# An Efficient Technique for Preventing Mimicry and Impossible Paths Execution Attacks

Danilo Bruschi Lorenzo Cavallaro* Andrea Lanzi

Università degli Studi di Milano
Dipartimento di Informatica e Comunicazione
Via Comelico 39/41, I-20135, Milano MI, Italy
{bruschi, sullivan, andrew}@security.dico.unimi.it

## Abstract

*In this paper we propose a new strategy for dealing with the impossible path execution (IPE) and the mimicry attack in the $N$-gram based HIDS model. Our strategy is based on a kernel-level module which interacts with an underlying HIDS and whose main scope is to "randomize" sequences of system calls produced by an application to make them unpredictable by any attacker. We implemented a prototype of such a module on a Linux system in order to experimentally verify the feasibility and efficacy of our idea. The results obtained are quite encouraging, furthermore it turned out that our module is quite efficient, as it affected the performance of a testbed web server with a slowdown factor of only* $5.9\%$.

## 1 Introduction

An Intrusion Detection System (IDS) is a security technology attempting to identify (in quasi real time) and isolate computer systems intrusions. A very broad classification generally adopted distinguishes between Host Intrusion Detection Systems (HIDSs) and Network Intrusion Detection Systems (NIDSs). Host-based IDSs mainly monitor operating system activities on specific hosts in order to detect intrusion attempts, while Network-based IDSs examine network traffic. Any category of IDS can be further divided into two sub categories on the basis of the mechanism adopted for detecting malicious activities. More precisely, we distinguish between signature-based IDS (also referred to as misuse detection) and anomaly-based IDS. A misuse detection IDS detects attacks as instances of attack signatures, i.e., sets of rules or filters which characterize a malicious event. Anomaly detection instead focuses on normal system behaviors, rather than attack behaviors, i.e., a normal behavior profile is created for any activity performed on the system, which has to be monitored, and any deviation from such a profile is flagged as a potential attack. In this paper we are interested in the models of anomaly-based HIDSs based on the $N$-gram model, initially introduced by Forrest *et al.* [4, 6].

The $N$-gram model is very simple and very efficient but it is characterized by a relatively high degree of false alarms, mainly because *correlations* among syscalls are lost, since there is no provision for storing information about the syscalls *coordinates*. In [10] it has been shown that such a weakness can be exploited for defeating these HIDSs; with two particular forms of attacks, namely the mimicry [11] and IPE [10]. Consequently, various authors started to propose variations to the $N$-gram model in order to improve its "precision", that is, its ability to correctly detect a computer intrusion, with a particular attention to both the IPE and mimicry attack. All these models try to overcome the limitations of the original model adopting a better characterization of a program behavior. Such a characterization is obtained by saving for any considered syscall, additional information such as the value of the program counter, the stack configuration, and information regarding the control flow graph (see [9, 10, 3, 5]). However, even these models suffer of some limitations. For example, in [10, 3] it has been shown that the models proposed in [10] and in [9] are not able to deal with some forms of IPE, while in [11, 7] it has been shown that all the models above mentioned are susceptible, with various degrees of resistance, to some forms of *mimicry* attack. In this paper we propose a new strategy for preventing IPE and mimicry attacks, which is based on the notion of syscall obfuscation introduced in [2]. Such a strategy is based on the con-

*Currently visiting at the CS Dept. of SUNY at Stony Brook, USA.

struction of a kernel-level module, the *obfuscator*, which interacts with an underlying HIDS. The main scope of such a module is to "randomize" the sequences of system calls produced by an application, in order to make them hardly predictable.

In this case, any attacker who wants to execute a traditional mimicry against a process $p$ has to know $p$'s traces, but if they are hardly predictable, it will be very difficult for him to perform such an attack. The same reasoning can be applied, with some changes, to the case of IPE attacks. Preliminary experimental results have shown that besides its effectiveness, our strategy is also efficient.

The paper is organized as follows. In § 2 we describe some works about the HIDSs research area. In § 3 we introduce some preliminary notions about mimicry and IPE attacks as well as on syscall management in Linux. In § 4 we describe our obfuscator model design as well as its integration with a $N$-gram anomaly HIDS (§ 5), while § 6 shows how our obfuscator module is able to defeat both mimicry and IPE attacks. In § 7 we give some experimental results about our module. Paper ends with § 8, where some final remarks are provided.

