UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Syscalls Obfuscation for Preventing Mimicry and Impossible Paths Execution Attacks

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

RAPPORTO TECNICO RT 10-06

Syscalls Obfuscation for Preventing Mimicry and Impossible Paths Execution Attacks

Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi

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

Abstract. We are interested in the models of anomaly-based Host Intrusion Detection Systems (HIDS) which have been built following the intuition that the "normal" behavior of a program can be characterized by the sequences of system calls (*N-gram*) it invokes during its executions in a sterile environment.

The N-gram model is very simple and very efficient but it is characterized by relatively high degree of *false alarms*, mainly because *correlations* among syscalls are lost. Furthermore, it has been shown that such a HIDS model is unable to detect two particular forms of computer attacks, namely *mimicry* and *Impossible Path Execution* (IPE). Quite recently, various authors started to propose variations to the N-gram model in order to improve its "precision", trying to overcome the limitations of the original model adopting a better characterization of a program behavior. However, even these models suffer of some limitations with respect to some forms of IPE and, with various degree of resistance, to some forms of mimicry attacks as well.

In this paper we address the IPE and the mimicry problem in the N-gram based HIDS model, and we provide a contribution which, for our point of view, will be very useful for the solution of the problem. More precisely, we devised a kernel-level module to which we will refer to as *obfuscator*, 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. Intuitively speaking, any attacker who wants to execute a traditional mimicry against a program, has to know at least a randomized trace which is, however, hardy predictable as its content strongly depends on the execution environment. The same reasoning can be applied, with some changes, to the case of IPE attacks.

We implemented a prototype of the obfuscator module on a Linux system and we have been able to experimentally verify that the idea is a viable solution to detection of both the mimicry and the IPE attacks. Furthermore, with respect to other solutions described in literature, it turned out that our module affected the performance of a testbed server with a slowdown factor of only 0.07%.

Key words: HIDS, Anomaly Detection, Mimicry Attacks, Impossible Paths Execution Attacks.

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 subcategories 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, which have been built following the idea initially introduced by Forrest *et al.* [5,8]. These systems are built following the intuition that the "normal" behavior of a program p can be characterized by the sequences of system calls it invokes during its executions in a sterile environment. In the original model the characteristic patterns of such sequences, known as N-grams, are placed in a database and they represent the language l characterizing the normal behavior of p. To detect intrusions, sequences of system calls of a given length are collected during a process runtime, and compared against the contents of the database. The Hamming distance between the collected string and l is computed, and when it exceeds a certain threshold, an alarm is raised by the HIDS.

The N-gram model is very simple and very efficient but it is characterized by a relatively high degree of false alarms [6], mainly because *correlations* among syscalls are lost, since there is no provision for storing information about the *position* where the syscalls are invoked. Furthermore, in [16] it has been shown that such a HIDS model is unable to detect two particular forms of computer attacks, namely the mimicry and IPE. Quite recently various authors started to propose variations to the N-gram model in order to improve its "precision", i.e. its ability to correctly detect a computer instrusion, 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 [12,16,4,7]). However, even these models suffer of some limitations. For example, in [16,4] it has been shown that the callgraph model proposed in [16] as well as the model proposed in [12] are not able to deal with some forms of IPE, while in [17,10] it has been shown that all the models above mentioned are susceptible, with various degrees of resistance, to some forms of *mimicry* attacks. In this paper we address the IPE and mimicry problem in the N-gram based HIDS model, and we provide a contribution which, for our point of view, could be very useful for the solution of the problem. More precisely, following an idea described in [3] we devised a kernel-level module to which we will refer to as *obfuscator* which interacts with an underlying HIDS, and whose main scope is to "randomize" sequences of system calls produced by an application in order to make them unpredictable by any attacker. Such a module, intercept a syscall s invoked by a process p, store the execution coordinates of s and force the kernel to execute s (or some other syscall s'), besides the normal execution, a further random number of times with "null" effects. Thus, the execution trace t produced by a single syscall s will be of the form $s\Sigma^*$, where Σ is the alphabet containing a symbol for any syscall. t will be registered by the underlying HIDS, which in our specific case, will be an N-gram HIDS, and anytime p is executed, and a syscall u is executed, the obfuscation module will be in charge of: verifying (using the syscall coordinates) that u belongs to the original code and not to a compromised version of it, and reproducing the same execution trace in accordance with that stored by the HIDS. Intuitively speaking, any attacker who wants to execute a traditional mimicry against p has to know at least a randomized trace t, which is however hardy predictable as its content strongly depends on the execution environment. The same reasoning can be applied, with some changes, to the case of IPE attacks.

In order to evaluate the feasibility of our idea we implemented a prototype of the obfuscator module on a Linux system. Using such an implementation we have been able to experimentally verify that the idea is a viable solution to detection of both the mimicry and the IPE attacks. Furthermore, with respect to other solutions described in literature, it turned out to be very efficient. We measured the overhead imposed by the obfuscator on an Apache web server during the navigation of a small dynamic web site. It turned out that our module affected the performance of the server with a slowdown factor of only 0.07%

The paper is organized as follows. 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. Technical details about the obfuscator implementation are given in § 7 and in § 8 preliminary experimental results will be reported. § 9 faces few evasion techniques that might be used against our approach as well as countermeasures that can be deployed in order to avoid such evasions. § 2 gives related works in the area whilst the paper ends with § 10 where some final remarks will be provided.

