UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze Matematiche, Fisiche e Naturali

DOTTORATO DI RICERCA IN INFORMATICA
XX CICLO
SETTORE SCIENTIFICO DISCIPLINARE INF/01 INFORMATICA

Comprehensive Memory Error Protection via Diversity and Taint-Tracking

Tesi di
Lorenzo Cavallaro

Relatore
Prof. R. Sekar

Co-relatore
Prof. D. Bruschi

Coordinatore del Dottorato
Prof. V. Piuri

Anno Accademico 2006/2007

UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze Matematiche, Fisiche e Naturali

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE
XX CICLO

Comprehensive Memory Error Protection via Diversity and Taint-Tracking

PhD Candidate
Lorenzo Cavallaro

Adviser
Prof. R. Sekar

Co-adviser
Prof. D. Bruschi

PhD Coordinator
Prof. V. Piuri

Academic Year 2006/2007

Abstract of the Dissertation

Comprehensive Memory Error Protection via Diversity and Taint-Tracking

by
Lorenzo Cavallaro

Doctor of Philosophy
in
Computer Science

Università degli Studi di Milano
2008

Memory errors in C and C++ programs are one of the oldest classes of vulnerabilities. Attackers have been exploiting these errors since late 80's and these issues are still a real and concrete threat. To date, several countermeasures to combat memory error vulnerabilities have been proposed. They cover a broad range of Computer Science disciplines, going from safe programming language solutions, anomaly detection approaches, and information-flow (also known as taint analysis) based strategies, to techniques that modify the underlying compiler, the operating system and underlying hardware, and system libraries. Among the others, transformation techniques which aim to provide artificial diversity, or are based on taint analysis approaches, seem to be the most promising and effective against a broad class of memory error vulnerabilities. Unfortunately, as protection mechanisms improve, so do the attacks, and existing transformation techniques which aim to provide artificial diversity or to perform taint analysis either cannot deal with all the memory errors, or they provide only *probabilistic* protection (e.g., artificial diversity) or, again, they have a high rate of false positives when dealing with some memory error vulnerabilities (e.g., pointer and non-pointer data corruption). This dissertation aims to provide comprehensive solutions to memory error vulnerabilities. Recognizing the effectiveness of the aforementioned diversity, taint-tracking, and anomaly-based detection strategies,

we propose two different approaches that are able to deal with a broad class of memory error vulnerabilities. In our first approach, we extend the concept of process diversification. So far, diversification has been applied on a process by itself, for instance by adopting address space and instruction set randomization schemes. Our approach, instead, couples diversification and replication together. It applies a form of diversification that involves a process P along with P_r, the process *replica* of itself. By monitoring P and P_r behavior, and by replicating data on particular rendez-vouz points, our *diversified process replicæ* approach detects behavioral divergences triggered by memory error exploits. In most cases, our strategy gives a *deterministic* protection. The second approach takes advantage of taint-tracking and anomaly detection techniques. The proposed strategy transforms a given benign application P into P_T, a taint-enhanced version of P. Then, by coupling taint analysis and anomaly detection, our *taint-enhanced anomaly detection* approach dynamically analyzes *sinks*, that is, relevant events (e.g., system calls or security sensitive functions) of the transformed taint-enhanced application, during a so-called training or learning phase. Taint information as well as different models are used to automatically infer the security policies which represent the behavioral profile $\mathcal{M}$ of the protected process P_T. Subsequently, during a so-called detection phase, similarly to any anomaly-based detection strategy, single events of P_T observed at run-time are checked one at a time to see whether they are consistent to the learnt behavioral profile $\mathcal{M}$. Should $\mathcal{M}$ be inconsistent with respect to these traces, an alarm will be raised. The dissertation ends by providing experimental results and comparison to existing and similar techniques. Moreover, performances and effectiveness of these approaches are also discussed, as well as their weaknesses, limitations and possible improvements.

Acknowledgement

This dissertation would have not been possible without the help, the presence, the comfort and the professionalism of several people and friends. I wish to thank my tutor and co-adviser Prof. Danilo Bruschi for having always allowed me to freely express myself and my ideas the way that I liked the most. This way, I have been able to work on topics that I enjoy and, hopefully, I will keep doing it in the future.

I am extremely in debt with my adviser, Prof. R. Sekar. He taught to me how research has to be done. He also supported and provided me with motivations even when almost everything seemed to be lost. It is not just a matter of hard work. It is also a matter of curiosity, imagination, and interests in different topics. I spent a wonderful period overseas and I have grown both under the personal and professional point of view.

I am also extremely in debt with the external referees of this dissertation, namely Prof. David Evans (thank you also for having invited me to University of Virginia), Prof. Engin Kirda, and Prof. Christopher Kruegel. I wish to thank them for the time and energy they have spent on the dissertation as well as for the insightful comments and suggestions they provided.

I would also like to thank my "old" friends as well as the "new" one I met at Stony Brook University. In particular, I wish to thank Andrea Lanzi for all the research work we have done together and all the discussions and brainstorming we had, but mostly because of his friendship. He has always been a source of inspiration to me. I wish to thank Lorenzo Martignoni which I always admired for his strength and skills and for his friendship as well. I hope to collaborate with him as well as soon as possible. Of course, a warm thank goes to all the rest of the Laser lab at Dipartimento di Informatica e Comunicazione of Università degli Studi di Milano (in no particular order): Mattia Monga, Giampaolo Fresi Roglia, Emanuele Passerini, Roberto Paleari, Alessandro Rinaldi, and Alessandro Rozza. Thank you for making the work place a better and funnier place. A special thanks goes to Linda Pareschi for her friendship and also for a practical help during this

latest period.

On the USA side, I met several nice and smart people which I wish to thank in this dissertation. Again, in no particular order, they are: Alok Tongaonkar, Weiqing Sun, Munyaradzi Chiwara, Yves Younan, Wei Xu, Sandeep Bhatkar, Saad Arif, Daniel Finke, Emir Malikov, Jennifer Dixson, Ezio Bartocci, Oliviero Riganelli, and Ilaria Zanardi. I wish to thank them for their friendship and I hope to stay in touch with all of them even after my overseas experience will be over and our life will be split apart.

A special thanks, of course, goes to Orgoglioni. They are too many to be singularly cited but I wish to thank all of them for their friendship, the wonderful time they were able to give to me and, I am sure, they will still give in the future.

I wish to thank my family as well for giving me all the love and support I needed during these hard PhD years.

Finally, I would like to thank Simona, which soon will become my wife and my family, for everything. Not even a little thing would have been possible without you.

I surely am far from being a good researcher, but I will do all of my best to become it. Hopefully, this dissertation will not be the end, but just the beginning of a wonderful journey.

to Simona, the love of my life

Contents

List of Figures

List of Tables

Part I

Introduction

You need the willingness to fail all the time.
You have to generate many ideas and then you have to work very hard only to discover that they don't work. And you keep doing that over and over until you find one that does work.

John W. Backus (1925 - 2007)

1 Introduction

Software security has become an increasing necessity for guaranteeing, as much as possible, the correctness of computer systems. Unfortunately, software vulnerabilities are omnipresent. In the last few decades, many new vulnerabilities have been discovered, and old ones have been continuously exploited.

Memory errors in C and C++ programs have been known for decades and are one of the oldest classes of software vulnerabilities. Attackers have been exploiting these errors since the days of the Internet Worm of 1988 (also known as Morris Worm) ([72]). Despite decades of research on memory error countermeasures, these software vulnerabilities are still a real and concrete threat, as a recent breakdown of the NIST National Vulnerability Database (NVD) of software vulnerabilities depicts in Figure 1.1.

Since their first public exploitation, a number of different types of memory error vulnerabilities have been discovered and, unfortunately, successfully exploited by attackers. To date, buffer overflows are probably the most common memory error vulnerability ([25]). As the name suggests, a buffer overflow vulnerability takes advantage of an erroneous or the lack of bounds checking on a buffer. As a direct consequence, buffer overflows can be exploited to write past the end of a buffer with the intent to corrupt adjacent memory locations. This carefully crafted corruption permits an attacker to generally execute *arbitrary code*, or, to perform actions that are as dangerous.

Memory errors can be eliminated by using type-safe languages such as Java. Unfortunately, these languages do not provide enough low-level control on memory management and data representation. These are features which are mostly required for systems software, therefore it is unlikely that these type and memory safe languages will substitute C or C++, at least in this context.

To date, a number of memory error countermeasures have been studied and proposed. For instance, to defeat stack-based buffer overflows [25], Cowan *et al.* [15] proposed StackGuard, a compiler patch that inserts a *canary* value right before the return address to detect attempts to corrupt a return address using

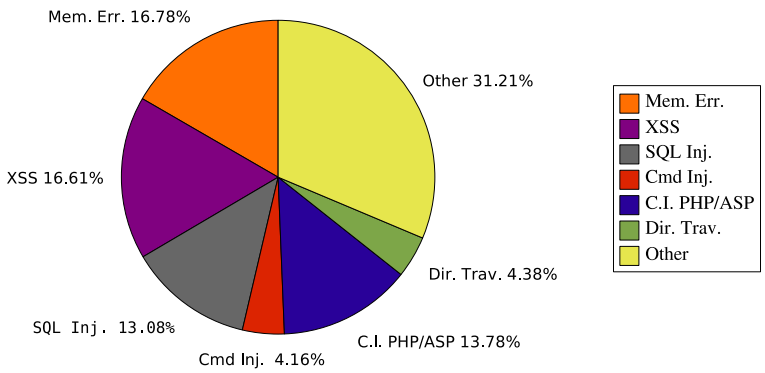

Figure 1.1: Breakdown of NIST National Vulnerability Database of software security vulnerabilities (2006 and 2007-Q1/Q2)

a buffer overflow. Unfortunately, not only has it been shown how to bypass StackGuard-based protections ([63]), but the offered protection is, most importantly, limited to a specific vulnerability. In fact, it has been designed to work only for stack-based buffer overflows. It has no effect, as is, for other kinds of buffer overflows, or for other memory error vulnerabilities, such as format strings [69, 35], therefore providing a very limited degree of protection.

The countermeasure proposed by Cowan *et al.* is just one of several others that have kept researchers busy providing different and progressively more complete solutions to memory errors in the past several years. Proposed solutions cover a broad range of Computer Science disciplines, going from safe programming languages ([43, 58]), anomaly detection ([87, 85, 28, 39, 70, 56, 54, 9]), and information-flow ([47, 13, 59, 89, 60]), to techniques that modify the underlying compiler ([15]), system libraries ([88, 82]), the operating system, or the hardware ([62, 79, 31]). Among these approaches, transformation techniques which aim to provide artificial diversity, or are based on information-flow approaches, seem to be the most promising and effective against a broad class of memory error vulnerabilities.

Transformation techniques that aim to provide artificial diversity to a program base their assumption on the fact that, generally speaking, memory error exploits leverage on the monoculture of systems. Therefore, once a memory error vulnerability is discovered on an application running on a particular operating system (OS) and architecture, it is fairly easy for an attacker to take over the rest of similar configurations (e.g., application, OS, architecture) which present the same type of memory error. In fact, this is possible because a process address space is organized always in the same way, for a particular OS and architecture,

and memory error exploits rely on corrupting particular memory locations (e.g., where a function return address is stored on the stack, or where a function global offset table (GOT) is stored in the vulnerable process address space) with suitable values. For this reason, memory error exploits can be thwarted by applying approaches that use diversity on computer systems, such as address space randomization (ASR) [79, 88, 67, 68], instruction set randomization (ISR) [24], and so on. Unfortunately, these approaches either cannot deal with all the memory errors or, even when successful, they provide only *probabilistic* protection.

On the other hand, information-flow (or taint analysis) based approaches focus their attention on how *untrusted* data is used and processed by a protected application (i.e., how information *flows* into the application), to check whether these data corrupt security sensitive memory locations. More precisely, taint analysis is a technique which aims to detect whether *untrusted* data is incorrectly used in security sensitive actions. To this end, taint analysis usually marks untrusted data coming from *taint sources* (e.g., data coming from the network, or other input-related system calls) as being tainted, and, as data propagates through memory, it propagates the taint information associated with the data itself. Any attempt to use security sensitive data which has been marked as tainted at security sensitive points, called *sinks* (e.g., particular system calls, or when a function is about to return), is generally a manifestation of an attack. For instance, a memory error exploit which aims to corrupt a code pointer, can overwrite a function return address with the intent to hijack the legal process execution flow. Naturally, the overwritten return address will be marked as tainted as a consequence of this attack attempt. The low-level implicit policy adopted by taint-based approaches does not allow to dereference tainted code pointers, as function return addresses are security sensitive data which should never become tainted. In fact, they are manipulated by the application code which is considered to be trusted.

Unfortunately, as protection mechanisms improve, so do the attacks. Recently, Chen *et al.* [14] pointed out that it is no longer necessary to exploit a memory error vulnerability with the goal to hijack a legal process' control-flow to cause harm. In fact, damage can also be caused by memory error attacks which do not target code pointers but, instead, aim to overwrite data and data pointers. Even if this seems to be a strict restriction, Chen *et al.* showed that these attacks can be as powerful as the classic ones (i.e., which corrupt code pointers). It may be argued that such attacks are not very common nowadays, as their exploitation require a better understanding of the application logic. As long as memory error attacks which corrupt code pointers will still be effective, attackers have no motivations to make their exploit harder to code and, most of all, succeed. Naturally, should existing research countermeasures become widely used, the attackers will modify their attacks to stick to this new technique, as confirmed by recent and related research [13, 56, 9, 8, 47] as well. Therefore, it is our belief that comprehensive memory error countermeasures should no longer ignore such a class of memory error attacks.

This dissertation aims to provide comprehensive solutions to memory error at-

tacks. To this end, we propose two different *program transformation* techniques which provide protection from a broad class of memory error attacks. While, both approaches deal with attacks which corrupt code and data pointers, the second approach also considers memory errors which corrupt arbitrary data as well. It could seem that the second approach is more effective and provide a better protection from memory error exploits to not justify the need for the first technique. However, as we will briefly see, this is not true. In fact, our first technique mainly offers a *deterministic* protection, while our second approach offers a *probabilistic* protection when data and data pointers corruption are involved. Moreover, the underlying mechanisms of the proposed approaches are different and this can suggest contexts where the proposed strategies find, respectively, a better deployment with respect to each other. In particular:

- In our first approach, we extend the concept of process diversification. So far, diversification has been applied on a process by itself, for instance by adopting address space and instruction set randomization schemes. Our approach, instead, couples diversification and replication together. It applies a form of diversification that involves a process P along with P_r, the process *replica* of itself. By monitoring P and P_r behavior, and by replicating data on particular rendez-vouz points, our *diversified process replicæ* approach detects behavioral divergences triggered by memory error exploits. In most and more frequent cases, our strategy gives *deterministic* protection.

- Our second approach takes advantage of taint-tracking and anomaly detection techniques. The proposed strategy transforms a given benign application P into P_T, a taint-enhanced version of P. Then, by coupling taint analysis and anomaly detection, our *taint-enhanced anomaly detection* approach dynamically analyzes *sinks*, that is, relevant events (e.g., system calls or security sensitive functions) of the transformed taint-enhanced application, during a so-called training or learning phase. Taint information as well as different models are used to automatically infer the security policies which represent the behavioral profile $\mathcal{M}$ of the protected process P_T. Subsequently, during a so-called detection phase, similarly to any anomaly-based detection strategy, single events of P_T observed at run-time are checked one at a time to see whether they are consistent to the learnt behavioral profile $\mathcal{M}$. Should $\mathcal{M}$ be inconsistent with respect to these traces, an alarm will be raised.

At first glance it could seem counter-intuitive to consider approaches which provide artificial diversity as program transformation techniques. Indeed, program transformation techniques aim to modify the original program in order to preserve its semantics. As we will see, both our approaches transform a program P so that it can be augmented with information used by the underlying protection mechanism adopted. Naturally, as every transformation technique guarantees, the semantic of P when non-malicious input is involved is preserved.

The dissertation ends by providing experimental results and comparison to existing and similar techniques. Moreover, performances and effectiveness of these approaches are also discussed, as well as weaknesses, limitations and possible improvements.

1.1 Dissertation Organization

The dissertation is organized as follows. Chapter 2 reminds the most important concepts about memory errors, what they are, what different kind of memory error vulnerabilities exist out there, and how they can intuitively be exploited. Chapter 3 introduces our first proposed memory error countermeasure, *diversified process replicæ*, while *taint-enhanced anomaly detection*, our second approach, is described in Chapter 4. Chapter 5 describes related literature, while we give some suggestions about future directions that can be taken to improve the proposed approaches in Chapter 6. The dissertation ends by giving concluding remarks in Chapter 7.

C++ and Java, say, are presumably growing faster than plain C, but I bet C will still be around.
Dennis Ritchie (1941 -)

2 Memory Errors

This chapter provides an introduction on memory error vulnerabilities in C programs. While it is out of the scope of the chapter and of the whole dissertation to provide a complete survey on memory error vulnerabilities and their common exploitation techniques, for which we redirect to [90], we would like to briefly summarize these concepts as we believe that they might help to better understand the rest of the dissertation. Also, we would like to remind the reader that there exists several interesting advanced memory error exploitation techniques that have been proposed over the years. We try to cite them as appropriate while describing common memory error vulnerabilities, although some of them will be probably missing from the discussion that follows as we prefer to spend more time on the defensive countermeasures proposed in the rest of this dissertation.

To the best of our knowledge, there is not a formal definition of a memory error. However, broadly speaking, it is possible to say that a memory error occurs when an object accessed using a pointer expression is different from the one intended. It is possible to classify memory errors in *spatial* or *temporal* errors or they can also be classified depending on the type of *corruption* they aim to pursue. In this case, it is possible to further distinguish among attacks which corrupt *code pointers*, *data pointers* and *non-pointer* data.

Spatial error. A spatial error occurs when a pointer which points outside the bound of its referent[1] is de-referenced. It is possible to further distinguish in the following categories:

(a) De-referencing non-pointer data. It occurs when an integer is erroneously assigned to a pointer p and p is subsequently de-referenced. Misinterpreting integer as pointers, under the right circumstances, can cause arbitrary corruption of process' memory locations. If sensitive, under a security point of view, memory locations are involved, the vulnerability can be exploited to fulfill the attacker's will.

[1] The referent of a pointer is the object the pointer is pointing to.

(b) De-referencing uninitialized pointers. It is similar to the temporal error described in the next paragraph except for the fact that the referent of the involved pointer does not change. It is fairly complicated to successfully exploit this vulnerability when dealing with static (global data) or dynamically allocated pointers (heap). In fact, static pointers are allocated once, at program start-up. Moreover, de-referencing an uninitialized static pointer will most likely cause the program to crash as the pointer is generally implicitly initialized to 0 (as it is stored in the `.bss` section), and usually any access to the first page of a process address space (AS) will generate a page fault (PF) as the page is usually unmapped (to catch `NULL` pointer de-referencing). Therefore, pointers allocated on the stack offer a better venue for an attacker (no matter whether the pointer referent is allocated on the (previous) stack frame(s), or on the heap, or on the data/bss). It is possible to imagine a situation where a function f is invoked which allocates and eventually initializes a local pointer variable p. When f terminates, the local stack frame is freed but the values are still stored at the same memory locations (i.e., the memory location which corresponded to p still holds a valid pointer value). Therefore, if f is invoked a second time, p will contain the same valid pointer value even *before* its explicit initialization. Under the right circumstances, p can be prematurely de-referenced and the memory error exploited.

(c) Valid pointers used with invalid pointer arithmetic. It is probably the most known scenario as it typically refers to out-of-bounds accesses of buffer-like variables. The classical buffer overflows (see § 2.1).

Temporal error. A temporal error occurs when a pointer which points to a referent which no longer exists is de-referenced (e.g., referent previously freed). Representatives of this category are dangling pointers [3] and double free [5, 52] memory error exploitation techniques[2].

As we introduced at the beginning of the Chapter, memory errors can also be classified depending upon the type of corruption they aim to pursue. Attackers have developed different, clever and interesting way to exploit memory error vulnerabilities. Moreover, as countermeasures are researched, developed, and deployed, several different exploiting techniques have been proposed to bypass them. In fact, as we briefly already noted, a memory error exploit aims to usually corrupt *code pointers*, *data pointers*, and *non-pointer* data of a vulnerable process, regardless of the underlying exploitation technique adopted.

[2]It is worth noting that the techniques for double free exploitations are actually generally enough to be used to corrupt heap management information which allows an attacker to write arbitrary bytes in arbitrary writable memory location of the vulnerable process (*write-anything-anywhere* primitive).

Code pointer corruption. This corruption refers to data belonging to a process address space which is used to *control* the process execution flow. This category mainly embraces (function) return addresses stored on the stack, application-specific function pointers stored on the stack or onto the data/bss segment, and other code pointers usually introduced by the underlying binary specification or programming language. For instance, global offset table (GOT) entries described by the ELF specification [81], and C++ virtual pointers table are examples of such pointers.

A memory error exploit usually aim to overwrite these code pointers. Depending upon how much freedom the attacker has (i.e., how "deeply" the vulnerability can be exploited), the code pointers can be *fully* or *partially* overwritten. The former case permits an attacker to (i) execute arbitrary injected code [25], (ii) execute already existing code by performing what is known as return-into-lib(c) [62] or return-into-text [74] attack, or, (iii) trigger an impossible path execution (IPE). On the other end, a partial overwrite can cause a more limited damage as it permits to trigger an IPE-like attack only. It is worth nothing that, technically speaking, returning-into-text techniques and IPE are achieved in the same way, i.e., to perform an IPE attack, it is necessary to return into the process code. However, return-into-text is a more general exploitation technique which is not limited to trigger an IPE attack (e.g., used to jump to a `jmp *%esp`, or `ret` instructions which are in turn used to reach and execute injected code).

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, it is impossible to jump into the *false* one[3]. 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 this kind of attack.

The host intrusion detection system (HIDS) research community has been proposing techniques to deal with this and other types of attacks which can be triggered by exploiting memory error vulnerabilities. However, as shown in [87, 28] some of these models are able to detect a subset of IPE attacks but fail in detecting all. To this end, Figure 2.1 depicts an example of code snippet originally proposed by [28] and slightly modified to better show how an IPE attack can be successfully perpetrated. Let us suppose that the function `is_regular(uid)` (line 20) invokes the `open` system call

[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 are present/used.