## 2 Related Works

The idea of using syscall obfuscation for preventing computer intrusions has been introduced in [2], where an obfuscation scheme based on the randomization of the system call mappings has been used for dealing with some type of buffer overflows.

The mimicry attack has been introduced in [10] and extensively described in [11], where it has been shown that it can be applied to all HIDS models based on syscall tracing. We briefly recall that this kind of HIDS work in two subsequent phases. The first phase, called learning phase, is aimed at collecting the sequences of syscall invoked by a process, during its execution in a sterile environment. Such information, considered the process normal behavior, is subsequently used in the production environment for detecting any deviation of a process from its normal behavior and for raising alarms. In order to improve the resilience of HIDS to mimicry attacks, many improvements have been recently suggested. All these improvements are based on the same strategy: to record, together with any syscall, additional information which enables the HIDS, in the detection phase, to correctly distinguish between syscalls invoked by a monitored process or injected by malicious code. The information used so far for accomplishing such a task is the value of the program counter at any syscall invocation and the call stack configuration at the time of syscall invocation ([9, 10, 3]). When such strategies are adopted, the only thing which an attacker can do is to execute a single syscall with his own malicious parameters, but due to the use of call stack configuration, at the end of the syscall execution the control will return to the original code.

However, it has been shown in [7] that even such a limited power is enough to a clever attacker for mounting a mimicry attack. More precisely in [7] the authors describe some techniques which enable an attacker to regain control of the program execution flow after a syscall is completed.

IPE attack has been described in [10]. The most significant contribution on such an issue is probably contained in [3]. In such a paper the authors propose an anomaly detection method that utilizes the return addresses information extracted from call stack for fighting, among others, the IPE attacks. In Section 6 we will show a form of IPE attack which evades such a model of HIDS. Instead, this attack is captured by the model proposed in this paper.

## 3 Preliminaries

In this section we recall some basic notions on *mimicry* and IPE attacks as well as on kernel mechanisms and few definitions, which will be used throughout the paper.

### 3.1 Mimicry Attack

The mimicry attack was first described by Wagner *et al.* [11, 10] as an attack that can be performed on syscall-based HIDS. In its simplest form, to which we will refer to as *traditional mimicry*, it basically consists of attacking an application by *mimicking* one of the legal syscall sequences stored by the HIDS. System calls contained in a legal sequence which are not worth for executing the attack will be "nullified", and all the remaining will be truly executed. The mimicry attack bases its success on two assumptions:

1. the attacker knows the system calls traces yielded by the process and stored by the HIDS.

2. the attacker has full control over the execution flow either executing a malicious code [8] or using the code inside dynamic library [12].

On the basis of such assumptions, an attacker can choose a trace that is meaningful for his attack, and build an injection vector that will permit him to execute the selected system calls trace. Such a trace will contain "dummy" syscalls, that is, those used only to simulate the legal sequence, which will produce "null" effects, and others used by the attacker for specifying the malicious behavior needed to gain control of the system.

Hence, as depicted in Figure 1, only the meaningful syscalls are executed successfully while the ones that have to be mimicked are "nullified", for instance by simply making them to fail due to incorrect syscall arguments.

*normal sequence*: $\mathtt{S}_1\ \mathtt{S}_2 \mid \mathtt{S}_3\ \mathtt{S}_4\ \mathtt{S}_5\ \mathtt{S}_6$

where the *normal sequence* is the sequence provided by the process, learnt by the IDS, whereas $\mid$ represents the location of the vulnerability and $\mathtt{S}_i$ represents a generic syscall $i$.

*attack sequence*: $\mathtt{S}_3^s\ \mathtt{S}_4^s\ \mathtt{S}_5\ \mathtt{S}_6^s$

where the *attack sequence*, built by the attacker, comprises the simulated "nullified" system call $i$ ($\mathtt{S}_i^s$) as well as the system call the attacker wishes to execute ($\mathtt{S}_5$).

**Figure 1. Traditional Mimicry Attack**

### 3.2 Impossible Path Execution Attack

If properly recognized, an impossible path can be exploited by an attacker in order to execute application code in a way that would not be possible otherwise; security-critical checks as well as "jumping" over unwanted (from a security viewpoint perspective) code can be, more or less, easily bypassed by Impossible Path Execution (IPE) attacks.

