Anomaly Detection Using Self/Nonself Discrimination

Lars Olsson

June 12, 2003

Abstract

In this thesis we show how computers can protect themselves from different forms of attacks, mis-configurations, and program errors. The work is inspired by the immune system and in a similar vein to the immune system our system learns how to distinguish self from nonself. The learning is done on a system call level and profiles are constructed for the analysed programs. The scheduler then decides how much processing time each process should have according to how “normal” the program behaves. Hence, this system can be seen as a homeostatic feedback loop where the analysis of the system calls is the sensor and the scheduler the actuator that tries to maintain a stable environment.

The system is implemented as a couple of modules to the Linux kernel and analyses each system call that is made by programs added to the system. To learn and analyse profiles of the system calls we have tried three different methods, a table lookup method, a feed-forward neural network, and an Elman recurrent neural network. Experiments show that this system can detect several methods of intrusion including buffer over-flow attacks, format string attacks, and Trojan code.

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 Aim 2

1.3 Background and Related Work 2

1.4 Overview 4

2 Biological Inspiration: Homeostasis and the Immune System **5**

2.1 Homeostasis 5

2.2 Immune Systems 6

2.2.1 Artificial Immune System Algorithms 7

2.3 Why Homeostasis and Immune Systems are Attractive Metaphors in Computer Science 7

3 Normal and Abnormal Program Behaviour **8**

3.1 Kernel Mode, User Mode, and System Calls 8

3.2 System Calls and Program Behaviour 9

3.3 Analysing System Call Sequences 10

3.3.1 Lookahead-Pairs 10

3.3.2 Neural Networks 12

4 Design and Implementation **14**

4.1 Requirements 14

4.2 Why a Kernel Implementation? 15

4.3 Design 15

4.3.1 System Call Hijacking 15

4.3.2 Profiles and Training 16

4.3.3 Scheduling of Abnormal Processes 17

4.3.4 User Interface 17

4.4 Implementation 17

4.5 The User's View of the System 20

5 Experiments and Results **21**

5.1 What Can Be Detected? 21

5.1.1 Buffer Overflow Attacks 22

5.1.2 Format String Exploits 26

5.1.3 Trojan Code 26

5.1.4 Summary of Detection Experiments 28

5.2 Off-line Experiments 29

6 Analysis and Discussion **30**
6.1 The Effectivness of Slowing Down System Calls 30
6.2 Performance of Different Learning Algorithms 30
6.3 How Different Are Abnormal Sequences From Normal Sequences? 31
6.4 Determinism of Sequences of System Calls 32
6.5 Problems With The Approach and Our System 33

7 Conclusions **35**

A Source code **40**
A.1 Development and Testing 40
A.2 Installation and Usage 40
A.3 The Sandbox Module 41
A.4 Makefile 42
A.5 gensyscallcode.pl 43
A.6 proc_interface.h 45
A.7 homeos_defs.h 46
A.8 homeos_profile.h 46
A.9 proc_io.h 48
A.10 sensor.h 48
A.11 syscalls.h 48
A.12 feed_forward.h 49
A.13 elman.h 49
A.14 profile_io.h 49
A.15 sandbox.h 49
A.16 homeos_utils.h 50
A.17 main.c 50
A.18 proc_interface.c 51
A.19 proc_io.c 53
A.20 sensor_module.c 58
A.21 syscalls.c 60
A.22 profile_io.c 86
A.23 homeos_profile.c 89
A.24 lookahead_profile.c 97
A.25 homeos_prof_saver.c 98
A.26 sandbox.c 99
A.27 homeos_utils.c 106
A.28 start_homeos 107
A.29 stop_homeos 107
A.30 homeossgui.tcl 107
A.31 timer.tcl 109

Chapter 1

Introduction

1.1 Introduction

As computer systems become more and more interconnected they become more vulnerable to various forms of deliberate attacks, configuration mistakes, and program errors. There is also a growing need for more autonomous and self-controlling systems since the abundance of computers makes it impossible for all computers to be under close supervision by humans. If computers are to be left alone for extended periods of time without human intervention they also need to know how to protect themselves. Hence, the major question for computer science is not performance anymore, but *survival*.

How should a system be designed in order to survive? If one looks at animals, which are experts at survival, they all seem to share some common design ideas. One of the most important ideas is that of *homeostasis*. Homeostasis is basically a way of maintaining stability, for example keeping a constant body temperature, through sensors and actuators coupled in feedback loops. One system existing in all mammals is the immune system that can distinguish between cells belonging to the animal (self) and cells not belonging to the animal (nonself). Once a foreign cell is found by the immune system, it is eliminated.

The work presented in this thesis investigates how ideas taken from biology can be used to create computer systems that can protect themselves with as little human help as possible. More specifically, we look at how an operating system can protect itself from processes that start to behave differently than the normal behaviour. A process can start to behave differently, and potentially dangerous for the computer, for a number of reasons. One reason is that there is an error in the program causing it to execute a not normal and potentially dangerous sequence of system calls. Another reason is mis-configurations, for example of a web-server, that potentially opens up security loop-holes or causes the web-server to not work properly. A computer can also be attacked by a foreign computer or hacker. A typical attack is to use a so called *buffer overflow attack*, which basically exploits some kind of weakness in the design or programming of the attacked process that causes the process to execute commands or code inserted by the attacker [1]. To detect abnormal behaviour we focus on the system calls that processes make to the operating system since these are (in modern operating systems) the only way a user space process can interact with the operating system and the hardware.

1.2 Aim

The aim of this work is to investigate how an operating system can protect itself from abnormal behaviour and to implement a real system for the Linux kernel. One difference between the work presented here and most work done in this area is that we have actually implemented a complete on-line system, not just a proof of concept off-line learning algorithm. This is important for several reasons. (1) Developing software for an operating system put several constraints on the software, for example speed and the use of some programming constructs. Thus, if only an off-line system is built it might be impossible or to slow to use in a real operating system. (2) By running the system as a part of the operating system it is easier to test on a number of different programs, since no collection of data is needed beforehand.

1.3 Background and Related Work

If one wants to build a secure system, there are three different approaches that one can follow[32]:

- Build a system in such a way that dangerous or unauthorized activities cannot take place. This is the best way, but is extremely difficult and may be impossible unless the system is used in an isolated environment without interactions with other computers.
- Detect illegal and dangerous activities as they happen. An ideal system would also stop the activities before they compromise the system.
- Detect the illegal activities after they have occurred and try to determine how much damage they have done.

Many systems, including Tripwire [20], use the third way by maintaining a set of cryptographically secure checksums for all files in a system. It is then possible to detect possible attacks by comparing the saved checksums with the actual checksums of files to find possible damage. Many virus-detection system also use a similar method where they once in a while scan local or network storage for signatures of known viruses.

In computer security *intrusion detection* is the area that focus on how to detect and respond to attacks from the outside. There are two basic approaches to intrusion detection [9]: *misuse intrusion detection* and *anomaly detection*. In misuse intrusion detection, known patterns (signatures) of known intrusion methods are used to identify attacks as they happen. In anomaly detection systems, on the other hand, it is assumed that the patterns of the intrusion are unknown, and an intrusion is then classified as something that is different from the normal behaviour of the system. Since misuse detection assumes that the pattern of intrusion is already known, it does not work for new kinds of attacks. In [6] Dorothy E. Denning presented a generic intrusion detection model shown in Figure 1.1. In the work presented in this thesis we use Denning's generic model to build an anomaly detection system since we are interested in detecting novel attacks.

Some current commercial and academic systems tries to implement the second method described above, detecting and stopping the attacks as they happen. In a recent study of the most popular intrusion systems [24], the effectiveness and stability of the tested systems were so bad that that no winner could be found in the test. One major problem was the high rate of *false positives*, which means false warnings of intrusions. This is a major problem since

Figure 1.1: Denning's generic intrusion detection model [6].

if a system generates too many false positives, it will either be turned off or most possibly neglected if it actually warns for a real intrusion [9]. Another, and even more severe error, is *false negatives*, where the system does not detect a real attack.

The seminal paper [12] by Forrest et. al. was the paper that established the analogy between the human immune system and intrusion detection. They did this by showing how correlations in fixed-length sequences of system calls executed by a process could be used as a signature, or profile, determining the "self" of each program. They analysed traces of system calls from several quite complex UNIX programs, including `sendmail` and `lpr`, and found that it was possible to build profiles of these programs using short sequences of system calls. The profiles could then be used to see if the process was behaving "normal" or not since different kinds of common intrusions changed the sequences of system calls. The system they developed was only used off-line using previously collected data and used a quite simple table-lookup algorithm to learn the profiles of programs. In [35], Forrest et. al. analyses alternatives to the table-lookup algorithm used in [12], including methods based on frequencies of different system calls, a data mining technique and Hidden Markov Models (HMMs). They show that HMMs have exceptional accuracy, but at a high computational cost. The simpler table-based methods performs almost as good and are much cheaper to execute.

Several other research groups then tried more complex machine learning algorithms. Ghost et. al. [16], tried several different machine learning algorithms including another table-lookup algorithm and a back-propagation feed-forward neural network. Most of the early algorithms used fixed sequence lengths to analyse the sequences. In [36], the authors describe how they have used a variable length sequence matching algorithm known as the Teiresias pattern-discovery algorithm to identify anomalies in sequences of system calls. Sekar and Uppuluri show in [29] how a language they have designed called REE (regular expressions for events) can be used in conjunction with an extended finite state machine to generate an intrusion detection system. In this system the user needs to create a specification of legal behaviour in REE for each process manually. A similar approach is used by Kerschbaum et. al. in [18] where they show how it is possible to add code to the operating system and the user programs to detect attacks. This, of course, assumes that the source code to both the programs and the operating system is available to the user. It also assumes, as in [29], that the user is knowledgeable enough to program and identify security holes and normal behaviour of the programs.

As far as we can tell, all examples discussed above except [18], were all implemented as proof of concept off-line systems and not as real-time intrusion detection systems embedded in the operating system. The first real anomaly detection system actually using the ideas developed in [12] was implemented by Anil Somayaji while he has writing his PhD thesis [32]. The system, *pH* (process Homeostasis), was implemented for the Linux 2.2 kernel and has been a great inspiration for the work presented in this thesis. In his thesis, Somayaji shows how the operating system can try to maintain a stable environment by slowing down processes that do not behave according to the learned profile. The user can then decide if the anomalies are part of the normal behaviour of the process or not. The system developed in this thesis is quite similar to [32], and we have even borrowed about 40 lines of code from Somayaji's system, see comments in the source code. One major difference is the design and another difference is some changes to Somayaji's profile training algorithm and that our system also can use different training algorithms.

The work described in [12] also inspired work in network intrusion detection, see for example [17] and [21]. Here the focus is the traffic between different machines. Instead of analysing system calls, self is defined as the normal pattern of ip-addresses and ports. Nonself is then traffic between ip-addresses and ports that is not in the set of normal traffic patterns. There have also been some work in combining system call trace analysis with other network based methods. In [15], for example, the authors describe the architecture of a complete agent-based system where different agents monitor different parts of networks, including system call analysis, network traffic, and authentication processes. [12] and the earlier work by Forrest et. al. such as [11] have also inspired much work in general pattern recognition algorithms, see for example [5] for a nice overview of the most common algorithms.

1.4 Overview

The rest of the thesis is structured as follows. In Chapter two we describe and discuss the biological inspiration for this work, namely homeostasis and immune systems. In Chapter three we describe how the behaviour of programs can be seen as sequences of system calls. We also shortly describe what system calls are and how they relate to operating systems and user space processes. We then describe the design and implementation of our system in Chapter four. In Chapter five we describe the experiments we have done and the results. Chapter six contains an analysis of the results and Chapter six contains our conclusions. Finally, Appendix A contains all the source code and instructions on how to build and use the system.

Chapter 2

Biological Inspiration: Homeostasis and the Immune System

This Chapter discusses the biological inspiration used in this thesis. First we describe the organising principle of homeostasis and then we describe some fundamental properties of one homeostatic system found in all animals, the immune system. We also discuss how this relates to computer science and why homeostasis and immune systems are good metaphors to use as inspiration when building secure and stable computer systems.

2.1 Homeostasis

All biological systems maintain a stable internal state by monitoring and responding to both internal and external changes [4]. This self-monitoring can be seen as one of the defining properties of life and is generally known as *homeostasis*. Although many homeostatic systems have been studied extensively, most are still not completely understood. One general feature seems to be the use of sensors that monitor some property of the body, for example body temperature. The sensors are then coupled to effectors [2], for example blood vessels in the extremities, that can contract when the body is too cold. This reserves a greater proportion of the body heat to the inner core of the body and hence makes the individual warmer.

Homeostatic systems generally have the following properties: a state that need to be maintained, a closed system where this state is to be maintained, a sensor that can detect the current state, and an effector that can change the state of the monitored state. Table 2.1 summarizes these properties and shows how they relate to one homeostatic system, namely temperature regulation.

Abstraction	Temperature Regulation
closed system	body of an individual
state	temperature
sensor	specialized nerve cells
effector	muscles, sweat, glands, and more

Table 2.1: The four defining properties of a homeostatic system.

To use biological metaphors in computing the right level of abstraction needs to be found.

One should also think about what abstractions in the artificial system that correspond to what abstraction in biology. Following [32], Table 2.2 shows how the biological abstractions found in Table 2.1 can be used to find abstractions to build an intrusion detection system.

Abstraction	Anomaly detection
closed system	individual computer
state	normal program execution
sensor	sequences of system calls
effector	delayed or canceled system calls

Table 2.2: The four defining properties of a homeostatic intrusion detection system based on analysing sequences of system calls.

There are several good reasons why analysing sequences of system calls is a good sensor. One is that system calls actually are the only way a program can interact with the operating system and hardware (see the next Chapter). Another good reason is that they are fairly easy to detect and that the detection can be done without much overhead. Once abnormal behaviour is detected it should also be dealt with. In this system we have been inspired by [32] and slow down processes that start to behave abnormal. There is also an option of killing executing processes or letting all abnormal system calls return an error code.

2.2 Immune Systems

The immune system is a complex homeostatic system that is responsible for defending the body from misbehaving and foreign cells. A good overview of the immune system for computer scientists can be found in [31] and a more biological overview can be found in [4].

The immune system consists of a large number of cells and molecules which interact in a number of different ways to detect and eliminate dangerous agents (pathogens). Since the interactions depend on chemical bonding they are localised. Surfaces of the immune system cells are covered with receptors, some which bind other immune system cells or molecules to achieve communication and signaling to mediate the immune response. Other cells bind chemically with with pathogens. Most of these cells travel around the body in the blood and lymph systems, without centralised control and very little, if any, hierarchical organisation [17]. Detection and elimination of pathogens is the result of trillions of cells interacting following simple, local rules.

