Static Analysis on x86 Executables for Preventing Automatic Mimicry 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. In 2005, Kruegel *et al.* proposed a variation of the traditional mimicry attack, to which we will refer to as *automatic mimicry*, which can defeat existing system call based HIDS models. We show how such an attack can be defeated by using information provided by the Interprocedural Control Flow Graph (ICFG). Roughly speaking, by exploiting the ICFG of a protected binary, we propose a strategy based on the use of static analysis techniques which is able to localize critical regions inside a program, which are segments of code that could be used for exploiting an automatic mimicry attack. Once the critical regions have been recognized, their code is instrumented in such a way that, during the executions of such regions, the integrity of the dangerous code pointers is monitored, and any unauthorized modification will be restored at once with the legal values. Moreover, our experiments shows that such a defensive mechanism presents a low run-time overhead.

1 Introduction

In their seminal work [13,12] about anomaly-based Host Intrusion Detection System (HIDS), Forrest *et al.*, introduced the idea that anomalous behavior of a process p can be detected by learning the sequences of system calls executed by p in a sterile environment, comparing them against those executed in a production environment, and detecting any significant deviation among them. Such an approach has been investigated by many researchers, who proposed several improvements over the original model, thus obtaining more efficient and more precise (i.e., which recognize broader classes of intrusions) anomaly detection HIDS. In [31,30] Wagner *et al.* observed that all the system call-based HIDS suffer a particular form of attack called mimicry. In its simplest form, to which we will refer to as *traditional mimicry*, it basically consists of forcing a process to execute an attack vector by *mimicking* the system calls sequences and learnt by the HIDS. Subsequently, several strategies (see [23,11,5]) have been proposed for inhibiting traditional mimicry attacks.

In a recent paper, Kruegel *et al.* [17] observed that even if the introduction of such techniques in anomaly-based HIDS [5,11,23] has significantly reduced the possibility to perform successful traditional mimicry attacks [26,25,31], they do not impose any kind of restriction on the execution of arbitrary code which does not directly invoke system calls (i.e., system call-free code). For example, a piece of code that is able to modify writable memory segments represents a threat by itself. This observation, brought Kruegel *et al.* to devise a variation of the traditional mimicry attack which is able to hijack a program execution flow, execute malicious system call-free code, relinquish the execution flow to the diverted program to regain it later on.

This malicious code is usually executed as a preamble of in-trace syscalls. Its main objective is either to change the value of the system call parameters in order to eventually execute arbitrary code, or to modify the value of some control-dependent data variable in order eventually influence the process execution flow. In [17] a proof of concept tool is provided which is able to automatically identify, inside a program, the instructions which can be used for such a scope. For this reason we refer to such an attack as automatic mimicry. More precisely, the main goal of the automatic mimicry is to elude HIDS checks by continuously diverting the process execution flow in order to execute arbitrary code with the purpose of changing system calls parameters without directly invoking any system call. However, most of the time these steps cannot be completed at once. Thus, any piece of malicious code has to take care of continuously regaining the control of the execution flow. Such a task is usually performed by modifying appropriate code pointers.

On the basis of the previous observation we devised a strategy for containing automatic mimicry attacks. Such a strategy consists of localizing, inside a IA32 binary p, all the *dangerous regions* $a_i, \cdots, a_n$, where by dangerous region, known also as *liveness* area, we mean the code area between the definition d and use u of the values v of the system calls parameters. After the liveness areas have been determined we collect, at run-time, for any area a_i $1 \leq i \leq n$, the "trusted values" $t_1, \cdots, t_k$ of the code pointers defined in a_i. Subsequently, we instrument the process p image so that at run-time code pointers in a_i will always be restored to their corresponding trusted values, before their use. Consequently the attacker will not be able to regain the control of p's execution flow and the attack will not be feasible.

A static analysis tool and a kernel-level module on a Linux system have been developed in order to assess the viability of our approach. Several experiments has been performed both for verifying the correctness of the approach and its overheads. The results obtained showed that our strategy defeats the automatic mimicry attack guaranteeing a low overhead impact in term of process execution time.

The paper is organized as follows. Related works are described in § 2 while § 3 introduces some preliminary notions about static analysis and automatic mimicry attacks that will be useful throughout the paper. The core of our code pointers integrity verification is faced in § 4, and § 5 shows the effectiveness of this defensive mechanism. Technical details and experimental results are given in § 6 and § 7, while conclusion, future works and final remarks are given in § 8.

2 Related Works

Generally speaking, memory error exploits which corrupt code pointers aim at pursuing two main goals (or a combination of them), that is, (i) to perform IPE attacks [30] (to bypass security critical checks), and (ii) to execute arbitrary malicious code.

Several strategies have been proposed to deal with this problem. Some of them, such as StackGuard [7] (SG) and Pro Police Stack Detector (also known as Stack Smashing Protector, i.e., SSP) [10], aim at protecting the integrity of particular code pointers (e.g., return address for SG and mainly return address and saved frame pointer for SSP). However, beside stack smashing attacks [18], they do not address other kind of memory error exploits based on the corruption of others code pointers (GOT, `.dtors`, heap management information, and so on).