2 Related Works

The idea of using syscall obfuscation for preventing computer intrusions has been introduced by [3], 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 [16] and extensively described in [17], where it has been shown that it can be applied to all HIDS models based on syscall tracing. 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: record together with any monitored sycall additional information which enables the HIDS, in the monitoring phase, to check that the syscalls the monitored process is executing are invoked by the process itself and they are not invoked by any injected code as well as enabling the HIDS to check whether syscall-aware functions, that is, functions which eventually invoke syscalls, return in their right position of application code. Doing so, the attacker will not be able to keep control over the execution of his malicious code. The information used so far for accomplishing such a task are the value of the program counter at any syscall invocation and the call stack configuration at the time of syscall invocation ([12,16,4]).

When such strategies are adopted, the only thing which an attacker can do is to force an application to execute a single syscall which deviates from the normal behavior, but due to the use of call stack configuration, at the end of the syscall execution the control would return to the original code. Thus, given such a constraint, the realization of a mimicry attacks seems to be possible only from a theoretical point of view.

Instead, in [10] it has been shown that even such a limited power is enough to a clever attacker for mounting a mimicry attack. More precisely in this paper the authors describe some techniques which enable an attacker to regain control of the program execution flow after a syscall is completed. In particular, the alternation of invoking system calls and regaining control can be repeated until the desired sequence of system calls is executed.

IPE attack has been described in [16]. The most significant contribution on such an issue is probably contained in [4]. In such a paper the authors propose an anomaly detection method that utilizes return addresses information extracted from call stack, for fighting, among the others, the IPE attacks; various strategies, of increasing complexity, for performing such an attack have been introduced, and it has been shown that such an attack, when suitably crafted, can evade detection by all of the syscall-based HIDS.

Today both mimicry and IPE represent the biggest conceptual weaknesses of the HIDS concept as originally introduced by Forrest *et al.* and subsequently developed by other authors. The idea we present in this paper can be used for implementing a strategy which can contribute to solve such an issue.

3 Preliminaries

In this section we recall some basic notions on *mimicry* and *Impossible Path Execution* 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.* [17,16] 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 the legal sequence which are not worth for executing the attack will be "nullified", and all the remaining will be truly executed.

Recently, Kruegel *et al.* [10] propose a variation of the traditional mimicry attack, to which we will refer to as *automatic mimicry*. Up to date, automatic mimicry can defeat all existing syscall-based HIDS models.

Traditional Mimicry Attack The traditional mimicry attack bases its success on two assumptions that are usually satisfied:

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 once it has been hijacked toward the execution of the injected [11] or already present [18] malicious code.

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 somehow 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 those 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 actually executed successfully while the ones that have to be mimicked, are "nullified", by simply making them to fail due to incorrect syscall arguments, for example.

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

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

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

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

Fig. 1. Traditional Mimicry Attack

It is not difficult to see that the following proposition hold.

Proposition 1 (Traditional Mimicry Attack) *An attacker A can perform a traditional mimicry attack on a process p only if he knows the system call traces yielded by p.*

Automating Mimicry Attack One of the key point that a successful traditional mimicry exploits, is the possibility, for the attacker, to control the execution flow once it has been diverted. In fact,it is rather easy for an attacker, to perform such a task successfully[1].

[1] Assuming no particular OS protection mechanisms, such as Address Space Layout Randomization (ASLR) [15,1,13] and non-executable data area [14,9,15] are deployed.

Recently, however, Kruegel *et al.* [10] proposed a variation on the traditional mimicry attack as well as a *proof of concept* tool, which enable an attacker to have full control over the execution flow of a process by manipulating particular code pointers, thus circumventing the syscalls coordinates checks. In such a context, a vulnerability is usually triggered by overwriting particular *code pointers*, such as stack return addresses, Global Offset Table (GOT) entries[2], function pointers, longjmp buffers and so on.

3.2 Impossible Path Execution Attack

Impossible paths can be defined as a sequence of instructions that can never be executed under normal circumstances due to a particular program structure. A typical example of this situation is represented by an *if () then ... else ...* statement. If the CPU ends up by executing some instructions in the *true* branch, there is no way to jump into the *false* one[3] no matter how many training runs we performed. It is simply an impossible path to follow due to the structure of the program and the *if/then/else* semantic. If properly individuated, an impossible path can be exploited by an attacker in order to execute application code in a way that would not otherwise be possible; 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 [19,4] 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 [4] 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 [12,16,8]. 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` represent a regular user or not (implementation not shown). Afterwards, the *true* "if" branch is 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 pro-

[2] A dynamically-linked ELF executable makes use of a Procedure Linkage Table (PLT) and a Global Offset Table (GOT) in order to call library functions. The application performs a direct call to the library function PLT entry which in turn ends by indirectly calling the relocated library function thanks to the value stored in its GOT entry filled by the run-time dynamic linker (rtdl). The rtdl resolves the symbols at some point so that subsequent calls are performed without involving the rtdl itself, but by simply using the address stored in the corresponding GOT entry. From a security viewpoint, an unmonitored overwrite of such an entry can make code hijacking easily possible.