The classic view is to the see the immune system as a system that distinguish *self* from *nonself*, where self is the body and nonself all pathogens. However, many pathogens are not dangerous, and eliminating them might even harm the body. Hence, A better definition is then to say that the immune system distinguishes between *harmful* nonself and self [17]. When a harmful pathogen has been detected it must be deleted. Different pathogens have to be eliminated in different ways and hence the immune system needs to choose the right kind of effector to eliminate a particular pathogen. Not everything is known how this works but signaling in the form of cytokines probably plays a part in regulating the B and T cells, the most common cells in the immune system.

2.2.1 Artificial Immune System Algorithms

In recent years several researchers in computer science have started to develop algorithms, especially for pattern recognition, that are inspired by biological immune systems. [5] is a good overview of the most well known algorithms. Here we will describe one of these algorithms, the *negative selection algorithm*.

One of the ways that the immune system achieves self-tolerance (correct discrimination between self and nonself) is by allowing the detector cells to mature in isolated settings. T cells, for example, mature in the thymus where they undergo several stages of maturation, including genetic rearrangements, positive selection and negative selection. Of special interest to this work is negative selection, in which T cells that bind strongly with self-proteins are destroyed. This principle can be used to design a change-detection algorithm as follows: Suppose that we are given a collection of data. This collection is defined as self and our goal is to monitor self for changes. The data could be sequences of system calls, an executable file, or a file containing some kind of data. (1) Generate a set of detectors which fail to match self. (2) Use the detectors to monitor the protected data. (3) If a detector finds a match a change has occurred in the monitored data and we can also know where since we know what pattern matched the detector. Observe that this algorithm uses the "closed world" assumption. If the detector does not match self it matches some part of nonself.

2.3 Why Homeostasis and Immune Systems are Attractive Metaphors in Computer Science

As discussed in [10], immune systems have several properties that make them attractive from a computer science point of view. They are *distributed* across the whole body with no central controller and hence there is no single point of failure. This is especially an attractive feature in so called *carrier-class* systems [13], which usually are systems used in demanding environments such as telecom systems and defense where no single point of failure is allowed and a high reliability is necessary. Immune systems are also *diverse*, which enhances the robustness on both individual and population level, since different people may be vulnerable to different pathogens. The components of immune systems are also constantly created, destroyed, and moved around the body, leading to a *dynamic* system that increases temporal and spatial diversity. Another interesting aspect is that the immune system is *self-protecting*, since the immune system actually protects itself using the same mechanisms that protects the rest of the body. The last, and maybe most important feature of the immune system, together with its distributed nature, is that it can *adapt*, i.e. learn, to recognize and respond to new foreign pathogens. All of these features added together sounds like the ideal design for a distributed computer system.

What is needed, then, is a way to design and structure computer systems as homeostatic systems using immune systems and other forms of feedback loops for control. In this thesis we investigate how one of these systems, namely a security system based on analysing sequences of system calls, can be designed. In a sense, this can be seen as a low-level reflex system, similar to movements that are carried out without consultation of the brain, such as moving your hand from a hot stove. To build a really stable and secure system built on these principles, other feedback systems, designed around for example log analyzers, network usage, and other system properties, must be designed.

Chapter 3

Normal and Abnormal Program Behaviour

This Chapter describes how user programs interact with the operating system and the hardware solely via a set of system calls. First, this interface is described and then different ways of representing program behaviour by sequences of system calls are discussed.

3.1 Kernel Mode, User Mode, and System Calls

All modern CPUs can run in at least two different modes usually called *kernel mode* and *user mode* [33][3]. If a process (an executing instance of a program is called a process) is executed in user mode it cannot directly access the kernel data structures or the kernel programs. When a program runs in kernel mode these restrictions no longer apply. In some primitive operating systems, like for example MS-DOS, user mode programs are allowed direct access to the hardware. In Linux, the operating system used in this work, and other modern operating systems such as Windows NT, the user program issues a request to the operating system when it wishes to access a hardware resource. If the request is granted the operating system interacts with the hardware device on behalf of the user program.

User space programs make requests to the operating system (kernel) via a set of system calls. If a system call is called by a user program the arguments to the system call, for example a file name and mode to the `open` call, are placed in specific registers in the processor. The user program then executes a special hardware-dependent instruction (`int 0x080` in x86 CPUs) and the CPU then changes from user mode to kernel mode. All of this is usually done by the standard library (`libc`) but it is also possible for user programs to make system calls directly to the kernel. The system call dispatcher in the kernel then checks if the system call and parameters are well-formed and allowed. If something is wrong, an error code is returned and the CPU changes back to user mode. Otherwise, the kernel executes the system call and returns the status of the executed call before the CPU changes back to user mode.

Figure 3.1 shows the relation between the kernel and user programs. System calls are in fact the only way user space programs can interact with the kernel and examples of system calls are `write`, `read`, `chmod`, `chdir`, and `execve`, the only way to execute a new process.

Figure 3.1: Kernel Mode (gray colour) and User Mode with the system call interface.

Figure 3.2: The different classes of computer program behaviour.

3.2 System Calls and Program Behaviour

When a program is used it is used because the user expects it to perform a certain task. If the program cannot perform the task, maybe because of faulty input or a broken network card, the user expects the program to halt the execution and inform the user. This can be defined as the legal program behaviour. If, on the other hand, the program does something that it is not intended to do, because of a programmer error, mis-configuration, or a deliberate attack by another user, this can be seen as illegal or abnormal behaviour. The way a program normally behaves on a particular computer with a particular configuration can then be seen as the normal behaviour. This is especially true for programs that execute for a long time, such as web and ftp-servers. Figure 3.2 displays this relation between normal, legal, and abnormal program behaviour.

Since a user space program only interacts with the kernel, and hence all hardware devices such as network interfaces and hard drives, through system calls, the behaviour of a process can be seen (from the kernel's point of view) as a sequence of system calls and the arguments to the system calls. In all work so far regarding anomaly intrusion detection systems, except [22], only the sequences of system calls and not the arguments to the calls have been studied. This is probably due to the complexity of analysing the arguments.

If we define the sequences of system calls that a program executes during normal execution as self, the problem of detecting abnormal behaviour is then the problem of finding sequences of system calls that are not in the profile of normal behaviour, that is, nonself. To build these profiles a number of different machine learning algorithms can be used. A few of these are described in the next Section.

3.3 Analysing System Call Sequences

Since most useful programs execute thousands or even millions of system calls during their lifetime, it is necessary to focus on sub-sequences, or windows of system calls. Also, since a system that analyses system calls should be able to detect abnormal behaviour while it is occurring, it also needs to execute fast since it will most probably be part of the kernel. The problem is then to build a profile of sequences of system calls that define the self of the program. More formally, given

$$
\begin{aligned}
C &= \text{the alphabet of all possible system calls} \\
c &= |C| \text{ (221 in Linux 2.4)} \\
T &= t_1, t_2, \ldots, t_\tau | t_i \in C \text{ (The trace of calls)} \\
\tau &= \text{the length of T} \\
w &= \text{the window size where } 1 \leq w \leq \tau
\end{aligned}
$$

the problem is then to build

$$
P = \text{the profile (a set of patterns associated with } T \text{ and } w)
$$

and to detect whether a given sequence S exists in P. An algorithm should also be able to say how abnormal S is, given P, maybe using Hamming distance or another way of measuring the difference between two sequences.

3.3.1 Lookahead-Pairs

In [32], Anil Somayaji uses an algorithm called lookahead pairs to build profiles. Here a window is slided over the sequence of system calls, recording for each system call what call(s) that came before it in the current window. More formally, given the definitions given above in Section 3.3 , we can define a profile P_{pair} as consisting of a number of pairs of system calls as:

$$P_{pair} = \{\langle s_i, s_j \rangle_l : \quad s_i, s_j \in C, 2 \leq l \leq w$$
$$\exists p : \quad 1 \leq p \leq \tau - l + 1,$$
$$t_p = s_i,$$
$$t_{p+l-1} = s_j\}$$

To make this more concrete, consider the following sequence of system calls:

`brk, open, read, fstat64, old_mmap, open, mprotect`

where `brk` is the first system call and `mprotect` the last in the sequence.[1]. Suppose that the window size, w, is 3, then the profile will look as Table 3.1. Note that since `open` follows both

current	position 1	position 2
brk		
open	brk, old_mmap	fstat64
read	open	brk
fstat64	read	open
old_mmap	fstat64	read
mprotect	open	old_mmap

Table 3.1: The sequence represented with a $w = 3$.

`brk` and `old_mmap`, `open` has both these two in the entry for position 1. To determine if a given sequence is in the profile we check new traces against the profile using the same method. Given this profile, consider what will happen if the following sequence is presented:

`brk, open, mprotect`

According to the profile `mprotect` can follow `open`, but since there is no entry for `brk` in position 2 for `mprotect`, this sequence will be classified as abnormal. Using this method it is also possible to measure how abnormal a sequence is by counting the number of entries in the sequence that are not in the profile. The example sequence above would have an abnormal count of one but a sequence like `mprotect, read, open` would have an abnormal count of two.

How can we know when the training of the profile is finished? In [32] the author describes a two-step heuristic that depends on the total number of calls seen so far and how many calls that have been seen since the profile was last edited. Let *total_count* be the total number of seen calls and *last_mod_count* the number of calls seen since the profile was last edited. The profile is *frozen* when

$$\frac{total\_count}{(total\_count - last\_mod\_count)} > 4$$

Then, if no changes changes has been made to the profile after *normal_wait* seconds, the profile is said to be normal and ready to be used. If the profile is changed when it is frozen it is thawed and the frozen flag set to 0.

[1]Note that these names are actually encoded as numbers between 0 and 255 in the Linux kernel. This mapping can be found in the asm/unistd.h header file.

Figure 3.3: The architecture of a feed-forward neural network (F) and a Elman recurrent neural network (E). The nodes labeled I are input nodes, the nodes labeled H are hidden neurons, and the nodes labeled O are the output nodes. In the Elman recurrent net the nodes labeled C are the context (memory) nodes.

3.3.2 Neural Networks

Another way to build profiles of traces of system calls is to use neural networks [16]. In this thesis we have used to different types of neural networks, the classic *back-propagation feed-forward* [14] and *Elman recurrent* neural networks [8].

The architecture of each type of network is shown in Figure 3.3. For both types of networks each input neuron represents one call in the window of system calls, Hence, if a window size of 8 is used the network has 8 input neurons. For the feed-forward network there is one output neuron whose value depends on whether the input sequence is abnormal or not. Elman networks have memory neurons and are used to predict sequences. Usually they have as many output neurons as input neurons where the values of the output neurons should be the next sequence. Hence, given a sequence, for example $1, 2, 1, 3, 4$, and a window size of three starting from the left, the input is $1, 2, 1$. Then the output should be the next sequence, that is, $2, 1, 3$. More formally, given an input sequence (i_{n-2}, i_{n-1}, i_n), the corresponding output neurons should have the values of (o_{n-1}, o_n, o_{n+1}). The sum of the absolute differences between the next input values and the values of the output neurons is used to calculate the error for the training.

The training is done using the back-propagation algorithm [14] for both the feed-forward network and the Elman network. Since the space of possible sequences is much bigger than the space of normal sequences the training starts by training the networks with random data. This random data is classified as abnormal behaviour. Then the network is trained to detect the training (normal) data as normal. To get good performance a number of networks can be trained on the same data using the methods described in [16], such as varying the number of hidden nodes since the optimal number of hidden nodes is not known before training. Because of this we have only been able to train the neural network profiles off-line. They can then be used in the on-line system described in the next Chapter.

To compute how abnormal a sequence is we use the *leaky-bucket algorithm* described in [16]. The leaky-bucket algorithm keeps a memory of recent events by accumulating the networks output. This value is then is slowly leaking out. Thus, if there are several abnormal system calls after each other the abnormal count will quickly accumulate a large value. If, on the other hand, the calls are normal, the anomaly counter will "leak" away the anomaly value and eventually reach zero.

Chapter 4

Design and Implementation

This Chapter describes the design and implementation of our system. First we discuss the requirements and different possible ways to implement the system and the reasons we finally decided on the used design. Then the design is described and the Chapter then ends with a description of the most interesting parts and data structures of the actual implementation. All the source code can be found in Appendix A together with a description of all files and instructions on how to build and use the system.

4.1 Requirements

The major goal when building this system was to design a reactive system that can protect the computer from processes executing unwanted system calls. This goal can be divided in to the following requirements:

- The system should monitor processes on a system call level.
- Profiles should somehow be created for the monitored programs, using some kind of machine learning algorithm. The implemented algorithms are so far an algorithm similar to the algorithm described in [32], a regular feed-forward neural network, and an Elman recurrent neural network [8]. At this moment it is only possible to train the neural network profiles off-line, but trained profiles can be used in the on-line system.
- When the profile for a program is stabilized all processes executing this program should be monitored for abnormal behaviour. Preferably, the algorithm should also be able to say how abnormal the process is behaving.
- The computer should then protect itself by slowing down the abnormal process. The user should be notified when a process is behaving abnormal and should be given the opportunity to accept the abnormal behaviour as normal and incorporate that in to the profile or kill the process.
- The system should be stable and fast enough to be run on production servers. It should also be fairly easy to install and not require the user to patch and recompile the whole kernel.

4.2 Why a Kernel Implementation?

Given the requirements listed above we decided to implement the whole system as a part of the Linux kernel. This was necessary for several reasons. The first reason is that we wanted to monitor processes on a system call level. This is actually possible from user space programs, using the `ptrace()` system call, but this is apparently not working perfectly and also slows down the computer significantly [32]. Another option would be to patch the standard `libc` c library. But, since it is possible for programs to bypass this library and make direct calls in to the kernel this option would never be able to catch all system calls. The second reason was that since the system should be able to change the scheduling of abnormally behaving processes, we needed to put at least a part of the code in the kernel. Since context switches between kernel and user mode are very expensive, from a performance point of view, we realised that a complete kernel implementation was the way to go.

Also, by putting the whole system in the kernel, the system becomes much more secure since it is extremely hard for a hacker to change or insert new code in to the running security module [1] . It is important to note though, that even though the system is implemented as a part of the Linux kernel, it is possible to build, install, and run it without recompiling the actual kernel. The reason for this is that the system is implemented as two kernel modules. Kernel modules are basically code, for example device drivers, that can be loaded and linked in to the kernel when needed and then removed when no longer needed[27][3]. This design has several advantages, and some disadvantages, as discussed below.

4.3 Design

Conceptually the system consists of six different parts. (1) First we need to access all system calls as they are executed. (2) Each executable that we wish to monitor should be represented by a profile. (3) Then we need to train the profiles of the monitored programs. (4) These profiles are then used to see if a process is behaving abnormal. (5) Abnormally behaving are then slowed down depending on how abnormal they behave. (6) What programs/processes to monitor and the administration of the system is done through a user interface. This interface also shows information about processes and executing processes. data. Figure 4.1 shows the relation between these parts and the rest of the operating system. In reality, the system is divided in two different modules. The first module is responsible for accessing all system calls as they are executed.