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. The memory error vulnerability present in the code snippet of Figure 2.1 can be exploited to execute arbitrary command with full (superuser) privileges. In fact, a regular user camouflaged as an attacker, by entering the true branch of the `if` statement (lines 21-27) and by exploiting the stack-based buffer overflow in `read_next_cmd` at line 8, is able to divert the program P execution flow in order to enter the false branch which eventually will give him full privileges.

It may be argued that IPE attacks could be difficult to perform since they depend on too many factors (e.g., program structure, vulnerability "at the right position") but, however, as pointed out by Feng *et al.* [28], they should not be left unconsidered since it may be quite easy, for an attacker, to deliberately introduce the right conditions in the program source code that may lead to an execution of an impossible path.

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

Figure 2.1: Code snippet that can be exploited by an IPE attack.

Data and data pointer corruption. Differently from the previous point, this memory corruption do not directly target data used to *control* the process execution flow. This classification comprises all the program's data which are *sensitive* under a security point of view. Therefore, their corruption can be as dangerous as code pointers corruption, even if it does not give an attacker the ability to directly execute arbitrary code or perform IPE-like attacks. Data and data pointers corruption have been recently described by Chen *et al.* in [14], and, as pointed out by the authors, these attacks can be as dangerous as the ones corrupting code pointers.

Pointer. In certain situations they might be the target of an overflow

and, if subsequently used, can permit an attacker to either bypass vulnerability-specific memory error countermeasure ([63]), indirectly corrupt other security sensitive data, or simply *write-anything-anywhere* in the address space of the victim process.

Non-pointer. Security sensitive data are crucial for the correctness of a program. For instance, it is easy to understand how a variable `auth` could be set to a non-zero value to indicate a successful authentication process. The corruption of such security sensitive data due to memory error exploits would compromise the security of the whole process (e.g., unauthorized access, code execution with full privileges). Similarly, cryptographic keys stored into the process address space can be corrupted and replaced by means of this type of memory error. Of course, a similar argumentation can also be made for data used as arguments of security critical functions (or system calls).

In the following, we briefly remind the most common memory error vulnerabilities that can be found in software. We also remind that every vulnerability can be exploited to eventually corrupt code pointers, data pointers, or non-pointer data.

2.1 Buffer Overflows

In the C programming language, when a buffer (or array) is declared, space for it is reserved as a consecutive sequence of elements of the same type (e.g., buffer of bytes). No ancillary information on the buffer, such as its length, are stored elsewhere[4]. A buffer overflow occurs when data is written past the end of the involved buffer. This is due to the lack of or improper check on the size of the buffer (bounds checking) during a copy operation and, as a direct consequence, memory locations adjacent to the overflown buffer are corrupted. Depending on where the buffer is stored, it is possible to talk about stack-based, heap-based, and static data buffer overflows.

2.1.1 Stack-based Buffer Overflows

Stack-based buffer overflows [25] are probably the most common, well-understood memory error vulnerabilities. As the name suggest, they happen on the stack when a buffer overflows and overwrites adjacent memory regions. The most common way to exploit a stack-based buffer overflow is to write past the end of the buffer until the function (saved) return address, stored on the stack is

[4]That is, the programmer should keep track of every buffer length. Sometimes, this is easy and can be done implicitly. For instance, C strings are represented as sequence of `NULL`-terminated bytes (`char`). The `NULL` marker (byte whose value is 0) represents the end of the string.

reached. The corruption of this code pointer permits to execute arbitrary code. The example shown in Figure 2.2 depicts a classical scenario.

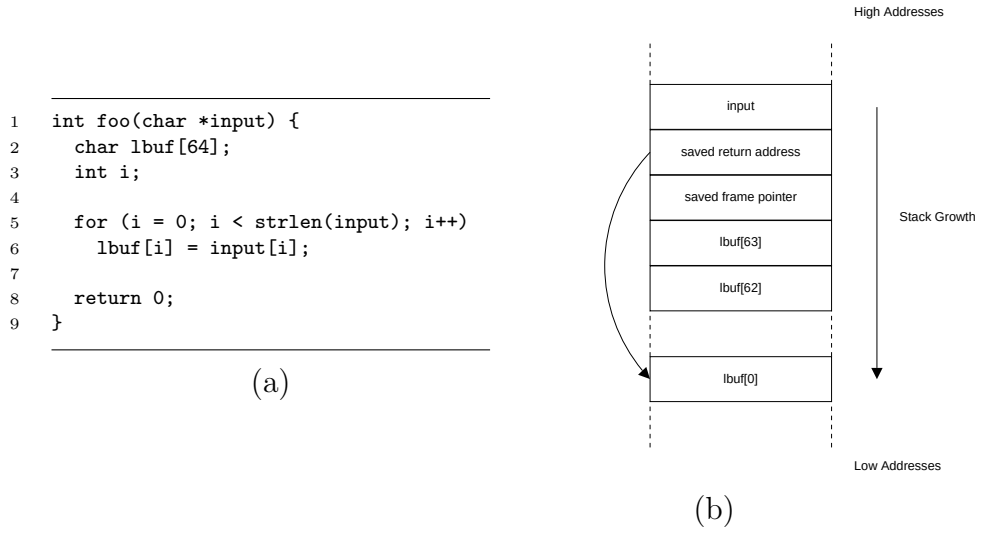

Figure 2.2: Stack-based buffer overflow vulnerability.

When the function `foo` is invoked, the CPU pushes the function return address onto the stack and the execution begins at the `foo` entry. Afterwards, the function *prologue* – if any – is executed. This has the effect to save the value of the frame pointer register, which points to the caller stack frame, onto the stack, in order to be able to set the new stack frame as current, by initializing the frame pointer register appropriately. Next, function local variables (`lbuf` and `i` in this order) are allocated and subsequent instructions are executed. The `for` loop starting at line 5 does not check whether the destination buffer is big enough to hold a copy of `input`, the argument passed to the function `foo`. As a result, a memory error exploit can write past the end of `lbuf`, corrupting the saved frame pointer[5], and either jump back on the injected code [25], as shown in the (b) column of Figure 2.2, or jump to execute already existing code [62] (for instance, if non-executable stack/data countermeasure is present [41, 71]), thus, executing arbitrary code in both cases or, again, corrupting security sensitive data [14].

2.1.2 Heap-based Buffer Overflows

One of the initial way that has been proposed to exploit heap-based buffer overflows was to overflow heap allocated buffers with the intent to overwrite function pointers that were stored adjacent next to the overflown buffer [50]. Alternatively, corruption of C++ virtual pointers table which is usually stored at the beginning of a dynamically allocated object, would allow to invoke arbitrary (malicious) code at the object's method invocation [64]. Tampering with adjacently stored

[5]We remind that controlling the saved frame pointer is sufficient to eventually execute arbitrary code [46].

data was also possible, but attackers' opportunities were not as appealing as the ones which could be gotten by exploiting stack-based buffer overflow (there, a code pointer *always* exists on the stack).

Anyway, in 2001, others techniques targeting heap allocated objects were proposed which corrupted heap management information. In fact, as a function return address is stored on the stack and could potentially be reached by means of overflows or other indirect attacks (e.g., pointers corruption, format string vulnerabilities), heap management information are stored right before the data which is directly usable by the program. Therefore, if we have two adjacent heap allocated buffers, writing past the end of the first will overflow into the second, corrupting its heap management information. Without going into much details, the attack exploit the way some internal heap management functions work which, eventually, act on the doubly linked list of free chunks the underlying heap memory allocator maintains (see [22], for instance). In particular, one of these function is the macro `unlink`. Its instructions remove an element from the free list (either because it has to be allocated, or because two free chunks border and they have to be coalesced into a bigger unique free chunk to avoid having lots of small fragmented chunks), as shown in the following (snippet shown in (a) is equivalent to the one shown in (b). The latter only explicits offsets in the structure).

```
1  chunk2->fd->bk = chunk2->bk;
2  chunk2->bk->fd = chunk2->fd;
```

(a)

```
1  (chunk2->fd + 12) = chunk2->bk;
2  (chunk2->bk + 8) = chunk2->fd;
```

(b)

The values `chunk2->bk` and `chunk2->fd` are part of the heap management information stored in-band (i.e., at the beginning of the chunk) and they point to the previous and to the next free chunk, respectively. Let us suppose that `chunk1` data is under the attacker control (i.e., allocated) and that `chunk2` is a free chunk which is adjacent to (i.e., it follows) `chunk1`. Let us also suppose that `chunk1` overflows into `chunk2`. As a consequence, `chunk2` heap-management information are corrupted. To see how this information can be corrupted to permit the attacker to overwrite an arbitrary memory location and, usually, to eventually execute arbitrary code, let us also suppose that `chunk2->fd` holds A and `chunk2->bk` holds S, attacker supplied values. When the overflown chunk is freed, it will be merged with `chunk2` as they border and they are both free chunks. However, `chunk2` has to be removed from the list of free chunks as it is no longer a chunk by itself, but it has been coalesced with `chunk1`. The `unlink` macro is now involved and its two instructions shown above have the effect to write the value S at the memory location $A + 12$ ($A + 12 = S$) and the value A at the memory location $S + 8$ ($S + 8 = A$). For exploits which aim to corrupt code pointers, S is usually the address where the shellcode[6] is stored, while A

[6] A *shellcode* represents a sequence of bytes which allows an attacker to execute *arbitrary* code. Historically, the final attacker's goal was to execute the shell interpreter `/bin/sh` and, even if this no longer holds, it is the reason why this sequence has been called this way.

represents the memory location of a code pointer which is necessary to trigger the execution of the malicious code. Due to the aforementioned simple math, if the attacker supplied $A - 12$ instead of A it would definitely achieve the effect to write at A the intended value[7]. More generally, this attack represents a write-anything-anywhere primitive, therefore allowing the attacker to perform more sophisticated attacks.

2.1.3 Static Buffer Overflows

These overflows happen on the data segment (i.e., `.data` and `.bss` sections which are usually stored within the same writable segment [81]) and they are similar to the heap-based ones, except that they do not usually target heap management information or at least, not directly. Through a static buffer overflow, it is possible to overwrite ELF-specific data, as `.dtors` (that can be seen as a table of application-specific function pointers that will be invoked upon application termination) [44], and process GOT entries, but this is highly constrained on the way the program has been linked. In fact, current linker scripts usually put all these sections *before* the data section. Therefore, in this case, the corruption of those data is possible only through indirect overwrite (i.e., the corruption of a pointer which will be subsequently used to corrupt other memory regions). To summarize, in its simplest form, a static overflow aims to corrupt application-specific function pointers stored on the data segment itself, which would allow for a direct code execution, data pointers, which would allow to indirectly corrupt other memory regions, or non-pointer adjacent security sensitive data.

2.2 Format String Vulnerabilities

Format string vulnerabilities [69, 35] are a more recent discovery than buffer overflows. Discovered at the end of the '90, they represent a serious and dangerous vulnerability. Unlike buffer overflows, format string vulnerabilities are easily used as a write-anything-anywhere primitive, therefore potentially corrupting the whole address space of a victim process. Beside this, format bugs (another name for this kind of vulnerabilities) can also be exploited to *arbitrary* read the whole address space of a process. This way, disclosing confidential data (e.g., cryptographic keys, pseudo-random number used by some memory error countermeasures [15, 79]) as well as dumping the whole address space content of a victim process becomes possible.

This vulnerability affects the `printf` family of functions. These variable arguments functions take usually a *format string* as argument and a series of other arguments, accordingly to the formatting string. If the format string is under the

[7]The careful reader has probably noted that the second instruction corrupts 4 bytes of the shellcode starting at offset 8. This is not really a big constraint, as the injected code can be easily written to jump over the corrupted sequence.

control of an attacker (e.g., `printf(buf)`), the vulnerability can be exploited. What would happen if the function would have fewer arguments than the one expected (and specified by the formatting string)? The missing arguments would be looked up into and retrieved from the stack or, more generally, from the process' address space. Depending upon the formatting directive used, double words can be directly (e.g., `%x`) or indirectly (e.g., `%s`) retrieved. Moreover, the number of bytes written so far by the these `printf` family of functions can also be written at next address to be retrieved from the stack (typically), by using the `%n` or one of its variants (e.g., `%hn`, `%hhn`, `%`k`$n`). A typical exploitation of this vulnerability requires to reach the buffer controlled by the attacker (e.g., by popping double words off from the stack) which represents the format string itself so that, by using the aforementioned formatting directives, arbitrary memory regions can be corrupted (write-anything-anywhere primitive). As a consequence, arbitrary code can be executed or security sensitive data becomes under attacker's control.

2.3 Integer Overflows

Integer overflows [10] are not memory errors by themselves. However, incorrect integer handling can trigger memory errors, such as buffer overflows or write-anything-anywhere-like primitive, depending on the involved integer misinterpretation. The issue arises due to the way integer are represented on computers. For instance, on IA-32 an `unsigned int` type is usually 4 bytes, while 2 bytes are needed for an `unsigned short int` type. If the value assigned to an `unsigned short int` variable is $2^{16} - 1 - k$, that is far from its maximum value of k, adding $k+1$ will cause the variable to wrap around and its value will become 0 (a similar reasoning can be made for underflow). This can be used to bypass security checks or write to (almost) arbitrary memory regions, especially when `unsigned int` variables are involved[8].

A more subtle way to exploit integer overflows is caused by the fact that two different representations are used depending whether the considered integer is *unsigned* or *signed*. For instance, let us consider the following example which tries to prevent buffer overflows conditions by performing explicit bound checking.

```
1  ...
2  char buf[64];
3  ...
4  void *safe_copy(void *dst, const void *src, short len) {
5
6     if (len > 64)
7        len = 64;
8     return memcpy(dst, src, len);
9  }
```

[8] In fact, `unsigned int` variables are 32 bits wide and so they can be used to address the whole default user space process address space on IA-32 machines.

The condition at line 6 tries to check that the number of bytes that must be copied from `src` to `dst` does not exceed `dst` buffer size (64). Unfortunately, the argument `len` given to the function `safe_copy` is a *signed* short, while the one used by `memcpy` is *unsigned*. This means that by giving a negative number as `len`, the bound checking enforced at line 6 will be useless. However, the `memcpy` function will interpret this length argument as a non-negative (unsigned) value, bigger than the buffer size, therefore causing `buf` to overflow. For instance, using $0x8000$ as `len` will bypass the check at line 6 as `len` will be interpreted as -32768, while the same length will be interpreted by `memcpy` as 32768, which in turn will cause memcpy to write past the end of the buffer `buf`.

Part II

Research Work

Imagination is more important than knowledge.

Albert Einstein (1879 - 1955)

3 Diversified Process Replicæ

Diversity plays a crucial role for the survivability of every biological species and, quite recently, the concept, has also been applied to computer programs [49, 68, 24, 67, 33, 65, 79]. Researchers in the computer security field started to apply different kinds of software transformations such as address space layout randomization [79, 67], instruction set randomization [24, 33] and several forms of more general program transformation techniques [68] in order to strongly thwart memory error attacks, no matter whether such diversities are made available by the OS kernel or by automated user space transformation approaches. By memory error exploits we mean all those techniques that an attacker may use for exploiting a particular vulnerability (see for example [25, 62, 69]) by overwriting and thus corrupting suitable memory addresses. The final purpose is to hijack a program P execution flow to either execute arbitrary code or to bypass security mechanisms.

One of the main drawback of such approaches is their *probabilistic* nature. In fact, software diversity applied on a process P can just improve the likelihood of resisting to some form of memory error exploits. Moreover, it has been observed that the existing forms of process diversification might be eluded by means of information leakage (see for example [69]) or are not so effective in protecting a process or, again, cannot protect from all the existing memory corruption attacks [4, 40].

In this dissertation, we provide a different interpretation of the notion of software diversity, independently conceived by Cox *et. al* in [7] as well. Such an interpretation is based on the concept of process *replica*. Given a process P, we define P's replica as a process P_r which behaves identically to P even if it presents some "structural" diversity from it.

By adopting such a notion of diversity, it is possible to devise mechanisms for detecting attacks in a deterministic way. The idea is very simple. A process and its replica fed by the same external non malicious input will behave in the same manner. However, a malicious input will modify some particular part of the internal P structure (as in the case of any memory error exploits) so that either

the P or its replica P_r will eventually start to behave in a different detectable way, giving the opportunity to block the attack with a *deterministic* protection.

More precisely, in our solution, a process and its replica only differ in their address space layout. In particular, we make the following contributions:

1. We devise a model which defeats memory error exploits targeting absolute memory addresses as well as those which *partially overwrite* a memory address. The former, independently addressed by Cox *et al.* in [7] as well, refers to all those exploitation techniques an attacker may use to overwrite a suitable memory object with an absolute memory address value in order to hijack a process execution control flow. The latter, instead, permits to overwrite a memory object with a partial value, thus allowing a relative execution flow hijacking. This latter class of attacks, generally known as *Impossible Path Execution* (IPE) attacks, can permit an attacker to bypass critical application-based security checks. Even if at first glance it might argued that IPE attacks are not so realistic, as pointed out in [28], this class of attacks are becoming a serious real security threat.

2. We give a complete characterization and propose a solution with respect to shared memory management, one of the biggest practical issue introduced by the approach proposed in [7], as well as diversified process replicæ, the one herein described. This issue, and others, has to be solved to permit a real and practical deployment of the whole strategy. Moreover, preliminary ideas on how to deal with synchronous signals delivery between a process and its replica are faced as well.

3. We developed a prototype proof-of-concept using the `ptrace` system call, on a little endian 32-bit Intel Architecture host running on a 2.6.x Linux kernel. Even if the performance results might not seem enthusiastic at first glance, conceptually speaking the idea is correct and seems to be a viable way towards systems survivability.

It is worth noting that existing techniques which provide a *probabilistic* protection [24, 33, 67, 68, 88, 79] do already provide acceptable solutions to memory errors. However, we would like to briefly cite fewer limitations which more justify a *deterministic* protection as the one proposed in the following.

Some existing memory error techniques are either too specific (e.g., [63, 15, 31]) or they are constrained by the underlying architecture or, again, they show other drawbacks. For instance, the approach proposed in [15] and similar techniques are too specific and thus fail to give comprehensive protection to memory errors. Moreover, they can easily be bypassed [63]. Address space layout randomization techniques (e.g., the one provided in [79] – or the one natively adopted by the 2.6.x Linux kernel) have some drawbacks as well. For instance, (i) they are not very effective on 32-bit architecture, as shown by [40] due to architectural constraints (e.g., memory pages aligned on 4KB boundary at least, which

cut off the 12 least significant bits that cannot be randomized), (ii) they are not effective against information leakage attacks, and (iii) they do not provide protection to impossible path execution (IPE) attacks – that is, attacks able to perform relative jumps to bypass security critical piece of code due to a *partial address overwrite*, for instance. Address space obfuscation/randomization (ASR) techniques, as the one provided by [8, 68, 67] provide a better and more comprehensive protection than the previous approaches by using a fine-grained address space randomization. However, the transformation proposed is more invasive than the one performed by the approach proposed in the following. ASR requires source code, and dealing with assembly code present in the source code is problematic as well. Furthermore ASR does not generally provide protection from IPE, and only probabilistic protection to information leakage attacks which eventually corrupts code/data pointers is provided. Moreover, this protection highly depends on the vulnerability encountered (see § 2). For instance, a format bug will have more chances to be successful than a buffer overflow, in this scenario. The strategy proposed in the following sections only partially shares some of the aforementioned limitations and drawbacks while achieving or improving the protection power.

3.1 Preliminaries

This section reminds fewer concepts about the Executable and Linking Format (ELF) [81] specification. Moreover, some remarks on a process address space layout are given at the end of the section.

3.1.1 Executable and Linking Format

The ELF specification [81] describes the format of *executable*, *shared* and *relocatable* objects. While we are not concerned with the latter one, in the following, we briefly recall on a fewer concepts about the others.

An ELF executable object (`ET_EXEC`) is an object file that holds a program code and data ready for the execution by the underlying operating system (OS). Two different cases have to be considered, for this type of executable object:

statically-linked binary. The object file contains all the code and data needed for its execution, that is, any external reference to library code and data is properly retrieved, relocated and correctly linked into the object file by the link editor, which eventually produces the desired executable object.

dynamically-linked binary. It contains only the executable program code and data while references to any external libraries or, more generally, shared objects (`ET_DYN`) referenced by the executable, will be resolved and managed at run-time by *rtdl*, the run-time dynamic linker (see also § 3.1.2).

An ELF executable object usually hold *absolute* code and data, no matter if the object is statically or dynamically linked. That is, the virtual addresses the object is mapped at are fixed. Moreover, any relocation information of the considered binary is generally stripped and thus it cannot be neither re-linked nor relocated anymore (obviously, dynamically-linked binaries have all the required information the dynamic linker will use for binding external references to their definition at run-time).

An ELF shared object (`ET_DYN`), instead, holds code and data that is usually dynamically linked into a process address space. Since different processes may use a different number of shared objects, such objects cannot contain absolute code and data references. Thus, they might potentially be mapped at different virtual addresses into the processes address space that make use of them. For such a reason, shared objects contain *position independent code*[1] (PIC) in order to permit the rtld to dynamically load the object into a process address space at an "arbitrary" base address and to correctly perform dynamic resolution of its symbols.

3.1.2 Process Address Space

The address space of a user-space process consists of all the virtual memory addresses a process may access [19]. Usually, on a *vanilla* Linux kernel running on a 32-bit Intel Architecture, a process, running in user-mode, is allowed to access the first 3GB of its address space while the whole 4GB is generally addressable in kernel mode.

For convenience and to ease the management of virtual memory a process address space is usually divided into regions each of which hosts particular parts of the ELF object being mapped. A typical division for a Linux process tries to map `ET_EXEC` ELF object text segment starting at the virtual address `0x08048000` ([81]) followed by its whole data segment (both `.data`, `.bss` and the start of dynamic heap). Everything must reside on a page boundary and it is necessary to honor any existing displacement present in the physical object file. If the executable object is dynamically-linked the kernel maps the run-time linker, usually `ld-linux.so`, which in turns eventually maps all the shared objects used by the executable, usually starting at the address `0x40000000`[2]. Finally, the kernel sets up the mapping for the stack region that grows downward, towards lower memory addresses starting from the address `0xbfffffff`, the last virtual memory address addressable in user space.

