To be decided
— Individual Research Project —

Giuseppe Pes
giuseppe.pes12@doc.ic.ac.uk
Supervisor:Prof. Cristian Cadar
Course: XXXX, Imperial College London

March 19, 2013

Abstract

MY ABSTRACT

Acknowledgements

I would like to acknowledge ...

Contents

1 Introduction **1**
1.1 System Call interceptor . . . 1
1.2 State of art in the system call interceptor . . . 2
1.3 Issues . . . 3
1.4 Requirements . . . 3
1.5 Design goal . . . 3

2 Ptrace **5**
2.1 Ptrace tracing mechanism . . . 6
2.2 System call virtualization . . . 7
2.3 Memory access . . . 8
2.4 Multi thread applications . . . 9
2.5 Aborting a system call . . . 10
2.6 Conclusion . . . 11

3 Kernel-based tracing mechanisms **12**
3.1 Utrace . . . 13
3.1.1 Setting up a System call interceptor mechanism using Utrace . . . 14
3.2 Kernel hybrid interposition architecture . . . 15
3.3 Kernel delegating architecture . . . 16
3.4 Kernel Probes . . . 18
3.4.1 System call interceptor using kernel probes . . . 19
3.5 Seccomb-bpf . . . 19

4 Binary Rewriting **20**
4.1 Static binary rewriting . . . 20
4.2 Dynamic binary rewriting . . . 20

5 Results analysis **21**

A Appendix Linux System Call Convention **22**

B Appdx B **23**

List of Figures

Listings

2.1 Synopsis ptrace system call 5
2.2 Condition that identifies SIGTRAP signals 7
2.3 Condition that identifies exclusively system call entry and exit 7
3.1 Synopsis utrace_attached_engine 14
3.2 Synopsis utrace_set_events_task 14

Chapter 1

Introduction

Chpter introduction

1.1 System Call interceptor

Regardless the nature of an application,resources (such us files,socket, shared memory etc) can be accessed only via the system call interface exposed by the operating system. The sequence of system calls invoked fully characterised the application's behaviour, that then can be monitored and regulated.

A **system call interceptor** is a powerful technique for investigating all system calls invoked by a program. It usually works by interposing a third agent,called *monitor*,between the operating system and the application of interest. This third agent is able to trace and regulate all application's interactions with the operating system.

The applicability area of a system call interceptor is rather broad spreading from debugging application such as GDB and auditing (record-replay) to security enhancements (sandboxes,intrusion detection) and virtualisation(UML). In the rest of this section we try to cover all possible applications of a system call interceptor.

Debugging

Security enhancement

Virtualisation

The first kind of system call interceptor as been introduced for debugging purposes as tracing routine provided by the operating system. An example of this is ptrace [chapter1]. This tracing routine provides a mean of analysing requested issued by the program in order to help the developer to find a bug in the application.

Presenting a detailed analysis of LINUX for the purpose of building interposition tools.

DETECTION and CONFINAMENT

There has been a lot of research to improve a system call interception in terms of security and performance because it is the base mechanism for security tools such as sandboxes and introduction detection[ref].The resource control provides by an operating system is usually based on a discretionary access control DAC (Linux, UNIX-like system) model, where all programs executed by a user inherit his permission for accessing resources. This model is simple and quite effective, though it does not provided any protection against malicious code (back-door) and flaws (buffer overflow). Sandboxing software [ostia, jailer, sfi, systrace] have been developed to overcome these shortcomings. A sandbox provides a fine-grained control for the resource access by the sandboxed program. These software use a system call interceptor to monitor all system calls made by the sandboxed program and check them against a policies file which constrain a program to a correct behaviour. If one of this policy check fails the system in not executed and the possible malicious action is prevent.
A system call interceptor is widely used to .

Introduction detection via analysis of system call traces has received attention throughout recently years [ref]. The problem of a system call tracer is similar to that of a sandboxing tool previously described. The "viewer" is only interested in analysing what calls an application makes in order to find out anomalous system call sequences which may identify an introduction in the system [9].

As the system call sequence fully characterized the application's behaviour, in the case of an appliation's crash the system call can be reocrder and reproduced that crush in an debugging environment in order to find out the cause. [Where is it used]

Multi-variant code execution is a run time monitor technique that finds out and prevents malicious code from executing. The idea behind this method is to run two or more slightly different version of the same application in lockstep.At certain synchronization point their behaviour is compared against each other and if a divergence is found, a notification is raised. In multi variant execution,the invocation of a system call is a synchronization point as all variants must make the exact same system call with the same arguments. When one of these attempts to make a system call, this is intercepted by a monitor process that then try to synchronize these with all the different calls within a temporal window [15] + PROF.

A system call interceptor has been successfully used to implement a kernel hypervisor in [6, 5]. UML is a virtualisation technology which enables multiple linux Systems to run as an user application within a normal Linux system. The system call interceptor is used to redirect all system calls made by a program to the guest kernel. This software is used to kernel debugging and for sandboxing application.

Another interesting application of a system call interceptor is in [8]. CDE creates automatically portable software package using a system call interceptor to track the execution of x86-Linux and collect all data,code and environment required to run it in another Linux machine.

Why is it important. ?

Where can it be used ?

1.2 State of art in the system call interceptor

State of art/different type
Five different approach may be used to implement a system call interceptor in Linux :

library A library approach was the first used for implementing a system call interceptor [ref]. It relies on the fact that system call are accessed via wrapper functions, within the glibc library. A system call interceptor can thus be realised by linking the application of interest to a different library which contains an instrumented version of the wrapper functions. This approach has been implemented in [16]. Its major benefit is that its performance are almost the same as no instrumented application, but it can be easily bypassed invoking directly a system call using low level mechanism.

Kernel trace tool The kernel provides features such as ptrace and utrace for tracing and debugging programs. Although this features has been design for debugging purpose, they can be used to successfully build up a system call interceptor in user space without kernel modification (Utrace needs to write a module).

Binary rewriting Binary rewriting consists of modifying the binary program to insert new instructions which allow intercepting the system call made by that modified program. This techniques can be applied both statically as well as dynamically and can be done in a number of different ways, e.g. full binary rewriting, selectively rewriting only sensitive instructions, rewriting only system call instructions.

Seccomp mode-based mechanisms: is a simple sandboxing mechanism provided by the kernel. It is consist of two system calls seccomp and seccomp-bpf.

