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

Diversified Process Replicæ for Defeating Memory Error Exploits

Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi

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

RAPPORTO TECNICO RT 14-06

Technical Report RT 14-06, Dipartimento di Informatica e Comunicazione, Università degli Studi di Milano, 2006.

Diversified Process Replicæ for Defeating Memory Error Exploits

Danilo Bruschi, Lorenzo Cavallaro* and Andrea Lanzi
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
{bruschi, sullivan, andrew}@security.dico.unimi.it

Abstract

Diversity plays a crucial role for the survivability of every biological species and, quite recently, the concept, has also been applied to computer programs.

An interpretation of the notion of software diversity, is based on the concept of diversified process replicæ. *We define p_r as the replica of a process p which behaves identically to p even if it presents some "structural" diversity from it. This makes possible to devise mechanisms for detecting memory corruption attacks in a* deterministic *way. In our solution, a process p and its replica p_r only differ in their address space which is properly diversified, thus defeating* absolute *and* partial overwriting *memory error exploits.*

We also give a complete characterization and we propose a solution of shared memory management, one of the biggest practical issue introduced by diversified process replicæ. Preliminary ideas on how to deal with synchronous signals delivery between p and p_r are faced as well.

A proof-of-concept prototype working in user space has been implemented. Our experimental results show a 68.93% throughput slowdown on a testbed web server application in the worst-case, while only a 1.20% throughput slowdown has been obtained in the best-case.

1 Introduction

Diversity plays a crucial role for the survivability of every biological species and, quite recently, the concept, has also been applied to computer programs [16, 20, 8, 19, 11, 18, 23]. Researchers in the computer security field started to apply different kinds of software transformations such as address space layout randomization [23, 19], instruction set randomization [8, 11] and several forms of more general program transformation techniques [20] in order to defeat or at least 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 [9, 17, 21]) 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 [21]) or are not so effective in protecting a process or, again, cannot protect from all the existing memory corruption attacks [1, 13].

A different interpretation of the notion of software diversity has been provided by Cox *et. al* in [2]. 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 *certainty*.

In this paper we propose an improved version of the idea and the prototype described in [2] which, beside being simpler, is able to deal with a broader range of memory error exploits. More precisely, in our solution, a process and its replica only differ in their address space layout, while in [2]

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

the two processes were diversified by two factors, namely the address space and the instruction set. Even if our model is simpler, we are able to solve a series of problems which a previous model has not been able to completely deal with. In particular we make the following contributions:

1. we devised a model which defeats memory error exploits targeting absolute memory addresses as well as those which *partially overwrite* a memory address. The former 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, while the latter permits an attacker 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 [10], this class of attacks are becoming a serious real security threat.

2. protection is obtained by using only one *diversity*, namely non-overlapped processes address spaces, in contrast with [2], no matter which malicious code exploitation technique is used. This has the advantage of making the whole framework simpler while still defeating a broader range of memory corruption attacks.

3. we give a complete characterization and we propose a solution of writable shared memory management, one of the biggest practical issue introduced by diversified process replicæ approach which has to be solved to permit a real and practical deployment of the method. Moreover, preliminary ideas on how to deal with synchronous signals delivery between a process and its replica are faced as well;

4. we developed a prototype user space proof-of-concept using the `ptrace` system call, on a little endian 32-bit Intel Architecture host running 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. Moreover, the model can also form a basis for other security-related applications, such as malware collector and Host Intrusion Detection System (HIDS) training set learnt "in the wild".

The paper is organized as follows. In § 2 we recall preliminaries concepts about the "objects" targeted by memory error exploits, the Executable and Linking Format (ELF) specification [25] as well as reminding fewer observations on the address space of a process. § 3 shows some related works while § 4 outlines the idea of diversified process replicæ as well as the framework we devise and the diversification approach we adopt. § 5 focuses the attention on the mechanisms our framework adopts for achieving protection from memory error exploits. Effectiveness about the approach are given in § 6 whilst § 7 faces some practical issues such an approach may arise, such as shared memory, signals and non-determinism. Experimental results show in § 8 that the process replication with diversification approach yields a 68.93% throughput slowdown on a test-bed web server application on a worst case, while exhibiting only a 1.20% throughout slowdown on a best case. § 9 gives conclusion and places fewer observations about future works.