In [1], Abadi *et al.* propose Control-Flow Integrity (CFI), an approach to guarantee the integrity of the execution control flow of a protected application p. By forcing p's execution to dynamically follow only paths defined by its Control Flow Graph (CFG), their approach defeats attacks which, as a final goal, attempt to hijack a program execution flow to alter its behavior. CFI leverages on fewer assumptions to achieve its goals. In particular, it relies on non-writable code, and non-executable data segments. While, generally, these are common sense requirements, as noted by the authors, the assumptions can be somewhat problematic in the presence of self-modifying code, run-time code generation, and the unanticipated dynamic loading of code.

Program shepherding, proposed by Kiriansky *et al.*, monitors control flow transfers to enforce a security policy [16]. While CFI could be enforced by program shepherding, the approach proposed by Kiriansky *et al.* is more general. In fact, it prevents execution of data or modified code and ensures that libraries are entered only through exported entry points, without making any assumption a priori. Moreover, program shepherding provides sandboxing that cannot be circumvented, allowing construction of customized security policies. On the other hand, this monitoring technique may impose a quite moderate overhead for certain types of programs. Moreover, existing code attacks can be stopped only in some cases.

In [29], a technique based on process address space layout randomization (ASLR) has been proposed and realized by developing a kernel level patch which is in charge of loading the process' memory segments (code, data, heap, stack and `mmap`'d region) at different, randomized memory locations. Since no knowledge on the process behavior or structure is required, the approach can only guarantee the randomization of the segments base addresses but it lacks a more fine-grained randomization. Unfortunately, the approach is vulnerable to information leakage attacks or it has been proved to be not so effective on 32-bit Intel Architecture platforms [14].

Other address obfuscation techniques have been proposed in [21,20] by Bhatkar *et al.* as a particular form of program transformations to combat memory error exploits. Such approaches differ from the one proposed in [29] since they aim at providing a more fine-grained address space obfuscation. The objectives of these transformations are to randomize the absolute locations of all code and data in order to achieve protection from a broad class of memory corruption attacks, and to randomize the relative distance between different data objects in order to defeat relative addressing attacks, which are a subclass of non-control data ones [6]. To this end, various obfuscating transformations have been proposed; they range from the randomization of the base addresses of common memory regions (stack, heap, `mmap`'d area, text and static data), the permutation of the order of variables and routines, and the introduction of random gaps between objects. A further improvement over such an idea has been proposed in [21], where a source-to-source transformation on C programs has been developed to produce self-randomizing programs.

All the aforementioned randomization approaches share a common concept: they provide a *probabilistic* defensive mechanism that, in general, cannot provide *certainty* in protecting from memory errors exploits. In this sense, quite recently, newer approaches have been devised [9,4] that make use of diversified process replicæto provide protection from a broad class of memory error attacks which mainly corrupt application's code and data pointers. Even if the approach seems sound, promising, and an on-going research topic, it currently presents a quite high overhead, and fewer practical not fully solved limitations involving the management of shared memory, signals, and threads.

In this paper, we address the problem of memory error attacks which corrupt code pointers in order to perform an automatic mimicry attack. We believe that our technique can defeat most of the memory error attacks, while experiencing a low overhead and a transparent deployment[1] in all the HIDS architectures. It is worth noting that our technique is symbiotic with the HIDS and, consequently, several checks about stack integrity, some form of traditional mimicry and some IPE attacks, are performed by the HIDS itself.

3 Preliminaries

In this section we recall some basic notions about program static analysis that will be useful to understand our approach, as well as further remarks on the automatic mimicry attack.

Liveness Analysis. Given a program p, we use data-flow analysis techniques in order to gather information about the data used by p. In particular, we use the *liveness analysis* to define the liveness region of the program. From our point of view, a liveness region is a sequence of instructions where a particular system call parameter is alive; a parameter is alive if it holds a value that will/might be used in the future. Figure 1 shows the liveness area of the variable a, which is defined at line 6 and used at line 10. All application paths defined between the *definition* and *use* of the variable belong to the liveness area of the variable itself. More precisely:

- **Definition:** the *definition* of a variable occurs when it is defined either by input-related system call-aware functions, that is, functions that eventually invoke system calls (e.g., `read`, `recv`, `fgets`), or, in according to the classic definition, with an assignment of that variable. More precisely, we can classify the assignments in two main categories:
  - *dynamic assignment*. Such a kind of assignment is associated to the data coming directly from the input. In such an assignment are involved all the input-related system-call aware functions;
  - *static assignment*. Such a kind of assignment is associated to the data whose values do not come from input but are statically defined into the application by constant values.

[1] That is, without modifying neither the HIDS nor the binary code.

- **Use:** the *use* of a variable occurs when it is used by some security sensitive system call (e.g., `write`, `execve`, `read`) or by some function which eventually invokes security sensitive system calls (e.g., `fprintf`). Any modification of the variable achieved through arithmetic transformation, is not considered like an use. Roughly speaking, we consider the use of a variable when it is only used by some security sensitive routine.

```
int main(int argc, char **argv)

   int a, b, c;
   c = 40;
   b = 30;
   a = 25;
   b = 2 * c;
   c = c * 2;
   c = c + 1;
   b = a + 1;
   c = c + 10;
```

Fig. 1. The Liveness Area of the Variable a

For the sake of clarity, we have reported in Figure 2 an example of the liveness region of the parameter `cmd` of the `execl` system call-aware function, which is defined between line 26 and line 30. At line 26 the parameter `cmd` is defined through the `fgets` system call-aware function, whilst at line 30 `cmd` is used by `execl`. All the statements between these lines represent the zone where the `cmd` parameter is alive. In order to compute the liveness regions, we will apply the classic data-flow algorithms [19] according to the aforementioned description of *use* and *definition*.