Kernel enhancement Following a kernel based implementation, the system call interceptor is implemented within the operating kernel, and all the extension code is executed in kernel mode. A system call interceptor may be inserted modifying the source kernel [19] or it can be uploaded via a kernel module [18]. Both solutions offer the same power in terms what can be accomplished within the extension code, though the later does require neither to compile nor to patch the kernel, it is thus more portable. The kernel approach is exposed in more details in 3.

1.3 Issues

conflicts paramenters

1.4 Requirements

So far we have provided a general introduction to all possible mechanisms Linux provided to implement a complete system call interceptor, further all of them will be extended explening how a system call interceptor can be implemented using it, providing real example in which such mechanism has been used. Which features are needed by a system call interceptor? A system call interceptor requires a minimum number of capabilities to be effective and thus providing a good base which a sandbox can rely on: Monitor capacity: A system call interceptor must intercept all attempts to invoke a system call made by the untrusted process before that the system call is executed by the kernel. Furthermore, it also needs to provide a method to analysis the arguments of the system call (and to access to the applications data space if the real argument is located there) and returned values. This requirement is the core of the intercepting mechanism itself. Fine-granaid control: we should be able to specify which system call should be intercept and which should be not. Regardless the method used to implement it; a system call interceptor always introduces an overhead with respect to the normal system calls flow. This greatly improves the performance because we could, for instance, trace the system call which gain access to a new resource but avoid to trace those that use the resource already open. Preventing the system call execution: when a system call is invoked with unsafe parameters such as open(/etc/passwd);. The system call interceptor must have a means of aborting its execution without aborting the entire process and setting a proper return value i.g. EPERM. Monitoring all children: The system call interceptor must intercept and monitor all children of the traced processor. It is crucial that the children of a sandboxed process must be constrained to the father's policy rules. Dealing with multithread application: another important requirement it the possibility to trace all threads of the traced application efficiently, without stopping them if not necessary. Why characteristics should a System call interceptor have?

1.5 Design goal

-flexibility -compatibility -security -deployability -performance

-Security -realiability -sdfd Robustness : the capability, in case the tracer process crashes unexpectedly to terminate all the traced processes. Principle of Least Privilege: if the monitor is compromise the attack gains only the user priveldge and not the root. All of these of this techinique introduce an overhead in the normal computation, so we have to use a method to assess thei performance and/ /// . In this document we will introduce a detail analysis of Linux for the purpose of building interposition mechanisms. Which resource must be protected? File system Network access

QUESTION : 1 Problem ? 2 Sand box 3 What is a system call interceptor. ¡ security, portability, configurability.¿ 4 How can we asses them? How can we compare them? A fault domain is a set of hardware components computers, switches, and more that share a single point of failure

Design GOAL

Issues.
OVERHEAD

Remain Document structure

Chapter 2

Ptrace

Ptrace is a system call provided by Linux and other UNIX-like operating systems. It was designed mainly for debugging, but it has been used for tracing purposes as well. Ptrace allows a process to observe and control the execution of another process; the first process is referred to as the tracer or monitor process and the latter as the tracee or target process. The ptrace interface is:

```
long ptrace ( enum __ptrace_request request, pid_t pid, void *addr, void *data)
```

Listing 2.1: Synopsis ptrace system call

The tracee needs to be attached to the tracer by issuing a request with ptrace; the details regarding setting up the tracing mechanism are introduced in 2.1. All ptrace commands are per thread, thus in a multithread application each of its threads must be attached individually. This situation might be a source of potential race condition if not handled carefully. The section 2.4 surveys different problems regarding tracing multithread application and provides an effective solution. In this chapter, the term multithread process/application refers to an application/process whose threads have been created using the system call *clone* [10] with the CLONE_THREAD flag.

Once the tracing mechanism has been correctly initialized by the kernel, the tracee will stop each time a signal is delivered. The tracer will be notified via a wait system call and then it can control the execution of the tracee calling ptrace with the following requests:

PTRACE_SYSCALL : continues execution until next entry or exit from a system call.

PTRACE_CONTINUE : continues execution until the tracee receives a signal.

PTRACE_SINGLESTEP : continues execution until the next tracee instruction.

While the tracee is in a stopped state, the tracer is allowed to examine and change its memory and registers. Ptrace provides the following requests to perform these operations:

PTRACE_GETREGS/PTRACE_SETREGS : respectively get or set the values of the general purpose register of the tracee process.

PTRACE_PEEKDATA /PTRACE_POKEDATA : respectively write or read a word from the memory address space of the tracee process.

Accessing the tracee memory is necessary for retrieving the system call id (contained in the EAX register) and its arguments. This may be used in applications for auditing purposes such as strace [ref to strace], where all system calls and their arguments are recorded or for security purpose such as sandbox application [references] where each system call is tested against a policy specifying which arguments are allowed and which are not. Although this interface is easy to use, it provides very low bandwidth only 4 bytes per call can be retrieved. Each call implies two context switches to the kernel

space and back, introducing a significant overhead in the case of large buffers. This is one of the main limitations of ptrace and different techniques have been developed during the recent years to mitigate it. These are presented in the [Accessing tracee memory] section.

Ptrace has been successfully used to implement a user-space system call interceptor in numerous and different software, for example Orchestra (MVEC) [ref], Systrace (Sandbox) [ref], User Mode Linux (Software Virtualization), Goanna (Virtual File System). However, all these approaches suffer from different limitations due to the fact that ptrace has been designed for debugging single thread applications and not for tracing applications. An important requirement for an application such as a sandbox is the possibility to deny the execution of a system call when it does not satisfy the policy. This is also the base for building up a system call virtualization mechanism [ref UML], where all actual system calls must be nullified; see section [System call virtualisation section]. While other operating system, such Solaris provide a way of aborting a system call, ptrace does not offer any. In the section [aborting system call] different methods to overcome this shortcoming are introduced.Finally, another requirement missed by ptrace is the possibility to trace only a subset of the system calls provided by the operating system and leave others to be executed without any overhead.

This reduces the tracing overhead in several different cases, for example, in a sandbox application where only the system calls which might affect the security of the operating system should be intercepted. In [ref ?] all system calls which open or change a file descriptor (open, socket, bind) are intercepted, while those which use it (read, lseek, write) are not. The major disadvantage of the ptrace approach is that the performance may be noticeably reduced in a program which uses intensively system calls. However, an interceptor mechanism can be used for really different purposes and therefore different criteria can be used for choosing it. When performance is not a fundamental requirement such as for debugging or security, ptrace is a valid solution. It provides an easy way to set up the tracing mechanism compared with a kernel based mechanism and it allows to easy port the interception infrastructure between different operating systems.