[3] As suggested by "best programming practice", we assume no *spaghetti code* at all, and hence no local jump, i.e. `goto`, from one branch to the other.

gram p execution flow in order to enter the false branch which eventually will give him full privileges[4].

It may be argued that IPE attacks could be difficult to perform 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.* [4], 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 an execution of an impossible path.

```
u_char *read_next_cmd(void) {

   u_char input_buf[64];
   u_char *p;

   umask(2);
   ...
   strcpy(&input_buf[0], getenv("USERCMD"));
   /* memory leak? :-) */
   p = (char *)strdup(input_buf);
   return p;
}

void login_user(int uid) {

   char *cmd;

   /* why do you call it "poor programming style"?! :-) */

   if (is_regular(uid)) {

      /* unprivileged mode */
      cmd = read_next_cmd();
      setuid(uid);
      /* yes, system is safe ;-) */
      system(cmd);

   }
   else {

      /* superuser! */
      cmd = read_next_cmd();
      setuid(0);
      system(cmd);

   }
   return;
}
```

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

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

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 [8] (*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.

Fig. 3. IPE attack

3.3 Syscall Invocation and Kernel Information

In this section we briefly recall how the Operating System kernel reacts when a syscall is invoked.

When a system call is invoked by a process p, the CPU switches to the 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 others information about the process state onto its own stack as well, and after performing few sanity checks, it retrieves the right syscall number and it 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 of p to execute 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 are saved onto the kernel-mode stack;

2. *execute the syscall*: in this phase the kernel invokes the correspondent system call service routine;
3. *restore information*: in this phase the kernel restores the PRA and 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*;
– an N-gram based HIDS.

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.

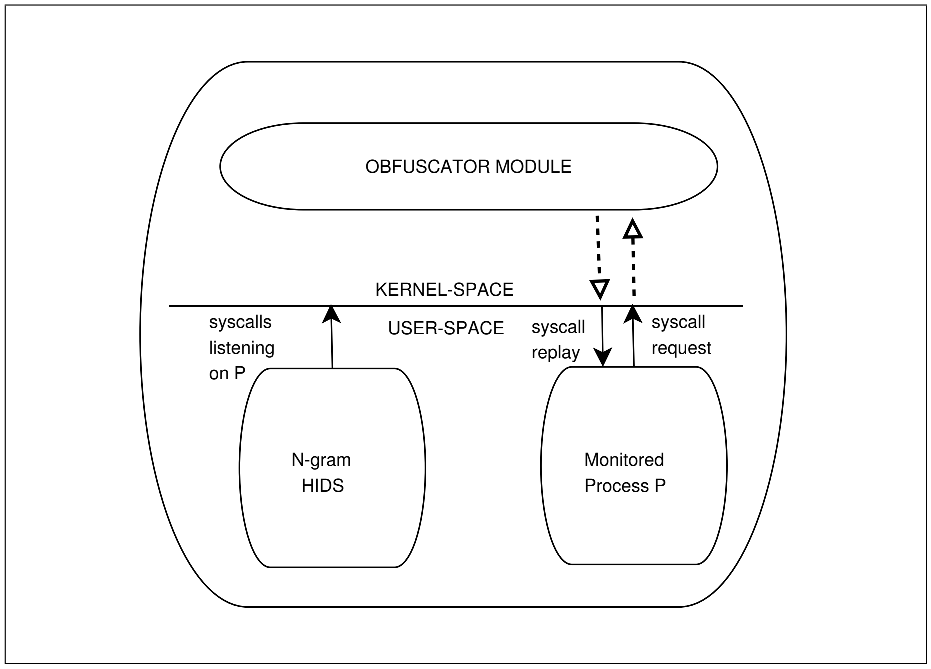

Fig. 4. System Architecture

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

The obfuscator module o 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[5] (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 same syscall or a different type of syscall. Generally speaking $t \in s\Sigma^*$, where Σ is the alphabet of all the system calls (see Figure 5).

Obfuscation phase

Giving a *normal sequence* $\mathtt{S}_1\ \mathtt{S}_2\ \mathtt{S}_3\ \mathtt{S}_4$, the obfuscator module produces an *obfuscated sequence* such as $\mathtt{S}_1\ \mathtt{S}_1^r\ \mathtt{S}_2\ \mathtt{S}_2^r\ \mathtt{S}_3\ \mathtt{S}_3^r\ \mathtt{S}_4\ \mathtt{S}_4^r$, where $\mathtt{S}_i^r$ represents the repeated system call i. It is this sequence that will be learnt by the IDS.

Fig. 5. Obfuscation Phase

In order to practically realize such a strategy two critical issues have to be addressed, namely how to distinguish between syscalls issued by a process and those issued by injected code, and how to execute "null" syscalls. In the following two sections we will explain the strategies we devised for solving such problems.

Recognize Proper Syscalls The first task which the obfuscator has to realize is to be able to distinguish between the syscalls generated by the original code and a compromised version of it. If the obfuscator were not be able to recognize the syscalls belonging to the monitored process, it would apply the obfuscation process on all syscalls even those provided by attacker, thus allowing the attacker to perform the attack successfully.

A syscall is said to belong to a given process, if it is called either from its own code segment, like a statically linked binary on UNIX systems, or from a code segment of mmap'd areas where usually dynamic libraries are mapped, like a dynamically linked binary on UNIX systems. Any other syscall called by other points of the process address space will not be subjected to the obfuscation process.