It is worth noting that such a mapping is applied to *every* process. Every process has the same view of its virtual address space which is a process' private resource.

[1] Indeed, even `ET_EXEC` ELF object can be made PIC in order to be mapped at a different base address by only experiencing a little performance slowdown.

[2] Even if using a 2.6.x Linux kernel, we are assuming the *legacy* address space layout.

3.2 Process Replication with Diversification

Process replication aims to create a process replica P_r of a given process P. To this end, P and P_r are artificially diversified so that each of them has a different non-overlapping memory address space layout. Thanks to the replication actions (§ 3.3) and diversification approaches (§ 3.2.2) both P and P_r will exhibit the same behavior as long as they are in the same environment and they are fed by the same benign input. However, malicious input that carries memory error exploits attempts will let the process and its replica to diverge in their behavior. The reason behind this lies in the fact that a memory error exploit should use an attack pattern usually comprising a given absolute memory address $\mathcal{A}$. Since P and P_r are artificially diversified (non-overlapping address space) and replicated, it is impossible that $\mathcal{A}$ is suitable for both processes. Any attempt to use $\mathcal{A}$ into P's and P_r's context will make them behave differently (generally one of them will eventually crash) giving the opportunity to spot the attack.

Partial address overwrite attacks can still be successful if we only ensure non-overlapping address space. However, such attacks class can be defeated if *relative distances* between P and P_r address spaces are properly diversified, as shown in § 3.2.2.

In the following we describe the model framework we devised as well as how diversity and replication are obtained and mapped by the framework.

3.2.1 Model Framework

The model framework is represented in Figure 3.1, and it is composed by three main elements: the process P, its replica P_r and a replicator and monitoring process T which we will call the tracer. Even if not further specified, it is clear that even T must be somehow protected.

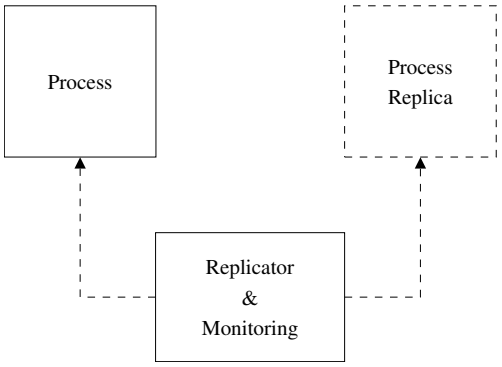

Figure 3.1: Model Framework

The main goal of T is to start, perform I/O replication and system calls management actions, and monitor the execution of P and P_r, while looking for any anomalous condition (see § 3.3).

Thus, T has to feed P_r with the same input given to P and it has also to correctly manage the system calls invoked by both processes so that they will exhibit the same behavior. To this end, P and P_r must be maintained *synchronized* by T and this is done on a syscall-based granularity by making P and P_r reach what we called a *rendez-vouz* point. The processes that are going to interact with P would not even notice the presence of P_r. Before going into the details of the diversification and replication approach, we can anticipate its effectiveness in defeating absolute and partial address overwriting attacks as show in Figure 3.5 (see § 3.4.1 for a description of the approach).

Details about the differences between P_r and P as well as the mechanisms adopted by T for "hiding" P_r while keeping P and P_r behavior consistent are the topics of the following sections.

3.2.2 Non Overlapping Processes Address Spaces

The diversity approach we adopted, independently conceived by Cox *et al.* in [7], aims to provide a non-overlapping address space between a process P and its replica P_r in order to defeat memory error exploits which aim to corrupt absolute memory addresses. By non-overlapping, we mean that no overlapping address spaces can be found when comparing the virtual addresses where the processes have been mapped at. A possible example is depicted in Figure 3.2. As reminded in § 3.1, usually every process is mapped starting at the same virtual memory address and the same applies for the stack region as well as memory mapping area created by the `mmap` system call. The main objective of address space diversification is to break such an assumption.

However, as noted at the beginning of § 3.2, the diversification proposed in [7] cannot deal with partial address overwrite attacks. In fact, these can still be successful even when adopting such a diversification between P and P_r. The reason behind this lies in the fact that partial address overwrite can permit *"relative jump"* to bypass security checks because "relative" distances between P and P_r address spaces are kept the same by default. Thus, our idea is to break this assumption here as well and to "shift" the address space of P's replica by k bytes. This way, relative distances between P and P_r address spaces are properly diversified thwarting partial address overwriting attacks.

In the following we describe the strategies adopted for reaching such an objective in the case of statically-linked binaries and dynamically-linked ones.

Statically-linked Binaries

In order to successfully diversify `ET_EXEC` ELF objects, we modified the default `ld` linker script[3] to achieve the following goals:

[3]Obviously, the same approach can also be applied to `ET_REL` ELF objects, that is, relocatable code.

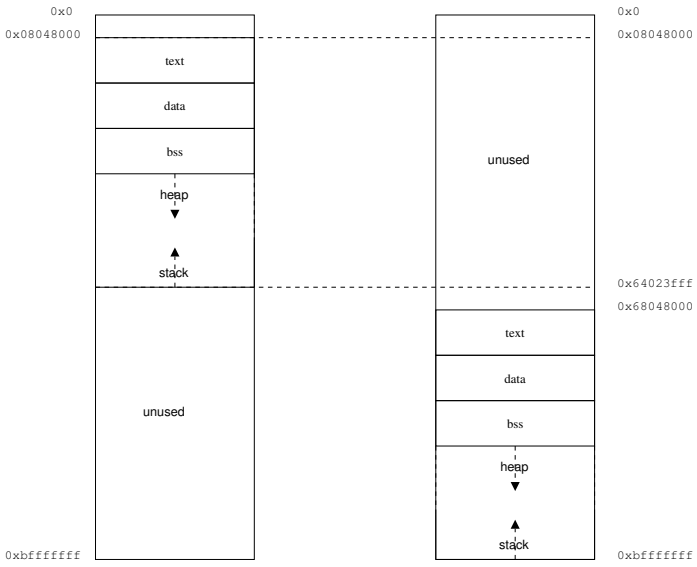

Figure 3.2: Diversified Process Replicæ

- load P_r starting at a custom address different from the one defined in the ELF ABI [81]; for our test purpose we initially used `0x68048000` instead of the default one (`0x08048000`). Obviously, a checking of the sizes of P and P_r (`.text`, `.data`, `.bss` segments as well as dynamically checking for heap expansions) are required to ensure non-overlapping processes address spaces;

  Note that, in order to achieve full non-overlapping address space, other regions, such as stack and memory mapped area (heap comes after the `.bss` segment so, it can be transparently handled by the modified linker script) have to be mapped at different addresses too. In order to accomplish this task and to be as transparent as possible with respect to the diversified executable object, we modified `ld-linux.so`, the dynamic run-time linker by using an approach similar to the one described in [88]. Of course, also the dynamic linker itself has to be "relocated" as well.

- modify the least significant byte (LSB) of the address at which `ET_EXEC` ELF object will be mapped at. This is achieved by inserting "junk" data right at the beginning of the `.text` segment description in the linker script, using the `LONG(`k`)` linker script keyword, taking care of the required alignment constraint[4] (e.g., 4-byte alignment). Thus, all the code is moved k bytes upward (towards higher addresses), thus shifting the executable entry point, as well as its code and data segments. This mechanism may be repeated as long as it is possible to obtain different "LSB values" for P and P_r thus

[4]Indeed, there are other keywords that may be used to achieve the same result. Moreover, due to sections padding and sections-to-segment mapping it may be necessary to carefully insert these junk bytes.

defeating any memory errors exploits that target partial memory address overwrite (IPE attacks).

In our initial test, the `0x68048000` has been shifted of 8 bytes and which gave us good results (see § 3.4.1).

Dynamically-linked Binaries and Shared Objects

Dynamically-linked binaries are a bit more tricky to deal with since the shared objects used by the executable have to be diversified in order to take full advantage of the entire diversification approach.

The "main" executable object (`ET_EXEC`) can be diversified as previously described while `ET_DYN` ELF objects, that is dynamic libraries or more generally shared objects, have to be properly handled. On the other hand, the base address of the considered shared object O is transparently diversified by the modified `ld-linux.so` described in the previous section. However, in order to achieve protection from partial address overwriting attacks, it is necessary to perform the same object "shifting" performed on statically-linked binaries. There are basically two alternatives.

1. The first one chooses to diversify shared objects when they are just going to be loaded by `ld-linux.so`, the run-time dynamic linker (*rtdl*), right after the *rtdl* maps the shared object O using the `mmap` system call but before they are used even by the *rtdl* itself, penalty the corruption of the in-memory shared objects data structures involved. The tracer T can easily handle this situation since the *rtdl* operates on behalf of the executing process and T monitors both P and P_r.

   Roughly speaking, after the *rtdl* maps a particular shared object segment m via `mmap`, T has to:

   (a) keep track in a table of the address returned by the mapping request as well as its length and the amount of desired shift (see next point);

   (b) shift the segment $\mathcal{M}$ just mapped by k bytes;

   (c) update relocation entries and the program header table (PHT) of the `ET_DYN` object as relocation references are shifted by k bytes;

   (d) give back to *rtdl* the `mmap`'d address displaced by the k-byte shift performed in order to permit the run-time linker to correctly reference the ELF header of the object O as well as all the others relevant ELF structures of O and the whole mapped region[5];

   (e) monitor any non-anonymous un-mapping request via `munmap`, in order to adjust by k bytes the address specified in the request and have the

[5]This is true for the segment that contains `.text`, `.rodata`, `.plt` sections and so on. Others loadable segments, such as the one holding "writable data", have to be subjected to the same shifting operation to honor the relative addressing that PIC objects exhibit.

kernel to correctly un-map the region, using the information stored in 3.

Unfortunately, this approach has limitations and drawbacks. Segments are usually padded during load time in order to obtain in-memory segments on a page boundary (e.g., 4KB-aligned) while respecting relative segments addressing. The shift operation exploits the padding introduced in order to use some unused in-memory room to shift the whole segment. Consequently, the aforementioned approach cannot be deployed on those segments whose size is already equal to a memory page. However, preliminary tests we conducted on a Debian GNU/Linux testing system reported that the percentage of shared libraries that would hardly take benefit of such an approach due to low-padding space is really low (about 0.4% on a 1947 sample). The great majority of the rests would be smoothly diversified. Nonetheless, we are currently investigating other solutions to undertake in order to achieve the same protection provided by the in-memory shifting operation for all the shared objects involved.

Another big drawback of this approach is the waste of (physical) memory that is required because of the shift operation (transparently handled by the copy-on-write (COW) kernel mechanism).

It is also worth noting that a kernel level patch has to be developed for handling the run-time dynamic linker since it also has to be modified by the same "run-time patching" mechanism applied to the shared objects. Object shifting to achieve LSB diversification has also to be applied to plugins loaded by means of `dlopen` library function which eventually invokes the `mmap` system call[6]. Moreover, a "stack shifting" has to be performed as well by the aforementioned kernel patch.

2. The second approach is simpler but it needs shared objects source code, which is not always available. It performs the address space "shifting" at compile time, by recompiling the needed `ET_DYN` object, using a custom linker script as it has been used for the statically-linked binary case.

Currently, our prototype supports the latter approach.

3.3 Replicator Module

The replicator and monitoring component T of the framework depicted in Figure 3.1 is in charge of (i) letting P and P_r reach a common execution point which defines what we have called *rendez-vouz* point to synchronize P and P_r

[6]In order to correctly perform this step, the tracer T can keep track of the object i-node whose file descriptors are used as argument to a non-anonymous `mmap`. This way it would be possible to perform the *shared object shifting* without incorrectly act on non-shared objects.

behavior, (ii) performing I/O replication and system calls management, and (iii) continuously monitor P and P_r, raising an alarm and terminating the both processes upon anomalous conditions are detected (attaks). In particular, T has to performs the following actions:

(i) It executes a process P and its replica P_r which has been previously diversified (see § 3.2). It is worth noting that T actually traces P and P_r execution using the `ptrace` system call. Such a system call permits T to "asks" the OS kernel to stop the execution of the traced processes every time they "enter" a system call s, that is before actually executing it, and right before they are willing to "exit" from s, that is after s has been actually executed by the OS kernel on behalf of P or P_r. It may also be observed that while performing these steps, T acts like a kind of a "high-level" scheduler whose purpose is better explained in the following items (however, the real "low-level" process scheduler remains the kernel).

(ii) It performs I/O replication on some I/O related system call invoked by P and P_r. Moreover, T has to correctly manage all the system calls invoked by P and P_r. To this end, T ensures that both P and P_r enter a system call s, reaching what we define a *"rendez-vouz"* point[7]. The main purpose of this synchronization point is to permit P and P_r to reach a common state in their execution flow f *before* actually execute s. This is necessary since, due to the peculiarity introduced by diversification and replication, different actions have to be taken depending on the considered system call and whether it has been invoked by P or by P_r. It is worth noting that if P and P_r receive the same *non-malicious input* they *behave* identically since they only differ in the memory locations they have been mapped at. Moreover, since T starts the execution of P and P_r, monitors them and takes the appropriate decision on a system call-based granularity, both P and P_r will end up by invoking the same system call s (with the same equivalent or comparable arguments). In particular, it is possible to classify the system calls depending on the actions T must carry out. In particular:

simulated system call. T enables the execution of s only to P. At the end of the system call, i.e., before enabling P to continue with its execution (that is at s exit), T *replicates* the effects produced by s onto P_r address space. For example, if s is represented by the `read` system call, T waits for P and P_r to enter s and it checks whether they both want to invoke it (also comparing all those immediate values that can be compared to, e.g., file descriptor, flags and mode if present). Afterwards, P invokes s and, once s is correctly executed, T replicates the data just read, if any, from P's address space to P_r's address space, accordingly modifying s' return value in P_r context as well. Non-erroneous and non-malicious read actions would not alter any code

[7]This term as a well defined semantic but here it is used with its more general meaning.

and data pointers stored in the process memory address space. P and P_r semantic will be identical and they will exhibit the same behavior.

executed system call. Both P and P_r execute s since it creates or modifies in-kernel process structures; such an execution is necessary since the actions performed and the values returned by s may be subsequently used by other system calls or a "simulation" would require too much effort to be done without kernel intervention (e.g., `mmap` or `mmap2`, excluding the "write" mode that deserve special treatment as further explained in § 3.5.1). A typical example is represented by the `open` system call since it creates in-kernel structures whose user level representation (i.e. file descriptor) might be used as an argument to other system calls that will be possibly executed by the involved processes (e.g., `close` can decrease an object file usage reference count);

carefully treated system call. There are fewer system calls, such as `mmap`, `mmap2` and IPC related ones like `shmat` and `shmget`[8], that have to be treated carefully since otherwise they may render inconsistent both P and P_r address spaces as well as the mapped objects. A step toward a possible correct treatment of such system calls is given in § 3.5.1.

Actually, due to the nature of our user-space approach, some system calls present a mixture of the first two points, that is they have to be somehow executed since they cannot be made to fail by our user-space prototype, but they also have to be simulated in order to provide consistency between P and P_r address spaces. A typical example of this situation is represented by the `getpid` system call: in order to guarantee a consistent behavior between the processes `getpid` invocations made by P and P_r have to yield the same process for both processes.

(iii) Finally, T continuously monitors P and P_r in order to check whether they receive signals so that proper actions can be taken. For example, during a classical successful memory error exploit, one process, say P, will keep going on while P_r, which has a different non-overlapping address space layout, will eventually crash letting T to correctly handle this situation by either raising an alarm or terminating P (see § 3.2). Thus, if we assume that in order to make real and useful damage on a system at least one system call has to be executed [87], this way no meaningful, from the attacker viewpoint, harm or damage can be successfully perpetrated against the protected system. In fact, it should be observed that both P and P_r have to synchronize themselves by reaching a rendez-vouz point. This means that *both* have to enter a system call s before it can actually be executed. So, if a process P is tricked into invoking a system call s but P_r is crashed, no rendez-vouz point will be reached and thus no system call will be invoked at all.

[8]Indeed, it depends on the considered kernel whether these represents actually a system call or a library function call that eventually invokes the same system call.

Recent research [14], however, showed that indeed is not always necessary to execute a system call to cause damage. Even if some memory error attacks which do not corrupt code pointers are currently caught by our approach, others are not. This is, unfortunately, a limitation of our approach. In particular, as long as pointers (code or data) are considered, there is a good confidence that one process, say P_r, will crash as soon as the pointer is dereferenced, while the other, say P, will not. This running process is clearly the real issue. Following this scenario, P can, (i) keep corrupting the process address space, or (ii) eventually invoking a system call. In (i) it has further choices. It can corrupt pointers or non-pointers data. In the former case the process will crash as soon as the pointer is dereferenced, while in the latter not. Clearly this could potentially be an issue. If the monitor (i.e., the "detection mechanism", the tracer T) resides within the process address space, and no further protections (ala software fault isolation) are used then it would be possible for the attacker to subvert the detection mechanism at once, by corrupting its internal data structure. Consequently, the attacker would be able to execute system calls without being detected anymore (no more process replica, no more replication). However, in our implementation, the tracer T uses the `ptrace` system call to accomplish its replication task (i.e., monitoring at system call level, replicating data, and so on). The tracer is therefore a separated process which does not share data neither with P nor P_r. A way by which a corrupted process P can tamper T's address space is, for instance, to use the `ptrace` system call as well. However, this is more tricky and can completely be avoided by using simple anti-debugging technique so that no external high privilege process can attach and monitor T (e.g., T executes `ptrace(PTRACE_TRACEME)` which implicitly does not allow any other processes to attach to itself anymore). The point (ii) poses no particular issue as we remark that the tracer T waits for P and P_r to reach their rendez-vouz point. One process, say P_r, has already been terminated due to a reference to an invalid memory access. When the other process, say P reaches the synchronization point, T knows that P_r will never do that, so it kills P as well. No damage is done on the system.

Further discussion about attacks on the proposed approach as well as argumentation on the protection provided are remarked in § 3.4.3.

3.4 Evaluation

In the following, we present the evaluation of our diversified process replicæ approach in terms of effectiveness and experimental results. The section ends by analyzing the security of the approach pointing out its weaknesses, limitations and, wherever possible, solutions to these issues.

3.4.1 Effectiveness

In order to validate the feasibility of the approach herein proposed we test its effectiveness with respect to memory errors exploits that aim at:

- overwriting memory addresses with absolute values needed to divert the legal process execution flow.
- corrupting least significant bytes of a memory address thus performing what has been so far called partial address overwriting.

The former method can be used by an attacker to exploit common memory corruption vulnerabilities, such as buffer overflows, heap overflows, format string bug, `jmp_buf` overwriting and so on, to usually execute arbitrary code. The latter method, instead, may be used to successfully perform what in literature known as an impossible paths execution (IPE) attack [85, 28, 16].

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[9]. It is simply an impossible path to follow due to the structure of the program and the *if/then/else* semantic. If properly recognized, 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. Usually, to perform a successful IPE attack, it suffices to overwrite the LSB of a suitable code pointer, such as stack return address, for example.

```
1    void foo(char *arg) {
2      char littlebuf[128];
3      ...
4      strcpy(littlebuf, arg);
5      return;
6    }
```

Figure 3.3: A typical stack-based buffer overflow vulnerability

Obviously a lot of sophisticated exploitation techniques exist, but for exposition purpose we consider only the simplest ones. Figures 3.3 and 3.4 depict code snippets showing respectively a stack-based buffer overflow vulnerability and a

[9] Again, as noted in § 2 and 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. Moreover, we are not considering any interpreted language.

```
1   u_char *read_next_cmd(void) {
2
3     u_char input_buf[64], *p;
4     u_char *e = getenv("USERCMD"), *q = &input_buf[0];
5
6     umask(2);
7     ...
8     while (*q++ = *e++) ;
9     /* memory leak? */
10    p = (char *)strdup(input_buf);
11    return p;
12  }
13
14  void login_user(int uid) {
15
16    char *cmd;
17
18
```

```
18    if (is_regular(uid)) {
19
20      /* unprivileged mode */
21      cmd = read_next_cmd();
22      setuid(uid);
23      /* yes, system is safe ;-) */
24      system(cmd);
25
26    }
27    else {
28
29      /* superuser! */
30      cmd = read_next_cmd();
31      setuid(0);
32      system(cmd);
33
34    }
35    return;
36  }
37
```

Figure 3.4: A typical security check that can be bypassed with an IPE attack.

security check that can be bypassed by performing an IPE[10]. In particular, Figure 3.4 depicts a situation where an attacker, camouflaged as a regular user, enters the true branch (lines 19-25) and exploits the stack-based buffer overflow (line 8) by overwriting the LSB of `read_next_cmd` return address. Once the function ends, the execution flow will return into the false branch (lines 28-33) ending up by running `cmd` as a privileged user, thus performing an IPE attack.

On the other hand, Figure 3.3 shows how the control-flow can be diverted by overwriting `foo` return address, pointing back into the vulnerable buffer itself which contains the malicious injected code.

Such attacks can be defeated by the process replication with address space diversification mechanism, no matter if they target absolute or partial address overwriting, as long as the address space is properly diversified with the approaches proposed in § 3.2.2. For example, consider the code snipped reported in Figure 3.3 and the stack layout of the process P associated to such a code and its replica P_r at the time the stack-based buffer overflow vulnerability is exploited, as reported in Figure 3.5. If the attacker were able to exploit the stack-based buffer overflow vulnerability, P and P_r would exhibit a different behavior. In fact, P_r will eventually reference an unmapped memory region in its address space and thus, it will be killed, along with P, by the replicator and monitor component t (or viceversa, that is P_r gets exploited and P is killed). The same holds for the IPE attacks described above.

3.4.2 Experimental Results

We conducted some experimental tests in order to evaluate the impact of the process replication with diversification approach herein described. To this end, a user-space `ptrace` proof of concept (PoC) which we developed, has been executed on a 1.3Ghz Intel Centrino with 512MB of RAM, running a Debian GNU/Linux with a 2.6 vanilla kernel. The PoC is in charge of correctly replicating and

[10]Example showed in Figure 3.4 was first proposed by [28] and slightly modified in [16].