The remainder of the chapter introduces two different mechanisms for setting a system call interception using ptrace. The first consist of tracing all system calls made by a program, while the second is based on virtualizing all system calls. The rest of the chapter presents different approaches that address the limitations discussed before.

2.1 Ptrace tracing mechanism

Although *ptrace* has been mainly designed for debugging purposes, it can be used successfully to build up a system call interceptor. To implement an effective system call interceptor, the tracer process must be able to intercept each system call made by the tracee process and its child processes.
There are two different ways to install the trace mechanism. The tracer might invoke the fork system call to create a child process which notifies the kernel its willingness to be traced by calling ptrace with the argument PTRACE_TRACEME. This request causes the kernel to set a traced flag (PT_TRACED) in the target process descriptor and stop it by setting its state to non-interruptible sleep. This approach is usually used before the tracing software initiates a new program invoking an exec system call. Alternatively, the monitor process can make a request to the kernel for tracing a running process via ptrace specifying the argument PTRACE_ATTACH and the target process pid. In this case, there are three conditions to be checked:

1. If the privileges of the tracer process allow it to trace the tracee process.

2. If the tracee process is being already traced.

3. If the target process belongs to a set of process that cannot be traced (init and itself).

If all previous controls are satisfied, the monitor process becomes the father of the traced process and the tracing flag is set on its process descriptor. Then, as the previous case, the tracee is put in a non-interruptible sleep state. This approach is useful when a process, which has been created by a

different process from the tracer, has to be traced as in the case the tracee spawns a new thread which must be traced as well.

It is worth noticing that in both cases the tracee becomes the father of the tracee process. In Linux a process has exactly one process father. Consequentially, this limits ptrace to track only one process per time making it a no-efficient mechanism for tracing multithread applications.

When the tracing mechanism has been correctly installed, all system calls made by the tracee are intercepted by the tracer. The tracee process is put in a non-interruptible sleep state by the kernel each time it receives a signal or attempts to invoke a system call. Once the tracee has been stopped, the tracer is notified via a signal SIGCHILD and then it is allowed to access the tracee address space. The tracer may handle this signal either through a wait-family system call or by installing a dedicated handler. In the case of system call interceptor, the only relevant notifications are those regarding the attempt of invoking a system call. Precisely, the tracer should be notified twice, at the entry point before the system call is executed and at the exit point before the traced process is resumed. The cause of the stop in the tracee can be identified through the status variable returned by the wait primitive or the integer argument in the handler function. The Linux signal library provides different macros to easily deal with this signal status variable [ref signal]. The cause that determines the stop of the tracee process can be retrieved as follows:

```
WSTOPSIG(status) & (WITSTOPPED(status) == SIGTRAP)
```

Listing 2.2: Condition that identifies SIGTRAP signals

Unfortunately, the status variable will assume the same value also when a SIGTRAP signal is delivered to the tracee for a different reason from invoking a system call. (WHICH?)
This ambiguity may be solved in two different ways. Further information about the source of the signal might be retrieved by issuing a ptrace request with PTRACE_GETSIGINFO. Although this will introduce a further system call for each SIGTRAP signal delivered to the trace, which in the case of a traced process are the most common. This performance decrease can be avoided using ptrace with the PTRACE_O_TRACESYSGOOD option. It ensures that when a signal trap is generated because the tracee is trying to make a system call the 7th bit of signal status

```
WSTOPSIG(status) & (WITSTOPPED(status)== SIGTRAP | 0x80)
```

Listing 2.3: Condition that identifies exclusively system call entry and exit

Now, the tracer can skip all false notifications and analyse only those regarding a system call. The tracee can be resumed by calling ptrace with SYS_SYSCALL as argument, its execution continuing until the next system call. The sequence of event triggered when the traced process invokes a system call is shown is Figure 1.

While the tracee process is in a sleep state, its memory and registers can be accessed by the tracer process. The tracee memory is accessed, before the system call is executed, to retrieve the system call identifier and arguments. This is used in software such as sandbox, for example [systrace, jail, c++] to check whether the system call invocation satisfies the requirements specified in a policy file or for auditing purposes in [systrace]. Furthermore, there is also the possibility to access the tracee memory when the process is stopped after the execution of the system call. This gives a means of analysing the system call return value, which is important for applying post policy rules in a sandbox application [jailer]. Post policy rules are used for those system call where a sensitive value is known only after the call is executed such as accpet, recv. Accessing the memory always introduces an overhead which has to be taken in account when performance is an important aspect of the application. Different approaches for accessing the memory of the tracee process are discussed in detail in [memory section].

2.2 System call virtualization

Virtualization has become increasingly popular and a new demand for Linux is to act as a hypervisor. Different virtualization technologies have been developed to cope this increasing demand, such

as instruction emulators QEMU, full or partial hardware emulation. System call virtualization is a technique to emulate the execution of a system call made by a process. Ptrace can be used to set up a system call virtualization mechanism in Linux. This approach has been followed by Jeff Dikk [two papers] developing UML whose goals was to run Linux kernel in user space for easing the debugging of kernel. A system call virtualisation needs a way to nullify system calls so that they would execute in such way as to cause no effects on the host. However, ptrace did not provide a means of aborting a system call [annulling a system call]. This shortcoming has been fixed introducing the possibility to change to change the actual system call. Annulling a system call can be done by substituting the actual system call with a getpid, which introduces a small overhead due to its execution.
When a system call is virtualised, there is no need to intercept the exit point because the system call has been nullified. This prevents the execution of two context switches between kernel space and user space which yields to a 50% improvement in the performance. All these ptrace enhancements have been introduced in the kernel mainline and they can be used by calling ptrace with the parameter SYSEMU [ptrace documentation]. As it has already been said, this prevents the system call to be executed and notifies the tracing parent only at system call entry.
The tracing mechanism can be installed in a similar manner has described in the previous chapter. The only difference is that when the traced software is resumed issuing ptrace with PTRACE_SYSEMU parameter instead of PTRACE_SYSCALL.
System call virtualization is implemented by the tracing thread intercepting and redirecting process system calls to the system call handler. It reads out the system call and its arguments, then annuls this in the host kernel and executes the virtual system call instead. When the emulated system call is finished; the return values are stored in the memory of the traced process. This process is depicted in the figure.
An example of this approach can be found in [GOANA], where ptrace has been used to create a user-level file system development environment. The bulk of the application consists of a monitor called GOANA which intercepts all system calls made by the traced process. Once the system call has been intercepted this is substituted with a prototype function. This provides an easy way to test a file system prototype because we do not need to deal with massive body of kernel-level code and due to the fact it runs on user space the system can be analysed using a powerful debugger such as gdb which would have been impossible to use in kernel mode.
The performance is still the main problem of this approach as well. Even though by using SYSEMU we reduced the context switch between User Space and Kernel space to 2. There is still the necessity to access the memory at the system call entry point to retrieve its arguments and after the execution of the system call is finished to store the return values. This is a critical part of each monitor process, this problem as in the previous case will be treat in detail in the next section.