4.3.1 System Call Hijacking

The first problem we faced was how to monitor each system call. Since the system is implemented as modules it is impossible to add code to the actual system call dispatcher. Instead, the system call hijacker module implements its own version of each system call. When the module is initialised each function pointer in the global `sys_call_table` array is replaced with the corresponding function implemented by us and the original pointers are saved (these are the pointers that are used to call the original system calls).

[1] But probably not impossible. See for example [26] in the hacker fanzine Phrack where the anonymous author describes how it is possible to patch a Linux kernel without inserting a new module. One problem though, with the method used in [26], is that the hacker needs to be root of the system (which is usually the goal of the attack).

Figure 4.1: The system design. Our system consists of the changed syscalls, train, process_syscall and delay_process module, profiles, and interface subsystems.

The reason to separate the system call module from the rest of the system was twofold. Firstly, by separating this functionality to a separate module it can also be used by other modules that need to monitor system calls, for example the sandbox module described in Appendix A. Secondly, and more important, is the possibility of race conditions [33] if the system is unloaded when no longer needed. Suppose that the whole system was built as one module. When the module is unloaded the original system call function pointers need to be put back in the `sys_call_table`. This in itself is no problem, since it is possible to turn off all interrupts and hence make the changes to the array atomic. The problem is that some system calls might take seconds, or maybe even days to execute, because they are waiting for some special event to occur. When these calls return, they return to the address of the corresponding function in the hijacking code. But, since this code is unloaded, this address can contain anything and a total system crash is likely to occur. The solution to this is to separate the hijacking code to a separate module that is never unloaded. Instead, when the user requests to unload the module it changes back to the original system calls, but does not remove itself (in reality, refuses to be removed) from memory. Since this module only occupies around 13k of memory this is not a high price to pay for a stable system.

4.3.2 Profiles and Training

The other module is responsible for all functionality relating to the program profiles and the interface to the user. Each program that is monitored has one profile. Since each program can be executed as several independent processes at the same time, each process has an individual state structure but shares the actual profile with all processes executing the same program. Hence, several processes can train and hence update a single process at the same time. This speeds up training if we assume that all instances of an executing program behaves similarly. Profiles are saved to file each time a process exits but they are also cached in memory for faster recollection. When the user adds a program to monitor via the user interface it is added to a list of monitored programs. Each time the `execve()` (the only way to start a new program) system call is executed the list with monitored programs is traversed. If the program is found the new process is added to the monitoring system. If the profile (the profile for the program this process is executing) is not found in the cache or on disk a new empty profile is created.

4.3.3 Scheduling of Abnormal Processes

If a process is behaving abnormal, it is slowed down by letting the process sleep while it is executing system calls. The length of the sleep depends on how abnormal the process has behaved. In the current implementation each process has a anomaly window consisting of the 128 latest executed system calls. This window is originally filled only with 0's but for each abnormal call the value is changed to 1. Thus, it is possible to keep track on how many abnormal call this process has done in the last 128 calls since a normal call will change the value back to 0. This value is used to calculate the number of milliseconds to sleep. Let A_b be the number of abnormal calls the last 128 calls. Then the process should sleep for $delay\_factor * 2^{A_b}$ jiffies. Jiffies is the internal time measurement in Linux and is generally $10ms$ on the x86 platform, which means that the operating system switches between processes 100 times per second. Observe that the amount of time that abnormal processes sleep increases exponentially depending on the value A_b.

4.3.4 User Interface

The user communicates with the system module by writing and reading to/from files in the `/proc/homeos/` directory which resides in the *proc filesystem*. The proc filesystem is a virtual filesystem (VFS), which means that it does not represent a physical device [3]. It is designed to allow easy access to information about processes (hence the name) and it is nowadays used by every part of the Linux kernel which has something interesting to report. We have also implemented a graphical user interface in the language TCL/TK that gives commands and read data from the system by writing and reading to/from the files in `/proc/homeos/`. Figure 4.2 shows this gui.

4.4 Implementation

The actual implementation of the whole system consists of around 9000 lines of C, including the around 3000 lines of `syscalls.c` that are generated automatically from a list of available system calls. Around 40 lines of the code have been copied from Anil Somayaji's system *pH* described in [32] and some data structures are also quite similar. The copied functions are all commented in the code with "Note: this function copied from pH". During development we have used the 2.4.17 kernel but it is probably possible to compile this system with all 2.4 kernels. This is a quite big and complex program and we can hence only describe certain parts of the implementation in this overview.

Central to the code is the struct `homeos_task_state` that represents a monitored process, see Figure 4.3. Another central data structure is the `homeos_profile` structure that represents the profile of an executable file. This structure is dependent on what learning algorithm

Figure 4.2: The graphical user interface.

```c
typedef struct homeos_locality {
    int first, total_lfc, max_lfc;
    unsigned char window[LOCALITY_WINDOW_LEN];
} homeos_locality;

typedef struct homeos_sequence {
    int last, length;
    unsigned char seq[MAX_SEQUENCE_LEN];
} homeos_sequence;

struct homeos_task_state {
    homeos_locality loc;
    homeos_sequence seq;
    int delay;
    int sleep;
    unsigned long count;
    int pid;
    char program[HOMEOS_MAX_FILE_NAME];
    struct proc_dir_entry *entry;
#ifdef __KERNEL__
    struct semaphore task_lock;
#endif
    homeos_profile *profile;
    homeos_task_state *next;
};
```

Figure 4.3: The three structs used to represent one executing process.

```c
typedef struct homeos_profile_data {
    unsigned long last_mod_count;
    unsigned long train_count;
    unsigned long sequences;
    unsigned char entry[NUM_OF_SYSCALLS][NUM_OF_SYSCALLS];
}homeos_profile_data;

struct homeos_profile {
    int normal, frozen;
    char filename[HOMEOS_MAX_FILE_NAME];
    char program[HOMEOS_MAX_FILE_NAME];
    long normal_time;
    int win_size;
    unsigned long count;
    int anomalies;
#ifdef __KERNEL__
    struct semaphore profile_lock;
#endif
    homeos_profile_data train, test;
    homeos_profile *next;
};
```

Figure 4.4: The structs that represent a lookahead profile.

that is used and Figure 4.4 shows this data structure for the lookahead-pairs algorithm. This algorithm is described in detail in [32] and in Section 3.3.1. Observe that both the central data structures contain semaphores. This is necessary since they need to be protected from the different problems that can occur in both single and multiprocessor systems due to concurrency [33][3].

When a system call is executed by a user process it first executes our version of the system call located in the system call module. These are located in `syscalls.c` and they all look almost the same, apart from `execve()` and `fork()`, since these have to deal with the execution of new programs and processes. All the other calls looks similar this

```c
asmlinkage ssize_t homeos_sys_write(unsigned int fd, const char * buf, size_t count)
{
    if(sensor(__NR_write))
        return saved_sys_write(fd, buf, count);
    else
        return -EINVAL;
}
```

where `sensor()` is the hook in to our system and `saved_sys_write()` a function pointer to the original system call. The function `sensor()` calls `homeos_do_system_call()` in the anomaly detection system and potentially other systems that are interested in system calls, like the sandbox described in Appendix A. In `homeos_do_system_call()` the `train()` function is called after the profile has been locked if the process is added to the list of monitored processes. If the profile for this process is normal, that is, it is considered stable, it is checked whether the executed system call is normal. If it is, the profile is unlocked and the function returns. If the call is abnormal, the process is put to sleep in the function `homeos_task_delay()`, using the `schedule_timeout()` kernel function. This function invokes the `schedule()` function which selects another process for execution. The number of jiffies the process is put to sleep depends on how abnormal the call was and is described above in Section 4.3.3. When the process is executed again it returns to `sensor()`. Then the real system call is executed. Observe that since the call is analysed *before* it is called, it is possible to disallow a call to be executed at all. This is an option to just putting a process to sleep if it is behaving abnormally.

4.5 The User's View of the System

The user can influence the running system in several ways. The user can add and delete programs to monitor. It is also possible to see the properties of currently monitored processes. The properties include if the profile of the process is frozen or normal, number of anomalies, if it is currently delayed, and more, see Figure 4.2. The user can also influence running processes in six different ways:

- A monitored process can be killed.
- A monitored process can be stopped being monitored.
- The profile of a process can be reset.
- A process' profile can be set to normal so that normal monitoring starts.
- A process can be *sensitized*.
- A process can be *tolerized*.

The tolerize and sensitize commands have the same meaning as in [32] and are inspired by similar processes in real immune systems. Tolerize means that the user regards recent program behaviour as normal even if the system classified it as abnormal. Sensitize, on the other hand, tells the system that the recently learned behaviour for a process should be forgotten by the profile.

Chapter 5

Experiments and Results

This Chapter describes the experiments we have performed and their results. The method used has been as follows. We begin by showing what types of behaviour that can be detected by looking at sequences of system calls. This is important since it shows what kind of anomalous behaviour a system like this at least theoretically is able to detect. This has been done by first designing a program that is vulnerable to a certain kind of attack. Then this program has been analysed using the program `strace` to find the sequence of system calls executed by the program. We have then tried the exploit to see if it results in another sequence of system calls. Finally, we have tried to use our system to detect the same exploit. This has been done by training the system on the normal behaviour of the program and then when the profile is considered normal we have executed the exploit.

The first behaviour we have looked at is just a change in the source code of a simple program. This has been done to show how small changes to the source code of a program change the sequence of calls. It also shows how programs are executed in Linux and is a good introduction to the other experiments. Then we describe how classic hacker attacks, such as buffer overflows, trojan horses, and format string exploits can be detected by analysing sequences of system calls.

Finally we describe some experiments done off-line with the different machine learning algorithms we have implemented to represent the profiles of programs. Here we use data for a number of different programs such as `sendmail`, `ps`, and `wu-ftp` downloaded from `http://www.cs.unm.edu/~immsec`. Here we show how the different algorithms, namely look-up tables, feed-forward neural network, and Elman recurrent network, can learn profiles and then detect anomalies.

5.1 What Can Be Detected?

When a program is executed in Linux many things happen before the actual program, the code that the programmer wrote, is executed [3]. First, the executable file needs to be found and then the executable format is determined. Then the file is copied in to the working memory and several memory areas are mapped. Finally, the `main` function of the program is called. To make this more concrete, consider the classic `hello world` program listed in Figure 5.1 that prints "hello, world" to the terminal and then exits.

To view the behaviour of a program on a system call level, the program `strace` can be used. This program outputs all system calls that a program makes and the arguments to standard

```c
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        char *msg = "hello, world!\n";

        write(1, msg, strlen(msg));

        return 0;
}
```

Figure 5.1: *hello.c*. A minimal "hello, world" program using the system call *write* directly instead of *printf* to use a minimal amount of libc code.

out. Figure 5.2 shows the output for *hello.c*. Here we can see that the program actually executes 18 system calls before `write` is called, the only system call that the programmer actually called in the source code. If the *hello.c* program is changed, for example to start another program if it is given a command line argument, the sequence of system calls changes. Consider the program *hello2.c* in Figure 5.3. If no command line argument is given it will give us exactly the same series of system calls as *hello.c*. But, if started with an argument the sequence will look like Figure 5.4. Notice how a small change in the code of the program can lead to a major change in the sequence of system calls executed by the program. This result shows that it is indeed possible to detect different behaviours of programs by analysing the sequence of the system calls that they execute. When we trained our system to learn the behaviour of *hello.c* it detected when we changed *hello.c* to the code of *hello2.c* (if started with an argument).

5.1.1 Buffer Overflow Attacks

Buffer overflows all involve a program that uses a fixed amount of storage to save some kind of external data [1]. If the program (programmer) does not ensure that the external data fits in the storage, it is possible to overwrite neighbouring memory locations, for example return addresses from functions. To make this more concrete, consider the small program in Figure 5.5. If this program is given input that is longer than 1024 bytes, the data following the 1024th byte will be written over other data on the stack. On Intel processors it turns out that the return address (where the program should start executing when it returns from `parse(char *arg)` is stored below the array `param[1024]` in the stack. Since the stack grows toward lower addresses [3], a string longer than 1024 bytes (in this case), will overwrite the return address. This usually results in a `Segmentation Fault` or `Illegal Instruction` since the return address is overwritten and usually points to a random position in memory. But, if a specially designed string is constructed where the return address points to an address where executable code is found, this code will be executed. Consider for example the program in Figure 5.6. This program is designed to output a sequence of bytes to standard output that can generate a buffer overflow in *breakme.c*. If *breakme.c* is executed with ordinary input less than 1024 bytes in size it will execute no system calls except the system calls that all programs execute described above. But, on the other hand, if *breakme.c* is given the output from *overflow.c* by executing `./breakme `./overflow 1009``, *breakme.c* will execute another program by calling `execve` even though there is no code for calling `execve` in the compiled

```
[larre@galadriel homeostasis]$ strace ./hello
execve("./hello", ["./hello"], [/* 35 vars */]) = 0
uname(sys="Linux", node="galadriel", ...) = 0
brk(0)                                  = 0x8049674
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=107844, ...) = 0
old_mmap(NULL, 107844, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                = 0
open("/lib/i686/libc.so.6", O_RDONLY)   = 3
read(3, "77ELF 06"..., 1024) = 1024
fstat64(3, st_mode=S_IFREG|0755, st_size=5772268, ...) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40032000
old_mmap(NULL, 1290088, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40033000
mprotect(0x40165000, 36712, PROT_NONE)  = 0
old_mmap(0x40165000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3,
0x131000) = 0x40165000
old_mmap(0x4016a000, 16232, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x4016a000
close(3)                                = 0
munmap(0x40017000, 107844)              = 0
write(1, "hello, world!", 14hello, world!)          = 14
_exit(0)                                = ?
[larre@galadriel homeostasis]$
```

Figure 5.2: The 20 system calls made by the hello world program.

```c
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        char *msg = "hello, world!\n";

    if(argc > 1)
        system("date");
    else
            write(1, msg, strlen(msg));

        return 0;
}
```

Figure 5.3: *hello2.c*. This program executes the program date if at least one command line argument is given, otherwise it prints “hello, world!” to the terminal.

```
[larre@galadriel homeostasis]$ strace ./hello2 22
...
-1, 0) = 0x4016a000
close(3)                                = 0
munmap(0x40017000, 107844)              = 0
rt_sigaction(SIGINT, SIG_IGN, SIG_DFL, 8) = 0/*first call due to the system("date")*/
rt_sigaction(SIGQUIT, SIG_IGN, SIG_DFL, 8) = 0
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
fork()                                  = 3210
wait4(3210, Sat Aug  3 19:21:42 BST 2002
[WIFEXITED(s) && WEXITSTATUS(s) == 0], 0, NULL) = 3210
rt_sigaction(SIGINT, SIG_DFL, NULL, 8) = 0
rt_sigaction(SIGQUIT, SIG_DFL, NULL, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0 /* last call for system("date") */
--- SIGCHLD (Child exited) ---
_exit(0)                                = ?
[larre@galadriel homeostasis]$
```