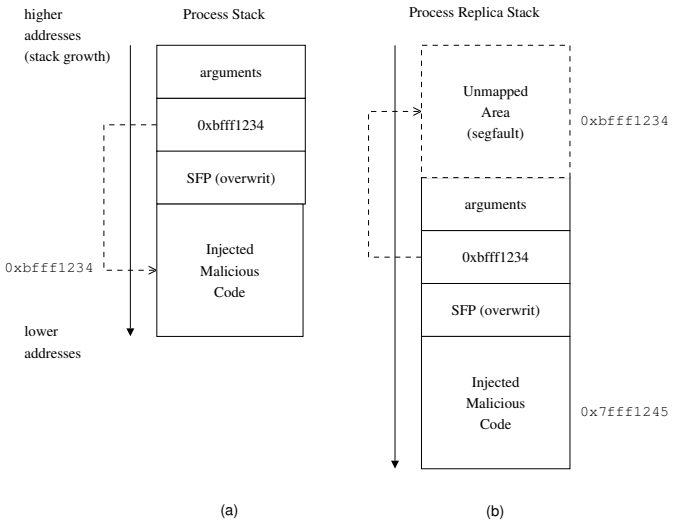

Figure 3.5: Diversified process replica for defeating *absolute* memory errors exploits

monitoring `thttpd` [42], a small and fast web server, as described throughout the Chapter. Moreover, `httperf` [17], an HTTP benchmark utility, has been used on three client hosts to assess the latency and throughput slowdown on an unsaturated 100Mbps LAN using 100 connections, 4 sessions per connection, 13 requests per connection, on a 7.5MB site. The last test case (#5), instead, was conducted using 10 connections on a 98MB site[11].

#	Throughput	MB/s (real system)	MB/s (DPR)	% slowdown
1	`thttpd` (mmap)	12386.9	12238.8	1.20%
2	`thttpd` (mmap-nocache)	12718.4	12496.5	1.75%
3	`thttpd` (read)	12599.5	12117.4	$\sim$ 3.8%
4	`thttpd` (read-nocache)	12603.7	7086.3	$\sim$ 43.8%
5	`thttpd` (read-nocache-single)	9134.5	2838.1	$\sim$ 69%

Table 3.1: Experimental results: Throughput.

Table 3.1 and 3.2 summarize the experimental results we achieved. In particular, we were quite surprised by the 1.20% throughput slowdown since, it was our belief that, due to the nature of the idea and of the PoC implementation, a more heavy performance impact and network slowdown (mainly caused by the need to simulate some system calls, such as the `read`) were expected. It is worth noting, in fact, that one of the more heavy system call the proof-of-concept must simulate is the `read` system call (as other similar input-related system calls, such as `readv`, `recv`, `recvfrom`, ...) since, as pointed out in § 3.3, it has to replicate

[11]The *real system* column shown in tables 3.1 and 3.2 refers to the execution of the original application `thttpd` *without* its replica but under `ptrace` monitoring to avoid including any overhead introduced by a simple system call tracing mechanism as the one brought by the `ptrace` system call. Of course, the column flagged as *DPR* refers to the approach proposed in this dissertation.

#	Latency	ms (real system)	ms (DPR)	slowdown
1	`thttpd` (mmap)	3.5	4.6	31%
2	`thttpd` (mmap-nocache)	3.5	4.5	29%
3	`thttpd` (read)	3.5	5.3	51%
4	`thttpd` (read-nocache)	3.7	21.6	$\sim 6x$
5	`thttpd` (read-nocache-single)	166	646	$\sim 4x$

Table 3.2: Experimental results: Latency.

data from one process to its replica, without actually letting the replica to execute the system call. However, further investigation on the testbed web server showed that, by default, `thttpd` uses the `mmap` system call, where available, in order to map VFS objects into the process address space, by avoiding any use of the slow `read` system call as much as possible and demanding to the kernel the loading of the remaining VFS object onto the process address space when needed. Moreover, the web server uses a cache system to avoid duplicate mapping or reading of VFS objects. This helped to give initial acceptable performances downgrade both for latency (31%) and for throughput (1.20%).

However, in order to be as much complete as possible and to better assess the introduced cost caused by the replication approach, we modified `thttpd` to force it to either use any combination of `mmap` and (simulated) `read` syscall with caching facility enabled or not. Table 3.1 and 3.2 report the combination we obtained and, as we expected, more realistic overheads were reported. Latency ranged from $4.5ms$ (31%) to $21.6ms$ ($\sim 6x$) on the 7.5MB web site, for non caching read operations, while it reached $646ms$ ($\sim 4x$) for the 98MB testbed web site. Likewise, we reported a throughput slowdown of $\sim 44\%$ to $\sim 69\%$ for non caching read operations on a 7.5MB and 98MB web site, respectively.

It is worth noting that the high overhead is mainly introduced by the replication task *and* by the underlying prototype implementation. In fact, every 4 bytes of data that need to be replicated, *at least* two `ptrace` system calls (and process context switch as well) have to be executed, as noted at the end of § 3.4.3. This is particularly onerous. A different implementation which does not make use of `ptrace` would certainly improve performances. Moreover, as it will be pointed out in § 6, parallel execution of P and P_r would further improve performances.

As a final note, it is worth noting that the overhead introduced by the `read` syscall simulation may be decreased if we were able to distinguish whether a read operation is performed on a regular VFS object file or from a socket or standard input, for example. In the former case, in fact, there is no reason to simulate the syscall at all, while in the latter case such a simulation is a must in order to guarantee for the correct processes behavior. Such an optimization would give better throughput on "download" operations (from a client perspective) while, unfortunately, would be practically useless on "upload" ones. Further speculations are given in § 6.

3.4.3 Discussion

The approach herein proposed provides a *deterministic* protection when a memory error exploit corrupts a 32-bit *code* or *data* pointer (absolute overwrite).

Generally speaking, to successfully exploit a memory error vulnerability, it is necessary that (i) a vulnerability allows for the corruption of some security critical data (e.g., code and data pointers in the considered scenario), and (ii) that these data are overwritten with known attack-provided values. Our replication approach correctly feeds both P and P_r with the same input. As P and P_r have disjoint address spaces (ASs), i.e., they do not share *any* address in common, thanks to the diversification strategy adopted by our approach, a memory error exploit which corrupts the aforementioned security critical pointers, will be valid on one address space (e.g., P), while will cause a segmentation violation on the other one (e.g., P_r), whenever the corrupted pointer is de-referenced due to references to *unmapped* portions of the involved process AS. This anomalous event is caught by the tracer T which terminates both processes, *before* any harmful activity can be performed. In fact, as it is clear that one process has no chance to execute even one instruction, the other one has. To better understand why no harmful actions can be performed anyway, let us consider the following scenario. Let us suppose that P is the process where the attack-provided address overwrites a code pointer with the intent to execute arbitrary code by referencing to a mapped page of P AS. On the other hand, the same address will cause P_r to crash as soon as the corrupted code pointer is de-referenced. On uni-processor machines, scheduling mechanisms might let P to execute *some* instructions related to the attack-provided input. However, this does not represent an issue as to cause harm to the system, eventually a system call has to be invoked. We remind that the tracer T waits for both P and P_r to reach the rendez-vouz point. Anyway, P might reach it (attack-controlled) while P_r will never. No matter on the (small) number of instructions executed by P due to scheduling mechanisms, eventually, P_r will generate a segmentation violation (code pointer corrupted with a wrong unmapped address), which in turn will cause the termination of itself and P as well. A similar but simpler reasoning can be made if a data pointer is corrupted as no attack-induced code can be executed at all, neither on P nor on P_r.

Things change a little if partial overwrites are considered. The underlying reasoning is similar, even if the protection provided by the address space *shifting* strategy proposed provides only a *probabilistic* protection. Let us suppose that P and P_r address spaces are disjoint and that P_r AS is shifted by k bytes with respect to the one of P. The consequence is that, for instance, a code pointer c of P points to an address whose least significant byte (LSB) is x, while the same pointer for P_r points to an address whose LSB is $x + k$. Of course, Both code pointers refers to the same object. Let us consider an attack scenario like the one proposed before. For instance, let us consider an attacker that wants to partially overwrite a code pointer to perform an IPE attack and skip security-

relevant checks (e.g., authentication check). Let us say that by replacing the LSB of c in P from x to y will allow the attacker to fulfill this goal. As the input is replicated by our approach, also P_r code pointer will have its LSB replaced by y. However, while this is the correct "offset" for P, it is not for P_r as its AS is shifted by k bytes. Of course, the correct value for P_r would be $y + k$ but then this would be incorrect for P. Most likely P_r will jump in the middle of an instruction, and it will either generate a segmentation fault due to illegal instruction decoding (`SIGILL` or `SIGBUS`). Alternatively, it will start executing some instructions which are not equivalent to the one executed by P as P_r will be "behind" P as it jumped k bytes less than the one expected. The processes most likely exhibit behavioral divergence and the tracer T will terminate both either upon receiving a segmentation fault-like signal or whenever they try to execute different system call (or the same acting on different parameters).

Unfortunately, as briefly discussed previously, the proposed approach does not provide protection when arbitrary non-pointer data are corrupted, as the example here below proposed by Mutz *et al.* in [56] depicts.

```
1  void write_user_data(void) {
2
3     FILE * fp ;
4     char user_filename[256];
5     char user_data[256];
6
7     gets(user_filename);
8
9     if (privileged_file(user_filename)) {
10      fprintf(stderr, "Illegal filename. Exiting.\n");
11      exit(1);
12    }
13    else {
14      gets(user_data); // overflow
15      fp = fopen(user_filename, "w");
16      if (fp) {
17        fprintf(fp, "%s", user_data);
18        fclose(fp);
19      }
20    }
21 }
22
23
24
```

The `user_filename` array obtained at line 7 (`gets` function) is subjected to a security check performed by the function `privileged_file` (line 9) that checks whether `user_filename` specifies a name of a privileged file or not. In affirmative case, the program prints an error message and quits. Otherwise (i.e., non privileged file), more data is read into the array `user_data`, through the function `gets` at line 14, and the file name specified by `user_filename` is opened at line 15. An attacker can overflow `user_data` by overwriting past its end, and overflowing into `user_filename`. As the overflow happens after the security check performed at line 9, an attacker can specify a legal file name for `user_filename` that will be replaced later on thanks to the overflow.

A limitation of our strategy is that it cannot defeat the aforementioned attack as no pointers corruption are involved and the considered vulnerability is a straight buffer overflow which corrupt adjacent buffers (*relative addressing* attacks). On the contrary, this would be possible if the vulnerability would make use of pointers to achieve its final goal. For instance, this is the case for a format bug which aims to corrupt a non-pointer security sensitive data if the format bug uses attack-provided addresses (as in most of the cases), as shown by the example in § 4.3.1.

An inherent drawback of the proposed solution is that replicating *legal* pointers values (i.e., non attack-provided input) from P to P_r or the other way around, is inherently seen as a manifestation of an attack and both processes are terminated (one will cause a segmentation fault which in turn will cause the termination of the other). Moreover, the prototype implementation uses the `ptrace` system call. Beside being an easy-to-use but slow approach for replicating information from one process address space to another (e.g., every replication of 4 bytes requires the execution of at least 2 `ptrace` system calls), it is not considered to be secure to provide local protection as well [32]. We believe this is not really a big concern, indeed. A different prototype implementation would simply do.

3.5 Practical Issues

Unfortunately, even if the idea of diversified process replication is simple and quite effective in combating a broad range of memory error exploits, there are some practical issues, namely shared memory, signals and non-determinism (e.g., threads) situations, that we have to cope with in order to successfully and broadly deploy such a defensive mechanism.

3.5.1 Shared Memory

Shared memory management is probably one of the biggest practical issue introduced by diversified processes replicæ.

In fact, as already pointed out in § 3.3, P and P_r have to synchronize themselves at each system call (rendez-vouz point) to let T to correctly perform the replication task. However, no system calls are invoked when shared memory is involved. It might not so clear at first glance where and how to achieve such a rendez-vouz point for synchronization. Moreover, it might also be unclear how to deal with a shared resource R in order to guarantee consistency between P and P_r behavior and R. In fact, as we will briefly see in § 3.5.1, it is fairly easy to make examples on how things can go wrong between P, P_r (behavioral divergence) and the involved resource R (data inconsistency).

For the sake of clarity and for explanation purpose, we would briefly remind how shared memory to achieve inter-process communication (IPC) is obtained and what resources are actually involved in the process. Depending on the needs, in fact, we may obtain shared memory either by means of `mmap` system call or by means of classical shared memory IPC form (`shmget`, `shmat`, ...)[12].

The main difference between the two approaches is that the former one provides shared memory by acting on a file system (FS) object O which, once mapped onto a process P address space (AS), will be shared to provide inter-process

[12]It is worth noting that it might happen that, on certain UNIX systems, IPC shared memory is obtained using the `mmap` system call. As we will see shortly, this does not interfere with our treatment.

communication. We can talk, in this case, of *non-anonymous* (shared memory) mapping.

On the contrary, the classical shared memory approach makes directly use of a memory area that will be shared among the processes that will attach to it. We can talk, here, of *anonymous* (shared memory) mapping. Without loss of generality, we will use the general term *shared memory* to refer to both approaches by default, unless differently stated, no matter if the resource being shared is a memory area or a FS object O. The main point, in fact, is that whenever a FS object is shared with such an approach, it is transparently accessed and modified, with the help of the underlying OS, without any I/O operation but only through memory accesses the process mapping O makes use of. Moreover, we will use the terms shared resource R, shared mapping, and shared memory interchangeably, unless differently stated.

It is worth noting that, however, not all the features provided by the aforementioned approaches are dangerous in our framework as well as in the model proposed by [7]. In the following we summarize how it is possible to obtain shared memory and whether the particular "type" of shared memory is suitable for inter-process communication (problematic case) or not.

`mmap`-based: can provide both anonymous (memory area) and non-anonymous mapping (FS object mapped onto a process address space).

1. non-anonymous can be further divided in:
   (a) *private mapping*, that creates a *private copy-on-write* mapping. It only provides what we call *intra-process communication*. That is, the resource R is shared only among parent/children relationship which only modify the memory associated with R in their AS; there is no modification of R at all. It is worth noting that, as long as the private mapping is not modified (copy-on-write), every modification of R made through a shared mapping (see next) is reflected into the private mapping as well;
   (b) *shared mapping*, that provides true *inter-process communication* among the processes mapping R; for this reason R can potentially be modified. Moreover, every modification performed on R is automatically reflected into the AS of the processes which map R.
2. anonymous that provides intra-process communication; the mapping is private and belongs to the process P's AS and its children, if any.

classical shared memory: can only provide anonymous (memory area) mapping. As above, it can be further divided in:

(a) *private mapping*, similar to the *intra-process communication* mapping provided by the `mmap`-based approach (point 2), with the exception

that the mapping is completely private and not on mapping modification;

(b) *shared mapping*, that, as in the `mmap`-based approach (point 1b), shares the resource R providing *inter-process communication* among the involved processes.

As we will soon describe in § 3.5.1, it is easy to see that the only problematic situations are (i) when a resource is actually shared, like in the `mmap`-based approach (point 1b), and (ii) in the classical shared memory (point b) approach.

We try to cope with the shared memory management issue with a step-by-step approach. We start with a simple scenario where *related-only* processes are involved (best case scenario easy to cope with). Next we move on a more tricky and realistic scenario when unrelated processes are involved (worst case scenario), to put the basis for a generic solution at the end of the Section.

By related-only processes, we mean a scenario where no external processes, beside P, P_r and their children (if any), are present. Synchronization between P and its children for accessing a shared resource R has to be properly done and it is not a side-effect introduced by our model.

We can anticipate that the main issue is that both P and P_r would end up by acting on the same shared resource R and this might cause inconsistency if not properly handled. The example described in § 3.5.1 shows such a situation (`mmap`-based (point 1b) approach) in a related-only processes scenario with no children (only P and P_r).

Data Inconsistency and Behavioral Divergence

The following example clarifies the main issue related to the management of shared memory regions in the process replication model. Even if the example is focused on a best-case scenario we show how it is easy to get data inconsistency and behavioral divergence between P and P_r.

Suppose that P creates a readable and writable[13] (`PROT_READ|PROT_WRITE`) non-anonymous shared memory segment (`MAP_SHARED`), that is a memory segment that maps a FS object O, via the `mmap` system call. Since both P and P_r are fed by the same input, also P_r will end up by creating the shared memory segment as well. In the following, we show a code snippet which P and P_r could execute following a different execution flow, thus exhibiting a different divergent behavior. As a direct consequence, O will be shared between P and P_r as well. This can be considered as the main *cause* of the issue, that is, P and P_r will start having an unwanted form of *inter-process communication*.

The consequences are that every modification made by P on the shared memory segment mapping O, will automatically be reflected onto P_r address space as well as into O itself (see § 3.5.1). As previously noted, if not properly handled this

[13]Note that read-only shared memory is not an issue. We will not further elaborate on this point here.

could lead to *data inconsistency* and *processes behavioral divergence*. Obviously, this is something to avoid as could be seen as false positive of the model that would bring the system in a stalled situation (termination) with a even worst side-effect of data corruption.

1. let `ptr` points to the `mmap`'d shared memory segment and suppose the first byte of O contains the value `A`. Suppose both P and P_r are ready to execute line `1` in the following code snippet (so, they have already been "scheduled" by R but they are waiting for being scheduled by the kernel).

```
1       if (*ptr == 'A')
2          *ptr = 'B';
3       else
4          *ptr = 'C';
5       ...
6       /*
7        * execute something based
8        * on the value held by *ptr
9       */
```

   Suppose the kernel schedules-in P[14]. As can be observed, since there are no system calls involved, there are also no rendez-vouz points; moreover, suppose that P executes the *true* branch, setting the byte pointed by `ptr` to the value `B`, before its quantum expires;

2. afterwards, let P be scheduled-out by the kernel scheduler which eventually schedules in P_r that starts its execution at line `1`; since `*ptr` has been changed by P and `ptr` points to a non-anonymous writable shared memory segment, P_r will enter the *false* branch, setting the byte pointed by `ptr` to the value `C`;

3. but since P_r is just a P's replica, it *must* exhibit the same behavior exhibited by P as long as both processes are fed by the same "good" input by R. This example shows a subtle way to feed P and P_r with different inputs. In fact, P thinks `*ptr` holds `A` while P_r not and such a situation might modify their behavior if further decisions are going to be taken based on the value stored in `*ptr`. Moreover, O might end up in an inconsistent status.

Related-only Processes

In this scenario we consider only P and P_r but no other external processes that might operate on the shared resource R. As highlighted in § 3.5.1 both P and P_r will act on the same shared resource R. The main issue is that they were not even suppose to share R between each other, starting, in this way, a form of *inter-process communication* between them as a direct consequence.

[14]Indeed, as noted elsewhere (§ 3.3), T is able to somehow control the scheduling of P and P_r by interacting with the kernel using the `ptrace` system call, but only from an high-level point. Actually, the kernel is in charge of performing the real process scheduling task and all the processes, even P, P_r and R, are involved.

Given this observation, the solution would seem to be trivial: to turn P_r inter-process sharing into an *intra-process* one so that P would not interfere with P_r behavior and viceversa, and R's data will be consistent with that they were supposed to be. Unfortunately, as pointed out in the previous section, an intra-process communication (private mapping) creates a *private* copy-on-write mapping. As long as the private mapping is not modified, it sees every modification made via a shared mapping, to the object being mapped. In this scenario, one process, P will create a *shared* mapping, while P_r will create a *private* one, falling in the situation just described. Thus, it is not sufficient to let P_r perform a *private* mapping, even in this simple scenario of related-only processes. In fact, due to scheduling policies, it could happen that the view of R is not always consistent. Moreover, for the same reasons, it could happen that P and P_r will exhibit an inconsistent behavior. Instead, in this scenario we can act in three different ways:

1. Allow P to create a *shared* mapping, while let P_r to create a *private* one. As we saw, the default behavior of "sharing and private" provided by the `mmap` system call, would not give us the behavior we are looking for (e.g., no IPC between P and P_r), as, even if P_r mapping is not shared (i.e., no changes made by P_r will be visible on O and in P mapping), the contrary is not true (i.e., P changes will be visible both on O and P_r mapping – therefore keeping a form of IPC between a process and its replica).

   Thus, we need to change the behavior of `mmap` when such mappings, created by a process and its replica, are involved. This can be achieved by modifying the underlying kernel, which represents the main downside of this approach.

2. Allow P and P_r to create the mapping they want to (e.g., shared) but redirect P_r to work on O_r, a replica of the FS object O mapped by P (selective on-demand FS replication).

   The downside of this approach is the necessity to replicate O. If O is too big, the replication would incur high overhead and would waste lot of disk space.

3. No *shared* mappings are allowed, neither for P nor for P_r. This has the advantage that no further handling is required, as P and P_r will work on their private copy of the memory mapped FS object O. The downside is that O will never be updated as all the changes are made on the memory which maps O. Therefore, if the processes are terminated and re-executed later on, they will start working on an old stale version of O (as all the changes made in the previous run in memory changes would be lost).

   A solution to this issue is to check when the mapped area $\mathcal{M}$ is unmapped and let T to flush the content of $\mathcal{M}$ into O (respecting mapping offsets, if any). T also checks that $\mathcal{M}$ of P and P_r are identical (i.e., consistent). If this is not true it means that there is a conflict as P and P_r have a

different view of the mapped object O. Therefore, the processes are killed and O is not updated at all. On the other hand, if the mapped regions $\mathcal{M}$ are consistent, `munmap` or `msync` system call are executed once for P, while `munmap` operation invoked by P_r would simply map O out of P_r address space (`msync` would behave similarly to a no-op system call – respecting the replication mechanism described in § 3.3). This has the advantage of not requiring any additional handling beside $\mathcal{M}$ flushing at `munmap` time (e.g., no selective FS object replication). This is the approach adopted by our prototype.

Things are more tricky when unrelated processes are to be considered. It is worth noting that whenever P_r start writing on a private mapping, the kernel disassociate the mapping with the file object. This is not an issue because the simple assumption we are claiming here is that no other external processes are working on the mapped object O. For this reason, P, P_r and their children, starting from an identical version (consistent) of O and executing the same operation (exhibiting the same behavior) in a deterministic way, produce the same output on O (consistency).