In order to recognize the syscalls that belong to the process and to correctly identify their call site, we define two spatial coordinates represented by FRAs, PRA pairs. Figure 6 shows a code snippet where it is possible to see:

Claim 1 *The PRA and FRAs address are enough in order to claim that a syscall is invoked by a given process as well as correctly infer if it is invoked from different memory locations.*

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 mon-

[5] 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.

itored 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.* [10] and few others described in § 9.

```
   0x080483a9:  push   %ebp
   0x080483aa:  mov    %esp,%ebp
   0x080483ac:  sub    $0x8,%esp
   0x080483af:  movl   $0x2,(%esp)
2) 0x080483b6:  call   0x80482a8    (PLT entry which calls 0x400d0fc0)
   0x080483bb:  leave               FRA
   0x080483bc:  ret
   ...
   0x080483c6:  mov    $0x0,%eax
   0x080483cb:  sub    %eax,%esp
1) 0x080483cd:  call   0x80483a9
   0x080483d2:  leave               FRA
   0x080483d3:  ret
   ...
   0x400d0fc0:  mov    %ebx,%edx
   ...
   0x400d0fc6:  mov    $0x3c,%eax
3) 0x400d0fcb:  int    $0x80
   0x400d0fcd:  mov    %edx,%ebx    PRA
   0x400d0fcf:  ret
```

Fig. 6. Spatial Coordinates

Assumption 1 (Function Return Addresses) *The function return addresses may be retrieved with low-overhead; for instance, we suppose that the binary was compiled with the frame pointer feature so that every function, even system call wrapper, presents a prologue and an epilogue.*

The Obfuscation Process In order to introduce unpredictability in the syscall traces, we have redefined the the kernel syscall invocation mechanism, for monitored processes, which will be performed as follows. When a syscall s is invoked by the process, the kernel will normally execute s but before the *restore phase* will take place, we modify the PRA and force a new invokation of s. Thus, once s is terminated, a kernel/user context switch will take place and the control will again return to s which, this time, however, will be invoked by skipping the execution phase (*simulated syscall*). In this way given a syscall s, the obfuscated trace will be of the form ss^*; however, the mechanism just described can be easily extended in order to produce obfuscated traces of the form $s\Sigma^*$.

In the latter case we obtain a system (obfuscator + HIDS) which is more robust against some types of attacks, such as *timing* and *bruteforce*, that will be showed in § 9. For the sake of simplicity we will refer to the first approach throughout the paper, unless differently stated.

5 Obfuscator & HIDS

In this section we will describe how our system can be used for detecting intrusion attempts. Such a task is performed in two phases, namely *learning* and *detection* phase. Scope of the first phase is to "collect" obfuscated syscall traces. While in the detection phase such traces will be used 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. Afterwards, the HIDS is enabled and the obfuscator starts to produce the obfuscated traces, while the HIDS performs its canonical learning phase. It is worth noting that the learning phase performed by the HIDS and obfuscator module must be the same, because if it were not the case the HIDS could learn traces which will not be obfuscated but could be used by the attacker to perform a successful mimicry attack.

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 that has to be obfuscated, invoked by the process p, 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;

   These information, which represents the syscall coordinates (see § 4) are used to determine whether s is invoked from different call site or not and, only in the affirmative case, they 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, 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. For performing such a task we referred to the work of Xu *et al.* who, in [19], have individuated 22 "dangerous" syscalls which can be used to take control on 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. In order to make effective the protection of dangerous syscalls, we also have to protect syscalls belonging to their neighborhood; more precisely we have to protect all the 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.

If we protected only dangerous syscalls we could enable an attacker who knows the trace before and after the protected syscall, to mimic it and reach the dangerous syscall and executes it at least once, performing successfully the attacks. So, to avoid this, 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 Control Flow Graph (CFG) associated with the binary of the monitored program, is built;
– basic blocks containing dangerous syscall-aware functions are individuated and, through the CFG, we also localize adjacent basic blocks which contain syscall-aware functions and whose flow reaches the dangerous syscall-aware functions basic block. These functions will represent the points on which 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 7, we show two examples of CFG 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.

This simple flow analysis allow us to cover all the paths which reach the dangerous syscalls. This is very important for our goal because if we were not able to cover all the flows, the attacker would be able to elude the obfuscation process by using a legal path that will lead him to reach the dangerous syscall (legal paths are the paths that are not learnt by the HIDS but may be considered like a false positive because the number of alarms associated to them, out-of-trace syscalls, are below the HIDS threshold). 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 way of the *real* syscall presents in such a sub-sequence.