Figure 5.4: The 18 first system calls are the same as Figure 5.2. Notice how a small change in the code of program can give this difference in the sequence of system calls that they execute.

```c
#include <stdio.h>
#include <string.h>

void parse(char *arg)
{
    char param[1024];
    int  localdata;

    strcpy(param,arg); /* potential buffer overflow */

    return;
}

main(int argc, char *argv[])
{
    parse(argv[1]);
}
```

Figure 5.5: *breakme.c*.A program that can be attacked by a buffer overflow. Note that the char array is 1024 characters long. *strcpy* does not check the length of the buffer it copies. Thus, if *arg* is longer than the buffer, the stack will be overwritten.

```c
#include <stdio.h>

/* assembly code used to create the code that is called in the buffer overflow.
This code basically calls the system call execve that executes the program
in EBX.
movl %esp, %ecx            # move ESP to ECX
xor %eax, %eax             # put 0 in EAX
push %eax                  # push EAX on to the stack
leal -0x7(%esp), %ebx      # put the offset (length of /tmp/sh) in EBX
add $0xc, %esp             # add 12 to ESP
push %eax                  # push EAX on to the stack
push %ebx                  # push EBX on to the stack
movl %ecx, %edx            # move ECX to edx
movb $0xb, %al             # move 11 (execve) to AL (the low half of AX)
int $0x80                  # tell the kernel that we want to execute a system call
*/

/* binary (machine code) version of the assembly code above */
char code[] = {
  0x89,0xe1,0x31,0xc0,0x50,0x8d,0x5c,0x24,0xf9,0x83,
  0xc4,0x0c,0x50,0x53,0x89,0xca,0xb0,0x0b,0xcd,0x80};

static inline getesp() {
    __asm__("movl %esp, %eax");
}

int main(int argc, char *argv[])
{
    int i;
    long unsigned esp;
    int offset = 200;

    for (i=atoi(argv[1]); i; i--)
        putchar(0x41);

    for(i = 0; i < 20; i++)
        putchar(code[i]);

    printf("/tmp/sh");

    /* calculate where in memory the code is inserted.
    this is the address that we replace the return address with */
    esp=getesp() - offset;
    fwrite(&esp,4,1,stdout);
}
```

Figure 5.6: *overflow.c*. A program that can be used to create a buffer overflow in the program presented in Figure 5.5 if the output from this program is used as input to the program in Figure 5.5. This program creates a string of bytes containing the machine code representation of executing the execve system call with */tmp/sh* as the executed program. */tmp/sh* is a symbolic link to any program. If the program is executed as *suid root*, superuser privileges, */tmp/sh* will also be executed with root privileges. The string of bytes first contain a number of NOP instructions. The length of this depends on the buffer size of the *param* buffer in Figure 5.5.

code for *breakme.c*. The output from strace can be found in Figure 5.7.

The first thing to note here is the argument string given to *breakme.c* that can be seen in the first `execve` system call. The A:s in the beginning are the NOP instructions that are needed to fill out the buffer. Then we can see the actual code that executes the system call (`\ùÄPSAÊ°Í`). Then follows the program to be executed, `/tmp/sh`, followed by the address that overwrites the return address in the stack. The next thing to note is the second `execve` call. Here we can see that `/tmp/sh` is executed and hence the buffer overflow has succeeded in executing another program!

This is of course bad news that it is possible to execute arbitrary programs by overflowing buffers. The good news is that a buffer overflow probably will execute a sequence of system calls that is not found in the normal behaviour of the program. Hence, a system like ours that detects changes from normal behaviour by analysing sequences of system calls will always detect a buffer overflow if it is correctly trained. Consequently, when we tried to train our system with *breakme.c* above, it managed to detect the buffer overflow when we started *breakme.c* with input from *overflow.c*.

5.1.2 Format String Exploits

Another kind of commonly used intrusion technique is *format string exploits*. Here the idea is to exploit the fact that the `printf` family of C functions, for example `fprintf` and `snprintf`, all use format strings to control the output [19]. If the programmer forgets to add a format string when he prints out a string that the user has typed in using any of these functions, the user can supply his own format string in the input. By cleverly crafting your own format string it is for example possible to read or even change the value of any variable in the program, see for example [28]. In itself, this is something that can not be detected by our system, unless the changed value of a variable causes the program to execute a piece of code never executed before.

The program in Figure 5.8 is a typical program that can be exploited by a format string exploit. The important thing to notice here is the call to `snprintf` that is done without a format string. The prototype for `snprintf` is

```
int snprintf(char *str, size_t size, char *format, ...)
```

Hence, in the program *vulnerable.c* the format string will be the argument given by the user. This works fine as long as there are no actual format strings, like `%d` or `%n`, in the string. But, if there is, this data will be taken from the stack.

Format string exploits can also be used to gain root access to a system. In a fashion somewhat similar to the buffer overflow described above it is actually possible to design an input string that contains the machine code for starting a shell. To test this we used a program called *fmtbrute.c* downloaded from [28] to generate this string. We then trained our system on *vulnerable.c*. Then, when the profile was normal we used the string generated by *fmtbrute.c* as input to *vulnerable.c*. As expected, our system succeeded in detecting the exploit that gives the exploiter a root shell (if *vulnerable.c* is started as suid root).

5.1.3 Trojan Code

The aim of this experiment is to see that the system can detect changes in the way a program behaves because the executable file has been replaced or recompiled with another configuration. This can for example be the case in a Trojan Horse attack, where a commonly used

```
[larre@galadriel buffer]$ strace ./breakme ‘./overflow 1009‘
execve("./breakme", ["./breakme", "AAAAAAAAAA...AAAAAAAAAA$ùÄPSAÊ°Í/tmp/sh‘øÿ>"],
[/* 35 vars */]) = 0AAAAAA á1ÀP
uname(sys="Linux", node="galadriel", ...) = 0
brk(0)                                  = 0x8049630
...
old_mmap(0x4016a000, 16232, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x4016a000
close(3)                                = 0
munmap(0x40017000, 107844)              = 0
execve("/tmp/sh", ["/tmp/sh"], [/* 1 var */]) = 0 /* the overflow */
uname(sys="Linux", node="galadriel", ...) = 0
brk(0)                                  = 0x804f5c4
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=107844, ...) = 0
old_mmap(NULL, 107844, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                = 0
open("/lib/i686/libc.so.6", O_RDONLY)   = 3
read(3, "77ELF 06"..., 1024) = 1024
fstat64(3, st_mode=S_IFREG|0755, st_size=5772268, ...) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x40032000
old_mmap(NULL, 1290088, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40033000
mprotect(0x40165000, 36712, PROT_NONE)  = 0
old_mmap(0x40165000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3,
0x131000) = 0x40165000
old_mmap(0x4016a000, 16232, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x4016a000
close(3)                                = 0
munmap(0x40017000, 107844)              = 0
brk(0)                                  = 0x804f5c4
brk(0x804f5ec)                          = 0x804f5ec
brk(0x8050000)                          = 0x8050000
```

Figure 5.7: The sequence of system calls executed by *breakme.c* if given the special overflow string written by *overflow.c*. Note that this trace has been edited in several places for space reasons. Firstly, in reality there are 1009 A:s (NOP instructions) in the beginning of the argument string to the execve call that starts .c. Then we have deleted some of the standard system calls that all processes execute, see Section 5.1. Note the second execve system call that is executed by the buffer overflow string.

```c
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char *argv[])
{
    char     foobuf[512];

    if (argc < 2)
        exit (EXIT_FAILURE);

    snprintf (foobuf, sizeof (foobuf), argv[1]);
    foobuf[sizeof (foobuf) - 1] = '\x00';

    exit (EXIT_SUCCESS);
}
```

Figure 5.8: *vulnerable.c.*A program that can be attacked by a format string exploit. Note that the snprintf function is called without a supplied format string. Hence the argument given to the program argv[1] will the format string. This program is taken from [28].

program, for example `su` has been replaced or recompiled with new code. This code usually leaves a backdoor that the hacker can use even though he no longer knows the root password. To test this form of attack we used an *sshd* backdoor [34], a source code patch for the Secure Shell that allows users to connect to a foreign machine using an encrypted communication channel. When this patch is applied it is possible for a user to login on the machine where the patched version is installed by typing a special password that is compiled in to the compromised program. If this password is used *sshd* gives the user a root shell without using the normal authentication and logging procedures. We first used the normal version of *sshd* to get a normal profile and to analyse the sequences of system calls manually with `strace`. We then tried the patched version and found that the profiles were similar. But, when we tried to login with the special password an `execve` call materialised where there had not been a `execve` call before. Hence, the system was able to detect this intrusion as well.

5.1.4 Summary of Detection Experiments

In the experiments above we have showed that several classical ways for a hacker to gain access to a system causes the attacked program to execute a different trace of system calls. Table 5.1 summarises the results. Observe that the only attacks that can not be detected

Attack	Changed trace	Detected by our system
Buffer overflow	yes	yes
Format string (new shell)	yes	yes
Format string (change/read variable)	no	no
Trojan code (sshd backdoor)	yes	yes

Table 5.1: Results of exploit experiments.

are attacks like changing/reading the value of a variable where no change in the sequence of system calls is made.

5.2 Off-line Experiments

To test how well the lookahead-pair, feedforward, and Elman network profile algorithms work we tested them on data downloaded from `http://www.cs.unm.edu/~immsec`. The different traces are described in Table 5.2 and more information about the data sets and how they were captured can be found on the website. The normal data sets were divided in to 5 equally

Program	Intrusion traces	Normal traces	Normal system calls
sendmail	3	147	1571583
ps	11	19	61440
wu-ftp	5	7	179916

Table 5.2: Data used in off-line experiments. Note that the intrusion traces are synthetic (attacks designed by the UNM researchers) for sendmail and real captured attacks for wu-ftp and ps.

sized sets and 80% of the data was then used for training and 20% for evaluation. We also used the intrusion traces to see if the trained networks could detect the intrusions. The neural networks used cross-validation for training and as described in Section 3.3.2 we tried neural networks with a number of different internal nodes for each program.

Program	Lookahead table	Feed-forward	Elman recurrent network
sendmail	0.0015%	0.02%	0.0012%
ps	0.001%	0.009%	0.001%
wu-ftp	0.009%	0.082%	0.007%

Table 5.3: Percent false postives. The results for the neural networks denotes the best trained network for each program. The percentage is the percentage of system calls that where classified as abnormal that were taken from tha normal evaluation data.

Table 5.3 shows the results for the best trained neural networks for each program and the lookahead-pair algorithm. Here we can see that the Elman recurrent neural network is slightly better than the lookahead-pair algorithm and that the basic feed-forward network performs worse than both of the other two methods. This was expected since the feed-forward contains no memory and hence has problems with classifying sequences.

The final experiment we performed was to see how well the different training algorithms can detect intrusions. Here we found that both the Lookup table algorithm and Elman neural network could detect all intrusions in the data described in Table 5.2. The feed-forward network could detect all the `sendmail` intrusions but only 3 of the `wu-ftp` intrusions and 8 of the `ps` intrusions.

Chapter 6

Analysis and Discussion

This Chapter discusses the results of our experiments and also discusses some general points regarding intrusion detection and the nature of system call traces.

6.1 The Effectivness of Slowing Down System Calls

The defense mechanism used in our system is to slow down or abort processes that behave abnormally. The idea is that by slowing the process the system administrator will have time to analyse the behaviour of the process or the process will run so slow that the attacker is unable to finish the attack. But, as we have seen in the experiments, even though almost all attacks are discovered, they might slow the process very little since the sleeping time depends on how abnormal a process is behaving. Thus, if an attack only makes very few changes to the normal sequence of calls the use will hardly notice the slowed down process. Of course this depends of how much the process is put to sleep.

Another problem of slowing down processes that behave abnormally is the possibility of Denial of Service (DOS) attacks [9], since provoking abnormal behaviour of a process will cause to execute slower. This can be at least solved by killing and restarting abnormally behaving processes.

6.2 Performance of Different Learning Algorithms

Altough the Elman neural network performed well there are a couple of problems with using neural networks in real on-line systems. One problem is execution speed. Even though neural networks are considered to be fast machine learning algorithms they are much slower than simpler table lookup methods. We have not done enough experiments to say anything certain but it seems like the neural network based algorithms slows down the computer around 15-20% while the table lookup methods slows down the computer around 5% when system call intensive applications are executed. Also observe that the Elman recurrent neural net, which performs better than the feed-forward neural net, is even slower since it contains more nodes since it uses an extra set of hidden nodes (the context nodes). Another problem is the fact that neural networks usually use floating point representations, which can cause problems when they execute in kernel mode. The reason for this is that the Linux kernel does not save the floating point registers when it switches from user mode to kernel mode and back [3] due to performance reasons. This can be solved in two ways. Either by implementing an integer based neural network or by saving the floating point registers each time before the neural network is executed and then restoring them afterwards. This is quite costly and since the system probably will execute up to hundreds or even thousands of times per second, this can not be considered a feasible option.

Another thing to consider with neural networks is their ability to generalise. This was the reason we first decided to try them since we considered generalisation as a solution to false positives. But, the more we have been thinking about it, the less attractive generalisation seems (in this context). The question is how a neural network can be trained to generalise so that only new normal behaviour is considered normal. Since an attack may consist of only one new system call, the neural network might say that this is normal behaviour because the way it was trained. Our philosophy is that even though false positives are bad, they are still better than false negatives (not warning for real intrusions), and hence our conclusion is that the simpler table lookup method is more secure than the neural network approach.

6.3 How Different Are Abnormal Sequences From Normal Sequences?

When we started to look the data generated by the experiments done in the last Chapter concerning different exploits we soon realised that there usually is a very small difference between abnormal behaviour and normal behaviour regarding what system calls that are used. Usually, the same system calls are used, and maybe the only difference is that the exploited program issues an `execve` call. We also found that the sequences seem quite regular and that sub-sequences mostly are fairly deterministic. The data downloaded from `http://www.cs.unm.edu/~immsec` reinforced that view. When we compared the different types of system calls for the `ps` program that where issued in the normal data and the exploit data, we actually found that they actually used exactly the same system calls, albeit in somewhat different sequences. This certainly does not hold for all possible types of attacks, but probably for the majority. One of the reasons for this is that the goal of the attacker is usually to gain *superuser* access [9], something that is most commonly done by executing (using `execve`) a new shell with superuser access. Up to that point, the program is probably behaving completely normal, which makes it hard to detect. Thus, a very simple but still quite effective intrusion system intended to defend the computer from intrusions with the intension to gain superuser privileges might only need to monitor programs for abnormal `execve` calls.

To confirm this idea with then did the same analysis on the live `wu-ftp` data from the same source. Here we actually found bigger differences between the normal and abnormal data. Three system calls, `unlink`, `getppid`, and `getpgrp` where only found in the exploit data and six other calls where only found in the normal data. One reason that more calls are found in the normal data is that the program might exit or execute another program when it is attacked. The unique calls only found in the abnormal data is the calls that perform the exploit.