2.3 Memory access

The monitor process is executed in user mode, it thus is not allowed to write or read the memory space of the traced process. The monitor needs to read the memory of the traced process in order to retrieve the contents of the indirect parameters such as strings or pointers. There are two kinds of arguments direct and indirect, the former is contained in the general purpose register, while the latter is contained in the process memory and its address can be found on the registers. The maximum argument number for a system call is six, so they fit in the register and there is no need to access the porcess stack.
Also, the monitor needs to write the target memory address, for instance, when the system call is emulated and the outcome of the computation need to be stored in the traced memory space.
Ptrace
One method to access the memory of the tracee process is provided by ptrace. The monitor process may read from the monitor process calling ptrace with PTRACE_PEEKDATA when the target process is suspended. The writing procedure can be performed in a similar manner by using PTRACE_POKEDATA. Unfortunately, ptrace transfers only 4 bytes per time, therefore it has to be

called many times to read or write a large amount of memory. Every call to ptrace requires a context switch from the monitor to the kernel and back, which decreases its performance since to make it, in fact, not a feasible way with a large amount of data.In order to improve the performance in accessing the traced memory space different approach can be taken.

Shared Memory

The monitor process has to ensure that the target process can access to the shared memory. In [Jailer] the shared memory is set up using a preload library technique. The preload library uses mmap() to read-only map the memory region in the targets process space , in addition, it loads some code routine. Another technique presented in [Orchestra] consists of replacing the first system call made by the target process with an appropriate one such as shmget, ipc after making a backup of the values contained in the registers. This makes the target to run the new syst em call and attaching the shared memory to the target process. After performing this operation, the original system call is restored by using the register values previously saved. In both cases a small code is injected to the space memory which copies the content of a buffer to another one. When the monitor process needs to access to an indirect arguments, it can retrieve the base address from the registers by using ptrace and then use this routine to copy all the buffer in the share memory. In the Jailer system, the values of registers are modified to point to this memory area for security reasons [See race condition].

FIFO

Another approach proposed in [Orchestra] is to use FIFO structures. FIFO is inter-process communication which allows the monitor process to communicate to the target process by writing to and reading from the FIFO. The pipe can be created using the system call mkfifo()[7] and then it can be managed via I/O system calls usually used to deal with files (in fact the FIFO is a file in the file system). The target process can easily open the FIFO by using the FIFOs name as parameters calling open. While, if the target process has been spawned by the monitor process, it inheritances the file descriptor corresponding to the FIFO open by monitor process. However, there is still need to use a routine to copy the values from the target process space to the pipe. Other ipc mechanisms such message queue [jailer] can be used in a similar fashion.

/Proc/$pid/mem

This is the most important because it provides a good solution. To write

2.4 Multi thread applications

Nowadays, most the applications are multithread. For example, a web server usually creates a new thread for each incoming request and let this newly created thread to accomplish it. This family of applications must be taken into account as possible target process for a system call interceptor and this requires a thoroughly analysis. As we already said in the introduction part, one of the most import requirements of a system call interceptor is to provide a means of intercepting all processes spawned by the traced process. Ptrace misses this aspect because it provides a tracing mechanism for single thread [ptrace documentation]; where each thread must be attached individually to the tracer process issuing a request with PTRACE_ATTACH. Moreover, a child process does not inherit the trace flag from the parent, so it can run untraced until the father attaches to it. These flaws make tracking multithreaded application difficult and error prone [race condition paper], so that some applications [jailer] decided to not support this kind of applications.

Nevertheless, ptrace offers a solution, though it is quite elementary. Setting the option PTRACE_O_TRACECLO [insert a note for other arguments], the tracee will be stopped at the next clone() function and the tracer will start tracing the newly cloned process. This solution, in fact, is not a feasible way to trace multithread application when performance are important because each time a signal is delivered to a traced thread, all threads in the traced poll enter in a stop state.

This introduces an unnecessary latency because the other threads (those that have not received any signal) must wait until the event is processed by the tracer and the entire application is resumed before being able to carry on their tasks. However, in application where performance is not important, for instance debuggers, this represents a valid solution. In fact, this approach is used by GDB. Another

issue related with this approach is that ptrace does not always guarantee that the forked processes are always automatically traced. Linux allows setting a flag on clone() (CLONE_UNTRACED) , which determines whether the child process can be traced or not. However, this can be easily solved noticing that the clone function will be intercepted by the monitor program and it can modify its arguments changing that flag to CLONE_PTRACE which makes the process traceable.
The common approach to deal with multithread application is to create a new tracer process [jain and sekar, jailer, orchestra, systrace,strace] for each new tracee process. Unfortunately, this solution raises a race condition which might yield to some system calls to escape from being intercepted. The new monitor is not allowed to trace the newly thread until the parent monitor detaches from it. When the parent monitors detaches from it the kernel sends a message to the tracee process and allows it to continue its execution. This leaves a small temporal window within some system calls might escape. This is critical problem in application such as sandbox, where all system calls must be intercepted for obvious security reason. A completely different approach is taken by strace where this problem is not tackled because a missed system will not comprise the entire application, while solving it will complicate the code, and the temporal window is rather small and the probability of missing a system call or more should be quite low.

A possible kernel enhancement, which solves this, would be to notify to the tracer when the newly thread has been created but it is not yet running. This would give the opportunity to safely attach a new tracer to the newly created thread without losing any call. Instead the tracer receives a tracing event only on the childs subsequent system call.
This problem has been addressed successfully in [Orchestra] as follows. When the tracee process spawns a new thread, automatically it is stopped by the kernel and the monitor start monitoring the newly created thread. At its first system call invocation, the monitor makes a backup of all new tracee process registers and then it substitutes the requested system call with a pause system call and finally lets it continue its execution. When the new process is resumed, it will enter in a pause state due to the injected system call. While this process is in the sleep state, the monitor is allowed to detach from it and the new monitor can safely attach to it. Once the new monitor has successfully attached, it restores the previous system call so that the execution of the new tracee process can continue. Although this solution works well, it introduces a small time overhead due to the execution of the pause system. Moreover, a communication between the new monitor and its father is needed to retrieve the backup information regarding the execution context of the new thread in order to restore the previous system call.