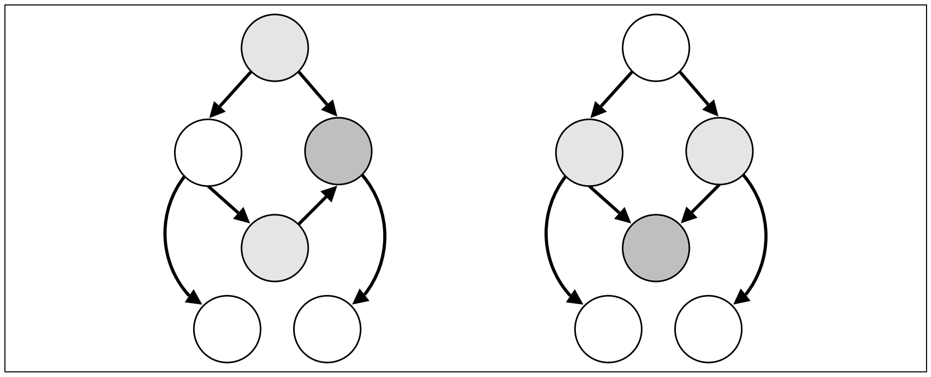

Fig. 7. Syscall Protection

5.2 Detection Phase

The detection of a computer misuse on a system provided by the obfuscator module just described and a HIDS, will be performed as follows. During a process p runtime, every time a syscall 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 perform the obfuscation operation on the syscall, and the HIDS will detect a legal trace of execution. On the other hand, if no successful look up is obtained, the obfuscator module will not obfuscate any syscall at all and the IDS will signal the anomaly.

6 Defeating IPE and Mimicry Attacks

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

The ability to recognize syscalls give the obfuscator the information needed in order to choose whether obfuscate a given syscall or not. So, the goal of such a mechanism is twofold in defeating both IPE and mimicry attacks as showed in the following Sections.

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 contains 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 8 we show the obfuscation process applied to the vulnerable code reported in Figure 8; 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), would yield the N-gram $\mathtt{S}_3^r$ S $\mathtt{S}_7^r$ never learnt by the IDS. The power of anomaly IPE signal detection may be increased by enlarging the number of the "simulated" syscalls used by the obfuscation process.

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:

O $\mathtt{S}_1^r$ O $\mathtt{S}_2^r$ U $\mathtt{S}_3^r$ S $\mathtt{S}_4^r$ E $\mathtt{S}_5^r$ if the expression E gives *true*
O $\mathtt{S}_1^r$ O $\mathtt{S}_2^r$ U $\mathtt{S}_6^r$ S $\mathtt{S}_7^r$ E $\mathtt{S}_8^r$ if the expression E gives *false*

which produces the following traces

O $\mathtt{S}_1^r$ O
$\mathtt{S}_1^r$ O $\mathtt{S}_2^r$
O $\mathtt{S}_2^r$ U
$\mathtt{S}_2^r$ U $\mathtt{S}_3^r$
U $\mathtt{S}_3^r$ S
$\mathtt{S}_3^r$ S $\mathtt{S}_4^r$
S $\mathtt{S}_4^r$ E

$\mathtt{S}_4^r$ E $\mathtt{S}_5^r$
$\mathtt{S}_2$ U $\mathtt{S}_6$
U $\mathtt{S}_6$ S
$\mathtt{S}_6$ S $\mathtt{S}_7$
S $\mathtt{S}_7$ E
$\mathtt{S}_7$ E $\mathtt{S}_8$

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 $\mathtt{S}_3^r$ S $\mathtt{S}_7^r$ trace which was not learnt before, rising up an alarm.

Fig. 8. Defeating IPE attacks

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 s_i at coordinates c_i or invoking s_i from another memory location different from c_i, such as data areas. In the latter case the obfuscator sees that the s_i is invoked from an *unknown* memory location with respect to the learning it performed so, obfuscation process will 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 s_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 Technical Details

In this section we provide the implementation details of our obfuscation module based on the Linux kernel, 2.4.x version, running on 32-bit Intel Architecture. We will describe the main memory structure needed by the obfuscation process and the algorithm used to apply the obfuscation on the monitored processes.

`task_struct` Modification We decided to store the process obfuscation information into the in-kernel `task_struct` structure. This structure is used by the kernel to hold information about the process, such as its state, its mapped address space, and so on. We chose such a structure since it is present in memory for all the process lifetime. In particular we modified the `task_struct` adding the following fields:

– *obfuscation state*: this flag is used by the obfuscator in order to know whether the process is in the obfuscation state or not;
– *syscall hash table*: every hash table entry contains the information (obfuscation parameter) provided by the obfuscation learning phase; more precisely we have PRA, FRAs, syscall number and rep syscall;
– *syscall return value*: the purpose of this field is to permit to return back to the process, i.e., it contains, the real syscall return value upon obfuscation termination.

Obfuscation Algorithm The obfuscation algorithm is implemented inside the code that is in charge of handling syscalls. Such a mechanism is implemented in `entry.S` and is divided into three main steps:

1. after the software interrupt is invoked by the process, the CPU switch in kernel mode and saves few information onto the kernel mode stack. Afterward, the execution is passed to the `system_call` entry point, where the kernel keeps going on to handle the system call trap, by saving the process state information onto its own stack and then retrieving the pointer to the `current` process as depicted by the following code snippet:

```
system_call:
   pushl %eax
   SAVE_ALL
   movl %esp, %ebx
   andl $0xffffe000, %ebx
   ...
```

2. some sanity checks are performed, such as testing whether the syscall is valid (the syscall is valid if the syscall number is inside a range of syscalls actually implemented in the kernel) so that the right syscall handler can be executed or not. The following code snippet describes such a phase:

```
cmpl $(NR_syscalls), %eax
jae badsys
...
call *sys_call_table(0, %eax, 4)
...
```