Automating Mimicry Attack. The main purpose of the automatic mimicry is to compromise an application overcoming any protection mechanism provided by an anomaly-based system call-based HIDS. The applications which can be compromised using an automatic mimicry attack have to satisfy some particular characteristics. More precisely, an application a has to contain a vulnerability that allows the injection of malicious code, and a sequence of system calls $s_1, \ldots, s_n$ which can be triggered for performing some unauthorized action. The main task which the injected code j has to perform is (i) to modify the code pointers inside a so that j can be executed before the legal in-trace syscall s_i is invoked, (ii) to relinquish the execution flow to a, and (iii) to eventually regain the execution flow to modify others code pointers used in a to change the behavior of a system call s_j.

For the sake of clarity and to better understand this evasion technique, we describe two successful automatic mimicry attacks[2] performed against the code snippet shown in Figure 3 and 4 proposed in [17]. In the former attack, the attacker exploits the stack-based buffer overflow [18] vulnerability related to the `strcpy` (line 8) in order to overwrite the return address of `check_pw`, and point it to the attacker code. In writing such a code he may follow two options, that is, to either (i) directly invoke an in-trace

[2] Assuming no particular OS protection mechanisms, such as Address Space Layout Randomization (ASLR) [29,20,27] and non-executable data area [28,15,29] are deployed.

```
1   #define CMD_FILE "commands.txt"
2
3   int enable_logging = 0;
4
5   int check_pw(int uid, char *pass)
6   {
7      char buf[128];
8      strcpy(buf, pass);
9      return !strcmp(buf, "secret");
10  }
11
12  int main(int argc, char **argv)
13  {
14     FILE *f;
15     int uid;
16     char passwd[256], cmd[128];
17
18     if ((f = fopen(CMD_FILE, "r")) == NULL) {
19        perror("error: fopen"); exit(1);
20     }
21
22     uid = getuid();
23     fgets(passwd, sizeof(passwd), stdin);
24
25     if (check_pw(uid, passwd)) {
26        fgets(cmd, sizeof(cmd), f);
27        if(enable_logging)
28           printf("uid[%d]: %s", uid, cmd);
29        setuid(0);
30        if (execl(cmd, cmd, 0) < 0) {
31           perror("error: execl"); exit(1);
32        }
33     }
34  }
```

Fig. 2. The Liveness Area of the Parameter *cmd*

system call, but, due to the system call coordinates checks [11], he cannot neither invoke a system call from an illegal call site nor returning into different location after the system call-aware function termination, or (ii) set `enable_logging`, overwrite the `printf` GOT entry with the address of the injected malicious code, fix the stack layout in order to restore the original `check_pw` return address (the one at line 26, supposing the function does not return 0) and saved frame pointer and, finally, voluntarily relinquish the execution flow to the application code. Since no system call has been executed so far, no HIDS checks are performed and everything runs smoothly until the execution flow reaches the `printf` at line 28. At this point, before executing the real syscall-aware library function that, however, has to be executed in order to keep the `write` syscall performed by the `printf` in-trace, the malicious code is executed in order to change the content of the `cmd` "string" so that arbitrary command will eventually be executed (line 30) with full privileges (thanks to the `setuid` at line 29). After this simple little black magic, the attack ends by relinquishing the execution flow to the application code so that the legal in-trace `printf` can be executed from the permitted call site with a correct return address.

Obviously, things can be much harder from the attacker perspective than the one just described. In fact, if the attacker is not able to find suitable GOT entries to overwrite, he has to find out different code pointers to play with (e.g., application function pointers), as depicted in Figure 4. In this scenario, the attacker can exploit the same vulnerability as in the previous example, but this time no suitable GOT entries are available in order to regain the control of the execution flow later on. However, by carefully looking at the code, `check_pw` return address can be forced to point to a malicious code that will set `enable_logging` and `uid` (a signed 32-bit integer). The former variable will be

```
#define CMD_FILE "commands.txt"

int enable_logging = 0;

int check_pw(int uid, char *pass)
{
   char buf[128];
   strcpy(buf, pass);
   return !strcmp(buf, "secret");
}

int main(int argc, char **argv)
{
   FILE *f;
   int uid;
   char passwd[256], cmd[128];

   if ((f = fopen(CMD_FILE, "r")) == NULL) {
      perror("error: fopen"); exit(1);
   }

   uid = getuid();
   fgets(passwd, sizeof(passwd), stdin);}

   if (check_pw(uid, passwd)) {
      fgets(cmd, sizeof(cmd), f);
      if(enable_logging)
         printf("uid[\%d]: \%s", uid, cmd);
      setuid(0);
      if (execl(cmd, cmd, 0) < 0) {
         perror("error: execl"); exit(1);
      }
   }
}
```

Fig. 3. First Vulnerable Program

used to reach the `do_log` function that will allow the attacker to regain the control of the code, while the latter will be used to exploit the vulnerability present in `do_log` function (line 14) that will enable the attacker to overflow the buffer and overwrite the return address with the malicious code address. In fact after the pointer arithmetic is performed on line 14, `do_log` return address is overwritten with the `cmd_id` value (controlled by the attacker as well), so that it can make it point into the malicious code. Once the execution flow is regained, the attacker can change the value of `cmd` parameter performing any privileged command.