6.4 Determinism of Sequences of System Calls

In *information theory* [30] *entropy* is used to measure the uncertainty, or impurity, of collections of data. Formally, the entropy of the data source X is defined as

$$H(X) = \sum_{i}^{N} p_i \log p_i$$

where N is the finite number of states that the data in X can be in and p_i the probability for a variable to be in state x_i. The entropy value is smaller when the data is pure and most variables belong to one state (or class). Another way of seeing entropy is to see it as the number of bits that is needed to represent the state of X.

Program	Unique calls	log_2	Entropy
sendmail	48	5.58	2.66
wu-ftp	48	5.58	2.16
xmms(mp3 player)	53	5.72	2.95
emacs	46	5.52	3.49

Table 6.1: The entropy for the traces of a number of different programs. Unique calls denotes the number of uniquely used system calls, log_2 the number of bits needed to represent one system call in trace if all calls have the same probability, and entropy the entropy of the trace.

Table 6.1 shows the entropy for the system call traces for a number of programs. The first thing to note here is that the average program uses around 50 different system calls. It is also interesting to note that the two server programs, `sendmail` and `wu-ftp` are the two programs with the lowest entropy. This means that these traces are more regular (and hence easier to predict) than for example `xmms` and `emacs`. These results are not surprising since `emacs` is a program that interacts with the user while `sendmail` and `wu-ftp` both are server programs that basically executes a loop that waits for clients to connect to the server.

Following work done in [22], we can measure the determinism and regularity in a trace of system calls by computing the *conditional entropy* of the trace. Let Y be a collection of sequences of system calls all of the length n (the window size). Each sequence is denoted as $(c_1, c_2, \ldots, c_{n-1}, c_n)$ where each c_i is a system call. Moreover, let X be a collection of subsequences where each subsequences is defined as $(c_1, c_2, \ldots, c_{n-1})$. Hence, X is the collection of all subsequences with a window one call shorter than Y. Then the conditional entropy $H(Y|X)$ measures how much uncertainty that remains in a sequence y after we have seen x. More formally, conditional entropy is defined as

$$H(Y|X) = -\sum_{i}^{N} \sum_{j}^{N} p(x_i, y_j) \log p(y_j|p_i)$$

This can be seen as a measure of how hard the data is to model.

To investigate our idea that the sequences of system calls are quite regular and deterministic, we computed the conditional entropy for a number of traces including data from `sendmail`, `wu-ftp`, and `ps` downloaded from `http://www.cs.unm.edu/~immsec` and `xmms` and `emacs` run on our system. The conditional entropy in this case measures how the $n-1$

Figure 6.1: Conditional entropy for normal traces and exploit traces.

first system calls in a sequence determines the nth system call where a result of zero means that it is completely deterministic. The traces for `sendmail` and `wu-ftp` contained 48 different system calls each and the `ps` trace consisted of 22 different calls. Figure 6.1 shows the results for a window size (n) between 2 and 20 for `sendmail`, `wu-ftp`, and `ps` and Figure 6.2 for `xmms` and `emacs`. Here we can see that the conditional entropy for the `wu-ftp` program and `ps` is highly similar. With a window bigger than 4 the conditional entropy is less than 0.1, suggesting that the value of the nth call is highly deterministic when the $n-1$ first system calls have been seen. For `sendmail` the conditional entropy starts of high but gets down to the values of `ps` and `wu-ftp` with a window bigger than six calls. With a window bigger than 12 calls the value is almost 0.0, indicating a highly predictive behaviour. The conditional entropy for the abnormal traces (exploits) are quite similar to the normal traces, the difference is that they are slightly less predictive with small window sizes.

In general, these results are not surprising, `sendmail` is a more complex program than the other two, which makes it harder to predict. In our system the window size is 8, which seems to be a reasonable size considering these results.

6.5 Problems With The Approach and Our System

During this work we have noticed a number of potential weaknesses and problems with this approach to anomaly detection:

- What if the attacker knows that the system is using a system call based anomaly detection system? It might be possible for a hacker to design special attacks, for example buffer overflow strings, that do not alter the trace of system calls. Attacks like this can not be detected by our system.
- Denial of service attacks. Since the protection method of our system is to delay abnormally behaving processes, this can be used for denial of service attacks [9]. A solution to this problem might be to restart important processes when they behave abnormally.

Figure 6.2: Conditional entropy for normal traces and exploit traces.

- When is a profile normal? Even though you wait for week without changes to a profile before it is regarded as normal, how can you be sure? This can lead to problems with false positives.
- Is generalisation a good thing for an anomaly detection system? The risk is that a generalisation might classify a dangerous attack as normal.
- Stability and Performance problems. The implemented system is stable and has been used for the days on our computer. But, there is at least one user program that causes problems. If our system is started while the music player `xmms` is playing music the graphical user interface for that program freezes. We have no idea why.

Chapter 7

Conclusions

In this thesis we have studied how operating systems can be designed to detect and respond to various forms of anomalous behaviour and we have also built a system based on the studied principles. To detect anomalies we analyse the sequences of system calls that user space applications call in the operating system kernel. We argue that this is the best level to place an intrusion detection system since this is the only way for programs to actually access files or hardware on modern computers.

To detect abnormal behaviour profiles are built for each application that models the normal sequences of system calls that the application calls. Thus, the problem of detecting abnormal behaviour is to distinguish the normal behaviour (self), from abnormal behaviour (nonself). There is a number of ways to build the profiles and in this thesis we have tried three different methods; a lookup-table, feed-forward neural networks, and an Elman recurrent neural networks. Results from the experiments show that the Elman network and feed-forward network are too slow to use in an on-line system since they need to execute for each executed system call as a part of the Linux kernel. Hence, speed is extremely important and of the tested methods only the lookup-table method is fast enough. In other experiments we have shown how several commonly used attack methods, including, buffer overflows, format string exploits, and Trojan code, usually change the sequence of system calls that the attacked program executes. Hence it is at least theoretically possible to detect these forms of attacks by monitoring what sequences of system calls applications execute.

There are also a number of problems with the suggested approach to anomaly detection. Maybe the most important is Denial of Service attacks that can exploit the fact that abnormally behaving programs are slowed down. Thus it is possible for an attacker to deny others to use an application by deliberately slowing it down. Another problem is to build good models of normal behaviour. If the model is not good enough it will alarm even though the program is behaving normal or not alarm when the program is behaving abnormal. Some researchers claim that the answer to this problem is generalisation, but how can you know if a generalising profile will not identify abnormal behaviour as normal, since we can not know beforehand what the abnormal behaviour will look like?

Even though there still are some problems with system call based intrusion detection it still has some advantages over other methods such as misuse detection. Two major advantages are that intrusions can be detected in real-time and that it is possible to detect novel attacks. Another major advantage is that the user does not need any specialised knowledge to use the system since it learns itself how the monitored programs normally behaves. There is also no need to analyse the source code and perhaps change the code since all monitoring and logging is done in the operating system.

To conclude we definitely believe that system call based anomaly detection has a future. As computers become more abundant and autonomous they need to use every method they can to survive. Discrimination between self and nonself will most probably be one of them.

Bibliography

[1] Aleph One, Smashing the Stack for Fun and Profit, in *Phrack* issue 49, `http://www.phrack.org`, 1998

[2] W. Ross Ashby, *An Introduction to Cybernetics*, Chapman & Hill, London, 1956, Internet (1999): `http://pcp.vub.ac.be/books/IntroCyb.pdf`

[3] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, O'Reilly, 2001

[4] Neil A. Campbell, *Biology* 4th edition, The Benjamin Cummings Publishing Company, 1996

[5] L. N. de Castro and J. Timmis, Artificial Immune Systems: A Novel Paradigm to Pattern Recognition, in L. Alonso, J Corchado, and C Fyfe editors *Artificial Neural Networks in Pattern Recognition*, University of Paisley, 2002

[6] Dorothy E. Denning, An Intrusion Detection Model, in *Proceedings of the 1986 Symposium on Security and Privacy*, 1986

[7] Patrik D'haesleer, Stephanie Forrest, and Paul Helman, An Immunological Approach to Change Detection: Agorithms, Analysis, and Implications, in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996

[8] Jeffrey L. Elman, Finding Structure in Time, in *Cognitive Science 14*, 1990

[9] Terry Escamilla, *Intrusion Detection*, John Wiley & Sons, 1998

[10] Stephanie Forrest and Stephen Hofmeyr, Information Processing in the Immune System, in Lee A. Segel and Irun R. Cohen editors, *Design Principles for the Immune System and Other Distributed Autonomous Systems*, Oxford University Press, 2001

[11] Stephanie Forrest, A. S. Perelson, L. Allen, and R. Cherukuri, Self-Nonself Discrimination in a Computer, in *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, 1994

[12] Stephanie Forrest, Steven Hofmeyr, Anil Somayaji, and Thomas A. Longstaff, A Sense of Self for Unix Processes, in *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, IEEE Computer Society Press, 1996

[13] Ibrahim F. Haddad, Charles Levert, Makan Pourzandi, and Miroslvaw Zakrzewski, DSI, Secure Carrier-Class Linux, in *Linux Journal*, issue 100, August 2002

[14] Simon Haykin, *Neural Networks* second edition, Prentice-Hall, 1999

[15] Guy H. Helmer, Johnny S. K. Wong, Vasant Honavar, and les Miller, Intelligent Agents for Intrusion Detection, in *Proceedings of the IEEE Information Technology Conference*, 1994

[16] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz, Learning Program Behaviour for Intrusion Detection, in *USENIX Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, 1999

[17] Stephen Hofmeyr and Stephanie Forrest, Architecture for an Artificial Immune System, in *Journal of Evolurionary Computation*, 2000

[18] Florian Kerschbaum, Eugene H. Spafford, and Diego Zamboni, Using Embedded Sensors for Detecting Network Attacks, in *Proceedings of the First ACM Workshop on Intrusion Detection Systems*, 2000

[19] Brian W. Kernigan, Dennis M. Ritchie, *The C Programming Language* second edition, Prentice-Hall, 1988

[20] Gene H. Kim and Eugene H. Spafford, Experiences with Tripwire: Using Integrity Checkers for Intrsuion Detection, Technical Report CSD-TR-94-012, Department of Computer Sciences, Purdue University, 1994

[21] Jungwon Kim and Peter Bentley, The Human Immune System and Network Intrusion Detection, in *Proceedings of the 7th European Congress on Intelligent Techniques and Soft Computing*, 1999

[22] Wenke Lee and Dong Xiang, Information-Theoretic Measures for Anomaly Detection, In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001

[23] McGraw Gary and Felten Edward, Understanding the Keys to Java Security - the Sandbox and Authentication ,*JavaWorld*, May 1997

[24] David Newman, Joel Snyder, and Rodney Thayer, Crying Wolf: False Alarms Hide Attacks, in *Network World*, `http://www.networkworld.com`, June 2002

[25] Lars Olsson, A Sandbox For Processes, Programs, and Users Implemented as a Linux Kernel Module, `http://www.shell.linux.se/~lars/sandbox.ps`, December 2001

[26] sd@sf.cz (anonymous), Linux on-the-fly kernel patching without LKM, in *Phrack* issue 58, `http://www.phrack.org`, 2001

[27] Alexander Rubini, *Writing Linux Device Drivers*, O'Reilly, 2001

[28] scut, Exploiting Format String Vulnerability, `http://www.team-teso.net/articles/formatstring`, March 2001

[29] R. Sekar and P. Uppuluri, Synthesizing Fast Intrusion Prevention/Detection Systems from High Level Specificiations, in *Proceedings 8th Usenix Security Symposium*, August 1999

[30] Claude E. Shannon and Warren Weaver, *The Mathematical Theory of Communication*, University of Illinois Press, 1949

[31] Lee A. Segel and Irun R. Cohen editors, *Design Principles for the Immune System and Other Distributed Autonomous Systems*, Oxford University Press, 2001

[32] Anil Somayaji, *Operating System Stability and Security through Process Homeostasis*, PhD. thesis, The University of New Mexico, July 2002

[33] Andrew S. Tanenbaum and Albert S. Woodhull, *Operating Systems: Design and Implementation*, Prentice-Hall International, 1997

[34] timecop, Root kit SSH 5.0, `http://www.ne.jp/asahi/linux/timecop`, January 2000

[35] Christina Warrender, Stephanie Forrest, Barak Pearlmutter, Detecting Intrusions Using System Calls: Alternative Data Models, in *Proceedings of the IEEE Symposium on Security and Privacy*, 1999

[36] Andreas Wespi, Marc Dacier, and Herve Debar, An Intrusion-Detection System Based on the Teiresias Pattern-Discovery Algorithm, in *EICAR Proceedings*, 1999

Appendix A

Source code

This chapter contains the source code for all the programs described in this thesis. We begin by describing our development method and then installation and usage are described. Before the source code listings there is also a short description of a sandbox module we have also developed as part of this project.

A.1 Development and Testing

Development of kernel modules is quite different from regular applications (see [27] for an excellent overview of kernel debugging and development). One problem is that almost all mistakes (like accessing a null pointer) crashes the whole computer and not just a single application. Debugging is also complex because it might be impossible to write to a terminal and infinite loops also causes the whole computer to freeze. It can also be tricky to test the code since it needs to be inserted in to the kernel and then somehow executed.

Because of these reasons we started by developing and testing as much as possible of the modules in regular programs using test harnesses. We also designed much of the code so that it is possible to use in regular applications by conditional compilation. The symbol `__KERNEL__` is used to decide whether the code should be included for the kernel or not. All important functionality was first tested on function level and then on module level. To manage the source code cvs was used and for all important functions we used regression tests to detect errors caused by new code or changes in old code.

A.2 Installation and Usage

To build the system one simply needs to type `make`. This compiles all files after the `syscalls.c` file has been created by `gensyscallcode.pl`. After compilation the modules `sensor.o`, `homeos.o`, and `sandbox.o` (see below) can be found in the same directory. This directory also contains a number of utility programs including `homeos_prof_saver`, a program used by the modules to save profiles, and `profile_reader`, a program that can be used to examine saved profiles.

To start the system the script `start_homeos` is used. This starts the `sensor.o` module and the `homeos.o` module by inserting them in to the kernel. It also starts the user-space program `homeos_prof_saver` that is used to save profiles to file. To interact with the system the user can then use the `homeosgui` program or by writing and reading directly to the files in the proc filesystem. To stop the system the script `stop_homeos` should be used. This script sends the `save_and_exit` command to the homeos module which saves all profiles before the `homeos_prof_saver` is terminated. Finally the homeos module is removed from the kernel.

A.3 The Sandbox Module

The source code listing also contains the code for a sandbox [23][25] module that can be used to configure what system calls a specific program, process, or user is allowed to execute. This can for example be used as a simple security system as described in Section 6.3 by disallowing the `execve` system call. The module is compiled when the other modules are compiled and can be inserted by typing (as root) `insmod sandbox.o`. The module is a new implementation but uses the same interface as described in [25].