3. At the end of the `system_call` syscall handler before returning back in user space, the kernel saves the syscall return value (`%eax` register) onto its kernel mode stack and set the things up for returning in user space, as shown by the following code snippet:

```
   movl %eax, 24(%esp)
ret_from_sys_call:
   ...
   RESTORE_ALL
```

Our modification is placed between the control for the syscall validity and the syscall handler invocation (second step); the pseudo code of our modification is the following:

```
get_process_information(pid)
if (process must be obfuscated) {

   if (task_struct[process].obfuscation == 1) {

   # rewind the return address at the previous
   # instruction, i.e. at the begin of int $0x80
      return_process_address -= 4

   # decrement the number of times the syscall
   # must be repeated
   rep_syscall-- ;
   if (rep_syscall == 0)
      task_struct[process].obfuscation = 0;
}
else {

   # look for the obfuscation parameter
18:    # into the syscall hash table
19:    check_syscall_obfuscator(PRA, FRAs, sysnum)
20:
21:    # tell the kernel to obfuscate any subsequent syscall
22:    # of this process but this one
23:    task_struct[process].obfuscation = 1
24:    exec_syscall_handler()
25: }
26:}
```

In our code, we can recognize two main obfuscation phases:

- the first phase is performed when the process gets marked in order to be obfuscated; this state is described by the code between lines 15 and 25.
- the second phase is performed when the process has already been marked for obfuscation so just syscall "bouncing" has to be performed as shown in the above code snipped between lines 3 and 14.

Every time a process invokes a syscall, the obfuscation code gets the process information (line 0) and, using them, checks in which state the process is; if the process must enter in the obfuscation phase, the code retrieves the syscall *obfuscation parameter* (line 19) for that syscall and update the `task_struct` obfuscation flag. Otherwise, if the process is already inside the obfuscation phase, the code rewinds the process return address in order to point at the begin of the syscall interrupt instruction again (line 7), updating the *rep syscall* parameter as well. When the *rep syscall* reaches 0 (line 13), the code sets the process obfuscation flag state to 0 (not obfuscated) which means that the process obfuscation on that syscall is finished.

8 Experimental Results

As a method to test the effectiveness of the obfuscator module devised in this paper, we tried the same mimicry attack performed by Wagner *et al.* [17] against the server `wuftpd` version `wu-2.4.2-academ[BETA-15](1)` running on a Debian GNU/Linux operating system with an N-gram based HIDS like the one in [8]. The empirical test were conducted on two kind of scenarios; the first with the obfuscator module "turned on" while the second one with it "turned off". The scenario which had not the obfuscator running showed, obviously, that a traditional mimicry attack was possible. On the other hand, such an attack totally failed in the other case. The same holds for IPE attack as well.

The rest of the 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.

Our module acts on the code used by the kernel in order to manage the syscalls, so we focus our attention on those routines, defined into the `entry.S` file. In order to measure time lapses we use the “timestamp counter” processor register. The counter, available on all kinds of Pentium processors is a 64 bit register that gets incremented at each clock tick. Using this measure we are able to provide the most accuracy measurement of the system.

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). See Table 1 for further details.
– *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). See Table 2 for further details.
– *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). See Table 3 for further details (the hash table holds 500 syscalls on average).

	Best time	Average Time	Standard Dev. Time	N. Trials
stack walk	60 μs (181185 ticks)	122 μs (367404 ticks)	507 μs (1523697 ticks)	10000
no stack walk	58 μs (174908 ticks)	117 μs (353906 ticks)	484 μs (1454984 ticks)	10000
Overhead	2 μs (3.4%)	5 μs (4.2%)	NA	NA

Table 1. Stack walking measurement

	Best time	Average Time	Standard Dev. Time	N. Trials
replay syscall	63 μs (188572 ticks)	114 μs (343328 ticks)	493 μs (1481410 ticks)	10000
no replay syscall	60 μs (180855 ticks)	105 μs (315372 ticks)	410 μs (1230820 ticks)	10000
Overhead	3 μs (5%)	9 μs (8.5%)	NA	NA

Table 2. Syscall replay measurement

	Best time	Average Time	Standard Dev. Time	N. Trials
Access Hash Table	61 μs (182040 ticks)	114 μs (342760 ticks)	437 μs (1313528 ticks)	10000
no access Hash table	60 μs (180757 ticks)	107 μs (322267 ticks)	443 μs (1330344 ticks)	10000
Overhead	1 μs (1.6%)	7 μs (6.5%)	NA	NA

Table 3. Hash table access measurement

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 9, 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`.