4 Defeating Automatic Mimicry Attacks

In this section we will explain the strategy we devised in order to prevent automatic mimicry attacks. Our approach is based on the use of the information contained in the Inter-procedural Control Flow Graph (ICFG) of the binary which has to be protected.

4.1 Defensive Strategy

A fundamental requirement of any *automatic mimicry* attack is the possibility to modify a process code pointers in order to execute the injected malicious code. Thus, automatic mimicry can be defeated if the integrity of such data is guaranteed. This is exactly the strategy we adopt. It is based on a three phases process: *code analysis*, *data collection*, and *code pointers restoring*. During the *code analysis* phase, the ICFG of the program *p* we want to protect is computed, and it is used to recognize the *dangerous regions*.

```
int enable_logging = 0;
int cmd_id = 0;
int uid_table[8192];

int check_pw(int uid, char *pass)
{
   char buf[128];
   strcpy(buf, pass);
   return !strcmp(buf, "secret");
}

void do_log(int uid)
{
   uid_table[uid] = cmd_id++;
}

int main(int argc, char **argv)

   if (check_pw(uid, passwd)) {
      fgets(cmd, sizeof(cmd), f);
      if (enable_logging)
         do_log(uid);
      setuid(0);
      if (execl(cmd, cmd, 0) < 0) {
         perror("error: execl"); exit(1);
      }
   }
}
```

Fig. 4. Second Vulnerable Program

Afterwards, the *data collection* phase collects the "trusted" values of the code pointers contained in p's dangerous regions. Such a phase is performed by executing p in a sterile environment. Finally, at run-time, the *code pointers restoring* phase restores the code pointers values collected during the data collection phase.

Code Analysis. The purpose of this phase is to determine the dangerous regions of p, using p's ICFG. In particular, we consider only nodes (basic blocks) that contain dangerous system calls, as defined by Xu. *et al.* [32]. Our method works as follows. Initially, we build p's ICFG, then:

- each node of the ICFG which contains a dangerous system call, is marked with u (i.e., we determine parameters' *use*);
- Let $p_1, \cdots, p_m$ be parameters used by a dangerous system call. For any p_j, $1 \leq j \leq k$ we collect the program locations where the parameter is defined[3], according to the definition given in § 3. To achieve this goal we use the standard equations defined by the data-flow analysis [19]. Such equations provide us the list of the defined variables on a per-basic block granularity. We mark all these nodes with d (i.e., we determine parameters' *definition*);
- subsequently, we visit the entire ICFG, and every time we meet a basic block marked by u, we perform the following steps. We apply the depth first search algorithm backward, starting from a node t marked with u and visiting the ICFG until we reach nodes marked by d, which contain the definition of the parameters used in node t. The sub-graph constituted by all visited nodes represents one of the dangerous regions we are interested in, and it is stored inside a database.

Figure 5, reports a fragment of a partial ICFG of the code depicted in Figure 3. In particular, we want to build a dangerous regions that is able to protect the parameter

[3] p is an `ET_EXEC` ELF executable so, code and data hold fixed absolute references.

`cmd` used inside the `execl` dangerous function. Gray nodes represent the dangerous regions built around the `execl` dangerous function and the leaf of the dangerous region is represented by the node marked with 4 (`fgets` function) in which the variable `cmd` is defined.

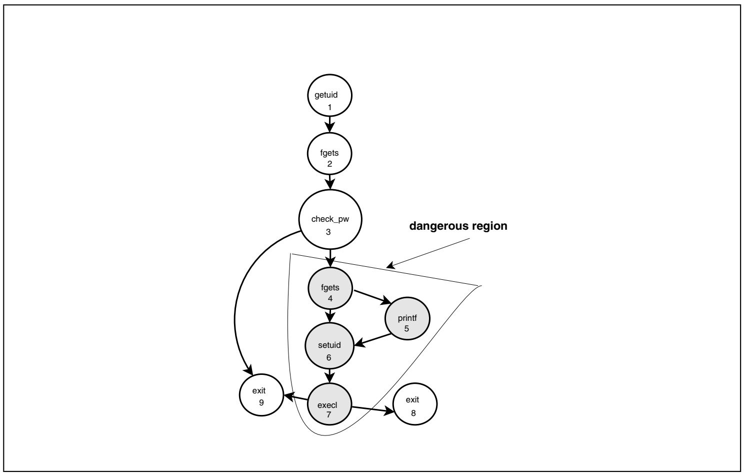


Fig. 5. Dangerous Region

Data Collection. After the code analysis phase, we collect the “trusted” addresses of the potential “dangerous” code pointers defined inside the dangerous regions. In particular, we collect the following information:

– *GOT Function Addresses*: For performance reasons we are interested only in the GOT addresses of the functions which are defined in dangerous regions[4]. GOT addresses are collected in two steps. Initially we execute the program p and we force the corresponding process to resolve all dynamic symbols collecting the address values; subsequently, we associate the GOT code pointers inside the dangerous regions with the discovered address values, and we store such relationships inside the database;
– *Function Pointers*: Another technique proposed by Kruegel *et al.* in order to perform the automatic mimicry attack is to exploit code pointers such as *application function pointers*. In order to collect such code pointers values, first we look for the program locations where the code pointers are defined, only inside dangerous regions; afterwards, through data-flow techniques we compute all code pointers destinations values and we save the correlation between program locations and such values into a database. Such operands represent the “trusted” values of the code pointers which will be used to check if an anomaly occurs during the program execution.