2.5 Aborting a system call

Aborting a system call

Having the possibility to abort a system call is an important feature. Lets consider the following scenario. A large application such as database has been sandboxed for security reason. It is possible that due to the nature of the application and the high number of system call requests, some policy checks fail. When this happen the system call should be nullified, but if this feature is not provided the only solution is to terminate the application even if there is not any real threat. Unfortunately Linux doesn't provide any means of aborting a system call. This lack has been one of the main reason because ptrace was not considerate a feasible way to implement a system call interceptor [Janus thesis]. Different solutions have been propose [C++ sandbox] and [Goanna], but this problem has been solved by [Uml Guy] who implemented a patch for the Linux kernel. This patch allows the tracer to substitute the invoked system call with another one by inserting a different system call id in the EAX register. Leveraging on the enhancement introduced for UML, the tracees system call can be substituted with the low-overhead system call getpid.
This technique for aborting a system call raises another problem, which value should be returned?
One possibility is EINTR which represent the error code for system call interrupted. However, this is not a good choice because some application may be coded in such a way that, if they receive EINTR

error they will retry the system call. If this happens in a loop the application gets stuck and the only way to break that loop is to kill the entire application. A better error code is EPERM, which stands for operation not permitted.

2.6 Conclusion

to write

Chapter 3

Kernel-based tracing mechanisms

Kernel tracing mechanisms were originally developed in response to the limitations of the existing tracing mechanisms such as ptrace. Since the early 2000, several sandbox tools have been developed using a system call interceptor implemented within the kernel [18, 19, 12, 7] either using a kernel enhancement (ptrace++) or a kernel module (mod_janus).

Two different approaches can be taken to develop a system call interceptor in the Linux kernel. As the source of the kernel is available, a completely new tracing infrastructure might be introduced in the kernel exposing its tracing functionalities to user space as in the case of *ptrace* or at kernel space as in the case of*utrace*. This approach requires a massive change in the kernel source that is difficult to implement and error-prone due to the complexity of the kernel. Furthermore, it is not a portable solution as it will not be inserted into the Linux source (see case utrace),and thus it should be dispatched as kernel patch which is not easy to install and require to compile the entire kernel.

An easier approach is to insert the tracing mechanism within a kernel module. It simplifies the installation process because it needs only to be compiled and loaded in memory. It also has the benefit that the tracing mechanism can be activated when the module is loaded and deactivated when it is removed. In fact, this is the implementation choice widely used for building up kernel-based interceptors [18, 19, 12].

Inserting code in the kernel space is always a risky operation because there is the possibility to introduce some bugs which may compromise the whole system. In order to reduce this eventuality, an **hybrid interposition architecture** has been firstly developed in [18]. Then it has become the prominent approach to build kernel-based interceptor. The attribute hybrid is due to the fact that it is not a fully kernel based implementation, as it is composed of two components : a kernel-level enhancement and a user-level process monitor. The kernel-level enforcement is the core of the system call interceptor as its main task is to provide to the monitor process a means of controlling the system calls made by another program.

This usually was accomplished by overwriting the address of a system calls within the system call table[1] with the address of the respective instrumented system calls.An instrumented system call runs a pre-call and post-call routine as well as calling the old system call. The extra routines executed are usually referred to as **extension code** and allow tracing component to have a full control of the system call. It can prevent a system call to be execute, access its parameters or return values.

These methods rely on the fact the system call table is modifiable. Since kernel 2.6, this possibility has been removed, the address of the system call table is not accessible and the pages on which it is contained are now read-only. However, different tricks[2] to overwrite the system call table exist, but they are not a reliable solution for implementing a system call interceptor.

A solution for this problem may be to use the instrumentation features provided by the kernel instead of modifying the system call table. Instrumentation features are used mainly for debugging and tracing purposes but they can be effectively used for implementing a system call interceptor as well. A Kernel instrumentation tool is a mechanism that allows for a extra code, usually called as *extension code*, to be insert into a specific location in the kernel code. When this point is hit, the

[1] Kernel structure *sys_call_table* which contains addresses of system call handlers.

[2] For example. the address of the system call table can be retrieved from the file /boot/System.map

extension code associated is executed.

There are two type of the kernel instrumentation:

Dynamic the instrumentation code is inserted in the specific location at run time. Kprobes [17]

Static the point and the instrumentation code is declared in the source file. Kernel makers/trace point used in LTTng [4] .

The primary advantages of a kernel based approach is low overhead due to the intercepting mechanism. The overhead for a system call interposition is determined completely by the extension code executed. Moreover, the kernel code runs at the highest privilege level and as such, can access all kernel structure as well as the address space of a user-space process avoiding the overhead due to the context switch. However, the power afforded by a kernel-mode intercepting mechanism entails some drawbacks. The primary one is that the extension code, as it runs at kernel-level, must respect the constrains of this environment. In this environment some assumption commonly used in user-space may not be true in kernel space [such us such as dynamic memory allocation].This make a kernel developing complicated and error-prone. Introducing an error in the kernel has serious consequences, because it may compromise the entire system or introduce security flaws. In addition a system call interceptor based on the kernel requires super user privilege to be installed that may make it difficult to be used in a multi-user environment.

Other factor that make this approach cumbersome for implementing a system call interceptor is that the code is not easily portable among different architecture or kernel version. The process state represented by the status of its register is architecture dependent, the ABI used to pass arguments across different kernel functions may depend on the Kernel version. Therefore a system call interceptor realized using a kernel mechanism needs to be adapted for the architecture of interest.

In the remainder of this chapter the models previously introduced are explained in better details. The first model presented is *Utrace*, it is a kernel tracing architecture initially implemented as kernel enhancement to replace ptrace. Although, this has never been inserted in the main line of Linux kernel, it is an interesting prototype which is worth analysing [the interesting thing is the performance]. The third and the forth chapter introduce two different model of hybrid interposition architecture. The first is implemented using a filtering architecture the second using a delegating one. The last part introduces the use of kernel instrumentation mechanism, it is mainly focus on the Kprobe architecture explaining how this mechanism can be used to instrument system call in order to build up an efficient kernel-based system call interceptor.

3.1 Utrace

Utrace has been developed principally in order to overcome the ptraces limitations, especially those regarding performance and race conditions. Utrace is an in-kernel API which can be used to build kernel-based tracing mechanisms. It has been used to implement a secure sandbox for web application in [14], or as base for as virtualization mechanism in KMview [3].