As shown in [13, 3], some HIDS models are able to detect some kind of IPE attacks but fail in detecting all of them.

Figure 2 depicts an example of code snippet originally proposed by [3] and slightly modified in order to better show how an IPE attack can be successfully perpetrated, while remaining completely undetected by the most prominent HIDS models such as those proposed in [9, 10, 6]. Let us suppose that the function `is_regular(uid)` (line 20) invokes the `open` system call twice in order to open, respectively, `/etc/passwd` and `/etc/group` to check whether the given `uid` represents a regular user or not (implementation not shown). Afterwards, the *true* "if" branch will be executed if the user represented by `uid` has no particular privileges, whilst the execution will fall into the *false* one otherwise. Entering the true branch and "jumping" into the false one represents an impossible path. In Figure 3, an undetected IPE attack sequence performed against an N-gram HIDS model is reported. In particular, a regular user camouflaged as an attacker, by entering the true branch of the `if` statement (lines 21-27) and by exploiting the stack-based buffer overflow in `read_next_cmd` at line 8, is able to divert the program $p$ execution flow in order to enter the false branch which eventually will give him full privileges[1].

[1]For the sake of simplicity we assume the `system` library function invokes only the `execve` system call.

It may be argued that IPE attacks are difficult to be performed since they depend on too many factors (program structure, vulnerability "at the right position", common syscall sequences spread all over the execution flow, and so on) but, however, as pointed out by Feng *et al.* [3], they should not be left unconsidered since it may be quite easy, for an attacker, to deliberately introduce the "right" conditions in program source code that may lead to the execution of an impossible path.

```
 1: u_char *read_next_cmd(void) {
 2:
 3:     u_char input_buf[64];
 4:     u_char *p;
 5:
 6:     umask(2);
 7:     ...
 8:     strcpy(&input_buf[0], getenv("USERCMD"));
 9:     /* memory leak? :-) */
10:     p = (char *)strdup(input_buf);
11:     return p;
12: }
13:
14: void login_user(int uid) {
15:
16:     char *cmd;
17:
18:
19:
20:     if (is_regular(uid)) {
21:
22:         /* unprivileged mode */
23:         cmd = read_next_cmd();
24:         setuid(uid);
25:         /* yes, system is safe ;-) */
26:         system(cmd);
27:
28:     }
29:     else {
30:
31:         /* superuser! */
32:         cmd = read_next_cmd();
34:         setuid(0);
35:         system(cmd);
36:
37:     }
38:     return;
39: }
```

**Figure 2. Code snippet that might be exploited by performing a successful IPE attack.**

### 3.3 Syscall Invocation and Kernel Information

In this section we briefly recall how the Operating System (Linux Kernel) reacts when a syscall is invoked.

When a syscall is invoked by a process $p$, the CPU switches to kernel mode execution and some information such as the program counter (PC) and few registers are saved by the hardware itself onto the kernel mode stack. Afterwards, the kernel saves other information about the process state onto its own stack as well, and after performing

The execution of the code snippet shown in Figure 2 yields the following syscalls sequences (normal sequence), accordingly to the syscall-based HIDS rules presented in [6] (*3-grams* traces)

`O O U S E` *true* branch (`S` and `E` at lines 24, 26)
`O O U S E` *false* branch (`S` and `E` at lines 34, 35)

These sequences produce the following traces (it is worth noting that `S` and `E` are invoked by different memory locations but the $N$-gram model does not take it into account, considering them as the same syscalls).

```
O O U
O U S
U S E
```

By exploiting the stack-based buffer overflow vulnerability at line 8 (Figure 2), the attacker diverts the execution flow so that the syscalls `S` and `E` at lines 34-35 will be invoked while keeping the execution flow in-trace, accordingly to the learnt syscalls traces above reported.