A.4 Make

```
#$Id: Makefile,v

#what compiler to
CC=gcc

#what kernel vers
#change this if y
KERNVERSION =‘una
KERNVERSION =2.4.

# lookahead, ff,
LEARNING = lookah


#flags to use whe
WARN= -Wall -Ws
COPT= -O2 -m486 -
-malign-loops=2 -
DEFINES = -D__KE
DEBUGFLAGS = -DDE
INCLUDES = -I/us
CFLAGS = $(COPT)

 ifeq ($(LEARNING)
   DEFINES += -DFE
 endif
 ifeq ($(LEARNING)
   DEFINES += -DEL
 endif
 ifeq ($(LEARNING)
   DEFINES += -DLO
   DEFINES += -DK
 endif

# Files for the n
HOMEOSSRCS = main
SENSORSRCS = sens
 homeos_profile.c
SANDBOXSRCS = san
HOMEOSHEADERS = p
 profile_io.h hom
SENSORHEADERS = s
 profile_io.h hom
SANDBOXHEADERS =

#utils sources
UTILSRCS= homeos

HOMEOS = homeos.o
SENSOR = sensor.o
SANDBOX = sandbox
HOMEOSOBJS = $(H
SENSOROBJS = $(S


 all: homeosmodule

 homeosmodule: .de

 sensormodule: .de
```

```
    @$(CC) -o $@

.PHONY: clean
#clean everything
clean:
    @echo "......
    rm -vf *.o *~
    .dependsandbo
```

A.5 gensy

```
#!/usr/bin/perl

#
#  gensyscallcode
#  needs the file
#
#  See license in
#
#  $Id: gensyscal
#

open(FILE, "sysca
    or die "can't
print STDERR "rea
%syscallsnums=();
while($line = <FI
    chomp($line);
    ($const, $num
#    print "const
    $syscallsnums
}

print STDERR "rea
while($line = <ST
    ($num, $name)
    $names[$num]
    $name =~ s/sy
    $callconstant
    $line = <STDI
    chop($line);
    $files[$num]
    $line = <STDI
    chop($line);
    $functions[$n
    $line = <STDI
}

print STDERR "sta
#the init_sensor(
print<<HEAD;
/*
 *  syscalls.c -
 *
 *  See license i
 *
 *  \$Id\$
 */

#include <linux/k
#include <linux/i
```

```
#ifdef SANDBOX
\tif(syscall_listc
\t\treturn 0;
#endif
\tif(running)
\t\thomeos_do_sys
\tif(call == __NR
\t\thomeos_exit_p
return 1;
}

/* our syscalls *

HEAD

%map=();
#calls
    for($i = 0;$i
    if($i != 119)
        $ourname
        $ourname

        if($map{$
        $map{$our
        $syscallc
        if($sysca
            $sysca
             $sysc
        }
        print "/*
        print"asm
        $args = $
        $args =~
        $args = $
        print "$a
        }
    }
    }
print "\n\n";


print STDERR "pri
#function calls
$counter = 0;
%map2=();
foreach $function
    if($counter !
    $function =~
    $ourname = $n
    $ourname =~ s
    if($map2{$our
    $map2{$ournam
    $arguments =
    $syscallconst
    if($syscallco
        $syscallc
    }
    print<<FUNCTI
$function
{
\tif(sensor($sysc
FUNCTION
```

```
int init_syscalls
{
\textern long sys
\tunsigned long f

\tif(running){
\t\tINFO("syscall
\t\tusage++;
\t\treturn 0;
\t}

\tINFO("init_sysc

\tusage++;
\tdown(&homeos_lo
\trunning = 1;
\tup(&homeos_lock

\t/* make the cha
\tsave_flags(flag
\tcli();

INIT

for($i = 0;$i < $
    if($i != 119)
    $ourname = $n
    $ourname =~ s
    print "\t$our
    $args = $func
    $args =~ /(\
    $args = $1;
    $syscallconst
    if($syscallco
        $syscallc
    }
    print "$args)
    }
}
#now change the t
print "\n\n\t/*_n
for($i = 0;$i < $
    if($i != 119)
    $ourname = $n
    $ourname =~ s
    $syscallconst
    if($syscallco
        $syscallc
    }
    print "\tsys_
    }
}
print<<ENDINIT;

\t/* now we can s
\trestore_flags(f
\treturn 1;
}

ENDINIT

print STDERR "pri
```

```
#define ADD_PROGR
#define REMOVE_PR
#define REMOVE_PR
#define CONFIG_FI

void homeos_proc
void homeos_proc
#endif
```

A.7 home

```
/*
 * homeos_defs.h
 *
 * See license i
 *
 * $Id: homeos_d
 */

#ifndef _HOMEOS_D
#define _HOMEOS_D

#define HOMEOS_PR

/* undef for less

//#define DEBUG

#define DEBUG_LEV
#define LIGHT_DEB
#define AVERAGE_D
#define STRONG_DE
#define PROC_DEBU
#define MAIN_DEBU
#define PROFILE_D
#define SYSCALL_D

#if defined(DEBUG
#define DEBUGMSG(
".%d:", __LINE__)
#elif defined(DEB
#define DEBUGMSG(
".%d:", __LINE__)
#else
#define DEBUGMSG(
#endif

/* INFO and ERR v
#ifdef __KERNEL__
#define INFO(stri
".%d:", __LINE__)
#define ERR(strin
".%d:", __LINE__)
#else
#define INFO(stri
".%d:", __LINE__)
#define ERR(strin
".%d:", __LINE__)
#endif

#endif
```

```
 * The arrays nee
 * otherwise it w
 * it has somethi
*/
typedef struct ho
    int normal, f
    long normal_t
    int win_size;
    unsigned long
    int anomalies
    unsigned long
    unsigned long
    unsigned long
    unsigned long
    unsigned long
    unsigned long
    unsigned char
    unsigned char
    char filename
    char program[
}homeos_file_pro


typedef struct ho

/**
 * A profile cons
 * and tests, som
 * locking the pr
 */
struct homeos_pr
    int normal, f
    char filename
    char program[
    long normal_t
    int win_size;
    unsigned long
    int anomalies
#ifdef __KERNEL__
    struct semaph
#endif
    homeos_profil
    homeos_profil
};

/**
 * The anomalies
 */
typedef struct ho
    int first, to
    unsigned char
} homeos_locality


typedef struct ho
    int last, len
    unsigned char
} homeos_sequence


typedef struct ho
```

A.9 proc

```
/*
 *  proc_io.h - p
 *
 *  See license i
 *
 *  $Id: proc_io.
 */
#ifndef _PROC_IO_
#define _PROC_IO_

#include <linux/f

#include "homeos

void add_profile_

int open_proc(str
int close_proc(st

/* write operatio
ssize_t write_pro
        size_t
int proc_info_wri
         unsig
int proc_config_w
            uns

int proc_save_pro
            uns
int proc_load_pro
            uns


ssize_t proc_proc
          size_t

/* read operation
ssize_t read_proc
        size_t
int proc_info_rea
         int co
int proc_config_r
           int

ssize_t proc_proc
        size_t

int proc_save_pro
         int co

int proc_load_pro
         int co

/*
int proc_process_
           int
*/
/* permissions */
int proc_permissi
```

A.12 feec

```
/*
 *  feed_forward.
 *
 *  See license i
 *
 *  $Id: feed_for
 */

#ifndef _FEED_FOR
#define _FEED_FOR

#define NUM_INPUT
#define NUM_HIDDE
#define NUM_OUTPU

struct ff_network
    double input_
    double hidden
    double output
    double hidden
    double output

    double learni
};

struct ff_network
void eval_net(str
void free_net(str
#endif
```

A.13 elm

```
/*
 *  elman.h - an
 *
 *  See license i
 *
 *  $Id: elman.h,
 */

#ifndef _ELMAN_H_
#define _ELMAN_H_

#define NUM_INPUT
#define NUM_HIDDE
#define NUM_OUTPU

struct elman_netw
    double input_
    double hidden
    double output
    double contex

    double hidden
    double output
    double contex

    double learni
};
```

```
    int allowscal
    struct u_node
}user_node;

/* information ab
typedef struct p_
    char program[
    int allowscal
    struct p_node
}program_node;

/* information ab
typedef struct pr
    int pid;
    int allowscal
    struct proc_n
}process_node;

#endif
```

A.16 hon

```
/*
 * homeos_utils.
 *
 * See license i
 *
 * $Id: homeos_
 */

char *homeos_get.
```

A.17 mai

```
/*
 * homeos - an i
 *
 *
 * Copyright (C)
 *
 * This program
 * it under the
 * the Free Soft
 * (at your opti
 *
 * This program
 * but WITHOUT A
 * MERCHANTABILI
 * GNU General P
 *
 * You should ha
 * along with th
 * Foundation, I
 *
 *
 * Contact me at
 * lars@shell.li
 *
 * or by snailma
 * Lars Olsson
```

```
 *
 * Release the se
 * from the proc
 *
 * Returns zero o
 */
void cleanup_home
{
    INFO("enter_c

    release_senso
    homeos_proc_

    INFO("exit_cl
}

/* macros to tell
module_init(init_
module_exit(clean
```

A.18 pro

```
/*
 *  proc_interfac
 *
 *  See license i
 *
 *  $Id: proc_int
 */

#include <linux/f
#include <linux/m
#include <linux/c
#include <linux/p
#include <asm/uac

#include "homeos
#include "proc_in
#include "proc_io
#include "homeos
#include "profile

/* rootdir name (
#define ROOT_DIR_N

/* subdir names (
#define PROCESSES

#define ADD_PROGI
#define REMOVE_PR
#define REMOVE_PI
#define SAVE_PRO
#define LOAD_PRO

/* info file name
#define INFO_FILE
#define CONFIG_FI

static struct pro
static struct pro
static struct pro
```

```
 * del_process −
 * @hstate: the p
 *
 * Deletes the pr
 */
void del_process(
{
    char name[6];
    int len;

    DEBUGMSG("del
    len = sprintf
    name[len] = '
    DEBUGMSG("nam
    remove_proc_e
}
/**
 * homeos_proc_r
 *
 * Register all e
 * This function
 * of init_module
 */
void homeos_proc
{
    /* Create the
    root_dir = pr

    if(root_dir){
        root_dir−

        // Create
        info_entr



        info_entr
        info_entr
        info_entr
        info_entr

        add_progr




        add_progr
        add_progr
        add_progr
        add_progr


        remove_pr




        remove_pr
        remove_pr
```

```
    homeos_unreg
    homeos_unreg
    homeos_unreg

    if(root_dir){
        if(add_pr
            remo
        }
        if(remove
            remo
        }
        if(remov
            remo
        }
        if(save_p
            remo
        if(info_e
            remo
        }
        if(config
            remo
        }
        if(proces
            remo
        }
        remove_pr
    }
    else {
        DEBUGMSG(
    }
}
```

A.19 pro

```
/*
 *  proc_io.c - p
 *
 *  See license i
 *
 *  $Id: proc_io.
 */

#include <linux/m
#include <linux/c
#include <linux/s
#include <asm/uac
#include <linux/p
#include <linux/s
#include <asm/sem

#include "homeos
#include "proc_in
#include "proc_io
#include "homeos
#include "sensor.


#define MESSAGE_L

#ifdef DEBUGMSG
#define MOD_INC_U
```

```
/**
 * write_proc - w
 *
 * Should never b
 * only read, doe
 *
 * Returns the nu
 */
ssize_t write_pro
            const
            size_
            loff_
{
    int i;
    char message[

    for(i = 0; i
        get_user(

    return i;
}

/**
 * proc_config_wr
 *
 * Should never b
 * add_program, r
 * calls the corr
 *
 * Returns the nu
 */
ssize_t proc_con
            unsig
{
    const unsigne
    char read_str
    unsigned int

    memset(read_s

    char_to_read
    copy_from_use

    /* i - 1 beca
       newline if
    if(read_strin
        read_stri
    else
      read_string

    DEBUGMSG("wri
    DEBUGMSG("cor

    /* addprogram
    if(!strcmp(fi
        add_progra
    else if(!strc
        remove_pr
    else if(!strc
        remove_pr
    else if(!strc
        homeos_co
```

```
    DEBUGMSG("pid

    if(!strcmp(m
        homeos_t
    else if(!strc
        homeos_s
    else if(!strc
        homeos_n
    else
        ERR("proc

    return i;
}

/**
 * proc_info_writ
 *
 * Should never b
 * only read, doe
 *
 * Returns the nu
 */
ssize_t proc_info
            unsig
{
    const unsigne
    char read_str
    unsigned int

    memset(read_s

    char_to_read
    copy_from_use

    return count;
}


/**
 * read_proc - re
 *
 * Should never b
 * only write an
 *
 * Returns the nu
 */
ssize_t read_proc
            char
            size_
            loff_
{
    char message[
    int bytes_wri
    static int fi

    if(finished){
        finished
        return 0;
    }

    bytes_written
```

```
        prof->tra
        prof->tes
        prof->tes
        prof->tes
}


/**
 * proc_save_pro
 *
 * this is a stat
 * profile depend
 * and line varia
 *
 * if there is no
 * to save:
 *
 * if state == HE
 * change state t
 * if state == TR
 * line < NUM_OF_
 * when line == N
 * to 0 and state
 *
 * if state == TE
 * line < NUM_OF_
 * when line == N
 * to 0 and state
 * by doing up on
 *
 * Returns bytes
 */
ssize_t proc_save
          int co
{
        unsigned
    char *header;

    while(profile
        interrupt
        /* someon
        if(profil
            DEBUG
            *eof=
            retur
        }
    }

    switch(state)
    case HEADER:
        header =
        create_he
        bytes_wri

        free_page
        state = T
        line = 0;
        break;
    case TRAIN:
        memcpy(pa
        bytes_wri
        line++;
```

```
 * process with t
 *
 * Returns the nu
 */
ssize_t proc_proc


{
    char message[
    int bytes_wri
    static int fi
    homeos_task_s

    if(finished){
        finished
        return 0;
    }
    if(strlen(pro
        pid = my
        hstate =
        if(hstate
            bytes




        }
        else
            bytes

    }
    else
        bytes_wri

    message[bytes

    finished = 1;
```

```
}
```