Unrelated Processes

This scenario is more tricky to deal with because, beside P and P_r, there are even unrelated processes which we do not have the control of and that want to interact with the shared resource R. The `mmap` semantic helps us out to provide up-to-dated version or R *as long as* the process which created the private mapping does not write into it.

Therefore, it is necessary (i) to find a way to allow P and P_r to execute instructions which operate on their mappings in an interleaved manner, and (ii) to ensure that no process (no matter if it is P, P_r, or E) will operate on R leaving it in an inconsistent state. If this is achieved, we only need to *replicate* the semantic of every write operation performed by P onto P_r address space, paying attention to update the shared resource R (e.g., the file system object) as well. It is fairly easy to guarantee the second requirement, as long as the involved processes (e.g., P, P_r, and E) use *synchronization mechanisms* whenever they attempt to access a shared resource R. We believe that this is not a strict requirement because without this assumption poorly written programs that make use of shared resources are going to break soon, even without any malicious intent by an adversary (it is a matter of processes/threads scheduling most of the time, which is, generally unpredictable or so)[15].

To achieve our goal, we propose an approach similar to *Fault Interpretation* [20]. The idea is simple: we exploit the CPU page fault (PF) exception

[15] "[...] What is normally required [when using shared memory], however, is some form of synchronization between the processes that are storing and fetching information to and from the shared memory region" [73].

to know whenever P or P_r attempts to access a given memory page(s) $\mathcal{M}$[16] of its own which refers to the shared resource O. To achieve this goal, we mark $\mathcal{M}$ of both P and P_r with no permissions. This task can be done by T which intercepts P and P_r system calls (see § 3.3), whenever the mapping is created. Alternatively, this task can be performed by T after the mapping has been created by injecting the code to invoke `mprotect` with the right parameters. This provides control over the execution of the instructions accessing $\mathcal{M}$, as every execution of those instructions will generate a PF exception which is caught by the tracer T. Let us see how this mechanism can be used with a proper mapping strategy to achieve our goal (i.e., data and processes behavioral consistency).

We let P to create a *shared* mapping, and P_r to create a *private* one. E is the external unrelated process which creates a *shared* mapping as well. As we have seen in the previous scenario, this means that as long as P_r does not write into its private mapping, no modification are reflected onto neither the mapped object O nor the existing (memory) mappings of O. Of course, our goal is to avoid that P_r writes into its private mapping, otherwise the copy on write (COW) mechanism will disassociate P_r mapping from the other shared ones. This would have the side effect to not show an up-to-dated version of O to P_r anymore. As we will briefly see, this is easy to achieve, by exploiting the fault interpretation mechanism described above.

When E writes into the shared memory mapping, this is reflected both into P and P_r address space (`mmap` semantic). Thanks to the assumption, E operates on the shared mapping only when it has acquired the lock. Therefore, anytime E operates on $\mathcal{M}$ it implicitly *updates* the mapping for P and P_r, providing them always with an up-to-dated version of O. Let us see now what happens when P tries to write into $\mathcal{M}$ (similar reasoning about the locking holds here as well). Whenever P tries to write on the shared mapping, it generates a PF (no permission on the mapping). T waits until the same happens to P_r, therefore defining a "new" rendez-vouz point (of course, the contrary is valid as well). At this point, if the instruction tries to *write* into $\mathcal{M}$, T executes the culprit instruction once (`stepi`) for P so that the instruction outcome can be propagated to every existing mappings and to O as well. Then, T needs only to replicate the instruction outcome/side-effects (e.g., `%eflags` updating) onto P_r address space. If P attempts to *read* from $\mathcal{M}$, then T just needs to execute the instructions on both P and P_r (after they reached the rendez-vouz point), so that they both can have the outcome of the executed instruction in their address spaces. In practice, any write action performed by P is executed once and replicated for P_r, while any write action attempted by P_r is ignored, as its outcome will be provided by P execution. To make a similarity, this is similar to the handling of a `read` system call, for instance.

[16] Whenever needed, we will use $\mathcal{M}'$ and $\mathcal{M}''$ to refer to P's and P_r's shared mapping respectively.

3.5.2 Signals and Non-Determinism

Unfortunately, shared memory does not represent the only critical issue that may arise due to the replication approach. Indeed, also signals handling and non-determinism should be analyzed, in order to guarantee a correct behavior of the process replication approach.

However, we believe that even if it is quite impossible for T to deliver to both P and P_r the same signal, which is asynchronous by nature, at the same time and at the same "point" (location and context), such a "delay" should not create significant differences in the behavior of P and P_r. This because both P and P_r have to reach their rendez-vouz point before the execution of every invoked syscall and, as already observed, this is guaranteed and carried out by T (see § 3.3).

Actually, since T catches every signals sent to P and P_r, it could delay the signal delivery a little bit and it can arrange the thing to fire up the received signal at each rendez-vouz point, thus achieving perfect synchronization with respect to signal delivering. The main problem with this approach is that, however, intensive CPU bound processes that make few system call could probably not benefit from this delayed action, but even in this case, the signal should be delivered at a given time chosen by T anyway.

Of course, things will break if, for instance, the state of a variable used in a system call depends on when the signal is delivered, or if there is a loop (containing no system calls) that needs to be interrupted. Unfortunately, the former case leaves nothing or little to be done. The latter case, instead, could be addressed by correctly identifying loops beforehand, and by transforming the application in order to insert dummies system calls inside the loop. This way, P and P_r will be forced to regularly synchronize themselves and that rendez-vouz would be the best one to deliver the caught signal.

Alternatively, when necessary, as shown in previous works ([34, 80]), we can leverage on CPU specific counters (`branch_retired`) and on the adopted diversification approach (§ 3.2.2) to turn an asynchronous event like a signal delivery to a synchronous one, even if absence of rendez-vouz points.

We also believe that, non-determinism situation should not generally pose a problem. In fact, since P_r is fed by the same input of P, it *must* behave identically to P, unless, as observed throughout the paper, the input received is a malicious one. Randomness should not be problematic since we believe that such data have to be collected *generally* via some sort of system calls[17]. Thus, as long as P input is correctly replicated into P_r address space, both processes will exhibit the same behavior unless relative-address data are involved. The situation could be different if threads are considered, as the outcome of an instruction executed on threads shared memory can be cause data inconsistency and behavioral divergence (even if this time the inconsistency and divergence will be inside the address

[17] An exception to this could be a random memory access used to provide a seed to initialized a pseudo-random number generator.

space of a given process). A preliminary solution could be to control the threads' scheduling mechanism so that a thread L of a process P would be scheduled in the "same order" (one after each other on UP machines) as L_r, the thread of its process replica P_r. Scheduling ordering might not be sufficient as it is necessary to understand whether L is going to operate on some shared resources because it acquired a lock, while L_r not because it has been scheduled out before acquiring the lock. Clearly, even this situation is problematic and, a naive solution, beside dealing with threads scheduling mechanisms, should consider also threads quanta (or number of instructions executed by the threads). For instance, L and L_r should be scheduled one after each other (on UP machines) or simultaneously (on SMP machines), and they should occupy the CPU for the same amount of time or number of instruction executed.

Prediction is difficult, especially about the future.

Niels Henrik David Bohr (1885 - 1962)

4 Taint-enhanced Anomaly Detection

Memory errors have been known for decades. Most likely, the first known public memory error exploitation can be traced out back to 1988, with the Morris Worm [72]. Since then, several researchers have been working on providing more or less comprehensive memory error protection mechanisms. Proposed solutions cover a broad range of Computer Science disciplines, going from safe programming languages [43, 58], anomaly detection [87, 85, 28, 39, 70, 56, 54, 9], and information-flow [47, 13, 59, 89, 60], to techniques that modify the underlying compiler [15], system libraries [88, 82], the operating system, or the hardware [62, 79, 31].

As we have shown in the previous chapter, techniques which aim to introduce artificial diversity to combat the software monoculture that is predominant nowadays, seem to be promising and effective. As we already noted, as protection mechanisms improve, so do the attacks. Recently, Chen *et al.* [14] pointed out that it is no longer necessary to exploit a memory error vulnerability with the goal to hijack a legal process' control-flow to cause harm. In fact, damage can be caused not only by corrupting code pointers, but also by tampering with application's data and data pointers as well. Even if this seems to be a strict restriction, Chen *et al.* showed that these attacks can be as powerful as the classic ones.

The diversified process replicæ approach described in the previous chapter represents a step toward providing a more comprehensive protection against memory error attacks which corrupt code and data pointers, by mostly providing a *deterministic* protection. Unfortunately, as we will see in the following sections and, as shown in [14], memory error attacks target *arbitrary* non-pointer data as well. In this situation, the approach previously describe fails to provide protection as this, usually, requires a better understanding of the internal application logic. To this end, we propose an approach which couples taint analysis and anomaly detection, to provide a more comprehensive protection against memory errors. In particular, we make the following contributions:

1. We propose *taint-enhanced anomaly detection*, a technique which couples taint analysis and anomaly detection. To the best of our knowledge, no

existing similar techniques have been proposed so far in the context of protection of benign applications. Anomaly detection or, better, learning-based approaches help to automatically infer security policies, as already shown by existing techniques [87, 85, 28, 39, 70, 56, 54, 9]. Unfortunately, these techniques have two major drawbacks. They (i) often exhibit high false positive rates issues as learning phases are hard to be exhaustive, and (ii) they are vulnerable to mimicry attacks [86, 85, 48, 78, 77] as attack-provided data can often stick to statistical learning-rules used to characterize the process behavior. By using taint analysis, we constraint these drawbacks as (i) unknown *untainted* traces seen during detection are *no more* considered as manifestations of attacks, and (ii) by largely constraining execution of foreign code and by enhancing learning rules with taint information to infer taint-enhanced security policies, mimicry-like attacks – even if still possible – are considerably constrained. More precisely, an attack involves a combination of a vulnerability, and an attackers ability to exercise this vulnerability. Anomaly detection techniques detect behavioral deviations that occur when a vulnerability (targeted by an attack) is exercised. Fine-grained taint information, instead, can provide information about the ability of the attacker to exercise this vulnerability, significantly increasing the odds that an attack is in progress.

2. Comparison with state of the art models shows that our approach is at worst as much as effective as the combination of all those work, for the class of attacks considered (i.e., memory errors). Moreover, false positives (FPs) one of the main drawbacks of anomaly based approaches, are considerably reduced.

3. We developed a prototype implementation of the proposed approach which transforms a program P to a taint-enhanced version P_T, semantically equivalent to the original one. Then, by leveraging on taint information, P_T is further enhanced to perform (i) a training phase where properties of *tainted* sinks' arguments, that is, relevant events (e.g., system calls or security sensitive functions) are learnt and modeled to generate a behavioral profile $\mathcal{M}$ of P_T, and (ii) a detection phase during which single events of P_T are observed at run-time and checked one at a time to see whether they are consistent to the learnt behavioral profile $\mathcal{M}$. Should $\mathcal{M}$ be inconsistent with respect to these traces, an alarm will be raised.

4.1 Preliminaries

For clarity, in the following we briefly remind the main concepts behind taint analysis and anomaly-based detection approaches. We then describe how our technique works by taking the advantages of these approaches while constraining their limitations and drawbacks.

4.1.1 Taint Analysis

Information-flow based techniques have been studied for decades [6, 29, 21, 51, 84, 57, 66]. Recently, these techniques, sometime known as taint analysis, have shown to be successful in thwarting memory error exploits [89, 59], and a broad range of software attacks [89, 60].

Taint analysis is a technique which aims to detect whether *untrusted* data is incorrectly used in security sensitive actions. To this end, taint analysis usually marks untrusted data coming from *taint sources* (e.g., `read`, `recv`, or other input-related system calls) as being tainted, and, as data propagates through memory, it propagates the taint information associated with the data itself. Any attempt to use security sensitive data which has been marked as tainted at security sensitive *sink*[1] (e.g., `open`, `write`, `printf`-like, code pointers dereference, or system call and functions of interest), is generally a manifestation of an attack. For instance, a memory error exploit which aims to corrupt code pointers, can overwrite a function return address with the intent to hijack the legal process execution flow. Naturally, the overwritten return address will be marked as tainted as a consequence of this attack attempt. The low-level implicit policy adopted by taint-based approaches does not allow to dereference this code pointers, as functions return addresses are security sensitive data which are never marked as tainted.

Taint information can be propagated in different ways. It is possible to propagate taint information based on data dependency, or based on direct control dependency. Implicit flows [21, 66, 38], which are a generalization of direct control dependencies, are generally not considered as benign programs usually offer little bandwidth to permit successful implicit flow exploitation [89].

Without going into much detail, data dependency taint propagation occurs whenever there is a direct data assignment, such as $x = expr$, where $expr$ has been previously marked as tainted.

On the other hand, direct control dependency occurs whenever the execution of an operation depends on the result of a tainted condition, as the example below depicts.

```
1    if (x == expr1)
2       y = expr2;
```

Here, y has to be marked as tainted only if the condition ($x = expr_1$) that guards the true branch ($y = expr_2$) has been marked tainted. The condition is marked tainted only if x or $expr_1$ has been marked tainted beforehand. The instructions which are guarded by the tainted condition form what in literature has been called a tainted scope.

[1] Taint analysis terminology considers *sink* to be a security sensitive function or system call of interest where a security policy should be enforced. Likewise, through the rest of this dissertation, we will use the term sink, system call, or function interchangeably, unless differently noted.

Tracking of control dependencies is easy to do, even in binaries, by associating a *taint label*[2] with the *program counter* ([89, 23]). Whenever the condition involved in a branch decision is *tainted*, the program counter is also tainted. An assignment causes the target variable to be tainted if the program counter is tainted, or if its right-hand side expression is tainted. The label of the program counter is restored at the merge point following a conditional branch.

We could avoid performing direct control-data dependency taint propagation but sometimes this is useful to catch application-specific taint data transformation (e.g., '+' → ' ') and look-up tables. However, as pointed out in [18, 13] data pointer taintedness and direct control-dependencies tracking should be enabled to improve detection capabilities (a good discussion on the utility of look-up tables is given in [18]).

4.1.2 Anomaly Detection

Broadly speaking, anomaly detection approaches build a behavioral model $\mathcal{M}$ of a monitored application P, during a *training* phase where some P behavioral properties are learnt.

Several anomaly-based models have been proposed in the past years [87, 85, 28, 39, 70, 56, 54, 9] and each of them learns different properties out of a given event of interest (e.g., system call). Some of them learn the sequence of system calls executed by a process [39], while others consider the calling context as well by representing the process control flow graph as a finite state automata (FSA) [70]. Others again, exploit call stack information to provide a snapshot of the function flow graph the process invoked to check whether an invoked system call complies with a call stack configuration learnt during a training phase [28]. Statistical properties of system calls arguments are considered by other models [56, 54], as well as data flow relationship between system calls arguments [9].

Generally speaking, the main goal of these approaches is to keep *learning* behavioral properties of a monitored process P with the intent to build a profile $\mathcal{M}$ of P. Then, when $\mathcal{M}$ is meaningful enough to represent an approximation of the overall P's behavior, the system starts a *detection* phase, where P events are observed at run-time and are checked, usually one at a time, to see whether they are consistent to the learnt behavioral profile $\mathcal{M}$. Traces inconsistent with $\mathcal{M}$ are considered to be *anomalous*, and as anomalous events are considered to be a manifestation of an attack, an alarm will be raised.

[2]Typically, the term "taint" is used in the context of data integrity, while "sensitive" is used in the context of data confidentiality. Similarly, the terms "taint-tracking" and "taint analysis" are used predominantly in the context of integrity, whereas the term "information flow tracking" and "information flow analysis" may be used in the context of data confidentiality as well as integrity.

4.2 Taint-enhanced Anomaly Detection

An important benefit of the taint analysis approach briefly described in the previous section is that it can accurately detect many classes of attacks *without* requiring application-specific policy development. In fact, the enforced policies are generic and easy to specify. However, to provide more comprehensive protection against other types of memory errors [14] or unknown types of attacks, it would be desirable to develop application-specific policies that can tightly constrain the behavior of the protected application. Development of such policies can be time-consuming. Moreover, it could be hard to describe manually, what a policy should look like. On the other hand, anomaly detection techniques have had an advantage in this regard: they do not require manual effort for developing behavior profiles; instead, profiles are automatically learnt using training data that is acquired during normal operation of an application.

The drawback of anomaly detection techniques is that in practice, they suffer from a high rate of false positives (FPs). This is because of the fact that training can never be exhaustive, and hence some unseen (but legitimate) behaviors will be classified as attacks. We believe that the discriminating power of anomaly detectors can be improved by combining them with fine-grained taint analysis. The intuitive justification for this is as follows. Note that an attack involves a combination of a vulnerability, and an attackers ability to exercise this vulnerability. Anomaly detection techniques detect behavioral deviations that occur when a vulnerability (targeted by an attack) is exercised. Now, fine-grained taint information can provide information about the ability of the attacker to exercise this vulnerability. More concretely, consider a system that detects anomalous system call arguments. Such a system may detect an anomalous argument to an `execve` system call, which raises a suspicion. If, in addition, this argument is tainted, then it significantly increases the odds that an attack is in progress. Based on this observation, we propose a mixed approach which couple anomaly detection techniques and fine-grained taint-tracking.

Since taint is a property of data, our proposed approach will be focused on learning properties of system call arguments rather than their names. Let Σ be the set of all the sinks, and $s(a_1, a_2, \cdots, a_n) \in \Sigma$ a generic sink, where $a_1, a_2, \cdots, a_n$ are sink's arguments. As aforementioned, our approach uses taint analysis and anomaly detection using a learning-based approach to learn taint information of sinks' arguments. For instance, our model considers all the system calls and some function of interest (e.g., `print`-like functions used in format string attacks) as sinks.

As other approaches [9, 70, 56], our analysis is *context-sensitive*. That is, it considers contexts for each system call that can be utilized to refine argument learning. For instance, our approach can distinguish between `open` system calls made from two different locations, and can thus learn different properties for the arguments of the two calls. This increases accuracy of the model if, say, one of the `open`'s is used to open a configuration file, while the other is used to open a

data file specified by a user. In fact, intuitively, the former system call uses an *untainted* file name argument, while the latter uses a *tainted* ones. This improves the likelihood to detect an attack should the untainted file name argument be marked tainted during the detection of our approach.

As other anomaly-based approaches, our strategy roughly consists of two phases, namely a *learning* phase, and a *detection* one. During the learning phase our approach build a profile $\mathcal{M}$ of a monitored application, based on the information we will describe here below. Afterwards, when the learning phase is terminated, the application is executed in detection mode and a new profile $\mathcal{M}'$ is created incrementally. Should $\mathcal{M}$ deviate from $\mathcal{M}'$, an alarm will be raised.

In the following, we describe what kind of taint information is learnt by our strategy, and how this can be used to thwart memory error exploits.

Using Coarse-grain Taint Information

Taint information associated to a sink argument a_i are learnt. For aggregate data such as `s`truct's and arrays, the taint status of all bytes of the data will be combined into one. Multiple taint values, e.g., explicit (data dependencies) versus implicit (direct control dependencies) taint, are learnt independently. As we will see in Section 4.3.1, to successfully detect particular class of memory errors, it is necessary to keep track of *how* taint information has been propagated.

At detection time, an alarm can be raised if an argument that was not tainted during training is now found to be tainted. This approach can detect many buffer overflow attacks that modify system call arguments, as opposed to modifying control flows. In fact, it is worth noting that taint analysis implicitly provides a form of control flow integrity, as, generally, tainted code pointers cannot be dereferenced (and therefore, the legal application control flow cannot be altered). Examples of documented attacks that can be detected by this extension are (a) an attack on the popular `WU-FTPD` that corrupts *userid* argument to a `s`etuid system call [14], and (b) an attack on `Netkit` telnet server that overwrites the name of a login program, which is subsequently used as an argument to `e`xecve system call [14].

Using Fine-grained Taint Information

For some aggregate data, the above approach may lose too much information by combining the taint values associated with the data. To improve precision, we can avoid this combination step. For instance, we can individually learn taint information associated with each field of a `struct`. This is particularly useful for some system calls, e.g., `recvmsg`, `sendmsg`, `readv`, `writev`, where a more detailed understanding of the involved data structure is required, in order to gather more meaningful taint-provided information). However, to improve performance, we limit this extension to only those fields specified in a configuration file. For arrays, user may select specific array elements for which taint information needs

to tracked individually. Currently, our proof of concept implementation does not exploit this feature.

Deriving Application-specific Taint-enhanced Policies

Policy-based and anomaly-based detection techniques possess complementary benefits: the main benefit of policy-based detection is a low rate of false positives, while their drawback is the effort required for policy development. In contrast, anomaly detection requires no such effort, but in practice, tends to suffer from a higher rate of false positives. Our strategy combines the strengths of the two approaches by using models built for taint-based anomaly detection to suggest (taint-enhanced) security policies that hold for the application. In this manner, the manual effort for policy development can be significantly reduced.

Learning whether a sink argument a_i is tainted or not already improves the accuracy of our approach and, as aforementioned, allows to detect some memory error attacks which corrupt data pointers (see for instance `WU-FTPD`, and `Netkit` in § 4.3.1). However, a better characterization of the argument considered is needed when more general memory error attacks are involved (e.g., [56]). To this end, for every sink argument a_i, to better characterize its usage, the adopted learning-rules characterize the following properties, depending on whether a sink argument a_i is *fully* or *partially* tainted.

(a) a_i is *fully* tainted. That is, each byte of a_i is tainted. The following argument's properties are inferred by the underlying learning-rules:

Maximum length. Since the argument is tainted, this property inferred during training phase (which is attack-free) gives an approximation of its probable maximum length l_{max} which is expected during detection phase. This helps to detect memory error attacks that try to overflow buffers with the intent to overwrite security sensitive data used at sinks (see [56] for instance). In fact, during detection these tainted arguments will exhibit a length $l > l_{max}$ which, as a direct consequence, will violate the inferred security policy.