2 Preliminaries

In this section we recall on fewer concepts about the data targeted by an attacker for diverting a process execution flow, as well as reminding fewer concepts about the Executable and Linking Format (ELF) [25] specification. Moreover, some remarks on a process address space layout are given at the end of the section as we think they might be useful throughout the rest of the paper.

2.1 Control-Flow Diverting Targets

An attacker can count on several different techniques in order to take advantage of a vulnerability. However, the main goal is nearly always the same, that is, to execute some malicious arbitrary code or to bypass some security checks an application makes use of. We are concerned with those vulnerabilities that require the attacker to "tamper" particular memory "items" in order to let him accomplish such a goal. In [19, 20], Bhatkar *et al.* made a wide and precise classification of the memory items that are usually suitable targets for memory error exploits. For the sake of simplicity, we summarize that classification in the following:

- code pointers; stack return addresses, stack, data or bss function pointers, Global Offset Table entries, C++ virtual pointers table, are examples of such pointers.

  Overwriting one of those pointers with an attacker supplied 32-bit value, usually lead to execution of arbitrary code. A partial overwriting, instead, can usually permit to bypass security checks.

- data "items" recently better defined as non-control-data [3], that may be ulteriorly subdivided in:

  – pointers. In certain situations they might be the target of an overflow and, if subsequently

used, can permit an attacker to "write-anything-anywhere" in the address space of the victim process. The worst consequence, again, is arbitrary code execution;

– non pointers. If used as arguments of security critical functions (or system calls) can be targeted by malicious code to change the semantic of (or to bypass) the deployed security mechanism.

2.2 Executable and Linking Format

The ELF specification [25] 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 § 2.3).

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.

2.3 Process Address Space

The address space of a user-space process consists of all the virtual memory addresses a process may access [6]. 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` ([25]) 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.

3 Related Works

Forrest *et al.* suggested preliminary ideas for building diverse computer systems [18]. 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.

Following such an idea, others researchers faced the problem of providing diversity to computer systems.

In [23], 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

[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.

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 a more fine-grained randomization. However, since run-time relocation is generally not possible, information leakage attacks or the not-so-strong effectiveness of ASLR on 32-bit Intel Architecture [13] can still defeat or thwart these protection mechanisms.

Other address obfuscation techniques have been proposed in [20, 19] by Bhatkar *et al.* as a particular form of program transformations to combat memory error exploits targeting both control and non-control data. Such approaches differ from the one proposed in [23] since they aim at providing a more fine-grained address space obfuscation. The objectives of obfuscation transformations are to randomize the absolute locations of all code and data in order to achieve protection from memory error exploits targeting memory address holding control-data (both absolute and partial overwrite), and to randomize the relative distance between different data objects in order to defeat relative addressing attacks, which are a subclass of non-control data ones [3]. 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 [20], where a source-to-source transformation on C programs has been developed to produce self-randomizing programs.

All the aforementioned techniques share a common concept: they provide diversity on a process itself and thus, they provide a *probabilistic* defensive mechanism that, in general, cannot provide *certainty* in protecting from memory errors exploits.

Recently, Cox *et al.* faced in [2] the concept of process replication with diversification herein improved. Their approach is based on the adoption of two different variations techniques, namely address space partitioning and instruction set tagging on a process and its replica. The former is used to provide protection against memory corrupting attacks that involve direct references to absolute addresses, while the latter is used to provide protection from code injection attacks. In this paper, we show that the address space partitioning variation is sufficient for guaranteeing protection against memory corrupting attacks that involve direct reference to or overwriting of absolute addresses (either partial or not) if *properly enhanced* (see § 4.2). Thus, it is our believe that instruction set tagging variation becomes quite useless. Moreover, the model proposed in [2], as ours one, introduces some unwanted issues that can negatively influence a practical "real" deployment. For example, 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.

4 Process Replication with Diversification

Process replication aims at creating 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 (§ 5) and diversification approaches (§ 4.2) 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 a. Since p and p_r are artificially diversified (non-overlapping address space) and replicated, it is impossible that a is suitable for both processes. Any attempt to use 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 § 4.2.

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

4.1 Model Framework

The model framework is represented in Figure 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.

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 catching for any anomalous conditions (see § 5).