The first difference with ptrace is that Utrace does not interact at all with user space, its interface its available only in the kernel space and all the extension code runs at kernel level. This implementation choice has been taken in order to avoid the overhead due to the context switch between user space and kernel space which is the major cause of low performance in ptrace.

The actors in an Utrace tracing mechanism are threads and tracing engines. A tracing engine is utrace's basic control unit, it is a piece of code defining the actions to be taken as consequences of an event occurring in the traced thread, such as invoking a system call. Typically, the utrace client is defined within a kernel module, and it establishes an engine for each thread of interest.

The Utrace interface provides the following basic facilities to build up an efficient tracing mechanism:

Event reporting: Utrace clients can register callbacks to be run when the traced thread issues a specific event of interest, i.e. system call entry/exit, exit, clone,etc.

Thread Control: The utrace client has full control of the running thread. It can inject signal, prevent a thread from running and abort a system call.

Thread machine state access: While the client is in a callback function, it can investigate the internal thread 's state by reading/writing the threads CPU registers and its memory space.

Utrace operates by inserting tracepoints at strategic point in the kernel code (these are called SAFE point more info) . When one of this points is hit by the traced kernel the callback associated with that event takes place. This callback, though happening in the context of the user process, occurs when this process is executing in the kernel space.

3.1.1 Setting up a System call interceptor mechanism using Utrace

The utrace client must be implemented in a kernel module in order to be able to access to the utrace interface. So, the mechanism will be activated when the module is loaded and deactivated when the module is removed. The tracing mechanism must ensure that a callback function is executed when the entry/exit system call event is triggered during the execution of the traced process.

A Utrace mechanism starts out by attaching an engine to a tread.

```
struct utrace_attached_engine * utrace_attach_task (struct task_struct * target,
                                int flags,
                                const struct utrace_engine_ops *ops,
                                void * data);
```

Listing 3.1: Synopsis utrace_attached_engine

Calling one of these function with the flag UTRACE_ATTACH_CREATE the engine is attached to the thread identified using its task_struck or PID. The structure utrace_engine_ops defines the callbacks function. In the case of a system call interceptor, the callback function of interest are those regarding the entry/exit of a system call.

Once the engine has been attached, the SYSENTRY and SYSEXIT need to be set in the engine. This can be accomplished using the following function :

```
int utrace_set_events (     struct task_struct *,
                            struct utrace_engine *,
                            unsigned long eventmask);
```

Listing 3.2: Synopsis utrace_set_events_task

Once the setting phase is completes, each time the traced process attempts to invoke a system the callbacks will be executed. During the execution of the callback function the tread is put in a QUIESCENCE status. This means that is stopped and will not start running when its status is accessed. Each callback takes as argument the task_struct which represents the state of traced process before the event has been triggered. Retrieving information from this structure such as system call number arguments depends on the architecture on the CPU architecture (i.e. x86 and x86_64 have different registers). The thread then can be resumed or aborted depending in the value returned by this function.

One of the important characteristic of a system call is to retrieve both direct and indirect arguments of a system call. The direct arguments can be retrieved from the CPU register. While the indirect ones are in the address space of the traced process and only their address can be retrieved from the CPU registers. The kernel provides two routine which allow to read (copy_from_user) from and write (copy_to_user) to the address space of a user process. Using these routines a value can be copied in the kernel space and analysed.

The system call interceptor, described so far it does not handle the case when the traced process spawns a new process. Utrace provide an event and its associated callback to handle this situation. If the traced process attempts to invoke a fork/clone system call, the CLONE event callback is called when the new child thread has been created but not yet started running (Note that this is the solution proposed in the previous chapter for ptrace in case of multi thread application).The newly thread cannot be scheduled until the CLONE tracing callback return. This allows the tracing mechanism

to create a new tracing engine and attach it to the newly process, ensuring that all system calls are correctly intercepted and analysed.

The main advantages of utrace is its performance. [Nice some examples]. Even though utrace seems a good solution for implementing a system call interceptor, it has been harshly criticized due to its kernel-based model. It suffers, as all kernel-based mechanisms, of the no-portability problem. It was intended to be a the ptrace killer application, but this is not happened because its interface can be used only within the kernel. A user need to have a base knowledge about kernel programming to write even an easy interceptor, which is not a common skill. These arguments caused the abandonment of utrace's development and it has never joined in the kernel main line.

3.2 Kernel hybrid interposition architecture

The first kernel hybrid interposition mechanism has been implemented in the second version of Janus sandbox [18] to provide a means of monitoring and modifying the system calls made by the sandboxed process. The necessity to develop a new interceptor mechanism raised from the limitation of ptrace in effectively aborting a system call. Before the enhancement introduced to support UML [6, 5], the only way offered by ptrace to prevent a system call from being executed was to terminate the entire program. This, obviously, was not a feasible solution for a sandbox tool because some false-threats can be thrown even though there is not any real threat. Event though the sandbox implemented with this intercepting mechanism is prone to vulnerabilities[ref], the same interceptor mechanism has been reproduced in other sandoxes tools such as MapBox [2], SysTrace [12] with some improvements.

An hybrid system call interceptor is composed of a tracing engine which completely resides within the kernel whose task is to track all system calls made of the traced program. It allows a monitor process to access the tracing features through an easy interface in user space .Typically the communication between the monitor process and the tracing engine takes place via a char device, for example in mod_janus is /dev/fcap. Once the tracing mechanism has been correctly set up all trap events associated with the traced can be retrieved by the monitor process through a select or poll system call. As in the case of ptrace, a relevant overhead is introduced due to the context switch between user space and kernel space. However, in the case of a kernel interceptor mechanism, this shortcoming can be mitigated by leveraging on power and flexibility of this method . For example, it allows a fine-grained control over the system calls allowing a monitor process to intercept only certain calls while leaving the other unmonitored decreasing the overall overhead.

This may look not a good choice for auditing or record-replay purpose because all system call and their parameters need to be recorded. That is true, but another important characteristic of a kernel interceptor mechanism is its flexibility. If we want to use this kind of interceptor for record-replay purpose, the monitor mechanism can be inserted within the kernel space reducing the overhead just to the execution of the extension code.

As describe in the previous few line the kernel approach is extremely powerful and flexible, though placing an entire system call interceptor in a kernel is not a trivial process and it can introduces errors and new vulnerabilities that can compromise the entire system.

