy. Oakland, CA: IEEE Computer Society Press, 1996. 120~128.

[12] Lee W, Stolfo SJ. Learning patterns from UNIX process execution traces for intrusion detection. In: Proc. of the AAAI97 Workshop on AI Methods in Fraud and Risk Management. Providence, Rhode Island: AAAI Press, 1997. 50~56.

[13] Okazaki Y, Sato I, Goto S. A new intrusion detection method based on process profiling. In: Proc. of the Symp. on Applications and the Internet. Nara: IEEE Computer Society Press, 2002. 82~90.

[14] Sekar R, Bowen T, Segal M. On preventing intrusions by process behavior monitoring. In: Proc. of the USENIX Intrusion Detection Workshop. Santa Clara: USENIX, 1999. 29~40.

[15] Uppuluri P, Sekar R. Experiences with specification-based intrusion detection. In: Lee W, Mé L, Wespi A, eds. Proc. of the 4th Int'l Symp. on Recent Advances in Intrusion Detection. Davis: Springer-Verlag, 2001. 172~189.

[16] Chari SN, Cheng PC. BlueBoX: A policy-driven, host-based intrusion detection system. ACM Trans. on Information and System Security, 2003,6(2):173~200.

[17] Xie HG, Biondi P. The LINUX intrusion detection project. 2002. http://www.lids.org

[18] Walker KM, Sterne DF, Badger ML, Petkac MJ, Shermann DL, Oostendorp KA. Confining root programs with domain and type enforcement. In: Proc. of the 6th USENIX Security Symp., Focusing on Applications of Cryptography. San Jose: USENIX, 1996. 21~36.