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

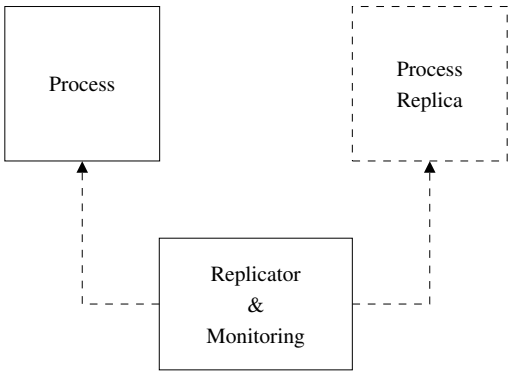

Figure 1. Model Framework

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 5 (see § 6 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.

4.2 Non Overlapping Processes Address Spaces

The diversity approach we adopted aims at providing a non-overlapping address space between a process p and its replica p_r. 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 2. As reminded in § 2, 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 § 4, partial address overwrite attacks can still be successful even when adopting non-overlapping address spaces 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 defeating or at least strongly 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.

4.2.1 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:

- load p_r starting at a custom address different from the one defined in the ELF ABI [25]; 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;

  Indeed, to achieve full non-overlapping address space, other regions have to be mapped at different addresses too that is stack and memory mapped area (heap comes after the `.bss` segment so, it can be transparently handled by the modified linker script). In order to accomplish this task and to be as transparent as possible with respect to the diversified executable object, a kernel patch is in charge of changing the base addresses used for stack and `mmap`'d area.

- 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 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 § 6).

4.2.2 Dynamically-linked Binaries

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.

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

[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.

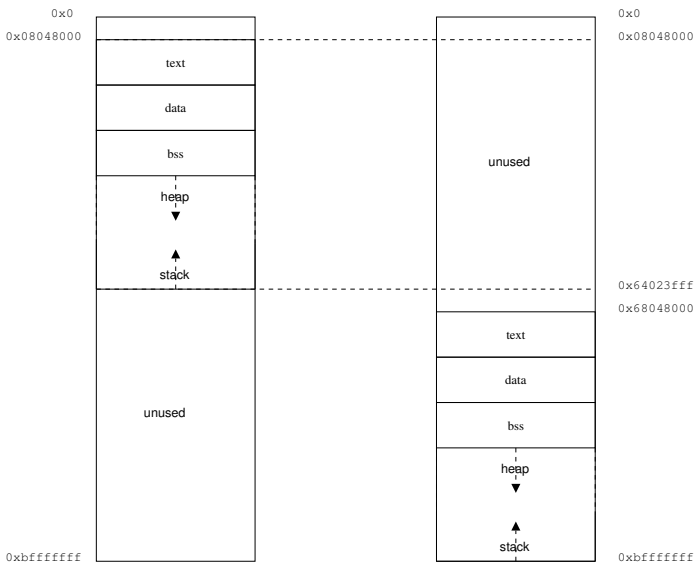

Figure 2. Diversified Process Replicæ

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 can be transparently diversified by the same kernel patch that is in charge of changing the stack and `mmap`'d areas since shared objects are mapped onto the latter one (§ 4.2.1). 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.

We chose 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:

1. 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);
2. shift the segment m just mapped by k bytes;
3. 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];
4. 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 kernel to correctly un-map the region, using the information stored in 1.

It is worth noting that it should not be necessary to update the o's ELF related structures to reflect the "new" *relative* position. In fact, shared objects have position independent code (PIC) and thus they do not hold absolute memory references. However, they need to honor *relative* addressing between the loaded segments and, as long as the shift operation is performed on all the loadable segments (`PT_LOAD`) of o, correct behavior is guaranteed.

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 test 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.

[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.

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.

Unfortunately, one 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.

Actually, our work in progress prototype still does not support all of these features but we are currently working on a toy application in order to see the viability of such ideas.

5 Replicator Module

The replicator and monitoring component t of the framework depicted in Figure 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 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) executes a process p and its replica p_r which has been previously diversified (see § 4). 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 this 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) 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 control-data values 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 § 7.1). A typical example is repre-

[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.

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

sented 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 § 7.1.

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 id to 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 up an alarm or terminating p (see § 4. 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 [27], 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.

Recent research [3], however, showed that indeed is not always necessary to execute a system call to cause damage. Even if some non-control-data attacks can currently be caught by our approach, others are not. This is, unfortunately, a limitation of our current method.