[4] We allow the attacker to regain control of the execution flow only in those locations of the application that are not dangerous to perform a successful automatic mimicry attack.

Code Pointers Restoring. The code pointers restoring is the last phase and it guarantees the integrity of the code pointers defined inside the dangerous regions. This phase is composed by two main steps. The first one, which takes place at loading time, performs a run-time *code instrumentation* of the process p, and loads the code pointers program locations and their trusted values inside a custom kernel data structures used by our kernel module component. The second step acts at run-time and performs the code pointers *integrity verification* step which is in charge of executing various checks on the "dangerous" code pointers and to restore the appropriate process execution flow.

run-time process instrumentation. The main goal of this step is to allow the execution of the *integrity verification* step in a transparent way without modifying neither the program source nor the binary code. It is performed as following. Initially it loads the kernel data structures containing the trusted values of the code pointers collected during the data collection phase. Subsequently, the code pointers instructions (i.e. `call`, `ret`) found inside the dangerous region (code analysis phase) are substituted by the `int3`[5] assembly instruction. The original instructions are saved inside the *saving instruction table* (see Figure 6) and will be restored after the integrity verification phase takes place.

Such a table will contain the op-codes of the substituted instructions and it is used to restore the execution flow of the process after the code pointers *integrity verification* step. Since the table is stored in p's address space, to guarantee that every tampering attempt is detected by the kernel before using the data provided by the table, its integrity is verified by using common cryptographic hash algorithms (SHA-1 and MD5). If the integrity cannot be satisfied, the kernel kills the process being protected, otherwise it is safe to use the data provided by the table to perform the next steps.

integrity verification. Due to the instrumentation process every time a potential dangerous function terminates its execution, or a (function) code pointer is invoked, the `int3` instruction brings the execution in kernel land to a custom module which performs the appropriate checks. Such checks are strongly dependent on the type of code pointers we are trying to protect, and they can be classified in three main sets:

- *GOT entries*: the trusted values of GOT entries are retrieved from the entry in the kernel memory structures associated to the substituted instructions, and replaced into the appropriate GOT entries locations of p;
- *Return addresses*: in order to get the appropriate return address we instrument the `call` statement associated to the called function f; consequently, whenever f is invoked the integrity verification module v will be able to retrieve and store inside its own memory structures the "active" f's return address (i.e., the address of the instruction next to the considered `call` statement). When f has to return (`ret` instruction) v will check if the return address is equal to the "active" return address retrieved during the call invocation; if so, the module

[5] Such an instruction issues a software interrupt and it is usually used by programs debugger.

will not perform any actions[6] and the process will continue its execution. Otherwise, v will restore the appropriate return address retrieved during `call` invocation, raising an alarm and allowing the process to continue its execution.
- *Function Pointers*: whenever a function pointer is invoked, the integrity verification module v will check if the function pointer belongs to the set learnt during the static analysis phase; if so, v will not perform any actions and the process will continue its execution. Otherwise, the module raises an alarm and stops the process execution.

restoring process control flow. After the integrity verification phase, the module checker must restores the normal process flow. Such a process is executed by a kernel module that will perform two actions according on the type of the substituted instructions:

- `call`: for each substituted `call`, the module copies inside the saving instruction table the `call` instruction followed by a `jmp` statement. Once the kernel restores the process flow after the integrity verification phase, it brings the execution flow to the appropriate `call` inside the table; at this point the call is executed and the address of the next instruction, that is `jmp`, is saved onto p's stack; afterwards, whenever the function invoked by the `call` returns, the program counter (`%eip` register) points to the `jmp` instruction which will jump to inside the executable process memory after the `int3` statement, restoring the normal execution process flow.
- `ret`: the module copies inside the saving instructions table the `ret` statement, which will bring the control flow to the appropriate process location, after the checks are performed.

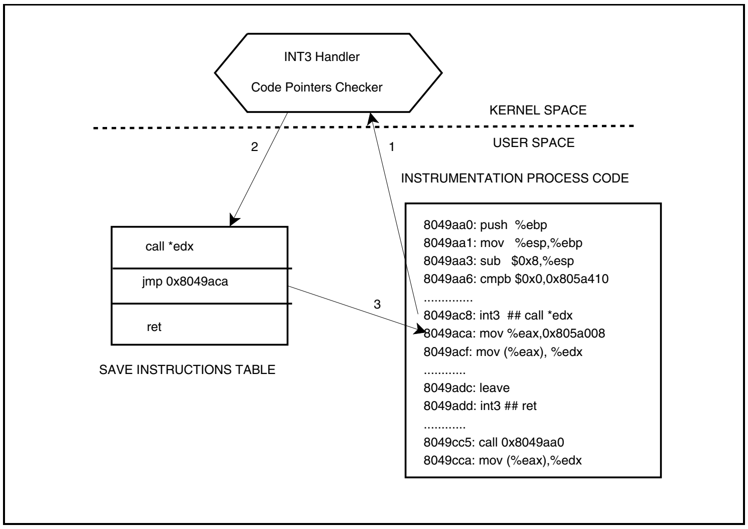
Figure 6, reports the steps of the restoring process flow. In step 1 the process raises an exception and the control flow is transferred to kernel land then, after the check on the trusted code pointers is performed, the module checker restores the program counter value to the entry associated to the substituted instructions (step 2). At the end, after the execution of the `call *%edx`, the flow returns to the `jmp` instruction inside the saved table, which brings the process execution flow to the appropriate program location (step 3).