Structural inference. There are situations where characterizing arguments' lengths is not enough. An attacker might not try to overflow any buffers but, instead, he might try to modify the normal structure of the considered argument to bypass some security sensitive checks. To this end, the structure of a_i is inferred so that each byte is clustered in proper byte classes. Currently, our model classifies uppercase letters (A-Z) to `A`, lowercase letters (a-z) to `a`, and numbers (0-9) to `0`. Each other byte belongs to a class on its own. For instance, if the model sees an `open("/etc/passwd", ...)` system call invocation, the finite state automaton (FSA) which is generated for the string `/etc/passwd` will recognize the language `/a*/a*`. We further simplify the obtained FSA by removing byte repetition, as we are not concerned about learning

lengths with this model. The final FSA will recognize the language `/a/a`. If during detection the structure of the considered a_i is different from the one learnt, an alarm will be raised.

It can be noted that for particular sinks, trying to infer their (tainted) arguments structure can rise FPs if the structure for that sink is highly unpredictable during learning (i.e., it keeps changing frequently). For instance, when arbitrary data D are read from the network, they are marked as tainted as network input is considered untrusted. Let suppose that D is subjected to some application-specific transformation (e.g., encoding/decoding) or application-specific sanity/security check, subsequently. Let $D^{'}$ be the transformed data. When $D^{'}$ reaches other sinks, such as output sinks (e.g., `open`, `stat`, `execve`) its structure will be either different from the initial one (e.g., encoding/decoding), or its structure will have a "fixed shape" (e.g., sanity/security check) when that particular output sink is reached. Therefore, to try to constrain FPs due to an incorrect characterization of the analyzed argument, and to make the learning phase less data-dependent, it makes sense to enable only the learning of the argument maximum length for particular sinks (e.g., `gets` at proper context, for example [56])[3].

(b) a_i is *partially* tainted. That is, a_i has both tainted and untainted bytes, the tainted portion is subjected to the learning of the aforementioned properties, while the following learning rules are considered for the untainted part (a_i can have different tainted/untainted portions and all of them have to be considered. However, our current implementation considers only the first found pairs):

Minimum length. When an untainted portion of a tainted argument is considered, what is important to remember is its minimum length l_{min}. In fact, considering its maximum length would be misleading as the argument would likely not have the same length all the time and, as the argument portion is untainted (i.e., trusted), we are more concerned on the fact that the argument will *always* have a minimum number of untainted bytes. Intuitively, this is an indication that the attacker will not be able to overwrite the whole untainted argument portion. In most cases the attacker will not be able to overwrite not even a byte of the untainted portion as usually, l_{min} will be identical to l_{max} (e.g., sinks arguments which operate on the same untainted data). However, for those situations where this is not true, l_{min} provides a lower bound under which it is not possible to go without raising an alarm.

Longest common prefix (LCP). Untainted arguments (or portion of them) should have a more regular "structure" or shape than the tainted counterpart (as they are not directly influenced by user input). Therefore,

[3]Although argument length depends on data, it is not data-dependent in the sense that it does not depend on any particular data value.

our approach learns the longest common prefix for every considered sink argument which is partially untainted. Should the learnt longest common prefix be different during detection, an alarm would be raised.

In practice, by using taint information, the taint-enhanced anomaly detection approach herein described *ignores* unknown untainted traces that are encountered during detection phase, but have not been learnt during the training step. A trace (or event, i.e., a system call or function of interest) is considered to be untainted if *none* of its arguments are tainted. Of course, for these events, no arguments characterization is made. As FPs usually arises because of an incomplete learning phase, we are able to lower the FPs rate as our strategy considers legal unknown untainted traces as *not* anomalous, therefore, not as manifestation of an attack. On the other hand, the aforementioned learning rules are adopted for every encountered tainted trace (i.e., an event which has one or more (partially) tainted argument).

Further discussion about attacks on the proposed approach as well as argumentation on the protection provided are remarked in § 4.3.4.

4.2.1 Implementation

As a first step, the approach proposed in this Chapter takes a program P_o as input and produces P, a semantically-equivalent taint-enhanced version of it. We based the taint analysis transformation on the approach developed by the Secure Systems Lab of Stony Brook University [1] to transform a program and taint-enhance it. The main goal of that approach [89] is to enforce taint-enhanced security policies on particular sinks (e.g., `printf`, `open`). Of course, it is a programmer (or administrator) duty to specify these policies. As we have described in § 4.2, there are situations where it is hard or almost impossible to manually come up with an effective taint-enhanced security policy, unfortunately. Therefore, by adopting dynamic learning rules, our approach *automatically* infers taint-enhanced policies as already described in § 4.2. As a direct consequence, the taint analysis prototype developed by [1] has been modified to fulfill the following goals.

- For *every* sink, that is for every system call or function of interest invoked by P, a wrapper $\mathcal{W}$ is introduced by the transformation approach. This enables to (i) learn properties of sinks' arguments, and (ii) mark some input as tainted (e.g., those coming from the network). Depending on the involved sink, wrappers can be inserted before or after its invocation. For instance, wrappers for sink responsible to mark inputs as tainted or not are invoked *after* the sink. The reason is simple. For these sinks, in fact, taint information is available only after the sink invocation and this information is used to build the application profile.

  Following this reasoning, it is possible to automatically infer and enforce taint-enhanced security policies for every sink s by using the information and learning rules described in section 4.2.

- Tracking of control dependency are fully enabled to be able to further enhance the inferred policies not only with taint information but also with information on *how* taint information is propagated throughout the application lifetime. As shown in 4.3.1, this information can help to further thwart memory error attacks which corrupt arbitrary non-pointer data.

In the following we detail the aforementioned steps, the building blocks of taint-enhanced anomaly detection.

1. P is monitored during the *training* phase and a log file is created. The log file includes sink's names and their context information (e.g., calling site), sink's arguments and, for each argument, taint information as well as further characterization, if needed, by using the model described in § 4.2. For instance, a typical log entry looks like the following:

```
read@0x8048f5c 3 arg0={ A:U } arg1={ A:U V[0-98]:T C:99:0:ls -la } arg2={ A:U }
```

   The meaning is as follows. The sink name (`read`) is followed by its calling site (`0x8048f5c`). Next, the number of arguments follows (`3`) and details about these arguments are considered. For instance, the entry for the second argument (`arg2`) tells that the address (`A`) where the sink buffer of size 99 (`V[0-98]`) is stored at is untainted (`A:U`), while the buffer content is tainted (`V[0-98]:T`). Moreover, the content of the tainted buffer which starts at offset 0 is `ls -la`. These information will be used by the next step.

   The learning phase is implemented by using a dynamic shared object loaded into P address space which wraps and overrides the original sinks behavior, re-invoking them whenever necessary.

2. The log file is analyzed *off-line* to build a profile $\mathcal{M}$ of the behavior of P by using the information provided by the previous step. In particular, (i) identical events, that is events whose names and call sites are identical are merged into a single event instance, and (ii) *untainted* events are inserted as part of $\mathcal{M}$ but no further information is gathered (or considered later on) for them.

   For instance, considering the previous example, the tainted sink `read` invoked at the calling site `0x8048f5c` has the first and third argument untainted, while the second argument a_1 is tainted. Moreover, l_{max}, the maximum length for a_2 is 99 while, accordingly to the learning rules described in § 4.2, its structure is `a -a`.

   The profile created during this step is serialized and re-loaded during the next step. This permits to update the profile whenever there is the need to do so, without re-building the out-of-dated one.

3. P is monitored during a *detection* phase. An *on-the-fly* behavioral profile $\mathcal{M}'$ of P is incrementally created. The information contained in $\mathcal{M}'$ are consistent with the one considered in 1. Should $\mathcal{M}'$ be inconsistent with $\mathcal{M}$ an alarm will be raised.

   The detection phase is implemented by using a dynamic shared object loaded into P address space which wraps and overrides the original sinks behavior, re-invoking them whenever necessary.

The anomaly detection part has been implemented by using the C, C++ and Python programming languages with approximately 15,000 lines of code.

4.3 Evaluation

In the following, we present the evaluation of our taint-enhanced anomaly detection approach. We evaluate our approach in terms of effectiveness in detecting memory error attacks, and false positives (FPs). As every learning-based approach, the training phase is the most important. Unfortunately, there are no meaningful publicly available traces that can be used to evaluate the effectiveness of a learning-based approach. Thus, we compare our model with other recent models and show that taint-enhanced anomaly detection is at least as powerful as the models herein considered. Moreover, we show that by considering only tainted events in contrast to every event, we are able to constraint FPs as unseen/unknown untainted events are not considered anomalous during detection. Thus, they are not considered a manifestation of an attack and no alarm is raised.

4.3.1 Effectiveness

It can first be observed that taint analysis by itself provides enough protection for memory error attacks which corrupt security sensitive code pointers, such as function return addresses saved on the stack or function pointers. In fact, it is possible to not consider these kind of memory error exploits in our effectiveness evaluation thanks to the taint analysis approach adopted (see [59, 89])[4]. Therefore, in the following we consider several different examples of attacks which corrupt data and data pointers.

Data and Data Pointers Corruptions

In [14], Chen *et al.* clearly described a memory corruption attack that does not alter the execution flow of the vulnerable program. They showed that these attacks are a realistic threats on real-world software and that the severity of

[4]Indeed, we could relax this restriction and enhancing our learning rules to consider whether a code pointer is allowed to be tainted or not. This is not the case generally, but when direct control dependencies are fully tracked, there might be the situation where a code pointer is marked as tainted based on control dependency taint propagation.

the resulting security compromise is as dangerous as the one caused by memory corruption attacks that alter a normal program control flow.

In their paper, Chen *et al.* showed that (i) configuration data, (ii) user input, (iii) user identity data, and (iv) decision-making data are critical to software security. They provide examples for each of this categories which we report here below, for clarity. Moreover, were necessary, we modify the example to show that our approach is effective regardless of the kind of memory error vulnerability (e.g., buffer overflow, format string).

For each of the proposed attacks, we downloaded the exploit or reproduced artificially the vulnerability and the attacks to show the effectiveness of our approach.

Format String Attack against User Identity Data

A version of `WU-FTPD` is vulnerable to a format string vulnerability in the `SITE EXEC` command ([14]). The default exploits alter the control-flow of the application to directly execute injected code or existing one. The attack proposed by Chen *et al.* aimed to keep the process' privilege level as high as possible (i.e., `root`). In this way, a regular authenticated user could upload a custom `/etc/passwd` file which allowed him to log in as `root` subsequently.

By considering the following code snippet, part of the function `getdatasock`, it is clear that this goal can be achieved by overwriting the `pw->pw_uid` field which contains the cached credential of the current authenticated user[5].

```
1  FILE *getdatasock(...) {
2      ...
3      seteuid(0);
4      setsockopt(...);
5      ...
6      seteuid(pw->pw_uid);
7      ...
8  }
```

Our approach can defeat this attacks in at least two different ways. In fact, we can consider whether `seteuid` argument is tainted or not, or we can detect structural divergence in the tainted arguments of the `printf`-like function used to exploit the format string vulnerability.

In particular, during the training step, our strategy learns that the `seteuid` argument `pw->pw_uid` at line 6 is always *untainted* during the (attack-free) learning phase[6]. On normal situation, where no memory error exploit is attempted,

[5] Although, it is not clear whether the attack proposed in [14] properly works or not (the used injection vector would have the effect to write 10 in the `pw->pw_uid` field, in the best case (for the attacker)), it should be possible to achieve what is claimed in the paper anyway, even if requires a slightly more complicated attack pattern.

[6] Even if `pw->pw_uid` is derived from user input, the result of this system call is marked as untainted. Moreover, also the result of the function `getpwnam`, which would likely be used to

this will be true also during detection phase. On the other hand, it will be tainted when `pw->pw_uid` is overwritten by a memory error attacks.

Considering the type of vulnerability (i.e., format string), our approach can also detect this attack as the `printf`-like function used in the `SITE EXEC` command uses a tainted format string. By using the learning rules described in § 4.2, we can detect any structural divergence of tainted `printf` arguments. In fact, being the training step attack-free, we are sure that the structure of the tainted format string learnt during the training phase will be different from the one observed at detection time (e.g., there will be tainted formatting directives).

Heap Corruption Attacks against Configuration Data

Null HTTPD The attack devised exploits a heap-based buffer overflow vulnerability. It aims at overwriting the `CGI-BIN` configuration string prefix to change it from the value `/usr/local/httpd/cgi-bin` (default value) to the string `/bin`. Every CGI script/program invoked by the client will be searched in this new CGI directory. For instance, the attacker will be able to easily invoke the shell interpreter.

In this scenario, it can be observed that the available options for the attacker are mainly two: (a) to either completely overwrite the original `CGI-BIN` configuration string, or (b) partial overwrite the configuration string. In this latter case, the goal could simply be to execute commands in a sub-directory of the original `CGI-BIN` or, alternatively, perform a directory traversal to reach the intended directory.

Depending on the attacker choices, it is possible to observe different scenarios. For the sake of simplicity, let us consider that the sink of interest here is the `open` system call.

(a) During the training step our approach would learn that the i-th argument of the sink s invoked at the context c is a combination of *untainted* data (i.e., the original `CGI-BIN` configuration string) and *tainted* one (i.e., the command derived from untrusted input).

When an attempt to exploit the memory error is made, it is clear that during detection phase the input will have no *untainted* component. Thereby, our system will raise an alarm for the anomalous event (untainted argument expected).

(b) This case is similar to the previous one because even if we still do have some *untainted* data, the observed length l of this data during detection phase is less then the one learnt during the training phase (with $l_{min} > 0$). Therefore, an alarm would be raised.

obtain information on the credential of a user, is marked as untainted. This holds for several other system calls/functions of interest, but not for others (e.g., the number of bytes read by `read` is marked as tainted as it can be used to influence loops or similar actions.

However, it is worth noting that l_{min} could be less or equal to l (e.g., argument a_i of s at context c operates on different untainted strings). To be successful, an attack should perform a directory traversal attack in order to backward-traverse the original directory while keeping the length of the *untainted* data consistent to what has been learnt. Since a directory traversal attack exhibit clear patterns, and the injected bytes are tainted as they come from the attacker, our structural inference learning rule will discover the divergence with the structure learnt during the training phase (unless, of course, such a pattern would have been learnt, at the same position, during such step).

Netkit Telnetd The attack devised exploits a heap-based buffer overflow vulnerability. It aims at overwriting the program name which is invoked upon login request by referencing the `loginprg` variable as showed by the following code snippet.

```
void start_login(char *host, ...) {
   addarg(&argv, loginprg);
   addarg(&argv, "-h");
   addarg(&argv, host);
   addarg(&argv, "-p");
   execv(loginprg, argv);
}
```

With this type of memory error attack, the daemon ended up by invoking `/bin/sh -h -p -p` (underlined characters are tainted).

Our approach detected this attack in a similar way as it detected the attack launched on `Null HTTPD` described above.

Stack Buffer Overflow Attack against User Input Data

The exploitation of this stack-based buffer overflow vulnerability was tricky but the authors of [14] where able to bypass the directory traversal check the application deployed by the application. In short, after the directory traversal check and before the input usage, a data pointer is changed so that it points to a second string which is not subjected to the application-specific sanity check anymore, thus it can contain the attack pattern (similar to a TOCTOU).

As the attack previously reported, also this memory error exploits can be detected in a similar way. In fact, if the tainted argument does not contain attack pattern during the training phase (e.g., directory traversal `../` patterns), an attack attempt during detection phase will present a different structure from the one previously observed.

Straight Overflow on Tainted Data

The following example has been proposed in [56] by Mutz *et al.*.

```
1  void write_user_data(void) {
2
3     FILE * fp ;
4     char user_filename[256];
5     char user_data[256];
6
7     gets(user_filename);
8
9     if (privileged_file(user_filename)) {
10      fprintf(stderr, "Illegal filename. Exiting.\n");
11      exit(1);
12    }
13    else {
14      gets(user_data);        // overflow into user_filename
15      fp = fopen(user_filename, "w");
16      if (fp) {
17        fprintf(fp, "%s", user_data);
18        fclose(fp);
19      }
20    }
21 }
```

The possible memory error attack is simple. The `user_filename` array obtained at line 7 (`gets` function) is subjected to a security check performed by the function `privileged_file` (line 9) that checks whether `user_filename` specifies a name of a privileged file or not. In affirmative case, the program prints an error message and quits. Otherwise (i.e., non privileged file), more data is read into the array `user_data`, through the function `gets` at line 14, and the file name specified by `user_filename` is opened at line 15. An attacker can overflow `user_data` by overwriting past its end, and overflowing into `user_filename`. As the overflow happens after the security check performed at line 9, an attacker can specify a legal file name for `user_filename` that will be replaced later on thanks to the overflow.

Our approach detects this data attack by learning the maximum length l_{max} of the tainted arguments of the `gets` invoked at line 7, and 14 (the training phase must be attack free). It is possible to infer the structures of their arguments as well, but due to the nature of the program, this might raise too FPs. Of course, using l_{max} by itself could raise FPs as well, as it highly depends on the accuracy of the learning step. Nonetheless, this does not depend on the value of the observed data, and therefore on the precision of the underlying statistical models.

Actually, this attacks and other similar memory errors which overflows adjacent variables could be solved in a different way, as we describe here below.

Roughly speaking, some modification to the way data is tainted are necessary. The taint propagation mechanism is kept unmodified – it just uses more bits to mark a data as tainted.

> As in every taint-based approach, we would mark *untrusted* input of interest as tainted. However, instead of marking the data as tainted or not, we *tag* that piece of data ideally with a 4-byte identifier (*label*). This identifier tracks the source, i.e. the context, where the tainted

piece of data came from[7]. As a consequence, the mechanism permits to know if a tainted tag which belongs to a sensitive memory location identified by x *flows* into y.

This solution is similar to what has been proposed in [83] which provides a dynamic information flow security (IFS) framework to ensure confidentiality of sensitive data. "[...] IFS policies consist of security *labels* and legal *flows*. Security labels are annotations associated with each storage location. Labels are used to classify information [...]. Flows are label pairs that determine valid information flow. For example the flow $l_1 \rightarrow l_2$ allows information to flow from label l_1 to label l_2."

This approach would exploit the advantages provided by a dynamic learning mechanism to learn information flow policies and to raise an alarm whenever a policy is violated. It is worth noting that this approach would, however, be somehow different to IFS because we do not aim at providing confidentiality. Therefore, generally, we will not observe any flow from a tainted data marked with the taint tag (label) l_1 to another one marked with a different tag l_2. If this happens, it means that some information *leaked* into another variable. By comparing this behavior and observing this anomaly at run-time, it will be possible to raise an alarm to stop the attack.

Coming back to the attack proposed by Mutz *et al.*, to better understand how our taint-enhanced anomaly detection would detect this attack, using the different taint marking strategy, let us consider the training and the detection phases of our approach.

Training During the training phase, each event which is a taint source is marked with a unique taint tag which comes directly from the calling context. For our purpose, let us consider the program line numbers as unique proper context's tags.

The taint propagation mechanism, then, propagates the corresponding tag.

At each sink point s we would collect its properties (e.g., sink's arguments) of s. In our example, both `user_filename` and `user_data` will be marked as tainted. However, the former will have a unique tag (7) as well as the latter one (14).

During training we should not learn any attack instance, therefore, at `fopen` and `fprintf` sink points we would correctly learn the right taint tag associated with the arguments of the considered sink points.

Detection During detection phase, when no attack are encountered, our approach should correctly detect that the tainted data with its corresponding learnt taint tag is consistent to what has been seen during the training phase.

[7] A similar idea slightly more complicated was introduced in [53] even if the final goal was different.

If we consider the data attack proposed in [56] it is clear how the `user_data` taint tag *flows* into the `user_filename` taint tag, as `user_data` data flows into `user_filename` as well. By comparing the actual model with the expected one, it is clear that an alarm would be raised.

It is worth noting that the model proposed in [56] can deal with this attack as well by leveraging the fact that two different system call are invoked (on the contrary, their previous context-insensitive model [55] would miss it), thereby two different context are considered. While our model would leverage on the same fact, there are two main differences with the approach proposed by Mutz *et al.*

(a) in this example, the approach proposed in [56] has to perform a good training phase. That is, the model learnt for the `gets` invoked in different context *must not* overlap. Our model, instead, would not consider data in this case because it would use the taint tag as the main source of information. The only requirement is that those instructions have to be executed at least once.

(b) our model should also be able to thwart other data attacks even in the presence of only one taint source s invoked by a particular context c. In fact, it is quite common to find direct control dependency in benign program used in security critical cases. Our approach should then be able to distinguish between a legal and illegal tainted data usage by leveraging on the type of taint propagation that has been performed.

In most of the case, a memory corruption attack causes a direct data dependency taint propagation to take place while a condition variable relevant from a security perspective "comes from" a direct control dependency taint propagation, as shown by the next example.

Format Bug to Bypass Authentication

This example has been proposed in [47]. As we will see, even if the vulnerability herein considered is a format string and can be detected by our approach by using the learning rules described in § 4.2, we will also show that our approach detects the data attack which aims to corrupt the non pointer variable `auth` by using taint information only. This is possible by considering the different types of taint propagation involved (i.e., data and direct control dependencies) during learning and detection (when the attack is actually performed) phases.

```
1  void do_auth(char *passwd) {
2     char buf[40];
3     int auth;
4
5     if (!strcmp("encrypted_passwd", passwd))
6        auth = 1;
7     else
8        auth = 0;
9
```

```
10      scanf("%39s", buf);
11      printf(buf);          // format string
12      if (auth)
13         access_granted();
14  }
```

The attack is simple. Normally, the variable `auth` is set to 1 or 0 depending on the fact that the right authentication credential is given as input or not (line 5). An attacker, can exploit the format string vulnerabilities at line 11 and overwrites `auth` with a non-null value so that the subsequent check of the credential (line 12) will grant privileged access.

Again, we can stop this attack in two different ways:

1. By modeling the format string (`printf(buf)`, line 11) itself we can learn whether tainted format directives have been used during the training step, along with their structure (structural inference on tainted arguments). If so, we check if the ones we are observing are consistent to the model we learnt. Since we learnt attack-free dangerous and tainted patterns, it should be impossible to mimicry those in order to perform a successful format bug attack, as this require the use of tainted `%` format string directives. (both memory corruption and information leakage).

2. If `access_granted` invokes a function or system call of interest S, we are able to detect the memory corruption attack by comparing the type of taint propagation that has been performed at S during detection with the one learnt during the training step. We remind that tracking of control dependencies is easy to do by associating a *taint label* with the *program counter* ([23, 89]). Whenever the condition involved in a branch decision is *tainted*, the program counter is also tainted. An assignment causes the target variable to be tainted if the program counter is tainted, or if its right-hand side expression is tainted. The label of the program counter is restored at the merge point following a conditional branch.