[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.

6 Effectiveness

In order to validate the goodness 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 correct 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 is know as an Impossible Paths Execution (IPE) attack [26, 10, 4].

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.

Obviously a lot of sophisticated exploitation techniques exist, but for exposition purpose we consider only the simplest ones. Figures 3 and 4 depict code snippets showing respectively a stack-based buffer overflow vulnerability and a security check that can be bypassed by performing an IPE[10]. In particular, Figure 4 depicts a situation where an attacker, camouflaged as a regular user, enters the true branch (lines

[9] 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. Moreover, we are not considering any interpreted language.

[10] Example showed in Figure 4 was first proposed by [10] and slightly modified in [4].

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

Figure 3. A typical stack-based buffer overflow vulnerability

```
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    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  }
```

Figure 4. A typical security check that can be bypassed with an IPE attack

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 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.

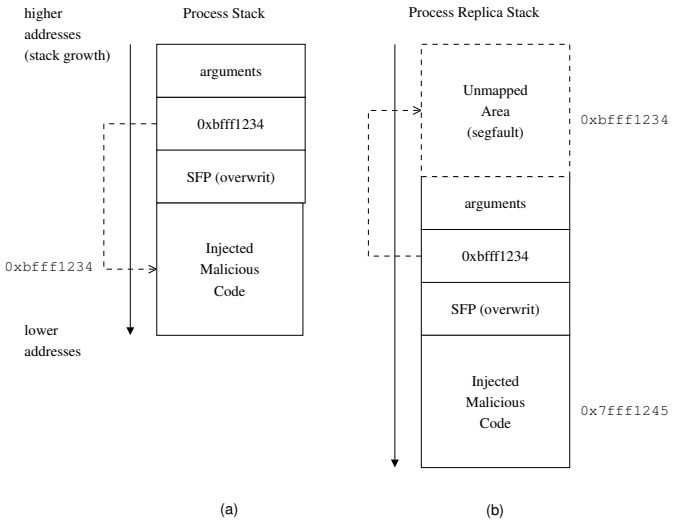

Figure 5. Diversified process replica for defeating *absolute* memory errors exploits

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 § 4.2. For example, consider the code snipped reported in Figure 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 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 regions 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.

7 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 situations, that we have to cope with in order to successfully and broadly deploy such a defensive mechanism.

7.1 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 § 5, 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 § 7.1.1, 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`, ...)[11].

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 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 [2]. As we will briefly see, we also leverage on one particular harmless "type" of shared memory that permit us to put the basis for solving the issue the process replication model introduces.

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 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;
   (b) *shared mapping*, that provide 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 involved in the IPC and viceversa.
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*, that resembles the *intra-process communication* mapping provided by the `mmap`-based approach (point 2);

(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 § 7.1.1, 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 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 the process replicæ model.

[11] 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.

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 § 7.1.1 shows such a situation (`mmap`-based (point 1b) approach) in a related-only processes scenario with no children (only p and p_r).

7.1.1 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[12] (`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 § 7.1). As previously noted, if not properly handled this 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[13]. 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.

To achieve our goal in order to propose a possible solution to the shared memory issue we remark on the following assumption that should hold among every *real* processes (that is, not a process and its replica) that are making use of using shared memory.

Assumption. *"[...] 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" [22]*

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).

7.1.2 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 § 7.1.1 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.

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

[13] Indeed, as noted elsewhere (§ 5), 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 idea here is simple: 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. As pointed out in the previous section, in fact, an intra-process communication (private mapping) is harmless. Thus, it is sufficient to let p_r perform a *private* mapping and because of the considered scenario of related-only processes this makes possible to ensure that the view of r is always consistent. Moreover, this makes possible to let p and p_r exhibit a consistent identical behavior.

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).

To make things easier we also operate without the Assumption given in § 7.1.1. In fact, the only involved processes here are p and p_r and it is necessary only to ensure that there is not any form of IPC between them. Technical details are given in § 7.1.5.

7.1.3 Unrelated Processes

This scenario is more tricky because here, 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.

By observing the example shown in § 7.1.1, it is possible to note that the issue is due to the fact that p and p_r start an unwanted form of IPC. In that example, as already pointed out, a straightforward solution is to force p_r to create a *private* mapping, thus disrupting any unwanted existing IPC form between p and p_r. This task can easily be performed by t which intercepts any system call invoked by the monitored processes (§ 5).