5 Effectiveness

In this section we will describe some properties of our defensive mechanism and in particular we will show its effectiveness on two practical examples. It is worth noting that in order to evaluate the effectiveness of our strategy we must consider the goals of our defensive mechanism and some properties of the application execution context. More in details:

- our defensive mechanism is symbiotic with the HIDS. Consequently, for performing a successful attack an attacker must elude both the HIDS and the code pointers integrity verification checks;

[6] The attacker should pick and change the address inside the set learnt in the static analysis phase. Such an attack, however, fall into the IPE category and the HIDS will be able to detect it.



```
8049aa0: push  %ebp
8049aa1: mov   %esp,%ebp
8049aa3: sub   $0x8,%esp
8049aa6: cmpb $0x0,0x805a410
..............
8049ac8: int3  ## call *edx
8049aca: mov %eax,0x805a008
8049acf: mov (%eax), %edx
............
8049adc: leave
8049add: int3 ## ret
............
8049cc5: call 0x8049aa0
8049cca: mov (%eax),%edx
```

Fig. 6. Process Execution Flow Restoring Step

– the main goal of our defensive strategy is to defeat the attacks that use code pointers as a way to divert a process execution flow. It is worth noting that a vulnerability could occur inside the dangerous regions in some positions where the attacker must not use any code pointers to change the value of the syscall parameters; such an attack is generally known as non-control data [6] and it is not currently addressed by our approach[7];

In the following we will show how our technique works on the two examples of automatic mimicry attack described in [17].

5.1 GOT Protection

We apply the mechanism described in § 4 to the automating mimicry attack presented in Figure 3 . Our goal is to protect the *cmd* parameter used by the `execl` syscall. After the code analysis phase, we have defined the dangerous region of the program line 26-30, the Kernel module checker *c* has loaded inside its memory structures the trusted values of the code pointers (i.e `execl`'s GOT line 30, `fgets` (line 26), `printf` (line 28), `setuid` (line 29) etc.) and the process has been instrumented. At this point, during the program execution, whenever the attacker exploits the vulnerability inside the `check_pw` function, he probably will rewrite the GOT of a function defined inside the dangerous regions which will be executed before `execl` function (line 30). The next attacker's action is to relinquish the control of the code to the application after the `check_pw` function, and regaining it when that particular function will be invoked. But after the attacker relinquished the control code to the "original" application, when

[7] However, some forms of such attacks can be defeated by protecting the use and definition area of the sensitive variables.

the execution flow reaches one of the checks previously set, the program flow will pass to c that will rewrite the appropriate code pointers, blocking so the attack.

5.2 Code Pointers Protection

In the example shown in Figure 4 the attacker uses a return stack as code pointer to regain code. During the learning phase the Kernel module checker retrieves the return addresses of the dangerous functions, `do_log` function, and during the loading it substitutes the `ret` instruction of such a function, with the breakpoint instruction `ret`.

Whenever the attacker exploits the vulnerability in the function `check_pw`, he changes the value of the variable `uid` writing a value that, once used by `do_log` function, will overwrite the function return address of that function bringing the execution inside the malicious code. Once the attacker has modified the `uid` value, it relinquishes the process execution flow back to p which will follow the normal flow. Whenever the program is about to return from `do_log` function, the execution flow will pass, by means of `int3`, to the code pointers integrity verification module which will overwrite the function return address with the "trusted" one, preventing to malicious code to regain control on the p's execution flow later on.

6 Technical Details

In this section we provide technical details of the prototype we developed for a GNU/Linux system (kernel 2.4.28 version). We will describe the static analysis tool used to perform the static analysis phase, the process instrumentation performed at load time after the dynamic linker process takes place, and the modifications of the `int3` kernel handler performed in order to manage the code pointers integrity verification process.

6.1 Static Analysis Tools

Static analysis of p has been performed using a static analysis tool for ELF IA32 binaries, developed by our group. The core of the tool is written using the Python language and some parts using the C language. Such a tool is able to obtain the Inter-procedural Control Flow Graph (ICFG) and to perform the basic data-flow analysis of the program being analyzed. In particular, the tool works following these steps:

- initially the pre-processing phase is performed in order to recognize some important ELF information such as symbol table location, code section location, dynamic section information, and so on;
- after gathering the preliminaries information, the tool disassembles the instructions contained inside the code section and converts them to an intermediate form. We used the well-known recursive traversal algorithm defined in [22] to disassemble the binary;
- the tool computes the ICFG (§ 3), afterwards the program is converted into the SSA form using the standard Ferrant's algorithm [8];
- finally, the tool uses the classic equations of the liveness analysis defined in [2] to perform this analysis.