**Figure 3. IPE attack**

some sanity checks, it retrieves the right syscall number and executes the corresponding kernel code that actually implement that system call. Among the data saved onto the kernel mode stack, we are particularly interested in the *Process Return Address (PRA)* and the *Function Return Addresses (FRAs)*, where:

- PRA is the address of the next instruction in $p$ to be executed once the syscall has been served.
- FRAs are the function return addresses stored on the program $p$ stack that are retrieved whenever a syscall-aware function, that is a function which invokes a syscall $s$, is executed. Using these addresses it is easy to determine the unique call site of $s$. FRAs can be obtained by "walking the stack" using the frame pointer in order to return back into the caller stack frame until we hit the `main` return address.

Syscall execution process is usually performed in the following three phases:

1. *save information*: in this phase information about the state of the running process is saved onto the kernel-mode stack.
2. *execute the syscall*: in this phase the kernel invokes the required system call service routine.
3. *restore information*: in this phase the kernel restores the PRA and FRAs the saved values in order to enable the process to carry on its execution.

## 4 The Obfuscator Module

In this section we will describe the architecture of our model, depicted in Figure 4, that is composed by two main components:

- *obfuscator module*.
- a HIDS based on $N$-gram model.

While the latter has already been extensively described in literature, we will concentrate our attention on the obfuscator module, the core concept of this work.

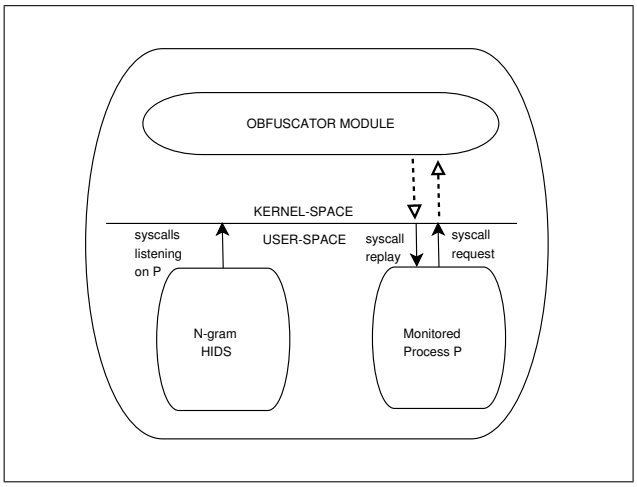


**Figure 4. System Architecture**

The main objective of the obfuscator is to introduce "random noise" in traces learnt by an HIDS, so that they cannot be replicated or mimicked by an attacker. The obfuscator works in a transparent way, without any modification of the underlying syscall-based HIDS.

The obfuscator module $o$ which we have devised works as follows. When a monitored process $p$ calls a syscall $s$, $o$ intercepts $s$ and checks if $s$ belongs to $p$ (i.e. it does not belong to an attack vector); in the affirmative case the syscall is executed with "null" effects $k$ times, and it is "normally" executed once[2] (where $k$ is a customizable obfuscator parameter, unknown to the attacker). Thus, the trace $t$ registered by the HIDS will contain $k$ instances of either the

[2]Technically speaking, there are user/kernel context switches but, if the syscall has just to be repeated, no kernel code implementing it will be executed during "null" syscalls execution.

same syscall or a different type of syscall. Generally speaking $t \in s\Sigma^*$, where $\Sigma$ is the alphabet of all the system calls.

**The Obfuscation Process**

In order to introduce unpredictability in the syscall traces, we have re-defined the kernel syscall invocation mechanism, which will work as follows. When a syscall $s$ is invoked by process $a$, the kernel will normally perform steps 1 and 2, but before the *restore phase*, it will modify the PRA in order to point to $s$ again. Thus, once $s$ is terminated, a kernel/user context switch will take place and the control will return again to $s$ which, this time, will be invoked by skipping the execution phase (*simulated syscall*). Technical details about such a technique are described in [1]. In this way given a syscall $s$, the obfuscated trace will have form $ss^*$; however, the mechanism described above can be easily extended in order to produce obfuscated traces of the form $s\Sigma^*$.

The obfuscation process works on any syscall produced by a monitored process and it turns out that, in case of a mimicry attack, the obfuscation process will be applied also to the injected syscalls nullifying any positive effect performed by the obfuscator. In order to avoid such a "weakness" the obfuscator has to be able to distinguish between "regular" syscall invoked by a process and "injected" syscalls. Consequently, we associate with any syscall two spatial coordinates represented by FRAs, PRA (see Section 3). At this point if the attacker injects the code in data area such as stack or heap and executes his own syscalls from there, these syscalls will be recognized as not belonging to the monitored program and, consequently, they will not be obfuscated. Thus, the resulting sequence will be out-of-trace and the HIDS will detect a trace different from the registered ones. The only way for the attacker to sneak the HIDS is either to predict the obfuscated sequence or to use different attack techniques such as the one proposed by Kruegel *et al.* [7] and few others described in [1].