As in the previous scenario, even here we have to grant only private mapping to p_r (no inter-process communication through shared resource between p and p_r but only intra-process) to achieve a *preliminary* data and behavioral consistency.

However, this is a necessary but not sufficient condition because an external process e might modify the resource r. This has the direct consequence that while p will see the modification, p_r will not and this might again lead to a behavioral divergence between them. We can call this situation a false positive of the model, where processes are "stalled" or terminated and clearly this is something we do not want to happen.

We need to let p_r always operate on an *up-to-dated* view of the shared resource r and to achieve this, we leverage on the Assumption given in § 7.1.1 (must hold) which as a consequence provides the following:

- it makes possible to decide *when* to perform the refresh operation. In other words, we are looking for a rendez-vouz point where p and p_r can synchronize themselves like in the normal case where only system calls were involved.

- with this new rendez-vouz point we wait untill p "acquire a lock" for r. The given Assumption avoids data inconsistency during the *refresh* operation in which it is possible to make p_r private mappings up-to-dated with respect to the current status of the resource r.

We remark on the fact that the Assumption given in 7.1.1 is not a strict requirement. Without it, in fact, poorly written processes that make use of shared resources are likely to exhibit anomalous behavior.

The main point is *how* and *when* to update the memory regions where r is referenced at. The answer to "when" can be partially given by analyzing the synchronization mechanisms a process p can use for gaining "mutual" access to r. Knowing such approaches can help in finding the answer to "how". We see two main different kinds of methods to obtain synchronization between processes, that is, *shared memory-based* and *system call-based* that we address in the following.

shared memory-based synchronization

It is possible to achieve synchronization for having granted mutual access to a shared resource r by *atomically* accessing a shared variable, usually using library functions like `sem_wait`, `pthread_mutex_lock` and `pthread_mutex_trylock`. Unfortunately, usually these functions do not execute any system call[14] but we need to find a way to decide when to perform the refresh operation at the right time without causing data inconsistency and processes behavioral divergence.

Preliminary results we conducted on these synchronization functions, showed that they end up by executing the assembly instruction `cmpxchg src, dst` (or a similar instruction) usually preceded by the `lock`[15] prefix to basically turn the instruction into an atomic instruction. In its general form, this assembly instruction atomically performs the following actions, where `accumulator` is the IA-32

[14] Please, note that `sem_wait` is actually a C library function that make use of the `futex` system call (Fast Userspace Locking system call) as well as atomic assembly instruction like `cmpxchg`.

[15] Indeed, there are many other instructions which `lock` can be applied to (see [14]).

eax register (or one of its "subpart") and ZF represents the Zero Flag IA-32 register [14].

```
1  if accumulator = dst
2     then
3        ZF ← 1;
4        dst ← src;
5     else
6        ZF ← 0;
7        accumulator ← dst;
8  fi;
```

For instance, pthread_mutex_lock on a GNU/Debian system, stable release, using glibc-2.3.5, makes use of the following instructions to atomically acquire a mutex, where %esi holds the address of the mutex (AT&T assembly syntax).

```
1  mov    0x8(%ebp), %esi
2  ...
3  xor    %eax, %eax
4  mov    $0x1, %ecx
5  lock cmpxchg %ecx, (%esi)
6  jne <loop_till_get_lock>
7  <mutex_acquired>
```

Even if we might also find cmpxchg/je pairs it is our believe that they might be handled in a similar way, accordingly to their semantic.

It is also worth noting that there are several kind of synchronization data types, such as mutex, semaphores, monitor, and so on. A precise characterization of them have to be done in order to understand how to cope with all the possible situations. It is our believe, however, that there are just fewer "low-level" situations to face since other "high-level" constructs make use of the low-level ones to achieve their goals.

Coming back to our initial issue, we need to guarantee that p_r mapping is up-to-dated with respect to the current content of r. The instruction used for acquiring a lock for r gives us information on whether the lock is successfully acquired or not. If it is, then it is possible to update p_r's memory area which refers to r (private mapping). Actually, as we will see in the following, the refresh operation has to be deferred until p and p_r reach a safe point.

To achieve our goal, we propose an approach similar to *fault interpretation* [7].