   More precisely, different situation can be considered:

   - When a tainted scope is created by a condition whose taint value has been determined based on *data dependency* only, every taint propagation in the tainted scope will be marked as due to direct *control dependency* only, unless the expression involved in the propagation is tainted; in this case, the union of the taint values is considered.
   - A similar situation happens when a tainted scope is created by a condition whose taint value has been determined based on *control dependency* only
   - On the other hand, when a tainted scope is created by a condition whose taint value has been determined based on both *data* and direct *control* dependency, every taint propagation in the tainted scope will inherit this label as well

For instance, let us consider an attack-free scenario. During the training phase, the variable `auth` is marked as tainted based on a tainted scope created by a condition whose taint values is based on data dependency only (lines $5-8$). Therefore, the taint label associated with `auth` is a control dependency taint value. The same happens when `auth` creates the tainted scope at line 12. Therefore, every taint propagation performed in `auth` tainted scope will inherit its taint label as well (control dependency only). If this is respected during detection as well, no alarm would be raised.

On the other hand, the exploitation of the format bug at line 11 has the effect to overwrite `auth` with a non-null value to be able to get successful unauthorized authentication. This tampering has the effect to augment the taint value of `auth` to be based on data dependency as well (`auth` retains the one given by control dependency). Therefore, every taint propagation performed in `auth` tainted scope will be based on both data and control dependency. This is inconsistent with respect to the profile learnt during training phase, and an alarm would be raised.

Integer Overflow Attack against Decision-Making Data

This attack has been proposed in [14] and aims at overwriting the variable `authenticated` so that authentication check will be bypassed even in case of erroneous authentication. This is shown by the following code snippet. It is very similar to the example shown above. The only difference is that the tainted scope created by `authenticated` is soon interrupted by a `break`. No further instructions are executed. We propose to insert harmless system calls pairs, just to insert useful sink where the learnt profile can be compared to (e.g., `getuid`/`Studi` pairs would do). If this is done, the scenario can be reduced to the one shown in the example above. Currently, our prototype does not support this scenario.

```
void do_authentication(char *user, ...) {
   int authenticated = 0;
   ...
   while (!authenticated) {
      type = packet_read();
      switch (type) {
         ...
         case SSH_CMSG_AUTH_PASSWORD:
            if (auth_password(user, password))
               authenticated = 1;
            break;
         case ...
      }
      if (authenticated)
         break;
   }
   do_authenticated(pw);
}
```

Untainted Format String Attacks

As pointed out in [18] it is possible to exploit format string vulnerabilities in such a way to being able of writing untainted data to untainted memory locations. This attack would be very hard to perform, however our taint-enhanced anomaly detection provides protection against it, because of the learning rules used on tainted sinks' arguments.

4.3.2 Models Comparison

For all the data attacks aforementioned, we summarize the effectiveness of our taint-enhanced anomaly detection as well as the one of related works ([56, 47, 9, 55, 13, 75]) in the following.

Format String Attack against User Identity Data [14]

Model	Detect?	Reasons/Comments
Anomalous System Call Detection [55]	**No**	The missing context information would create only one model for a given system call.
Exploiting Execution Context for the Detection of Anomalous System Calls [56]	**Yes**	Learning performed only on a subset of system calls. Therefore, `printf`-like calls will be missed. However, if the learning sets of the `seteuid` call overlap, is easier to perform mimicry (e.g., learning performed on 2 users allow one of them to impersonate the other).
Dataflow Anomaly Detection [9]	**Yes**	If the learnt sets overlap is easier to perform mimicry.
Secure Program Execution via Dynamic Information Flow Tracking [75]	**No**	The approach detects only code pointers corruption.
Defeating Memory Corruption Attacks via Pointer Taintedness Detection [13]	**Yes**	By denying *any* tainted pointer dereferencing the approach might exhibit high false positives rates.
Improving Software Security via Runtime Instruction-level Taint Checking [47]	**Yes**	The approach is not able to differentiate between good and bad taint usage. It might exhibit high false positives rates.
our approach	**Yes**	It learns policies on tainted format string – if any. Moreover, it is also possible to signal that a tainted format argument is used, during training.

Heap Corruption Attacks against Configuration Attacks [14]

Model	Detect?	Reasons/Comments
Anomalous System Call Detection [55]	**Prob. No**	Highly dependant on learning performed on the same system call on different contexts. The string length has to represent all these contexts, therefore it is easy to be able to overflow a variable. Others model might be bypassed by mean of mimicry.
Exploiting Execution Context for the Detection of Anomalous System Calls [56]	**Prob. Yes**	Highly learning-dependant.
Dataflow Anomaly Detection [9]	**Prob. Yes**	Same as above.
Secure Program Execution via Dynamic Information Flow Tracking[75]	**No**	The approach detects only code pointers corruption.
Defeating Memory Corruption Attacks via Pointer Taintedness Detection [13]	**Yes**	By denying *any* tainted pointer dereferencing the approach might exhibit high false positives rates.
Improving Software Security via Runtime Instruction-level Taint Checking [47]	**Yes**	The approach is not able to differentiate between good and bad taint usage. It might exhibit high false positives rates.
our approach	**Yes**	It learns policies on *untainted/tainted* arguments as well as their structural inference and longest common prefix.

Integer Overflow Attack against Decision-Making Data [14]

Model	Detect?	Reason
Anomalous System Call Detection [55]	**No**	There is profile as no system calls/function of interests are involved.
Exploiting Execution Context for the Detection of Anomalous System Calls [56]	**No**	Same as above.
Dataflow Anomaly Detection[9]	**No**	Same as above.
Secure Program Execution via Dynamic Information Flow Tracking [75]	**No**	Same as above.
Defeating Memory Corruption Attacks via Pointer Taintedness Detection [13]	**No**	(see pag. 5 and 8 of the paper).
Improving Software Security via Runtime Instruction-level Taint Checking [47]	**No**	Due to untainting operations.
our approach	**No**	However, it could be detected by adding a fictional sink in the tainted scope and learning the type of taint propagation as described at the end of the previous section.

Stack Buffer Overflow Attack against User Input Data [14]

Model	Detect?	Reasons/Comments
Anomalous System Call Detection [55]	**Prob. No**	Highly dependant on learning performed on the same system call at different contexts.
Exploiting Execution Context for the Detection of Anomalous System Calls [56]	**Yes**	
Dataflow Anomaly Detection [9]	**Yes**	
Secure Program Execution via Dynamic Information Flow Tracking [75]	**No**	The approach detects only code pointers corruption.
Defeating Memory Corruption Attacks via Pointer Taintedness Detection [13]	**Yes**	By denying *any* tainted pointer dereferencing the approach might exhibit high false positives rates. Highly vulnerability-dependant.
Improving Software Security via Runtime Instruction-level Taint Checking [47]	**Yes**	The approach is not able to differentiate between good and bad taint usage. It might exhibit high false positives rates. Highly vulnerability-dependant.
our approach	**Yes**	It learns policies on *untainted/tainted* arguments and their structural inference and longest common prefix.

Straight Overflow on Tainted Data [56]

Model	Detect?	Reasons/Comments
Anomalous System Call Detection [55]	**No**	
Exploiting Execution Context for the Detection of Anomalous System Calls [56]	**Yes**	Highly dependant on the learning. Overlapping sets make mimicry attacks easier.
Dataflow Anomaly Detection [9]	**Yes**	Highly dependant on the learning. Overlapping sets make mimicry attacks easier.
Secure Program Execution via Dynamic Information Flow Tracking [75]	**No**	The approach detects only code pointers corruption.
Defeating Memory Corruption Attacks via Pointer Taintedness Detection [13]	**Yes**	By denying *any* tainted pointer dereferencing the approach might exhibit high false positives rates. See also next.
Improving Software Security via Runtime Instruction-level Taint Checking [47]	**Yes**	The approach is not able to differentiate between good and bad taint usage. As above for the rest.
our approach	**Yes**	It learns the maximum length of the tainted argument and possibly its structure. Alternatively, it could use the proposed labeling schema. In this case, the approach would not highly depend on the learning as it would be sufficient to execute instructions at least once to learn *how* information flows.

Format Bug to Bypass Authentication [47]

Model	Detect?	Reason
Anomalous System Call Detection [55]	**Depends**	. . . on the learning (see below).
Exploiting Execution Context for the Detection of Anomalous System Calls [56]	**Depends**	. . . on the learning. However, the proposed model should hardly be able to infer the structure in this case. The other models might not be so useful (e.g., the string might have some non-consecutive `%` characters and the model will learn it (prone to mimicry).
Dataflow Anomaly Detection [9]	**Prob. Yes**	Highly dependant on the learning.
Secure Program Execution via Dynamic Information Flow Tracking [75]	**No**	
Defeating Memory Corruption Attacks via Pointer Taintedness Detection [13]	**No**	
Improving Software Security via Runtime Instruction-level Taint Checking [47]	**Depends**	Yes, if control-dependencies are not tracked. In this case, `auth` has to be manipulated by taintless-instruction. However, if control-dependencies are tracked, then the approach is not able to detect the attack. As pointed out in [18], control-dependencies tracking has to be enabled to improve detection capabilities.
our approach	**Yes**	It learns the right policies (or, alternatively, by using the extended proposed taint labelling schema).

4.3.3 False Positives

Usually, in anomaly-based approaches false positives arise when an unknown legitimate event is encountered during detection phase. Our approach does not consider legitimate events encountered during detection phase, as long as these events are *untainted* (i.e., no events arguments have to be tainted). Of course, it is not possible to say whether an unknown tainted event is a manifestation of an attack or not. Therefore, as in every anomaly-based approach, should an unknown tainted event be encountered during detection, most likely will raise an alarm. Nonetheless, as our approach uses some statistical models to characterize properties of sinks' arguments, FPs have to be expected as well. As every learning-based approaches, the learning phase is the most critical.

Not considering unseen/unknown untainted traces during detection phase already constrain FPs. To further constraint them we disable the structural inference and longest common prefix for some sink, should they show a too irregular and arbitrary structure. For instance, `read`, `recv`, and other input-related system calls are highly sensible to the input they receive. A poor characterization

of these input will result in either in a too specific or in a too generic model. The former will raise FPs, while the latter will give raise at FNs. However, it can be noted that, whenever possible, input will reach output sink. Therefore, it is possible to enforce statistical models more on output sinks than on some input, where lighter properties could be learnt (e.g., maximum input length seen so far).

The following table summarize the FP rates we experienced while testing `proftpd`.

#	App	# Traces (Learning)	# Traces (Detection)	FP	FP rate
1	`proftpd`	59,729	1,532,293	0	0%

The following table shows the percentage of tainted events the have been encountered during the learning phase. As it can be noted, only a small percentage of the whole profile has to be considered.

#	App	# Traces	# Taint. (%)	# Uniq.	# Uniq. Taint. (%)
1	`proftpd`	59,729	19,798 (33.15%)	330	39 (11.80%)

Finally, in the following table, we show the overhead introduced by our taint-enhanced anomaly detection.

#	App	slowdown (taint)	slowdown (taint-learn)	slowdown (taint-detect)
1	`proftpd`	3.10%	5.90%	9.30%

Various activity were performed during learning/detection. Among the others, the following were issued: change directory, download ($\sim$ 138MB), upload, recursive directory listing.

4.3.4 Discussion

The proposed approach couples taint analysis and anomaly detection techniques. To the best of our knowledge, no existing similar techniques have been proposed so far in the context of benign services protection. Anomaly detection or, better, learning-based approaches help to automatically infer security policies, as shown by existing techniques [87, 85, 28, 39, 70, 56, 54, 9]. Unfortunately, these techniques have two major drawbacks. They (i) often exhibit high false positive rates issues as learning phases are hard to be exhaustive, and (ii) they are vulnerable to mimicry attacks [86, 85, 48, 78, 77] as attack-provided data can often stick to statistical learning-rules used to characterize the process behavior. By using taint analysis, we constraint these drawbacks as (i) unknown *untainted* traces seen during detection are *no more* considered as manifestations of attacks, and (ii) as foreign code cannot be executed anymore and learning-based rules use taint information to infer security policies, mimicry-like attacks – even if still possible – are thwarted.

The taint-tracking mechanism of our *taint-enhanced anomaly detection* provides *deterministic* protection when a memory error exploit corrupts a *code* pointer (absolute or partial overwrite). In fact, code pointers are usually initialized and manipulated by application code (e.g., function return addresses), which is considered to be *trusted*[8], therefore untainted. As a direct consequence, mimicry attacks ([86, 85, 48, 78, 77]) which rely on hijacking the execution flow of the vulnerable process to either invoke in-trace system calls (or, more generally, sinks), or to corrupt security sensitive data by executing foreign code are no longer possible.

Unfortunately, while this class of arbitrary code execution and mimicry attacks are defeated, others – even if thwarted – could still be possible. More precisely, mimicry attacks can target sinks' arguments, or generically tamper the process address space with the intent to corrupt security sensitive data. In fact, learning-based rules are used to automatically infer a security policy to be enforced on system calls or functions of interest (i.e., sinks). Unfortunately, sometimes these rules could be either over permissive or over restrictive. False negatives (FNs) and false positives (FPs) are, respectively, the consequence of this characterization. Following this reasoning, it is possible to distinguish these cases:

Untainted sinks arguments. Taint information is used by the whole approach to infer security policies to be enforced on sinks during detection phase. This already gives a better process behavior characterization compared to previous models (see [87, 85, 28, 39, 70, 56, 54, 9], for instance). Generally, previous models had focused their attention on *every* event of the monitored process to better characterize its behavior. Of course, over simplified models carry minimal information and are more likely to be defeated by mimicry-like attacks. Likewise, over specialized events characterizations as well as unknown unseen events encountered during detection phase, would lead to high false positives rates. In our approach, instead, a sink argument a found to be *untainted* during training phase, *must* be untainted during detection as well. Therefore, a large number of mimicry attacks that aim to tamper with untainted data are no longer possible, as attack-provided data or, more generally, input data[9] are always considered untrusted and thus marked as tainted.

Moreover, untainted events, that is sinks whose arguments are untainted,

[8] As noted elsewhere, it is possible to relax this requirement as function pointers might be initialized based on tainted control dependency conditions. A conservative approach is to permit the code pointer to be either untainted or tainted due to control dependency taint propagation. In the latter case, an enumerated set of admissible and legal addresses learnt during training is maintained and checked against for consistency during detection. The main drawback is that a "selected" mimicry attack could be executed (instead of an arbitrary one). However, the ability to cause meaningful damage is constrained.

[9] Which, for our model, are network inputs.

encountered during detection phase are not considered as attacks' manifestations, therefore lowering false positives (FPs) rates.

As described in § 4.2, things change a little when a combination of tainted and untainted arguments are characterized. In fact, the untainted part is characterized by using the minimum length, and longest common prefix models. These models are highly dependant on the value of the data seen during training. However, these data are untainted, therefore (i) they cannot be modified by an attacker without raising an alarm (in our threat model, network inputs – and thus attacker-provided inputs as well – are always marked as tainted), and (ii) depending on the considered sink, they should be more "predictable", thus easier to characterize constraining FPs.

Tainted sinks arguments. Unfortunately, tainted sinks arguments can be controlled by an attacker. Therefore, mimicry attacks are still possible on the models – maximum length, and structural inference – used to characterize these tainted inputs. Anyway, as foreign code or already present code which does not logically follow the normal execution flow, cannot be executed anymore (tainted code pointers cannot be de-referenced anymore), it is harder for an attacker to stick to the models inferred during training phase. In fact, this extremely relies on the type of memory error vulnerability involved (see § 2) and the position where the vulnerability is located. A critique of the adopted models in this context follows.

(a) *Maximum length.* As described in § 4.2, the main purpose of this simple model is to provide an upper bound to the number of bytes considered for a given sink argument. An over permissive model (i.e., too high upper bound) would permit overflows to occur during detection phase. As a direct consequence, variables adjacent to the overflown buffer could be controlled by an attacker potentially missing attacks (e.g., mimicry-like). On the other hand, an over restrictive model (i.e., too low upper bound) would wrongly characterize a given sink argument. As a direct consequence, FPs would be more likely to occur.

(b) *Structural inference.* As already noted in related literature [54], there are cases where an attacker is able to craft its input in order to stick to the considered model yet being able to cause harm. As described in § 4, the purpose of the structural inference model is to learn the structure of a given sink argument. While the model herein considered could be more vulnerable to mimicry-like attacks, others (e.g., [54]) are not (see next).

We remark that some of the considered model are not new (see for instance [54, 56], which also propose a better structural inference model). However, it is important to note that the learning rules and models considered herein can be definitely replaced by more accurate models (e.g.,

temporal relationship among system calls arguments [9], other statistical models [56]). The strategy of coupling taint analysis with anomaly detection offers independent benefits from the underlying learning-rules and models adopted, as already pointed out in 4.2.

Security sensitive data. The adopted learning-rules not only consider whether a sink argument is tainted or not, but also keep track of *how* security sensitive data or, more generally, memory locations, have become tainted. In fact, as pointed out in § 4.2 and showed in § 4.3.1, taint marks carry different values depending on whether they originate from data or control dependency taint propagation, or a combination of both. As shown in § 4.3.1, this permits to thwart memory error attacks which target non pointers data.

It is worth noting that the approach proposed in this Chapter can, sometimes, detect memory error exploits attempts even before reaching a system call argument (e.g., format string and tainted formatting directive). While the approach described in the previous Chapter is generally vulnerability-independent, our taint-enhanced anomaly detection can be more successful depending on the underlying vulnerability considered (e.g., format string versus buffer overflow). For instance, one of the conditions that must hold to successfully exploit a format string vulnerability (e.g., § 2.2 and [69, 35]) is that the formatting string has to be controlled by the attacker. This means, that the format string has to be *tainted*. Therefore, our approach provides protection in two ways. First, during learning it signals that a particular formatting string is *tainted*. As there is no reason to have a tainted format string, the application could be fixed right away (e.g., `printf("%s", buf)` instead of `printf(buf)`). Second, if this is not possible, as the learning step has to be performed in an attack-free environment, no dangerous formatting directives can be learnt. Therefore, the inferred structure will not contain any dangerous tainted formatting directive (e.g., `%x` or `%n`).

In summary, taint-enhanced anomaly detection offers a *deterministic* protection for code pointers corruption and, depending on the specific scenario, for attacks which corrupt data and data pointers. In other cases, the offered protection is *probabilistic*. The catch is that learning phase is still an open issue (e.g., how long to learn for? Is the learning meaningful, i.e., synthetic – probably no but attack-free – versus "in the wild" – more meaningful but probably not attack-free? What about code coverage?), especially because it is not exhaustive and this most likely contributes to raise false positives during detection. The proposed approach, besides (i) providing *deterministic* protection for some class of memory error attacks, and (ii) thwarting mimicry-like attacks for others, it also constraints false positives. In fact, differently from other anomaly-based approaches ([87, 85, 28, 39, 70, 56, 54, 9]), we remark that in our approach, *unknown* and *untainted* events encountered during detection phase, are not considered as a manifestation of attacks, but as the fact that learning-based approaches are hard to be exhaustive in completely characterizing an application behavior.

Not everything that counts can be counted, and not everything that can be counted counts. (Sign hanging in Einstein's office at Princeton)

Albert Einstein (1879 - 1955)

5 Related Literature

The approaches proposed in this dissertation are mainly related to three research topics, namely *artificial diversity*, *information flow* (also known as taint analysis), and *learning-based anomaly detection*. In the following, we describe related works that have been done in these areas. Likewise, wherever appropriate, we extend the discussion to other interesting strategies as well.

5.1 Artificial Diversity

Forrest *et al.* suggested preliminary ideas for building diverse computer systems [65]. In their paper they observed that computer systems were mainly monoculture with no diversity at all. Due to this, a memory error exploit would be successful on almost all the computer systems belonging to the same "species". Hence, they proposed the use of several forms of randomization in order to introduce diversity into computer systems and, following such an idea, others researchers faced the problem of providing diversity to computer systems.

In [79], a kernel level patch has been developed in order to give the opportunity to load the memory segments of a process (code, data, heap, stack), as well as the shared objects the process makes use of, at different memory locations achieving what has been called address space layout randomization (ASLR). 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 of a more fine-grained randomization. Moreover, as run-time relocation is generally not possible, information leakage attacks or the not-so-strong effectiveness of ASLR on 32-bit Intel Architecture [40] can defeat these coarse-grained diversification mechanisms.

Other address obfuscation techniques have been proposed in [68, 67] by Bhatkar *et al.* as a particular form of program transformations to combat memory error exploits which corrupt pointers (code and data) and non-pointer data. Such approaches differ from the one proposed in [79] since they aim to provide a more fine-grained diversification via address space obfuscation. The objectives of the

obfuscation transformations are to randomize the absolute locations of all code and data of a process in order to achieve protection from memory error exploits targeting code pointers (both absolute and partial overwrite), and to randomize the relative distance between different data objects in order to defeat relative addressing attacks [14]. 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 [68], where a source-to-source transformation on C programs has been developed to produce self-randomizing programs with the intent to randomize a process address space [67, 68] and its data representation [8].

All the aforementioned techniques share a common concept: they provide diversity on a process itself and thus, they provide a defensive mechanism that, in general, only provides a *probabilistic* protection from memory error exploits.

Recently, Cox *et al.* faced in [7] the concept of process replication with diversification. Their framework is similar to the one proposed in this dissertation (see Chapter 3). As an application, they propose two different diversification approaches to defeat (i) memory errors which corrupt 32-bit addresses (address space disjointedness), and (ii) code injection attacks (instruction set tagging). We improve the diversification strategy adopted by addressing memory errors which partial overwrite pointers. Moreover, the model proposed in [7], as ours one, introduces some unwanted issues that can negatively influence a practical real model deployment. For instance, shared memory and synchronous signals delivery have to be properly managed to guarantee data and process behavioral consistency. To this end, we provide a solution that, to some extent, can represent a first step toward a more realistic model usage.