# 5 Obfuscator & HIDS

In this section we will describe how our system can be used together with a syscall based HIDS. More precisely, we describe how the learning and the detection phases of the HIDS will be performed when the obfuscator is active for detecting intrusion attempts.

## 5.1 Learning Phase

The learning phase is performed in two main steps. In the first one, the HIDS is disabled and the obfuscator determines the information it needs in order to properly obfuscate system calls (*obfuscator learning* phase). Afterwards, the HIDS is enabled and the obfuscator starts to produce the obfuscated traces, while the HIDS performs its canonical learning phase.

More precisely, the *obfuscator learning* phase can be further divided in two sub-steps performed in the following order:

1. *On-line Learning*
   During this step, the obfuscator retrieves, for every system call $s$, generated by the process, the following information:

   - *Process Return Address (PRA)*;
   - *Function Return Address (FRAs)*;
   - *Syscall Number*: this information represents the type of syscall which is invoked by the process; it can be retrieved by the obfuscator since it is stored onto the kernel mode stack;

   This information, which represents the syscall coordinates (see § 3), is used to determine whether $s$ is invoked from different call site or not and, only in the affirmative case, will be stored in the obfuscator repository.

2. *Off-line Fixing*
   During this step, the obfuscator scans its repository obfuscating each syscall in an unique way (more details on this strategy will be given in § 6.1), determining the following parameters:

   - *Rep Syscall*: represents the number of times that a syscall must be obfuscated.
   - *Real Syscall*: this syscall represents the syscall that is really executed by the kernel and invoked by the process.
   - *Simulated Syscall*: a set of syscalls which contains the syscalls used to perform the obfuscation process. These syscalls and will never be executed by the kernel but will be registered by the HIDS inside the current trace.

In the naive model we can obfuscate all syscalls executed by $p$ but this approach implies big overhead during the *detection* phase. In order to optimize such an aspect, we decided to restrict the number of syscalls to obfuscate. Xu *et al.* in [13], have recognized 22 "dangerous" syscalls which can be used to take control of a GNU/Linux system. These syscalls represent good obfuscation points to mitigate the mimicry attack, so we decided to focus the obfuscation process on them. However, if we protect only dangerous syscalls, an attacker who knows $t$ trace, could mimic it until the dangerous syscall is reached, thus performing a successful attack. To avoid such a drawback , we have to protect all syscalls that:

- are "close" enough to the dangerous syscalls, by belonging to a fixed customizable neighborhood and,
- belong to the paths of the control flow which contain the dangerous syscalls.

Once the appropriate syscalls to obfuscate have been successfully collected, we employ static analysis in the learning phase of the obfuscation module. More precisely, before the on-line obfuscator learning phase (see § 5.1, item 1) takes place, we need to localize all the flow paths belonging to the dangerous syscalls and all syscalls that belong to such paths. Such a task can be performed as follows:

- the Interprocedural Control Flow Graph (ICFG) associated with the binary of the monitored program, is built;
- basic blocks containing dangerous syscall-aware functions are recognized and, through the ICFG, we also localize adjacent basic blocks which contain syscall-aware functions and whose flow passes through the dangerous syscall-aware functions basic block. These functions will represent the points on which to apply the obfuscation process;
- finally, once we gather all the basic blocks we are interested in, we collect for all the syscalls involved, the syscalls information, such as syscall type as well as its call site.

In the Figure 5, we show two examples of ICFG where inside the dark-grey basic blocks we found the dangerous syscall, whereas inside the light-grey basic blocks we found the syscalls belonging to its neighborhood that will be subjected to the obfuscation process.

Things change a little bit if we want to optimize the "detection" of IPE attacks. In fact, as pointed out in § 6.1, the learning phase may simply consist in looking for equal $N$-grams (that can be suitable target for an IPE attack in the $N$-gram model) inside the *whole* program syscall trace and obfuscate that sub-sequence in order to create a "unique" program syscall trace. For each sub-sequence obfuscated we store the addresses (FRAs and PRA) and the obfuscation parameters of the *real* syscall presents in such a subsequence.