Fault Interpretation

The idea is simple: we exploit the CPU page fault (PF) exception to know whenever p is writing into given memory page(s) m[16] of its own which refers to the shared resource r. To achieve this goal, we mark m of both p and p_r as read-only. This task is performed by t which intercept p and p_r system calls (see § 5) whenever the mapping is created. Further technical details are given in § 7.1.5.

In particular, whenever p or p_r want to write to their shared mapping m, they acquire a lock (Assumption given in § 7.1.1) in m. Since m is read-only, the CPU will raise a PF exception which cause a segmentation violation signal (SIGSEGV) to be delivered to the faulty process (caught by t). Roughly speaking, t waits until p and p_r reach this new rendez-vouz point triggered by the PF. The first time the PF is raised is because p and p_r want to acquire a lock. At this point:

1. t releases m_p protection (i.e., it gives read/write permission), let p execute the lock-type faulty instruction (ptrace single-step), and re-protect m_p.

2. t interprets the outcome of the lock-type instruction, and either:

   (a) it *refreshes* p_r's shared memory mapping of r *only if* the lock was successfully acquired and the shared region was marked as *unlocked*. Moreover, t marks a *lock* meta-data information we use bound to m_p and m_{p_r} to true.

   Actually, it might be noted that we need to defer the refresh operation if the shared memory area used to acquire the lock differs from the one which r is being mapped at or if a system call-based synchronization approach is being used. We speculate on this in § 7.1.4 where we sketch the steps towards a generic solution. Or

   (b) it let p_r skip the lock-type instruction without performing any update operation of r, if the lock was not acquired, or

   (c) it marks m_p and m_{p_r} as unlocked. Actually, this step is pretty useless in this scenario (Assumption given in § 7.1.1, no system call-based synchronization and the shared memory area used for acquiring the lock holds r as well) while it will become necessary for finding a generic solution (§ 7.1.4).

   In any case, t arranges to let p and p_r continue with their execution;

3. t executes every non lock-type instruction issued by p and p_r that tries to write into a shared region mapped by them performing the same steps as carried out in 1.

It is worth noting that we want p_r to execute the code in its critical section because this way we permit it to execute instructions that might also modify its private state.

[16] Whenever needed, we will use m_p and m_{p_r} to refer to p's and p_r's shared mapping respectively.

system call-based synchronization

System call-based synchronization approach to acquire "mutual access" to a shared resource r is less tricky to deal with. In fact, system calls invoked by p and p_r can be treated and managed like all the other system calls, as explained in § 5.

If it were possible to know *which* system call s is used for granting synchronization to a given resource r, then it would be possible to perform the refresh operation whenever the tracer t handles s. Unfortunately, if multiple shared resources are present it would be rather hard to infer which system call is responsible for such a synchronization (and, however, t might not infer in a good way).

However, we achieve the refresh operation by exploiting the fault interpretation approach above introduced. We further speculate on this scenario in the next Section.

7.1.4 Toward a Generic Solution

So far it should be clear how to deal with certain shared memory scenarios, but it is still not so clear how to discriminate and know if p makes use of a syscall-based synchronization approach or a shared-memory one.

At first glance, it seems that it should be necessary to know which kind of synchronization p (and so p_r) makes us of. In fact:

- *syscall-based synchronization*. It poses almost no particular problem. Every memory access in the shared mapping causes a fault which indicates it is safe to refresh the shared memory (we are operating under the Assumption given in § 7.1.1). Should we perform a refresh operation for *every* writable shared memory access? Which shared memory segments this "locking primitive" refers to? These are plausible questions for which we try to give acceptable answers in the following;

- *shared memory-based synchronization*. Again it should pose no particular issue in the way it has been addressed. However, as above noted, the approach seems to not work if the shared memory area used by the locking instruction differs from the one which refers to r.

It might be argued that, since we are under the Assumption given in § 7.1.1, the aforementioned steps carried out in by the Fault Interpretation approach could be simplified. In fact, a naive solution might be to make a refresh at every memory access to r that is not a `lock`-type instruction. However, the major drawbacks of this naive approach are that could generate too much overhead and it does not work with a system call-based synchronization approach and if the shared memory area used for acquiring a lock differs from the one which refers to r.