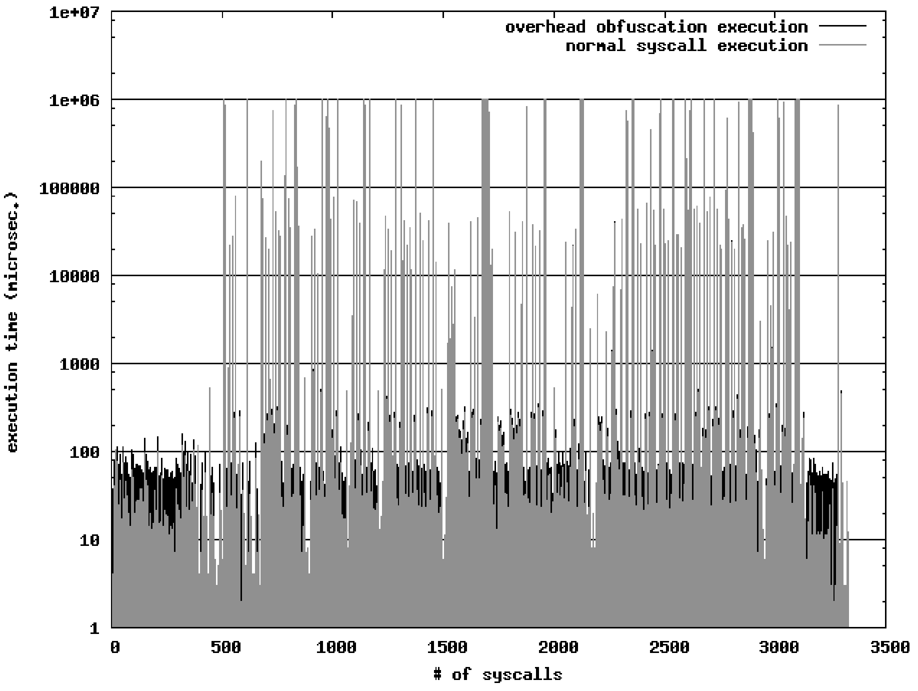

Fig. 9. Total Syscall Execution Time

In Figure 10 and 11, we have reported the same data-set collected during the surfing, showing for each syscall made by the Web server the normal execution time on average (light grey) and the overhead execution time on average (dark). We have labelled each data with the correspondent syscall number.

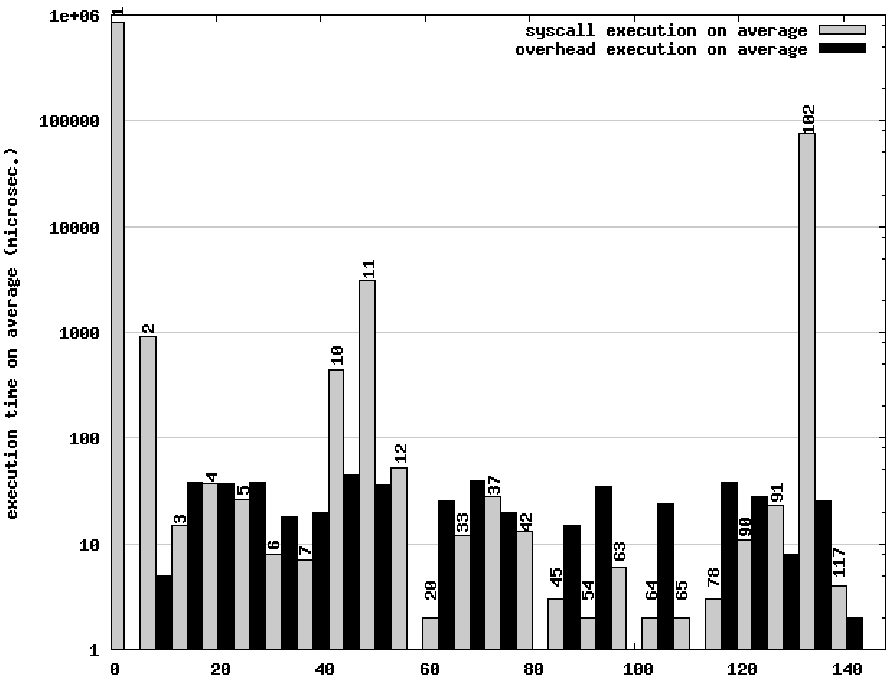

Fig. 10. Average Syscall Execution Time (1)

Our measurements show that for the whole surfing of the web site the Total Syscall Execution Time is $106.481s$; we have added only $75.131ms$ delay for the obfuscation process, that is the 0.07% total overhead. Consequently, we can consider our system a very low-overhead one.

9 Evasion Techniques

We recognized some evasion techniques that might be used to elude the obfuscator system. For each technique we will describe its base concept and we will discuss about the countermeasures that can be adopted.

9.1 Bruteforce Attack

In order to elude the obfuscator module, an attacker could perform a *brute force* attack trying to obtain the obfuscated trace yielded by the obfuscator module. In fact, if an at-