A.20 sens

```
/*
 *  sensor_module
 *
 *  See license i
 *
 *  $Id: sensor_m
 */

#ifndef __KERNEL
#define __KERNEL
#endif

#include <linux/m
#include <linux/c

//#if defined(CON
//#define MODVER
//#include <linux
//#endif

#include <linux/k
#include <linux/i
#include <linux/u
#include <linux/s
#include <linux/p
#include <asm/uac
#include <linux/s
#include <linux/s

#include "homeos
#include "sensor.
#include "syscall
#include "homeos
#include "proc_io
#include "homeos

/*
 * Macros for des
 * Used by the co
 */
MODULE_DESCRIPTIO
MODULE_AUTHOR("la

#define SAVE_AND_

static void free_

/**
 * The list of pr
 */
static homeos_pro

/**
 * my_atoi - a at
 * @s: A pointer
 *
 * Parse the int
```

```
    temp = progra
    prev = progra
    while(temp->n
        temp = te
        prev = pr
    }
    if(temp->next
        prev->nex
        return;
    }
    prev->next =

}

/**
 * get_program_fr
 * @program: the
 *
 * Returns NULL i
 */
static homeos_pro
{
    homeos_progra

    for(temp = pr
        if(!strcm
            retur
    return NULL;

}

/**
 * add_program_t
 * @prog: the pro
 *
 * Adds the progra
 */
static void add_p
{
    homeos_progra

    if(prog == NU
        DEBUGMSG(
        return;
    }
    DEBUGMSG("add
    if(programs =
        programs
    else {
        while(tem
            temp
        temp->nex
    }
}

/**
 * profile_progra
 * @program: the
 *
 * Returns 1 if t
 * otherwise 0
```

```
void remove_progra
{
    homeos_progra
    INFO("remove_

    prog = get_pr
    if(prog != NU
        remove_p
        free_prog
    }
    else
        INFO("uns
}

/**
 * remove_process
 * @pidstr: the p
 */
void remove_proce
{
    int pid;
    INFO("remove_

    pid = my_atoi
    homeos_exit_p
}

/**
 * init_sensor -
 *
 * Init the sysca
 */
void init_sensor(
{
    INFO("enter_i

    programs = NU

    homeos_init_

    if(init_sysca
        INFO("ok,
    }else{
        INFO("no_
    }

    INFO("exit_in
}

/**
 * release_sensor
 *
 * Release the sy
 */
void release_sens
{
    INFO("enter_r

    if(release_sy
        INFO("rel
    }else{
        INFO("can
```

```
#include <asm-i38
#include <asm/sem
#include <asm/ato
#include <asm-i38
#include <asm-i38
#include <asm-i38
#include <linux/u
#include <linux/t
#include <linux/t
#include <linux/n
#include <linux/u
#include <linux/u
#include <linux/n
#include <linux/t
#include <linux/s
#include <linux/s

#include "homeos
#include "sensor.
#include "homeos

struct mmap_arg_
    unsigned long
    unsigned long
    unsigned long
    unsigned long
    unsigned long
    unsigned long
};

struct sel_arg_st
    unsigned long
    fd_set *inp,
    struct timeva
};

struct semaphore
DECLARE_MUTEX(hon
static int runnin
static int usage

int (*syscall_lis

void register_sys
{
    syscall_lister
}

void unregister_s
{
    if(syscall_lis
    syscall_lister
}

static int sensor
{
#ifdef SANDBOX
    if(syscall_lis
        return 0;
#endif
    if(running)
        homeos_d
```

```
asmlinkage unsign

/*  __NR_getpid *
asmlinkage unsign

/*  __NR_mount */
asmlinkage unsign

/*  __NR_umount *
asmlinkage unsign

/*  __NR_setuid *
asmlinkage unsign

/*  __NR_getuid *
asmlinkage unsign

/*  __NR_stime */
asmlinkage unsign

/*  __NR_ptrace *
asmlinkage unsign

/*  __NR_alarm */
asmlinkage unsign

/*  __NR_oldfstat
asmlinkage unsign

/*  __NR_pause */
asmlinkage unsign

/*  __NR_utime */
asmlinkage unsign

/*  __NR_access *
asmlinkage unsign

/*  __NR_nice */
asmlinkage unsign

/*  __NR_sync */
asmlinkage unsign

/*  __NR_kill */
asmlinkage unsign

/*  __NR_rename *
asmlinkage unsign

/*  __NR_mkdir */
asmlinkage unsign

/*  __NR_rmdir */
asmlinkage unsign

/*  __NR_dup */
asmlinkage unsign

/*  __NR_pipe */
asmlinkage unsign
```

```
/*  __NR_sgetmask
asmlinkage unsign

/*  __NR_ssetmask
asmlinkage unsign

/*  __NR_setreuid
asmlinkage unsign

/*  __NR_setregid
asmlinkage unsign

/*  __NR_sigsuspe
asmlinkage unsign

/*  __NR_sigpendi
asmlinkage unsign

/*  __NR_sethostn
asmlinkage unsign

/*  __NR_setrlimi
asmlinkage unsign

/*  __NR_getrlimi
asmlinkage unsign

/*  __NR_getrusag
asmlinkage unsign

/*  __NR_gettimec
asmlinkage unsign

/*  __NR_settimeo
asmlinkage unsign

/*  __NR_getgroup
asmlinkage unsign

/*  __NR_setgroup
asmlinkage unsign

/*  __NR_select *
asmlinkage unsign

/*  __NR_symlink
asmlinkage unsign

/*  __NR_oldlstat
asmlinkage unsign

/*  __NR_readlink
asmlinkage unsign

/*  __NR_uselib *
asmlinkage unsign

/*  __NR_swapon *
asmlinkage unsign

/*  __NR_reboot *
```

```
/*  __NR_iopl */
asmlinkage unsign

/*  __NR_vhangup
asmlinkage unsign

/*  __NR_vm86old
asmlinkage unsign

/*  __NR_wait4 */
asmlinkage unsign

/*  __NR_swapoff
asmlinkage unsign

/*  __NR_sysinfo
asmlinkage unsign

/*  __NR_ipc */
asmlinkage unsign

/*  __NR_fsync */
asmlinkage unsign

/*  __NR_clone */
asmlinkage unsign

/*  __NR_setdomai
asmlinkage unsign

/*  __NR_uname */
asmlinkage unsign

/*  __NR_modify_l
asmlinkage unsign

/*  __NR_adjtimex
asmlinkage unsign

/*  __NR_mprotect
asmlinkage unsign

/*  __NR_sigprocm
asmlinkage unsign

/*  __NR_create_m
asmlinkage unsign

/*  __NR_init_moc
asmlinkage unsign

/*  __NR_delete_m
asmlinkage unsign

/*  __NR_get_kern
asmlinkage unsign

/*  __NR_quotactl
asmlinkage unsign

/*  __NR_getpgid
asmlinkage unsign
```

```
asmlinkage unsign

/*  __NR_sched_ge
asmlinkage unsign

/*  __NR_sched_se
asmlinkage unsign

/*  __NR_sched_ge
asmlinkage unsign

/*  __NR_sched_yi
asmlinkage unsign

/*  __NR_sched_ge
asmlinkage unsign

/*  __NR_sched_ge
asmlinkage unsign

/*  __NR_sched_r
asmlinkage unsign

/*  __NR_nanoslee
asmlinkage unsign

/*  __NR_mremap *
asmlinkage unsign

/*  __NR_setresui
asmlinkage unsign

/*  __NR_getresui
asmlinkage unsign

/*  __NR_vm86 */
asmlinkage unsign

/*  __NR_query_mo
asmlinkage unsign

/*  __NR_poll */
asmlinkage unsign

/*  __NR_nfsservc
asmlinkage unsign

/*  __NR_setresgi
asmlinkage unsign

/*  __NR_getresgi
asmlinkage unsign

/*  __NR_prctl */
asmlinkage unsign

/*  __NR_rt_sigre
asmlinkage unsign

/*  __NR_rt_sigac
asmlinkage unsign
```

```
/*  __NR_lchown32
asmlinkage unsign

/*  __NR_getuid32
asmlinkage unsign

/*  __NR_getgid32
asmlinkage unsign

/*  __NR_geteuid3
asmlinkage unsign

/*  __NR_getegid3
asmlinkage unsign

/*  __NR_setreuid
asmlinkage unsign

/*  __NR_setregid
asmlinkage unsign

/*  __NR_getgroup
asmlinkage unsign

/*  __NR_setgroup
asmlinkage unsign

/*  __NR_fchown32
asmlinkage unsign

/*  __NR_setresui
asmlinkage unsign

/*  __NR_getresui
asmlinkage unsign

/*  __NR_setresgi
asmlinkage unsign

/*  __NR_getresgi
asmlinkage unsign

/*  __NR_chown32
asmlinkage unsign

/*  __NR_setuid32
asmlinkage unsign

/*  __NR_setgid32
asmlinkage unsign

/*  __NR_setfsuid
asmlinkage unsign

/*  __NR_setfsgid
asmlinkage unsign

/*  __NR_pivot_ro
asmlinkage unsign

/*  __NR_mincore
```

```
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage int ho
{
    int error = 1
    char *filenam

    filename = ge
    error = PTR_E
    if(IS_ERR(file
        goto out;

    if(profile_pro
        homeos_ac

    if(sensor(__N
        error=do_
    if(error == 0
        current->
```

```
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage int ho
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage unsign
{
    if(sensor(__N
        return sa
    else
        return -E
}
```

```
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage int ho
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage unsign
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
```

```
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage int ho
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}
```

```
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
```

```
    if(sensor(__N
        return ol
    else
        return -E
}

asmlinkage int ho
{
    if(sensor(__N
        return ol
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}
```

```
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage int ho
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage int ho
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage int ho
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
```

```
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage unsign
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}
```

```
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage ssize_
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage ssize_
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
```

```
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}
```

```
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage int ho
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
```

```
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage ssize_
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage int ho
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}
```

```
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
```

```
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage long h
{
    if(sensor(__N
        return sa
    else
        return -E
}

asmlinkage ssize_
{
    if(sensor(__N
        return sa
    else
        return -E
}


void init_sys(voi
{
    INFO("init_sy
}

int init_syscalls
{
    extern long s
    unsigned long

    if(running){
        INFO("sys
        usage++;
        return 0;
    }

    INFO("init_sy

    usage++;
    down(&homeos_
    running = 1;
    up(&homeos_lo
```

```
saved_sys_ni
saved_sys_oldu
saved_sys_uma
saved_sys_chr
saved_sys_ust
saved_sys_dup
saved_sys_get
saved_sys_get
saved_sys_set
saved_sys_sig
saved_sys_sge
saved_sys_sse
saved_sys_set
saved_sys_set
saved_sys_sig
saved_sys_sig
saved_sys_setl
saved_sys_set
saved_sys_old
saved_sys_get
saved_sys_get
saved_sys_set
saved_sys_get
saved_sys_set
old_select =
saved_sys_sym
saved_sys_lst
saved_sys_rea
saved_sys_use
saved_sys_swa
saved_sys_reb
old_readdir =
old_mmap = (u
saved_sys_munr
saved_sys_tru
saved_sys_ftr
saved_sys_fchr
saved_sys_fcho
saved_sys_get
saved_sys_set
saved_sys_ni
saved_sys_sta
saved_sys_fst
saved_sys_iop
saved_sys_soc
saved_sys_sys
saved_sys_set
saved_sys_get
saved_sys_new
saved_sys_nev
saved_sys_nev
saved_sys_unar
saved_sys_iop
saved_sys_vha
saved_sys_ni
saved_sys_vm8
saved_sys_wai
saved_sys_swa
saved_sys_sys
saved_sys_ipc
saved_sys_fsy
saved_sys_clo
```

```
saved_sys_get
saved_sys_cap
saved_sys_cap
saved_sys_sig
saved_sys_ser
saved_sys_ni
saved_sys_ni
saved_sys_vfo
saved_sys_get
saved_sys_mma
saved_sys_tru
saved_sys_ftr
saved_sys_sta
saved_sys_lst
saved_sys_fst
saved_sys_lch
saved_sys_get
saved_sys_get
saved_sys_get
saved_sys_get
saved_sys_set
saved_sys_set
saved_sys_get
saved_sys_set
saved_sys_fch
saved_sys_set
saved_sys_get
saved_sys_set
saved_sys_get
saved_sys_cho
saved_sys_set
saved_sys_set
saved_sys_set
saved_sys_set
saved_sys_piv
saved_sys_min
saved_sys_mac
saved_sys_get
saved_sys_fcn
saved_sys_ni
saved_sys_ni
saved_sys_get
saved_sys_rea
saved_sys_ni


/* now change

sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
```

```
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
```

```
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl

    /* now we can
    restore_flags
    return 1;
}


int release_sysca
{
    extern long s
    unsigned long

    if(usage > 1)
        INFO("sys
        usage--;
        return 0;
    }
    if(usage < 1)
    return 0;

    INFO("release

    down(&homeos
    running = 0;
    up(&homeos_lo

    /* make this
    save_flags(fl
    cli();

    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
```

```
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
sys_call_tabl
```

```
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl
    sys_call_tabl

    /* now we can
    restore_flags
    return 1;
}
```

A.22 pro

```
/*
 *  profile_io.c
 *
 *  See license i
 *
 *  $Id: profile_
 */

#ifdef __KERNEL__
#include <linux/f
#include <linux/s
#include <linux/s
#include <linux/v
#include <asm/uac
#include <linux/f
#include <asm/ato
#include <asm/sem
#include <linux/s
#include <linux/s
```

```
        while ((*
                *

                s
                d
                i
        }
        *dest = '

        return re
}

/**
 * homeos_init_p
 * @program: the
 * @filename: the
 *
 * Creates a new
 *
 * Returns the ne
 */
static homeos_pr

{
    homeos_profil
    int i;

    DEBUGMSG("ini

#ifdef __KERNEL__
    prof = (home
#else
    prof = (home
#endif
    strcpy(prof->
    strcpy(prof->

    for(i = 0;i <
      memset(prof
      memset(prof
    }
    return prof;
}


/**
 * copy_fprof_to_
 * @profile: the
 * @fprof: the fi
 *
 * Copies the con
 */
static void copy

{
    int i;

    DEBUGMSG("cop
    profile->norm
    profile->froz
    profile->norm
    profile->coun
```

```
    profile->next
    sema_init(&pr

    /*
       save the c
       and set it
       so that we
       routines.
    */
    orgsfs = get_
    set_fs(KERNE

    fprof = (hom

    fi = filp_ope
    if(IS_ERR(fi)
        DEBUGMSG(
    }
    else {
        if(fi->f_
            retva


            /* fi
            if(re
                D
            }
            else
                /
                D

                c
            }
        }
        filp_clos
    }

    set_fs(orgsfs
    current->fsui
    current->fsgi

    vfree(fprof);
    free_page((un

    return profil
}

/**
 * homeos_save_p
 * @profile: the
 *
 * Adds the profi
 */
int homeos_save_
{
    INFO("saving_

    if(save_profi
        save_pro
        INFO("sav
    }
    else
```

```
    homeos_file_p

    char *filenam
    fprof = (hom
    INFO("no_kern

    homeos_profil

    new = homeos
    if(new != NUL
        DEBUGMSG(
    new->count =
    new->frozen =
    new->normal =
    new->next = N
    new->anomalie
    new->normal_t
    new->train.la
    new->train.tr
    new->train.se
    for(i = 0;i <
        memset(ne
        memset(ne
    }


    INFO("trying_
    if((file = fo
        printf("c
        //return
    }

    if(file && (f
        printf("c
        fclose(fi
    }
    else if(file)
        fclose(fi
        copy_fpr
        new->next
    }
    free(filename

    return new;
}
#endif
```