A more generic solution makes use of the following observations, derived from the Assumption given in § 7.1.1: in its simplest form, an operation on a shared resource r can be seen as a regular expression pattern lw+u where l and u identify respectively the lock and unlock operation (either syscall-based or shared memory-based) and w+ is a regular expression pattern that identifies one or more write access to the shared resource r (we are not considering read-only accesses because are not of interest).

Whenever t encounter a l pattern it stores information about the type of synchronization l represent. Moreover, t associates every shared memory area obtained by p with some meta-data, such as a boolean *lock* variable and a set representing *active* l patterns (that is, l pattern that are not "balanced" by a corresponding u pattern).

Whenever a w pattern is encountered, t checks which shared resource mapping r this w refers to. Afterwards, it checks whether the corresponding *lock* variable is true. If it is not, t binds all the active l patterns encountered so far to r's meta-data, performs a *refresh* of p_r's shared area and set the corresponding *lock* variable to true (t interprets, to some extent, the outcomes of a synchronization attempt). Otherwise, it means that this w pattern does not represent the first memory access in the shared memory area and thus it is operating on an already up-to-date view of the shared resource r.

Whenever a u pattern is encountered, t looks up the corresponding meta-data set of active l. If at least a match is found, the *lock* variable of r is set to false.

Even if this approach seems to be viable, we are still unaware whether all the active l patterns are necessary or redundant to get a correct synchronized access to r. As we the naive approach, we are still conducting experiments to assess how this method impacts on the performances.

We are currently investigating on the possibility to spot race conditions if the Assumption would not hold using the our approach.

7.1.5 Shared Resource: Technical Details

No matter on the resource being involved, i.e., anonymous or non-anonymous mapping, the tracer t has to perform several actions in order to avoid IPC between p and p_r as well as guaranteeing an up-to-date version of the shared resource on which p_r will work on.

We remark that intra-process communication (private mapping) has to be established even in the related-only processes best-case scenario to guarantee data consistency. On the other hand, the unrelated processes worst-case scenario also requires to refresh the private mapping to guarantee processes behavior consistency.

Intra-Process Communication (Private Mapping)

Here, the task of t is fairly easy. The action to be performed depends on the type of shared mapping the process is requesting:

- *non-anonymous mapping*. A private mapping which avoids IPC form between p and p_r is obtained by changing the `MAP_SHARED` flag of every `mmap` system call invoked by p_r, which is already intercepted by t (§ 5), with `MAP_PRIVATE`;
- *anonymous mapping*. As above but the involved system call is `shmget` and the flag is `IPC_PRIVATE` (which actually is a special key and not properly a system call flag).

Updating the Shared Resource

Here, the task of t is more tricky. For this purpose, in fact, in order to update the memory mapping of p_r, t injects code into p_r's address space. Moreover, t retrieves all the information (data) needed to correctly perform such operations. The action to be performed depends on the type of shared mapping the process is requesting:

- *non-anonymous mapping*. Using the `ptrace` system call, t injects the code to let p_r invoke the the right system calls to correctly update the involved existing mapping, with respect to the current status of the shared resource r.

  In particular, the mapping is updated by invoking the following system calls:

  1. `munmap`, to destroy the existing mapping;
  2. `open`, to open the FS object o which the shared mapping referred to. t takes care of storing the object name and the flags originally used by p_r to perform the same task (it suffice to correlate the file descriptor obtained by the `open` system call, subsequently used by the `mmap` system call);
  3. `mmap`, to establish a new fresh up-to-date private mapping (`MAP_PRIVATE`) with permissions suitable for the techniques explained in § 7.1.3. It is worth noting that the mapping has to be obtained at the same original virtual memory (VM) address using `MAP_FIXED` `mmap` flag, otherwise all p_r references into m will be dangling and invalid;
  4. `close`, to get rid of the file descriptor obtained by the previous `open` used by the `mmap` system call. This has the effect of keeping the o reference count coherent with the value it has to be;

- *anonymous mapping*. The situation is similar to the non-anonymous mapping with the difference that the involved system call is `shmdt` (instead of `munmap`), `shmget` (instead of `open`) using `IPC_PRIVATE` to request a private mapping, and `shmat` (instead of `mmap`) with a non-null VM `shmaddr` as system call argument (similar to `MAP_FIXED` of `mmap`).