### 5.2 Detection Phase

During the execution of a process $p$, every time a syscall belonging to the dangerous region is invoked, the obfuscator retrieves the current process obfuscation parameters from the kernel mode stack. In particular, it also checks whether the FRAs belong to the application code (process or library code areas) in order to be sure that all the return addresses have not been tampered with by the attacker. Afterwards, the obfuscator compares these obfuscation parameters with the ones saved during the learning phase. If a matching is found, the obfuscator performs the obfuscation operation on the syscall; otherwise , if no successful look up is obtained, the obfuscator module will not obfuscate that syscall.

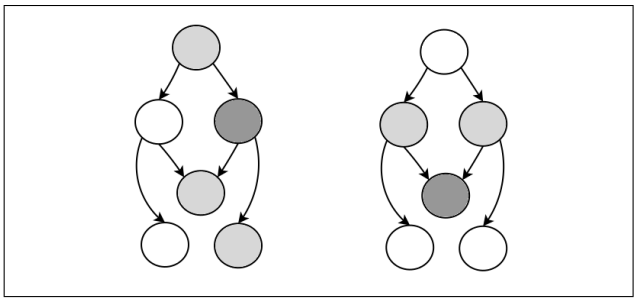

**Figure 5. Syscall Protection**

## 6 Effectiveness

In this section we will explain why the strategy we implemented through the obfuscator module, is able to prevent *IPE* and *traditional mimicry* attacks.

### 6.1 IPE

As just explained in § 3.2, in the IPE attack the attacker is able to use some syscalls that follow the appropriate trace, learnt by the HIDS, but which are positioned in different code locations. If the syscall trace provided by the program did not contain equal substrings ($N$-gram), then the attacker would not be able to jump to other piece of code and most forms of the IPE attacks will not be feasible anymore. Consequently our idea is to look for equal $N$-gram inside the syscall trace program and obfuscate that sub-string in order to create a "unique" syscall program trace, that is the "unique" $N$-gram inside the HIDS database. In the Figure 6 we show the obfuscation process applied to the vulnerable code reported in Figure 2; whilst the code reported in Figure 2 presents equal $N$-gram `USE` in different code locations (the first one at 6, 24, 26 and the second one at 6, 34, 35) allowing the attacker to perform the IPE attack; after the obfuscation process takes place, the syscall called at different call site are obfuscated in different ways, so the jump to line 23 to 34 (IPE attack), yield the $N$-gram $\mathtt{S}_3^r$ S $\mathtt{S}_7^r$ never learnt by the IDS defeating so the attack.

Our model has the same detection power of the $Vtpath$ [3] and in some cases is able to detect some kind of IPE attacks that $Vtpath$ can not. In fact the $Vtpath$ does not manage the $null$ virtual path which occurs when the same syscall is invoked more times at the same program location (loop statement); such information leakage may be

**System call obfuscation: defeating IPE attacks**

Giving the following *normal sequence* learnt by the HIDS, if the obfuscator obfuscated every syscall with another one of different type, the HIDS would see:

$$\texttt{O}\ \texttt{S}_1^r\ \texttt{O}\ \texttt{S}_2^r\ \texttt{U}\ \texttt{S}_3^r\ \texttt{S}\ \texttt{S}_4^r\ \texttt{E}\ \texttt{S}_5^r \quad \text{if E is } \textit{true}$$
$$\texttt{O}\ \texttt{S}_1^r\ \texttt{O}\ \texttt{S}_2^r\ \texttt{U}\ \texttt{S}_6^r\ \texttt{S}\ \texttt{S}_7^r\ \texttt{E}\ \texttt{S}_8^r \quad \text{if E is } \textit{false}$$

Giving an *attack sequence* such as `U S E`, where `S` and `E` are those of, respectively, lines `34` and `35` of Figure 2, thanks to the obfuscator work, the HIDS would see the $\texttt{S}_3^r\ \texttt{S}\ \texttt{S}_7^r$ trace which was not learnt before, rising up an alarm.