Moreover, the tool is able to compute both the control dependencies and the classic data-flow analysis equations defined in [2].

6.2 Process Instrumentation

The process instrumentation phase is performed by the *instrumentator*, a program we have developed which trace the program p to be protected and, by using the `ptrace` system call, substitutes the appropriate instructions with the `int3` statement. Moreover, the instrumentator has to build the saving instructions table. Such a table will be mapped at a fixed address known by the kernel code pointers integrity verification module as well; this can be easily achieved by using the `mmap` system call with a `MAP_FIXED` flag.

6.3 **`int3`** Exception Handling

In order to perform the code pointers integrity verification checks, we have modified the `int3` kernel exception handler implemented in the `do_trap` kernel function (`traps.c`). In particular, when the `int3` exception is raised the control flow is transferred from user space application to the kernel code which calls the `do_int3` kernel function (see `entry.S`) which eventually invokes `do_trap`. Figure 7, reports a snippet of the `do_trap` function we modified to add our code pointers integrity verification.

More in detail, when the exception `int3` is executed, the `do_trap` function checks the exception number and the process name (line 7 and 8) which raises the exception. Consequently, if the process name[8] and the exception number are appropriate, we perform the code pointer checks and restore the flow as already explained in § 4 (line 10 and 11); otherwise the handler will work in the usual way and the code inside the *if* statement (line 7) will not be executed.

```
1    trap_signal(...)
2    {
3        struct task_struct *tsk = current;
4        tsk->thread.error_code = error_code;
5        tsk->thread.trap_no = trapnr;
6
7        if ((trapnr == 3) && !(strcmp(tsk->comm, "process_name")))
8        {
9          check_code_pointers();
10         restore_flow() ;
11         return;
12       }
13       else
14       {
15         if (info)
16            force_sig_info(signr, info, tsk);
17         else
18            force_sig(signr, tsk);
19         return;
20       }
21     }
22   }
```

Fig. 7. Modified `do_trap` Function

[8] Indeed, a check on the program's i-node number would be better. Otherwise, the check can be easily bypassed by a local attacker by using symbolic links, for example.

7 Experimental Results

In this section we will describe the system used in order to make our experimental test and then present the set of experiments we ran to collect the measurements about the overhead introduced by our code pointers integrity verification module. All the experiments have been executed on an Intel Pentium IV processor with 3 GHz clock, running a GNU/Linux Debian operating system, 2.4.28 Linux kernel and 128 MB of RAM.

Our module acts on the code used by the kernel in order to manage the checking performed at runtime, so we focus our attention on those routines, defined into the file `traps.c`. In order to measure time lapses we used the time-stamp counter processor register (`tsc`, using the `rdtsc` assembly instruction). 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 accurate measurement of the system.

In the first phase of our experiments we have measured two 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:

– *Context Switch*: 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 code pointers integrity verification process. The 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 the best-case).

– *Hash Table access time*: this measure represents the amount of time needed to access the hash table (saving instruction table) used in order to replace the trusted code pointers values and for bringing the execution flow back to the process. The hash table size depends on the number of different instrumentation locations defined by the application. For the test we conducted, our hash table contained 50 locations on an 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 the best-case).

As a second phase of our test we have considered three different kind of applications: server web `dhttpd` version $1.02a$, the `tftpd` server version $0.17 - 15$, and `sudo` application version $1.6.8p12$. Table 1 reportes the results of the static analysis phase; for each service we can see the number of the dangerous regions found and the total number of the code pointers that must be protected defined inside those regions.

After the static analysis phase we have performed the run-time analysis. For the HTTP server we have used a small web site with the following features: total size 500 KB, 12 static HTML pages, and 6 pdf documents (document's size 200 KB); for the `tftpd` server we have considered download and upload operations of a file which size

Table 1. Dangerous Regions

Services	# Dangerous Regions	# Code Pointers found in the Dangerous Regions
`dhttpd`	6	30
`tftpd`	7	39
`sudo`	14	75

Table 2. Runtime Analysis

Services	#Checks	Execution Time μs	Checking Time μs	Overhead
`dhttpd`	724	1407000	72400	5.1%
`tftpd`	467	1200000	37600	3.1%
`sudo`	110	99150	8800	8.8%

is 500 KB, and for the `sudo` utility we have considered the execution on the `cat` command on a small file. In Table 2 we have reported the run-time analysis overhead for each service, the number of the checks performed during the execution time, the total amount of the time spent by the process (we do not considered the I/O idle time), the time spent by the integrity verification process and, in the last column, the overhead inserted by our integrity verification module for those particular services.

8 Conclusion and Future Works

This paper presents a novel defensive technique based on the Inter-procedural Control Flow Graph; such a mechanism is represented by the code pointers checker module at the kernel-level, which is able to protect the HIDS against automatic mimicry attack with a low overhead.

One of the main problem of our defensive mechanism is represented by the accuracy of the static analysis phase performed on x86 binaries. In fact, the imprecision of such an analysis could increase both the false positive and negative in our system. In particular, there are two main problems which must be addressed when working on an executable binary: (1) the CFG's completeness and (2) the aliasing problem. In order to improve the CFG's completeness we can adopt the technique described in [24]. In this approach, the authors use the data-flow analysis in order to determine the values of the indirect calls so to improve the completeness of CFG. Instead, for the aliasing problem we can use the algorithm describe in [3]. This technique works on the x86 executable and has obtained good results. However, in future we think to work on the source code of the program in order to solve the problems that binary static analysis techniques arise.