A.23 hon

```
/*
 *  homeos_profil
 *
 *  See license i
 *
 *  $Id: homeos_
 */

#ifndef __KERNEL
#include <stdio.h
#endif
```

```
static void home
static void home
static void home
static inline voi

static void home
static void home
static inline voi

static void home
static void home
static inline int
static inline voi
static void home
static inline voi
static inline voi

/*
 * functions incl
 */
static inline int

static inline voi
static inline voi
static void home
            h

/*
 * Include traini
 *
 */
#ifdef LOOKAHEAD
#include "lookah
#endif
#ifdef FEEDFORWAR
#include "feed_fo
#endif
#ifdef ELMAN
#include "elman.c
#endif


/**
 * homeos_registe
 * @add_handler:
 *
 * Observe that t
 */
void homeos_regi

{
    add_handler_f
}

/**
 * homeos_regist
 * @del_handler:
 *
 * Observe that t
 */
void homeos_regi
```

```
 */
void homeos_sens
{
    homeos_task_s

    hstate = hom
    if(hstate ==
        ERR("tryi
        return;
    }
#ifdef __KERNEL__
    down(&(hstate
#endif
    sensitize_pro
#ifdef __KERNEL__
    up(&(hstate->
#endif
}

/**
 * homeos_normal
 * @pid: the proc
 *
 * Normalize proc
 */
void homeos_norm
{
    homeos_task_s

    hstate = hom
    if(hstate ==
        ERR("tryi
        return;
    }

    if(!hstate->p
        homeos_st
    else
        ERR("tryi
}



/**
 * sensitize_proc
 * @hstate: the p
 *
 * To sensitize a
 * data is reset.
 * should not be
 * so that the pr
 */
static inline voi
{
    reset_profile
}

/**
 * tolerize_proce
 * @hstate: the p
 *
 * When a process
```

```
            temp
        temp->nex
    }
}

/**
 * homeos_find_pr
 * @pid: the pid
 *
 * Returns NULL i
 */
inline homeos_ta
{
    homeos_task_s

    if(processes
        return NU
    }

    for(temp = pr
        ;

    return temp;
}

/**
 * homeos_init_p
 *
 * At the moment
 */
void homeos_init
{
    INFO("init_pr
    processes = N
    programs = NU
    running = 1;
}

/**
 * homeos_releas
 *
 * Frees all proc
 * from sleeping.
 */
void homeos_rele
{
    DEBUGMSG("rel

    running = 0;

    homeos_free_p
    homeos_free_p
    processes = N
    programs = NU
}

/**
 * homeos_save_a
 *
 */
void homeos_save
{
```

```
    temp = proces
    prev = proces
    while(temp->n
        temp = te
        prev = pr
    }
    if(temp == NU
        prev->nex
        return;
    }
    prev->next =
}

/**
 * homeos_exit_pr
 * @pid: the pid
 *
 * Saves the pro
 * the task_state
 */
void homeos_exit_
{
    homeos_task_s

    hstate = hom
    if(hstate ==
        /* ERR("h
        return;
    }
    homeos_save_

    homeos_free_p
}

/**
 * homeos_free_pr
 *
 * Frees all prog
 */
static void home
{
    homeos_profil

    INFO("free_pr

    for(p = progr
        temp = p-
        homeos_fr
    }
}

/**
 * homeos_free_pr
 *
 * Frees all proc
 */
static void home
{
    homeos_task_s

    INFO("free_pr
```

```
 * homeos_add_pro
 * @pid: the pid
 * @program: the
 *
 * Creates a new
 * adds to the ta
 */
void homeos_add_p
{
    homeos_task_s
   DEBUGMSG("hom
#ifndef __KERNEL
    hstate = (hom
#else
    hstate = (hom
#endif
    hstate->pid =

    homeos_get_fu

    strcpy(hstate
    homeos_init_t

    add_to_task_l
}



/**
 * homeos_set_run
 * @run: should b
 *
 */
void homeos_set_r
{
    running = run
}


/**
 * homeos_add_ca
 * @seq: the sequ
 * @value: the th
 *
 * Note: this fun
 */
static inline voi

{
        seq->last
        seq->seq[
}

/**
 * homeos_init_se
 * @seq: the sequ
 *
 */
static void home
{
    seq->last = M
    seq->length =
```

```
/**
 * homeos_add_anc
 * @hstate: the t
 * @val: the numb
 *
 * Note: this fun
 */
static inline voi

{
        int i = (

        if (val >
                h
                i




                }
        } else if
                h
                h
        }
        hstate->l
}

/**
 * get_profiles -
 *
 * Returns a poin
 */
homeos_profile *g
{
    return progra
}


#ifdef __KERNEL__
/**
 * homeos_do_dela
 * @delay: number
 * @hstate: the p
 *
 * Loops until th
 * is no longer 1
 */
static void home
                h
{
    hstate->delay
    while(hstate-
        current->
        schedule_
        hstate->d
    }
    hstate->delay
}
#endif

#ifndef __KERNEL
/* do nothing if
```

```
    train->train_

    if(homeos_ch
        if(profil
            profi
        homeos_ac

        train->se
        train->la
    }
    else {
        unsigned

        train->la

        if(profil
            retur

        normal_co
            train

        if(norma
           ((trai
            (norm
            profi
#ifdef __KERNEL_
            profi
              hor
#endif
             INFO(

           }
    }
}

/**
 * homeos_stop_m
 * @hstate: the p
 *
 */
void homeos_stop_
{
    hstate->profi
}

/**
 * homeos_start_m
 * @hstate: the p
 *
 * Starts to mon
 * flag to 1 and
 * test profile.
 */
static void home
{
    int i;
    homeos_profil
    homeos_profil
    homeos_profil

    test->last_mo
```

```
            lfc_c

            if(lf
                I


                s
                h
            }

            if(hs
                I



                t
            }
        }
        /*
          we must
          no othe
          we won'
        */
#ifdef __KERNEL__
        up(&hstat
        up(&hstat
#endif
        if(lfc_co
            home


    }
    return anoma
}
```

A.24 look

```
/*
 *  lookahead_pr
 *
 *  See license i
 *
 *  $Id: lookahea
 */

#include "homeos

/**
 * reset_profile_
 * @data: the pro
 *
 */
static inline voi
{
    int i;

    for(i = 0; i
        memset(da
}

/**
```

```
        data->seq

    reset_profile
}
```

A.25 hon

```
/*
 *  homeos_prof_s
 *
 *  See license i
 *
 *  $Id: homeos_
 */

#include <stdio.h
#include <errno.h
#include <string.
#include <sys/sta

#include "homeos


int get_next_int(
{
    char tmp[100]
    char *next;

    next = strstr
    memset(tmp, 0
    memcpy(tmp, h

    return atoi(t
}

int get_next_long
{
    char tmp[100]
    char *next;

    next = strstr
    memset(tmp, 0
    memcpy(tmp, h

    return atol(t
}

void parse_header
{
    char *next, *
    char tmp[100]

    /*
      filename:nc
      train_last_m
      test_last_m

      %s:%d:%d:%l
    */

    next = strstr
```

```
        printf("%
        fprof->fi
        fprof->no
        fprof->fr
        fprof->no
        fprof->co
        fprof->ar
        fprof->tr
        fprof->tr
        fprof->tr
        fprof->te
        fprof->te
        fprof->te
}

int main(void)
{
    FILE *file;
    unsigned char
    homeos_file_p
    int len;
    int i,j;
    char *exit =

    /* loop and s

    while(1){
        if((file
           ==NUL
            print
            retur
        }

        fprof = (
        //printf(

        fgets(hea
        fclose(fi

        /*
           just f
           should
        */
        header_bu
        parse_hea

        //print_h

        create_di

        for(i = 0
            if((f
               ==
                p
                r
            }
            len =

            fclos
        }

        for(i = 0
```

```
MODULE_DESCRIPTIO
MODULE_AUTHOR("la

#define ROOT_DIR_N
#define ADD_PROGR
#define REMOVE_PRO
#define ADD_PROCI
#define REMOVE_PR
#define ADD_USER
#define REMOVE_US
#define INFO_FILE

static struct pro
static struct pro
static struct pro
static struct pro
static struct pro
static struct pro
static struct pro
static struct pro

/*****************

/* atoi */
static int my_ato
{
    int res = 0;
    int mul = 1;
    char *ptr;

    for(ptr = str
    {
        if(*ptr <
            retur
        res += (*
        mul *= 10
    }

    return res;
}

/* return -1 if n
int get_next_int(
{
    char tmp[100]
    char *next;
    int ret = -1;

    next = strstr
    if(next == NU
        return -1
    memset(tmp, 0
    memcpy(tmp, h
    ret = my_atoi

    return ret;
}
/***** program, p
```

```
    DEBUGMSG("ad
    if(process_no
        process_n
    else {
        while(tem
            temp
        temp->nex
    }
}

static void remo
{
    process_node

    if(process_no
        ERR("proc
        return;
    }

    if(process_no
        process_n
        return;
    }

    temp = proces
    prev = proces
    while(temp->n
        temp = te
        prev = pr
    }
    if(temp->next
        prev->nex
        return;
    }
    prev->next =

}


static void add_
{
    user_node *te

    if(user == NU
        DEBUGMSG(
        return;
    }
    DEBUGMSG("ad
    if(user_nodes
        user_node
    else {
        while(tem
            temp
        temp->nex
    }
}

static void remo
{
    user_node *te

    if(user_nodes
```

```
/*
 * Returns 1 if t
 * to do the acti
 */
static int proces
{
    process_node

    for(;tmp != N
        if(tmp->p
            if(tm
                r
            else
                r
        }
    }

    return 1;
}

/* returns one if
int sandbox_allov
{
  float d1 = 1.56
  float d2 = 0.21

  float d3 = d1 +
  if(d3 > 4)
    printk("d");

  if(!strcmp("tes
    INFO("call:%d
    printk("%d\n"
  }
    if(user_actio
        program_ac
        process_ac
         return 1;
    else
         return 0;
}

static void free_
{
    program_node

    DEBUGMSG("fre

    for(p = progr
        temp = p-
        kfree(p);
    }
}

static void free_
{
    process_node

    DEBUGMSG("fre

    for(p = proce
```

```
    while(next){
        int call
        DEBUGMSG(
        if(call =
            break
        if(call >
            calls
        next = st
    }
}

static void add_p
{
    program_node
    char *next =
    char tmp[256]

    next = strstr
    memcpy(tmp, s
    tmp[next - st
    DEBUGMSG("add

    node = get_pr
    if(node == NU
        node = (p
        memcpy(no
        set_allow
        node->nex
        add_prog
    }
    else{
//      next = st
        set_allow
    }
    if(program_no
        DEBUGMSG(
}

static void remov
{
    program_node

    prog = get_pr

    if(prog == NU
        ERR("tryi
        return;
    }
    remove_progra
    kfree(prog);
}


static void add_p
{
    process_node
    char *next =
    char tmp[256]
    int pid = -1;

    next = strstr
```

```
    }
    else{
        set_allow
    }
    if(user_nodes
        DEBUGMSG(
}

static void remo
{
    int uid = -1;
    user_node *pr

    uid = my_atoi

    DEBUGMSG("rem

    proc = get_us

    if(proc == NU
        ERR("tryi
        return;
    }
    remove_user_
    kfree(proc);
}


/************ en


/**************

ssize_t proc_info
        unsig
{
    const unsigne
    char read_str
    unsigned int

    memset(read_s

    char_to_read
    copy_from_use

    return count;
}

ssize_t proc_info
        int co
{
        unsigned
    program_node
    process_node
    user_node *ut
    int i;

    bytes_written


    for(; tmp; tm
        bytes_wri
```

```
    if(read_strin
        read_stri
    else
      read_string

    DEBUGMSG("wri
    DEBUGMSG("cor

    /* addprogram
    if(!strcmp(fi
        add_progra
    else if(!strcr
        remove_pr
    else if(!strcr
        remove_pr
    else if(!strcr
        add_proce
    else if(!strcr
        remove_us
    else if(!strcr
        add_user(
    else
        ERR("Shou

    return count;
}


/*
 * Called when so
 * the entry in t
 * to do everythi
 * write.
 */
static int proc_p
{
    if(op == 4 |
        return 0;

    return -EACCE
}

/*
 * Called when so
 * Increments the
 * to unload the
 */
static int module
{
    MOD_INC_USE_CO

    return 0;
}
/*
 * Called when so
 * Decrements the
 */
static int modul
{
    MOD_DEC_USE_CO

    return 0;
```

```
        remove_p
        remove_p
        remove_p
        remove_p

        add_proc



        add_proc
        add_proc
        add_proc
        add_proc


        remove_u



        remove_u
        remove_u
        remove_u
        remove_u

        add_user_



        add_user_
        add_user_
        add_user_
        add_user_


    }

    register_sysc

    init_sys();

    if(init_sysca
        INFO("sys
    }
    else{
        INFO("sys
    }
    INFO("exit_in

    return 0;
}

/*
 * Here's our exi
 *
 * We put back th
```

```
 *  See license i
 *
 *  $Id: homeos_
 */
#ifdef __KERNEL__
#include <linux/k
#include <linux/i
#include <linux/u
#include <linux/s
#include <linux/p
#include <asm/uac
#include <linux/s
#include <linux/s
#include <linux/v
#include <linux/f
#include <linux/p
#include <linux/s
#endif

#include "homeos

#ifdef __KERNEL__
char *homeos_get
{
    struct dentry
    struct vfsmou
    struct mm_str
    struct vm_are
    int result =
    struct task_s

    task_lock(tas
    mm = task->mm
    if (mm)
        atomic_in
    task_unlock(t

    down_read(&mm
    vma = mm->mma
    while (vma) {
        if ((vma-
            vma->
            mnt =
            dentr
            resul
            break
        }
        vma = vma
    }
    up_read(&mm->

    strcpy(buf, d

    return buf;
}

#else

char *homeos_get
{
    return buf;
}
```

```
    scrollbar $f.
        -command
    grid $f.list
    grid rowconfi
    return $f.lis
}

proc Normalize {
    puts stdout n
}

proc Sensitize {
    puts stdout s
}



proc Show_Menu {
    set popup [tk
    $popup add co
    $popup add co
}

proc List_Select
    frame $parent
    set choices [
        -width 10
        -selectmo
    pack $parent.

    bind $choices
        [list Sho

    foreach x $va
    $choices inse
    }
    return $choic
}

proc Write_To_Hom
    set fileOut [
    puts $fileOut
    close $fileOu
}

proc Create_Proc
    set fileIn [c

    #pid
    #for some pe
    gets $fileIn
    gets $fileIn
    append str [f
        [expr [st
    #program
    gets $fileIn
    puts stdout $
    append str [f
    [expr [string
    #frozen
    gets $fileIn
```

```
    }
}

proc Program_Add
    global progra

    set line $pro

    Write_To_Hom

}

proc Program_Rem
    global progra
    set line $pro


    Write_To_Hom
}



wm title . homeo
frame .top -bord
pack .top -side

set listp [List_

append str [form
#program
append str [form
#frozen
append str [form
#normal
append str [form
#delay
append str [form
#count
append str [form
#anomalies
append str [form
#pcount
append str [form
#last_mod_time
append str [form
```