In the remainder of this section we analyses the system call interceptor implemented in in the second version of Janus sandbox. This has been chosen because other system call interceptors are implemented in a similar fashion and its source code is available on line.

The entire system call interceptor is implemented within a kernel module called mod_janus. To be able to use it the module must be loaded inside the kernel memory. One the module is correctly loaded in memory, the system call table is saved and a char device is created in the dev directory. This char device is used to carry out the communication between the intercepting engine within the kernel and the the monitor process in user space.

Before starting tracing a process, the monitoring process need to allocate the resource for supporting the tracing operations. This is accomplished by calling the open system call on the char device /dev/fcap, which returns a descriptor representing a monitor structure that can be used to track a process. Once the monitor has been created, the monitor process is attached to the traced process by issuing a request via ioctl with parameter BIND and the descriptor of the monitor previously created.

Furthermore, this function takes two additional parameters that give the possibility to the monitor process to specify for which system call entry/exit it will be notified. This is the first difference and improvement respect to the tracing mechanism provided by ptrace, which is based on an approach of all-or-nothing.

The system call interceptor is installed by overwriting the system call's addresses within the system call table with the address of a redirecting routine. The main task of this routine is to perform some preliminary checks (such as whether the process is already traced) and redirect the program flow to the tracing engine. This must be really concise and fast as it is invoked even by programs that are not being traced. It is usually implemented with few lines of assembler code.An interesting example of such routine can be found in the Janus source [ref source] in the file ent.S.

When a program issue a system call that has been replaced with the routine, the system call is not executed and the control flow is redirected to the tracing engine. The first check taking place here is whether there is a a monitor associate with this process. If the result is positive a request for a EVENT_CALL_ENTER is issued, otherwise the real system call is executed. Let's consider the case of the system call has been called by a traced program.When the event EVENT_CALL_ENTER is issued the traced program is put in a deep sleep as well as notify this event to the monitor process to its next call to either select or pool system call. The event type then can be retrieved with a read system call on the file description. While being in a sleep state the traced process can be accessed by the monitor process, as in the case of ptrace there are several different ways to access the memory of the tracee process, those one presented in the section 2.3 are still valid also in this case. However, for completeness of the subject we reported the method adopted by Janus. When the process is stopped the monitor process can request access to the system call's arguments by issuing a request via ioctl with argument FC_IOCTL_FETCH_ARG. When the monitor calls this function, it must specify the argument to be retrieved, its type[3] and a user space buffer on which the argument will be stored. Particularly interesting is the type TYPE_PATH_FOLLOW, when this option is specified the path name returned is expanded and canonicalized by the kernel preventing race condition on the path name [**?**].

Once the monitor has terminated its operation on the tracee process its execution can be resumed or denied specifying the right argument on a write request on the descriptor. If the tracee process is allowed to continue the system call is executed. Moreover, if a exit trap for this system call has been specified a similar sequence of event as that one describe above takes place when the execution of the system call is completed. The only difference is that the event issued is the EVEN_CALL_EXIT.

An exception of the previous approach is the fork/clone system call. The exit point of this system call is always trapped, the pid of the newly created process is retrieved from the task structure of the father (field p_cptr) and then it is stopped before it can make any system call. This allows the monitor process to start monitoring this process as well.

(I am not sure about this the code looks like there is a temporal windows within the child process might invoke some system calls)

It is worth mentioning the concurrent strategy adopted in this intercepting model. The monitor process can handle multithread application using a multiplexing model, in which a single monitor listens to the events associated to all threads of interest at the same time. This model may be implemented by creating a set of file descriptors on which all thread descriptors are inserted. Then the select function may be used to listen to the pending requests on these file descriptors. This implementation choice has been made because it would significantly reduce overhead over load. However, as showed in [7], this reduces the scalability of the system as the single monitor becomes the performance bottle neck.

3.3 Kernel delegating architecture

The interceptor model presented in 3.2 suffers of a series of security problems (race condition [ref]) which makes its design and implementation substantially error prone to race conditions. This is due

[3]type indicates the type of the argument being requested, for example TYPE_SCALAR will store a scalar argument in the destination buffer, while TYPE_STRING will store a string

to the fact the action of checking a resource such as the system call parameters and the action of use is not an atomic operation therefore the resource may change between the two moments. This is usually called TOCTOU race condition. In order to overcome the limit of this architecture an alternative intercepting architecture has been developed in [7]. It is usually referred as *delegating architecture*.

A delegating architecture is composed of tree main components.

Kernel module: A small kernel module whose task is to prevent system calls made by the traced program from being executed. Furthermore, it also provides a trampoline instruction that redirects the system call back to the emulation library. (This can be easily accomplished overwriting the system call table for example).

Emulation library: The emulation library resides in the program's address space. When a system call is made by the traced program, the trampoline instruction within the kernel module is hit and a call back to the specific handler in the emulation library is issued. Then, the emulation library converts this system call request into a IPC request to the agent. In addition, to boost up subsequent system calls from the same point in the program execution, the handler analyses the instructions, if they have the expected form, applies a runtime patch to jump directly to the handler. This avoids the expensive context switch from user space to kernel space and back for subsequent system calls.

This library needs to be installed in the program's address space before the program starts running,so that all its system calls are intercepted and executed through the agent. The solution adopted in [7] is to modify the ELF loader so that the program is loaded in memory only after that the emulation library has been installed.

The communication between emulation library and an agent take place over a UNIX domain socket. This communication model has been chosen because it allows a file descriptor to be passed between the process and the agent. This is a crucial feature as it allows delegation of accesses to resources (i.e. open files and socket), while permitting the process to use them directly.

User-level agent The user-level agent is responsible for handling request for system calls from the emulation library. This is the most [delicate] and complex component, it must executes system calls on behalf of the traced process as well as providing a normal system call interface, as the monitored process should not be aware to be tracked. However, the Linux system call interface is rather complex and accomplish this is not an easy task. In [7] the only system calls analysed are those allowed in an sandboxed environment, and this does not offer a comprehensive view of all issues which may raise using this system call interceptor. For example, the case where the traced process invokes a mmap system call is not analysed. This may raise an issue because the new memory is attached to the process who invoked the system call, which is the agent in a delegating sandbox. We try to cover all problems linked to the delegation agent by subdividing the system call in subgroup and providing an possible implementation which should not be bound to sandboxing environment.