Another problem of our approach is the size of the dangerous regions. In fact, sometimes there exists a great distance between the definition and use of a particular variable; consequently, if the region's size is very large, the attacker could have more chances to perform the attack successfully. In fact, if the vulnerability is positioned inside the dangerous regions the attacker can change the value of the system call parameters successfully without using any code pointers. We are investigating for improving our technique in order to solve these issues and to mitigate other attacks such as the non-control data.

Acknowledgements

We thank the anonymous reviewers for their helpful comments that improved the quality of the paper. We thank Monirul Sharif for help in reviewing the paper and we thank Christopher Kruegel and Lorenzo Martignoni for their extensive comments and suggestions.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: CCS '05: Proceedings of the 12th ACM conference on Computer and communications security, pp. 340–353. ACM Press, New York (2005)
2. appel, a.w.: Modern compiler implementation in c. Cambridge University Press, Cambridge (2004)
3. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)
4. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-Variant Systems: A Secretless Framework for Security through Diversity. In: 15th USENIX Security Symposium (2006)
5. Bruschi, D., Cavallaro, L., Lanzi, A.: An Efficient Technique for Preventing Mimicry and Impossible Paths Execution Attacks. In: 3rd International Workshop on Information Assurance (WIA 2007) (April 2007)
6. Chen, S., Xu, J., Sezer, E., Gauriar, P., Iye, R.K.: Non-Control-Data Attacks Are Realistic Threats. In: 14th USENIX Security Symposium (2005)
7. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. of the 7th Usenix Security Symposium, pp. 63–78 (January 1998)
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
9. Bruschi, D., Cavallaro, L., Lanzi, A.: Diversified Process Replicæ for Defeating Memory Error Exploits. In: 3rd International Workshop on Information Assurance (WIA 2007) (April 2007)
10. Etoh, H.: GCC extension for protecting applications from stack-smashing attacks (ProPolice) (2003), `http://www.trl.ibm.com/projects/security/ssp/`
11. Feng, H., Kolesnikov, O., Fogla, P., Lee, W., Gong, W.: Anomaly Detection using Call Stack Information. IEEE Symposium on Security and Privacy, Oakland, California (2003)
12. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy, p. 120. IEEE Computer Society Press, Los Alamitos (1996)
13. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion Detection Using Sequences of System Calls. Journal of Computer Security 6(3), 151–180 (1998)
14. Shacham, H., Page, M., Pfaff, B., Goh, E.-J.: On the Effectiveness of Address-Space Randomization. In: CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security, pp. 298–307. ACM Press, New York (2004)
15. iSec.pl Development Team. kNoX - Implementation of non-executable Page Protection Mechanism (February 2005)
`http://www.isec.pl/projects/knox/knox.html`
16. Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure execution via program shepherding. In: Proceedings of the 11th USENIX Security Symposium, pp. 191–206, Berkeley, CA, USA, USENIX Association (2002)
17. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating Mimicry Attacks Using Static Binary Analysis. In: Proceedings of the USENIX Security Symposium, Baltimore, MD (August 2005)

18. Elias Aleph One Levy. Smashing the Stack for Fun and Profit. Phrack Magazine, vol. 0x07(#49), Phile 14–16 (December 1998)
19. Nielson, F., Nielson, H., Hankin, C.: Principles of Program Analysis (1999)
20. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In: 12th USENIX Security Symposium (2003)
21. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In: 14th USENIX Security Symposium (2005)
22. Schwarz, B., Debray, S., Andrews, G.: Disassembly of Executable Code Revisited. In: Proceedings of the Ninth Working Conference on Reverse Engineering (2002)
23. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In: IEEE Symposium on Security and Privacy, Oakland, California (2001)
24. De Sutter, B., De Bus, B., De Bosschere, K., Keyngnaert, P., Demoen, B.: the static analysis of indirect control transfers in binaries. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, pp. 1013–1019 (June 2000)
25. Tan, K.M.C., Killourhy, K.S., Maxion, R.A.: Undermining an anomaly-based intrusion detection system using common exploits. In: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (2002)
26. Tan, K.M.C., McHugh, J., Killourhy, K.S.: Hiding intrusions: From the abnormal to the normal and beyond. In: Information Hiding, pp. 1–17 (2002)
27. The Linux Kernel 2.6 Development Team. The Linux Kernel 2.6 (February 2005), `http://lwn.net/Articles/121845/`
28. The OpenWall Development Team. The OpenWall Project (February 2005), `http://www.openwall.com`
29. The PaX Team. PaX: Address Space Layout Randomization (ASLR) `http://pax.grsecurity.net`
30. Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In: IEEE Symposium on Security and Privacy, Oakland, California (2001)
31. Wagner, D., Soto, P.: Mimicry Attacks on Host Based Intrusion Detection Systems. In: Proc. Ninth ACM Conference on Computer and Communications Security (2002)
32. Xu, H., Du, W., Chapin, S.J.: Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 21–38. Springer, Heidelberg (2004)