Moreover, in both cases, t has to act as a proxy between the shared resource r and p_r mapping. For this reason, t attaches itself to the anonymous mapping as well in order to be able to replicate the data into p_r address space. As a direct consequence of this and as a current limitation, if address-derived data are involved in the shared resource r, then the refresh operation will make things break on p_r. We are currently working on solution to deal with this subtle issue.

7.2 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 § 5).

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.

Alternatively, when necessary, as shown in previous works ([12, 24]), we can leverage on CPU specific counters (`branch_retired`) and on the adopted diversification approach (§ 4.2) to turn an asynchronous event like a signal delivery to a synchronous one, even if absence of rendez-vouz points.

We are currently studying on the feasibility of the idea and on its impact on the performances.

We also believe that, non-determinism situation should not pose a problem at all. 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. 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, like noted in § 7.1.

8 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, 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 monitoring `thttpd` [15], a small and fast web server. Moreover, `httperf` [5], an HTTP benchmark utility, has been used on three client hosts to assess the throughput slowdown on a 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.

Table 1 summarizes the experimental results we achieved. In particular, we were quite surprised by the 1.20% throughput slowdown since, it is our believe that, due to the nature of the idea and of the PoC implementation, we were expecting a more heavy performance impact and network slowdown mainly caused by the need to simulate some system calls, such as the `read`. 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 § 5, it has to replicate data from one process to its replica, without actually let the replica 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 VFS object "parts" onto the process address space. Moreover, the web server make use of a cache system to avoid duplicate mapping or reading of VFS objects which yield good performance in our test cases.

In order to be as much complete as possible and to better assess the throughput slowdown caused by the replication approach, we modified `thttpd` in order to force it to either use any combination of `mmap` and (simulated) `read` syscall with caching facility or not. Table 1 reports the combination we obtained and, as we expected, we report a throughput slowdown of 43.78% till 68.93% for non caching read operations on a 7.5MB and 98MB web site, respectively.

It is worth noting that the slowdown inducted 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.

9 Conclusions & Future Works

The notion of process replication with diversification herein faced, gives the opportunity for detecting a broad range of memory error exploits targeting absolute addresses overwriting as well as *partial overwriting* ones. In fact, by carefully ensuring (i) non-overlapping address spaces between p and p_r, and (ii) *different relative distances* between p and p_r address spaces, it is possible to obtain complete protection from these memory errors with *certainty*, in a deterministic way.

A characterization and a practical solution for the management of writable shared memory mappings, one of the main practical issue the process replication approach may suffer, is described. Preliminary ideas on how to deal with synchronous signals delivery between p and p_r are faced as well.

Moreover, in order to validate the goodness and effectiveness of the approach herein proposed, a proof-of-concept prototype working in user space has been developed. Experimental results report a 68.93% throughput slowdown on a testbed web server application in the worst-case, while only a 1.20% throughput slowdown has been obtained in the best-case.

Our future works are currently focused on providing a full implementation of our proof-of-concept prototype as well as to valuate the theoretical and practical feasibility of the others solutions and scenarios. In fact, as noted at the beginning of the paper, even if the performance results might not seem enthusiastic at first glance, and there are some technical issues to be completely solved as well, conceptually speaking the idea is correct and seems to be a viable way towards systems survivability. Moreover, the model can also be exploited as a basis for others security-related applications, such as malware collector and to build a Host Intrusion Detection System (HIDS) training set "in the wild".

#	Throughput	MB/s (real system)	MB/s (diversified process replica)	% 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	3.83%
4	`thttpd` (read-nocache)	12603.7	7086.3	43.78%
5	`thttpd` (read-nocache-single)	9134.5	2838.1	68.93%

Table 1. Experimental results

Acknowledgement

We would like to thank Prof. R.C. Sekar for his valuable comments on this work and for giving us suggestions for dealing with signals management.

References

[1] 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.

[2] 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.

[3] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *14th USENIX Security Symposium*, August 2005.

[4] 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.

[5] 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.

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] George W. Dunlap Samuel T. King Sukru Cinar Murtaza Basrai 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`.

[13] 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.

[14] Intel. *IA-32 Intel© Architecture Software Develop er's Manual*, volume 2A: Instruction Set Reference, A-M. Intel, June 2006. `http://download.intel.com/design/Pentium4/manuals/25366620.pdf`.

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

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

[17] 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.

[18] 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.

[19] 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.

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

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

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

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

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

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

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

[27] 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.