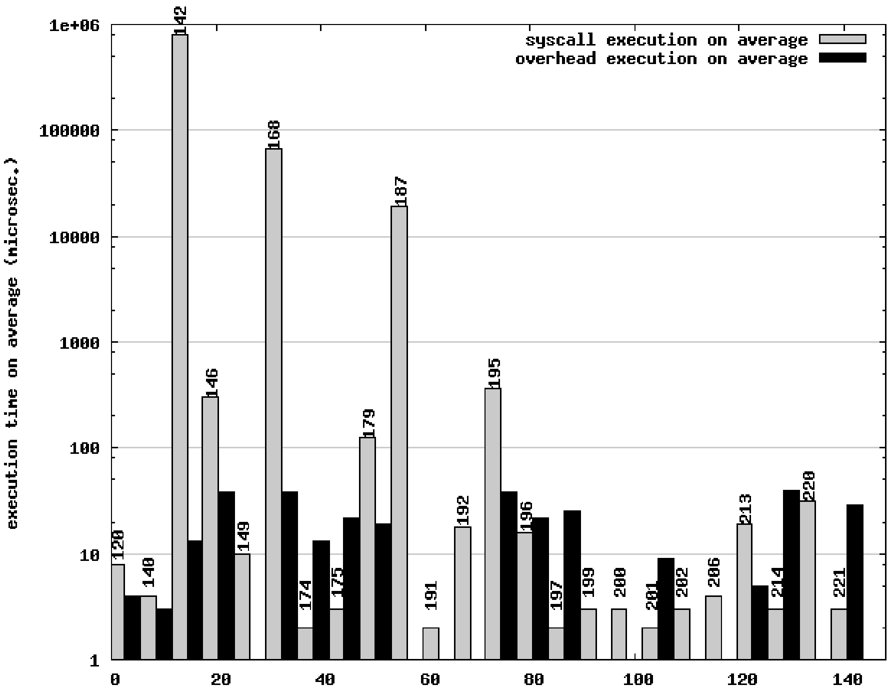

Fig. 11. Average Syscall Execution Time (2)

tacker were able to build the appropriate obfuscation trace, then the HIDS would accept it, even if the code was executed in data area such as stack or heap. Easy combinatorial formulæ can be used to show that such an attack is not feasible.

9.2 Timing Attack

In the case our obfuscator module adopts the strategy of repeating the same syscall k times, a timing attack can be realized for predicting the number of times that a particular syscall has been repeated. In order to achieve such a task, the attacker must gather the following information:

- *Tse (Time syscall execution)*: this measure represents the amount of time that a particular "real" syscall, takes in order to be executed.
- *Tsr (Time syscall replay)*: this measure represents the time used to replay the syscall, that is the time to perform the context switch between kernel-space to user-space and vice versa.
- *Tne (Time network environment)*: this measure is needed only if the attacker performs this attack in network environment, and it represents the round-trip delay introduced by the network path.

– *Tas (Time structures access)*: this measure represents the time to access the structure of obfuscator used to retrieve the *obfuscator parameter*. This time may be accuracy compute because the obfuscator structures are known.
– *Ttr (Time total response)*: this measure represents the times of reply to perform a particular operation on behalf of the protected process, for sake of simplicity we consider that the operation is associated to one syscall.

Once an attacker retrieves these information, he is able to compute the formulæ 1 in order to obtain the number of times a particular syscall is issued.

$$Nreps = \frac{Ttr - Tas - Tne - Tse}{Tsr} \tag{1}$$

In order to mitigate such an attack we can obfuscate the execution time by introducing a fixed quantum time delay as big as the Tsr time. These quanta of time obfuscate the real number of times that a syscall is repeated, so the attacker will not be able to build the appropriate trace.

9.3 Automatic Mimicry Attack

The key aspect of the automatic mimicry is the execution flow regain, that is a technique the permit to regain the control of the execution flow at some points, in order, for example, to permit the attacker to provide malicious parameters associated to "dangerous" syscall. Moreover, this attack does not even need to know a proper syscall trace and this is the main reason why our obfuscation technique fails to detect it. However, the main obfuscator architecture herein proposed, may be used, properly adapted, in order to prevent such an attack as well. Preliminary ideas about defeating automatic mimicry attacks using our obfuscator model as main framework are an ongoing project and can be found in [2].

10 Conclusion & Future Works

This paper presented a novel defensive technique, represented by the obfuscator module, which works in transparent way and low overhead, and it is used in order to protect the N-gram based HIDS against known attacks such as mimicry attack and most forms of IPE. We are working in order to improve the obfuscator to defeat the automating mimicry and all forms of IPE, to detect by data-flow analysis non-data control attacks and to improve the signal anomaly in order to distinguish between anomalies yielded by false positive and by some attacks.

References

1. S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In USENIX *Security Symposium*, 2003.
2. 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.

3. M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. *Technical report, CMU department of computer science*, 2002.
4. 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.
5. 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.
6. A. K. Ghosh and A. Schwartzbard. A Study in Using Neural Networks for Anomaly and Misuse Detection. In USENIX *Security Symposium*, 1999.
7. J. T. Giffin, S. Jha, and B. P. Miller. Detecting Manipulated Remote Call Streams. *11th USENIX Security Symposium*, 2002.
8. S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.
9. iSec.pl Development Team. kNoX - Implementation of non-executable Page Protection Mechanism. http://www.isec.pl/projects/knox/knox.html.
10. 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.
11. Elias "Aleph One" Levy. Smashing the Stack for Fun and Profit. Phrack Magazine, Volume $0x07$, Issue $\#49$, Phile 14 of 16.
12. 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.
13. The Linux Kernel 2.6 Development Team. The Linux Kernel 2.6. http://lwn.net/Articles/121845/.
14. The OpenWall Development Team. The OpenWall Project. http://www.openwall.com.
15. The PaX Team. The PaX Project.
16. D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy, Oakland, California*, 2001.
17. D. Wagner and P. Soto. Mimicry Attacks on Host Based Intrusion Detection Systems. *In Proc. Ninth ACM Conference on Computer and Communications Security.*, 2002.
18. Rafal "Nergal" Wojtczuk. The Advanced return-into-lib(c) Exploits: PaX Case Study. Phrack Magazine, Volume $0x0b$, Issue $0x3a$, Phile $\#0x04$ of $0x0e$, December 2001.
19. 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.