System calls fall in few subcategories :

Process context dependent There are few system calls that they result is bound to the execution context, as the agent resides in a different execution context this rises an issues. For example, in the case of the client calls mmap this must be executed by the agent, but the new memory space is attached to memory of the caller that in this case is the agent. For this case, a possible solution is that the agent modifies the argument of mmap so that the new memory area is shared between the two processes. [there may be more problematic cases].

Resource access In Linux resources are accessed via descriptors. Applications starts with an file descriptor space containing only input,output and error file descriptors. To grant access to an additional resource the monitored process must execute a system call (i.e. open, socket), that then will be intercepted and executed by the agent. The resulting descriptor

is passed to the monitored process via Unix domain socket. Once the resource has been correctly opened, the monitor process can modify the object referred to by the descriptor by passing it to the agent, for example this happens in the case of ioctl or bind system call.

Id management The process's identity is represented by the user and group id. To perform accesses on the process's behalf the agent must assume the identity of the monitored process. This is accomplished by reproducing the identity state of the client within the agent, and all system calls (setgid, getuid) update this internal state. When the agent performs a call on the client behalf's, it assumes the identity saved in this internal state.

Signals The monitored process's signal are send by delegating the call to the kill system call to the agent.

Spawn new process When the client process invokes a system call such as clone or fork, the emulation library notifies this to the agent. The agent then spawns a new agent process and returns a new domain socket to communicate with the newly agent.Finally, the monitor process via the emulation library calls into the kernel to execute new fork.

This model has been mainly developed to overcome the securities issues in the filtering model. The structure of the delegating model itself solves the problem linked to the TOCTOU as the system call are invoked from the agent which resides in a different address space from the monitored process. This make impossible for an the multithread process to change the argument of a system call after this has been delegated to the agent. [More info] [More info about the overhead due to the delegation]

3.4 Kernel Probes

Kprobe [11] is a simple lightweight instrumentation mechanism developed by IBM and it has been introduced in Linux Kernel Version 2.6.9. Kprobes allows a user to dynamically insert a *probepoint*[4] in a specific kernel location. When a probepoint is hit a user-defined handler is executed. This will be executed in the context of the process where the probepoit has been hit. Kprobes has been mainly used for debugging purposes because a debug routine can be inserted easily in the kernel without recompiling it, for kernel tracing application as SystemTap[1], for performance evaluation, for fault-injection,etc.

Kprobes operates by overwriting the first byte of the probed instruction with a breakpoint instruction (e.g., int3 on i386 and x86_64). The original instruction is copied into a separate region of memory.When the probed point is hit by the CPU a trap fault occurs, the CPU register are saved and the control passes to the Kprobes manager via the kernel notification chain[5]. Then, the Kprobes manager executes a user-defined routine *pre_handler*. After that, the original instruction must be executed. This is run in single-step mode out of the normal program flow. This solution called *"single-step out of line"* or *"execute out of line"* (XOL) allows a probe mechanism to work with multiple processes at the same time. [more info] When the execution of the probed instruction is completed, the control returns to the kprobes manager which executes a user-defiend post_handler. A nice description of the kernel probes can be found at [17].

A probe point can be registered using the function *register_kprobe()* specifying the address where the probe is to be inserted and what handlers is to be called when the probe point is hit. Recently,the possibility to insert probe point through symbolic name has been introduced in the kernel. This facility is particularly useful as a routine can be probed just using its name. Currently three different types of kprobes are supported :

Kprobe Kprobe can be inserted at any location within the kernel.

Jprobe Jprobe is inserted at entry point of a kernel function and it provides a convenient way to access the function's arguments.

[4]Probepoint is the address where the instrumentation is registered

[5]The kernel notification chain is the communication systems used within the Linux kernel, it follows a **Publish-subscribe** model.

Kretprobe Kretprobe , usually called return probe, is inserted at the end point of a kernel function.

The over head introduced using kprobes is to be taken into account when the performance is a important part of an application. The overhead is principally due to the execution of two exceptions for each probe instruction. A series of kprobes, called kprobe-booster, has been developed and integrated in the kernel to reduce this overhead. Their implementation rely on the fact that the post_handler is not always used and, if so, the second exception can be avoided using a jump instruction. This enhancement reduces by half the overhead due to kprobe instrumentation. This result was easily imaginable because the number of execpetion has been halved. A further improvement has been proposed in [13] where jump instructions are used instead of the break instruction. They claim to have achieved performance 5 times better than a normal probe.

3.4.1 System call interceptor using kernel probes

[to write]

3.5 Seccomb-bpf

[to write]

Chapter 4

Binary Rewriting

4.1 Static binary rewriting

4.2 Dynamic binary rewriting

Chapter 5

Results analysis

Appendix A

Appendix Linux System Call

In this section we introduce how system calls are implemented in Linux and at least the ABI convention used to pass parameters across different functions.

Appendix B

Executable and Linkable Format

ELF

Bibliography

[1]

[2] Anurag Acharya, Mandar Raje, and Ar Raje. Mapbox: Using parameterized behavior classes to confine applications. In *In Proceedings of the 9th USENIX Security Symposium*, pages 1–17, 2000.

[3] Renzo Davoli. Kmview @ONLINE, October 2007.

[4] Mathieu Desnoyers and Michel R. Dagenais. The lttng tracer: A low impact performance and-behavior monitor for gnu/linux. 2006.

[5] Je Dike. Linux as a hyper visor. 2001.

[6] Je Dike. User-mode linux. 2001.

[7] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *IN NDSS*, 2003.

[8] Philip J. Guo and Dawson Engler. Cde: Using system call interposition to automatically create portable software packages. 2011.

[9] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2010.

[10] Linux man-page Project. Linux programmer's manual clone. `http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm`, 2013.

[11] Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Jim Keniston. Probing the guts of kprobes. 2006.

[12] Pervos. Improving host security with system call policies.

[13] DjprobeKernel probing with the smallest overhead. Masami hiramatsu and satoshi oshima. 2007.

[14] Ocvtavian Purdila and Andreas Terzis. A dynamic browser containment environment for countering web-base malware. 2006.

[15] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in userspace. In *In Proceedings of the European Conference on Computer Systems*, 2009.

[16] Mark Seaborn. Plashglibc, 2008.

[17] Goswami Sudhanshu. An introduction to kprobes @ONLINE, 2005.

[18] David A. Wagner. Janus: an approach for confinement of untrusted application.

[19] Guido Van t Noordende, dm Balogh, Rutger Hofman, Frances M. T. Brazier, and Andrew S. Tanenbaum. A secure jailing system for confining untrusted applications. In *International Conference on Security and Cryptography (SECRYPT*, pages 28–31.