5.2 Information Flow

Information flow analysis has been researched for a long time [6, 29, 21, 51, 84, 57, 66]. Early research was focused on multi-level security, where fine-grained analysis was not deemed necessary [6]. More recent work has been focused on language-based approaches, capable of tracking information flow at variable level [61]. Most of these techniques have been based on static analysis, and assume considerable cooperation from developers to provide various annotations, e.g., sensitivity labels for function parameters, endorsement and declassification annotations to eliminate false positives. Moreover, they typically work with simple, high-level languages. In contrast, much of security-critical contemporary software is written in low-level languages like C that use pointers, pointer arithmetic, and so on. As a result, information flow tracking for such software has been primarily based on runtime tracking of explicit flows that take place via assignments.

Recently, several different information flow-based approaches, often known as

taint analysis as they are concerned with data integrity, have been proposed [59, 89, 13, 47, 76]. They give good and promising results when employed to protect benign software from memory errors [59, 89], and a broader class of attacks [89] by usually relying, for instance, on some implicit assumptions which are common grounds on benign software (e.g., no tainted code pointers should be de-referenced, no tainted SQL directive should be used). Other researchers (e.g., [47, 76]) extended basic taint-tracking techniques in order to generically address attacks which corrupt data and data pointers [14] as well. Preliminary results seem promising, even some of them require architectural change ([47]) and it is still unclear whether they can thwart a broad range of memory error attacks while exhibiting only a limited rate of false positives.

5.3 Learning-based Anomaly Detection

The underlying idea of any dynamic learning-based anomaly detection approach is to monitor an application P in order to characterize its behavioral profile $\mathcal{M}$, during an initial period often known as training or learning phase. Afterwards, the same application P is monitored again generating $\mathcal{M}'$, a behavioral profile of P observed during its normal operations. Should $\mathcal{M}'$ somehow differ from $\mathcal{M}$, an alarm will be raised. While learning-based approaches usually share the aforementioned steps, they usually differ in the way the application behavior is inferred.

The idea of using syscall obfuscation for preventing computer intrusions has been introduced by [49], where an obfuscation scheme based on the randomization of the system call mappings has been used for dealing with some type of buffer overflows. Following this idea, Forrest *et al.* [30, 39] introduced a learning-based anomaly detection strategy in order to characterize the behavior of an application P. This system is 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 host intrusion detection system (HIDS).

The N-gram model is very simple and very efficient but it is characterized by a relatively high degree of false alarms [36], 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 [85] it has been shown that such a HIDS model is unable to detect two particular forms of computer attacks, namely the *mimicry* [86, 85, 48, 78, 77] and *impossible path execution*

(IPE) [28, 85] attacks. 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 for instance [70, 85, 28, 37]). However, even these models suffer of some limitations. For example, in [85, 28] it has been shown that the callgraph model proposed in [85] as well as the model proposed in [70] are not able to deal with some forms of IPE, while in [86, 48] it has been shown that all the models above mentioned are susceptible, with various degrees of resistance, to some forms of mimicry attacks.

In a recent paper, Kruegel *et al.* [48] observed that even if the introduction of such techniques in anomaly-based HIDS [11, 28, 70] has significantly reduced the possibility to perform successful traditional mimicry attacks [78, 77, 86], 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 instance, the *execution* of 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 [48] 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. It is worth noting that the research works described in this dissertation (see Chapters 3 and 4) catches any de-reference of corrupted code and data pointers, therefore defeating or, in some cases constraining, this particular threats. While this could be somehow misleading for the approach proposed in Chapter 3 as the technique does not involve any learning phase, it is indeed particularly important for the taint-enhanced anomaly detection approach proposed in Chapter 4.

On the basis of the previous observation (i.e., execution of foreign code), other

techniques have been recently proposed for containing automatic mimicry [48]. For instance, [12] proposes a strategy which consists of localizing, inside a IA-32 binary P, all the *dangerous regions* $a_i, \cdots, a_n$, where dangerous region, also known as *liveness* area, are code areas between the definition D and use U of the values V of the system calls parameters. After the liveness areas have been determined 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 are collected at run-time. Subsequently, the process P image is instrumented 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 be thwarted.

As pointed out in [48] and further reiterated by [14], for instance, it is clear that system call monitoring by itself is no longer sufficient to correctly characterize an application behavior. To this end, researchers proposed statistical models [56, 55] which try to characterize system calls *arguments* to improve the precision of the application's behavioral profile. Likewise, data flow relationship between system calls arguments [9] have been recently proposed to address broader classes of attacks (e.g., memory errors, race conditions).

5.4 Control-Flow Integrity

Originally, one of the main goal of a successful memory error exploitation was to execute arbitrary code. To this end, code pointers had to be eventually corrupted to be able to divert the normal process execution flow. Following this reasoning, several strategies have been proposed to guarantee code pointers integrity which in turn provides control-flow integrity. Some of them [15, 26], aim to protect the integrity of a limited set of code pointers (e.g., return addresses and saved frame pointers). Some of them [92, 91], offer a very good protection with a very limited performance slowdown for some code pointers corruption. Unfortunately, they leave several different venues to the attackers as the offered protection is not comprehensive (see § 2).

In [2], 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, attempts 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 [45]. While CFI could be enforced by pro-

gram 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 apriori. 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.

Part III

Future Directions & Conclusions

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

Brian W. Kernighan (1942 -)

6 Future Directions

The research works proposed in this dissertation aim to provide comprehensive memory errors protection. The previous chapters showed that the proposed approaches are effective against a broad range of memory error vulnerabilities. Nonetheless, the methods are far from being complete and, as pointed out in the respective chapters, they suffer from some limitations and drawbacks. In the following, we aim to highlight possible future directions that can be taken to improve the underlying ideas proposed in the dissertation.

6.1 Diversified Process Replicæ

The main drawbacks of the approach described in Chapter 3 refer mainly to *performance* issues, *relative-addressing* attacks, and *side-effects* introduced by the adoption of the approach (e.g., threads management). In the following, we sketch possible directions to be examined in the future to improve the diversified process replicæ approach.

6.1.1 Optimizations

One of the main drawback of the proposed approach is about the introduced overhead. In fact, as pointed out in § 3.4.2, as P and its replica P_r have to be executed, the introduced overhead is of at least 100%. To this percentage it is necessary to add the overhead introduced for replication management which is, depending on the cases, not always negligible. Moreover, the prototype implementation uses the `ptrace` system call, which is known to be *slow* (in fact, it should be used for debugging purposes). To lower the overhead, it is possible to act in the following ways:

1. *Native execution.* As pointed out in § 3.4.2, there are situations where data replication from P to P_r is not needed. It is possible to let P and P_r to execute a file system (FS) input-related system call without resorting to

replication. For instance, an `open` and `read` issued on a *not shared* FS object can be executed by P and by P_r. The saving is about the overhead introduced by T for performing the replication task. We recall that this overhead is not negligible, especially with the current prototype (i.e., at least two `ptrace` system calls have to be executed by T to replicate 4 bytes from P to P_r address space[1].).

2. *SMP.* P and P_r have to be completely executed and this alone corresponds to an overhead of 100%, at least. However, it is worth noting that the main requirement for the whole approach about these processes is that they can synchronize themselves at particular rendez-vouz points. Therefore, before reaching such points, it is reasonable to execute P and P_r in *parallel*. Symmetric multi-processors (SMP) can greatly help in this direction. By scheduling one process, say P, on one CPU and the other, say P_r, on the other CPU, the overhead introduced due to sequential execution of P and P_r on uni-processor (UP) machines can be considerably cut down.

6.1.2 Dynamic Binary Translation

Another drawback of the proposed approach refers to the issues raised in § 3.5. While solutions have been proposed for shared memory and signals management, threads management represents still an open issue. As pointed out in § 3.5.2, to correctly handle situations where co-operating threads which act on shared resources, it should be necessary to (i) schedule a thread L followed by its replica L_r, and (ii) let them execute the same number of instructions (or the same quanta). This should guarantee consistent states for involved shared resources.

A way to achieve this goal is to use dynamic binary translation techniques, such as those offered by the Qemu processor emulator [27]. Moreover, by using dynamic binary translation, it is possible to directly work on *binaries* instead of requiring source code. Diversity is achieved by the translation approach: whenever a pointer (or a memory access) is de-referenced (or made), address relocation is performed. While this approach does not provide meaningful diversity for memory errors protection on a process by itself (i.e., attack-provided memory addresses will be relocated as well), it should be effective with the replication approach considered in this dissertation. In summary, dynamic binary translation would provide:

(i) Transparent diversification by not requiring program recompilation to provide diversity.

(ii) A faster implementation than the one provided by using the `ptrace` system call.

[1]This is the minimum requirement. In fact, the number of `ptrace` invocation is generally higher as system calls invoked by P_r have to fail and P_r status (e.g., system call return code) has to be kept synchronized with P as well. This requires other `ptrace` invocations.

A drawback of this solution is that protection for partial address overwriting attacks is lost and relative addressing attacks are not addressed either.

6.1.3 Program Transformation

Alternatively, instead of operating directly on the binary, it would be possible to improve the protection for partial overwrite and relative addressing attacks, by using lightweight diversification techniques similar to the one proposed in [8]. In particular, since the underlying diversified process replicæ approach can already deal with absolute code and data pointers corruption by providing a *deterministic* protection, it is possible to relax and weak the complexity of the transformations approaches described in [8] yet achieving a strong *probabilistic* protection for this class of memory errors. For instance, a subset of attacks which corrupt non-pointer data could be thwarted by inserting non-overlapping gaps between buffer-like variables of P and P_r.

6.2 Taint-enhanced Anomaly Detection

An open issue in the approach proposed in Chapter 4 concerns the *learning* phase. While it is out of the scope of this dissertation to give answers to this, so far, learning represents an open issue which affects every dynamic learning-based approaches. For instance, it is still unclear to what extent a synthetic learning – which is *attack free* – is as effective as the one performed in the wild – which has no guarantee to be attack free. Moreover, how long to perform the learning phase for is also another question that needs an answer.

Despite this issue, the combination of taint analysis and anomaly detection seem to be effective to provide protection to a broad class of memory errors. Motivated by this and by previous results on taint-enhanced policies enforcement ([89]), we believe that the approach can be extended to provide protection to a broader range of software vulnerabilities (e.g., web-based vulnerabilities). After all, as shown in this dissertation, the anomaly detection or, better, dynamic learning-based approach will help to automatically infer taint-enhanced policies representing the application behavior to be enforced. For instance, it would be possible to:

(i) Leverage on a context-sensitive analysis on taint-enhanced (transformed) PHP interpreter to provide more accurate characterization of sinks invocation.

(ii) Learn policies for SQL injection attacks that – among common attacks – deal with:

- 2^{nd} order SQL injection.
- dynamic construction of SQL query (e.g., fuzzy advanced search).

(iii) Leverage on the learning-based component to infer/learn safe attack pattern usage to constraint false positives and false negatives.

The important thing is not to stop questioning. Curiosity has its own reason for existing.
Albert Einstein (1879 - 1955)

7 Conclusions

Memory errors in C and C++ programs have been known for decades and are one of the oldest classes of software vulnerabilities. Researchers have been working on memory error protection mechanisms for decades. Nonetheless, it seems that this kind of vulnerability is far from being completely defeated.

This dissertation presented two program transformation techniques to provide comprehensive solutions to memory error attacks. Recognizing the effectiveness of artificial diversity, taint-tracking, and anomaly-based detection strategies, we proposed two approaches that are able to deal with a broad class of memory error vulnerabilities. In particular our first approach, *diversified process replicæ* extends the concept of process diversification. So far, diversification has been applied on a process by itself, for instance by adopting address space and instruction set randomization schemes. Our approach, instead, couples diversification and replication together. It applies a form of diversification that involves a process P along with P_r, the process *replica* of itself. By monitoring P and P_r behavior, and by replicating data on particular rendez-vouz points, our technique detects behavioral divergences triggered by memory error exploits. In most and more frequent cases, our strategy gives *deterministic* protection. Moreover, by giving solutions to unwanted side-effects introduced by the approach (e.g., shared memory and signal management), the dissertation provides a first step toward a more realistic deployment of the protection mechanism. *Taint-enhanced anomaly detection*, the second approach proposed by this dissertation, takes advantage of taint-tracking and anomaly detection techniques. The proposed strategy transforms a given benign application P_o into P, a taint-enhanced version of P_o. Then, by coupling taint analysis and anomaly detection, the approach learns a profile $\mathcal{M}$ of the transformed taint-enhanced application, during a so-called training or learning phase. Taint information as well as different models (e.g., structural inference, longest common prefix and data lengths) are used to automatically infer the security policies which represent the behavioral profile of the protected process P. Subsequently, during a so-called detection phase, similarly to any anomaly-based

detection strategy $\mathcal{M}$ is checked against $\mathcal{M}'$, the profile of P observed during normal runs and generated by the same learning rules adopted during the training phase. Should $\mathcal{M}$ be inconsistent with respect to $\mathcal{M}'$, an alarm will be raised.

Both approaches are able to deal with memory error attacks which corrupt code and data pointers, by generally providing a *deterministic* protection. Moreover, by leveraging on taint information and a learning-based approach, the second approach also deals with those attacks which corrupt arbitrary data, as well.

Bibliography

[1] Secure Systems Lab. `http://seclab.cs.sunysb.edu/seclab/`.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS '05: Proceedings of the 12th ACM conference on Computer and co mmunications security*, pages 340–353, New York, NY, USA, 2005. ACM Press.

[3] J. Afek and A. Sharabani. *Dangling Pointer: Smashing the Pointer for Fun and Profit.* Watchfire, 2007.

[4] Ana Nora Sovarel and David Evans and Nathanael Paul. Where's the FEEB? The Effectiveness of Instruction Set Randomization. In *14th USENIX Security Symposium*, August 2005.

[5] Anonymous. Once upon a free()... Phrack Magazine, Volume $0x0b$, Issue $0x39$, Phile $\#0x09$ of $0x12$, December 2001.

[6] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[7] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. In *15th USENIX Security Symposium*, 2006.

[8] Sandeep Bhatkar. *Defeating Memory Error Exploits Using Automated Software Diversity.* PhD thesis, Stony Brook University, December 2007.

[9] Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security*

and Privacy (S&P'06), pages 48–62, Washington, DC, USA, 2006. IEEE Computer Society.

[10] blexim. Basic Integer Overflows. Phrack Magazine, Volume $0x0b$, Issue $0x3c$, Phile $\#0x0a$ of $0x10$.

[11] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. An Efficient Technique for Preventing Mimicry and Impossible Paths Execution Attacks. In *3rd International Workshop on Information Assurance (WIA 2007)*, April 2007.

[12] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Static Analysis on x86 Executable for Preventing Automatic Mimicry Attacks. In *GI SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2007.

[13] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[15] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. B eattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer -overflow attacks. In *Proc. of the 7th Usenix Security Symposium*, pages 63–78, Jan 1998.

[16] D. Bruschi and L. Cavallaro and A. Lanzi. Syscalls Obfuscation for Preventing Mimicry and Impossible Paths Execution Attacks. Technical Report RT 10-06, Università degli Studi di Milano, 2006.

[17] D. Mosberger (main author), S. Eranian, and D. Carter. httperf - HTTP performance measurement tool. http://www.hpl.hp.com/research/linux/httperf/ – Hewlett-Packard Research Laboratories.

[18] M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing Hardware Architectures for Security. In *Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking (held in conjunction with the 33rd International Symposium on Computer Architecture)*, 2006.

[19] Daniel P. Bovet, and Marco Cesati. *Understanding the Linux Kernel, 2nd Edition*. O'Reilly, December 2002.

[20] Daniel R. Edelson. Fault Interpretation: Fine-Grain Monitoring of Page Accesses. In *USENIX Winter*, pages 395–404, 1993.

[21] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[22] Doug Lea. `dlmalloc` Memory Allocator. `http://g.oswego.edu/`.

[23] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Usenix Tech Conference*, 2007.

[24] Elena Gabriela Barrantes and David H. Ackley and Stephanie Forrest and Darko Stefanovic. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.

[25] Elias "Aleph One" Levy. Smashing the Stack for Fun and Profit. Phrack Magazine, Volume $0x07$, Issue #49, Phile 14 of 16, December 1998.

[26] H. Etoh. GCC extension for protecting applications from stack-smashing attac ks (ProPolice), 2003. http://www.trl.ibm.com/projects/security/ssp/.

[27] Fabrice Bellard. QEMU – open source processor emulator. http://fabrice.bellard.free.fr/qemu/.

[28] 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.

[29] J. S. Fenton. Memoryless subsystems. *Computing Journal*, 17(2):143–147, May 1974.

[30] 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.

[31] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2001. USENIX Association.

[32] Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[33] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[34] George W. Dunlap and Samuel T. King and Sukru Cinar and Murtaza Basrai and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2002. http://www.eecs.umich.edu/~kingst/revirt.pdf.

[35] gera and riq. Advances in format string exploitation. Phrack Magazine, Volume $0xb$, Issue $0x3b$, Phile $\#0x07$ of $0x12$.

[36] A. K. Ghosh and A. Schwartzbard. A Study in Using Neural Networks for Anomaly and Misuse Detection. In USENIX *Security Symposium*, 1999.

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

[38] J. A. Goguen and J. Meseguer. Security policies and security models. *sp*, 00, 1982.

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

[40] Hovav Shacham and Matthew Page and Ben Pfaff and Eu-Jin Goh and Nagendra Modadugu and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307, New York, NY, USA, 2004. ACM Press.

[41] Ingo Molnar. Exec-Shield, September 1999.

[42] J. Poskanzer. thttpd - tiny/turbo/throttling HTTP server. http://www.acme.com/software/thttpd/ – version 2.23beta1-3sarge1.

[43] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[44] Juan M. Bello Rivas. Overwriting the `.dtors` section. `http://www.synnergy.net/downloads/papers/dtors.txt`.

[45] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[46] klog. The Frame Pointer Overwrite. Phrack Magazine, Volume 9, Issue 55, Phile 8 of 19, September 1999.

[47] Jingfei Kong, Cliff C. Zou, and Huiyang Zhou. Improving Software Security via Runtime Instruction-level Taint Checking. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 18–24, New York, NY, USA, 2006. ACM Press.

[48] 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.

[49] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[50] Matt "shok" Conover & w00w00 Security Team. w00w00 on Heap Overflows. `http://www.w00w00.org/files/articles/heaptut.txt`, January 1999.

[51] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. pages 79–93, May 1994.

[52] Michel "MaXX" Kaempf. Vudo - An object supertitiously believed to embody magical powers. Phrack Magazine, Volume $0x0b$, Issue $0x39$, Phile $\#0x08$ of $0x12$, December 2001.

[53] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P'07)*, pages 231–245, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[54] D. Mutz, F. Valeur, C. Kruegel, and G. Vigna. Anomalous System Call Detection. *ACM Transactions on Information and System Security*, 9(1):61–93, February 2006.

[55] D. Mutz, F. Valeur, C. Kruegel, and G. Vigna. Anomalous System Call Detection. *ACM Transactions on Information and System Security*, 9(1):61–93, February 2006.

[56] Darren Mutz, William Robertson, Giovanni Vigna, and Richard Kemmerer. Exploiting Execution Context for the Detection of Anomalous System Calls. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, 2007.

[57] A. C. Myers. JFlow: Practical mostly-static information flow control. pages 228–241, January 1999.

[58] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[59] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *NDSS*, 2005.

[60] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting, 2005.

[61] Perl. Perl taint mode. `http://www.perl.org`.

[62] 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.

[63] Gerardo Richarte. Four different tricks to bypass StackShield and StackGuard protection, April 2002.

[64] rix. Smashing C++ VPTRS. Phrack Magazine, Volume $0xa$, Issue $0x38$, Phile $\#0x08$ of $0x10$, May 2000.

[65] S. Forrest and A. Somayaji and D. Ackley. Building Diverse Computer Systems. In *HOTOS '97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 67, Washington, DC, USA, 1997. IEEE Computer Society.

[66] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), January 2003.

[67] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *12th USENIX Security Symposium*, 2003.

[68] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *14th USENIX Security Symposium*, 2005.

[69] scut / team teso. Exploiting Format String Vulnerabilities, September 2001. version 1.2.

[70] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 144, Washington, DC, USA, 2001. IEEE Computer Society.

[71] Solar Designer. Non-executable Stack Path.

[72] Eugene H. Spafford. The Internet Worm Program: An Analysis. Technical Report CSD-TR-823, Department of Computer Sciences, Purdue University, IN, USA, November 1988.

[73] W. Richard Stevens. UNIX *Network Programming: Inter Process Communi cations*, volume 2, chapter 12, page 303. Prentice-Hall, 1999.

[74] Clad "RORIV" Strife and Xdream "ROJIV" Blue. Ret onto Ret into Vsyscalls. `http://seclists.org/bugtraq/2005/Apr/att-0312/ret-onto-ret_en_txt`.

[75] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[76] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[77] Kymie M. C. Tan, Kevin S. Killourhy, and Roy A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection*, 2002.

[78] Kymie M. C. Tan, John McHugh, and Kevin S. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Information Hiding*, pages 1–17, 2002.

[79] The PaX Team. PaX: Address Space Layout Randomization (ASLR). http://pax.grsecurity.net.

[80] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. In *ACM Transactions on Computer Systems*, pages 14(1):80–107, February 1996.

[81] TIS Committee. Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification, May 1995. Version 1.2.

[82] Timothy K. Tsai and Navjot Singh. Libsafe: Transparent system-wide protection against buffer overflow attacks. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, page 541, Washington, DC, USA, 2002. IEEE Computer Society.

[83] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *MICRO 37: Proceedings of the 37th annual*

IEEE/ACM International Symposium on Microarchitecture, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[84] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. 4(3):167–187, 1996.

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

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

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

[88] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. volume 00, page 260, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[89] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced Policy Enforcement: a Practical Approach to Defeat a Wide Range of Attacks. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[90] Y Younan, W Joosen, and F Piessens. Code injection in C and C++: A Survey of Vulnerabilities and Countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, Belgium, July 2004.

[91] Yves Younan, Wouter Joosen, and Frank Piessens. Efficient Protection against Heap-based Buffer Overflows without Resorting to Magic. In *Eighth International Conference on Information and Communication Security (ICICS)*, December 2006.

[92] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended Protection against Stack Smashing Attacks without Performance Loss. In *Twenty-Second Annual Computer Security Applications Conference (ACSAC)*, December 2006.