**Figure 6. Defeating IPE attacks**

exploited by an attacker to perform a successful IPE attack. As showed in Figure 7 the attacker exploits the vulnerability at line 13 (format bug), setting malicious $write$ parameters and bring the control flow back to write at line 10 (IPE Attack) making so a $null$ virtual path and eluding the $Vtpath$ model. Even if in the code showed in Figure 7 there's no any loop statement, the $Vtpath$ has to accept any null virtual path yielded by the program, because is not able to recognize if a particular syscall is invoked one or more times. Instead our model is able to defeat such an attack. As showed in Figure 6 every potential IPE program location is obfuscated in different way and the invocation of syscalls positioned in a particular location, one or more times, is detected.

```
1: char buffer[256] ;
2: ...
3:  setuid(0);
4:  fd = open(/etc/shadow, w) ;
5:  readsocket(luser, password, new_password) ;
6:  f_password = read_shadow_file(luser) ;
7:
8:  if(check_pwd(password, f_password))
9:  {
10:     write_shadow(fd, luser, new_password) ;
11:     update_memory_cache(luser, new_password) ;
12:
13:    snprintf(buffer, sizeof(buffer), newpassword) ;
14:    buffer[sizeof(buffer-1)] = '\0';
15:    strcat(buf, " password changed") ;
16:    syslog_entry(buf) ;
17:    }
18:    setuid(nobody);
19: close(fd) ;
20: ....
```

**Figure 7.** $Vtpath$ **Nullify Attack**

### 6.2 Traditional Mimicry

Suppose that an attacker is able to exploit a vulnerability and that he recognizes, inside the original program after the vulnerability, the trace $t = t_i, t_{i+1}, ..., t_k$ to be mimicked in order to perform a successful mimicry attack. Moreover, suppose that $c = c_i, c_{i+1}, ..., c_k$ is the corresponding array of obfuscation coordinates belonging to $t$. Thus, the attacker has two choices for executing $t$, namely either invoking the proper syscall $t_i$ at coordinates $c_i$ or invoking $t_i$ from another memory location different from $c_i$, such as data areas. In the latter case the obfuscator sees that the $t_i$ is invoked from an *unknown* memory location with respect to the learning it performed so, obfuscation process will not take place and the sub-sequence will be out-of-trace giving the HIDS, that runs on top of the obfuscator, the opportunity to raise an alarm. In the former case, instead, thanks to the $c_i$ syscall coordinate associated with the syscall $t_i$, the obfuscator will provide the appropriate trace. However, the attacker will not be able to regain the control of the execution flow because the PRA and the innermost FRA will make the execution flow to return at the memory location where the syscall-aware function was invoked.

## 7 Experimental Results

In this section will describe the set of experiments we ran to collect the measurements about the overhead introduced by our defensive mechanism and related results. For our experiments we used an Intel Pentium IV processor with 3 GHz clock, running Debian GNU/Linux operating system as a guest operating system on the VMWare 5.0 virtual machine, with the 2.4.30 Linux kernel and 92 MB of RAM.

In the first phase of our experiments we have measured three main pieces of code of our obfuscation model, providing three measurements for each of them: the best time, the average time and the standard deviation of execution; in particular we have:

- *Stack walk time*: this time represents the time needed to retrieve the FRAs sequence. To compute such a time we have considered that, for each protected syscall, we have to walk 6 stack frame on average. The obfuscator overhead in this case is $122\mu s \pm 507\mu s$ ($4.2\%$ overhead) on average (reporting $60\mu s$, i.e., $2\%$, on best).

- *Replay syscall time*: this measure represents the amount of time used to perform the context switch from kernel to user mode context and vice versa executed during the obfuscation process. The obfuscator overhead in this case is $144\mu s \pm 493\mu s$ ($8.5\%$ overhead) on average (reporting $63\mu s$, i.e., $5\%$, on best).

- *Hash table access time*: this measure represents the amount of time needed to access the hash table in order to retrieve the *obfuscation parameters*. This measure depends on the number of the syscalls invoked by the function called in the different program call site which are used to define the hash table size. We have considered that the hash table contained $500$ syscalls on average. Thus, the obfuscator overhead in this case is $114\mu s \pm 437\mu s$ ($6.5\%$ overhead) on average (reporting $61\mu s$, i.e., $1.6\%$, on best).

In the second phase of our test we have considered the server web Apache version 2.0.55-4, and a small dynamic web site with the following features: total size 500 KB, 12 static HTML pages, 6 CGI scripts, and 6 PHP scripts with an average page size of 4 KB. We have set our obfuscator module in order to replay four times the dangerous syscalls and their neighbors and we set the deep protection (the number of neighbor syscalls to obfuscate) to 2. In the first step we have collected data during the surfing of the site without the obfuscator module; afterwards we have measured the same surfing with the obfuscator on.

In the Figure 8, we have reported the *Total Syscall Execution Time* during the surfing with the obfuscator on, we show in light gray color the normal execution time of syscalls made by the Apache and in black one the overhead execution time inserted by our obfuscator module, the higher impulses in the middle of the graphic are associated to the I/O syscalls such as `new_select` and `poll`.

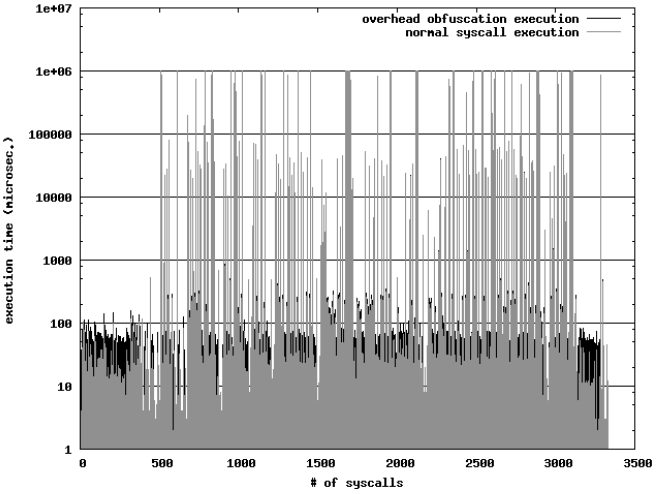


**Figure 8. Total Syscall Execution Time**

Our measurements show that for the whole surfing of the web site the Total Syscall Execution Time is $252.28ms$; we have added only $15.13ms$ delay for the obfuscation process, that is the $5.9\%$ total overhead. Consequently, we can consider our system low-overhead. More details about the measurements are described in [1].

## 8 Conclusion & Future Works

This paper presented a novel defensive technique, represented by the obfuscator module, which works in transparent way and low overhead (5.9%) with the higher accuracy than of the state of the art of HIDS [3]. We are working to use the obfuscator module in order to improve the false positive rate and to detect other kind of the IPE attacks.

## References


[1] D. Bruschi, L. Cavallaro, and A. Lanzi. Syscalls Obfuscation to Prevent Automatic Mimicry. Technical report, Dipartimento di Informatica e Comunicazione, Università degli Studi di Milano, 2006.

[2] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. *Technical report, CMU department of computer science*, 2002.

[3] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection using Call Stack Information. *IEEE Symposium on Security and Privacy, Oakland, California*, 2003.

[4] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 120, Washington, DC, USA, 1996. IEEE Computer Society.

[5] J. T. Giffin, S. Jha, and B. P. Miller. Detecting Manipulated Remote Call Streams. *11th USENIX Security Symposium*, 2002.

[6] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[7] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating Mimicry Attacks Using Static Binary Analysis. In *Proceedings of the USENIX Security Symposium*, Baltimore, MD, August 2005.

[8] E. A. O. Levy. Smashing the Stack for Fun and Profit. Phrack Magazine, Volume $0x07$, Issue $\#49$, Phile $14$ of $16$, 1998.

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

[10] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy, Oakland, California*, 2001.

[11] D. Wagner and P. Soto. Mimicry Attacks on Host Based Intrusion Detection Systems. *In Proc. Ninth ACM Conference on Computer and Communications Security.*, 2002.

[12] R. N. Wojtczuk. The Advanced return-into-lib(c) Exploits: PaX Case Study. Phrack Magazine, Volume $0x0b$, Issue $0x3a$, Phile $\#0x04$ of $0x0e$, December 2001.

[13] H. Xu, W. Du, and S. J. Chapin. Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths. *RAID LNCS 3224 Springer-Verlag*, pages 21–38, 2004.