Diversified Process Replicæ for Defeating Memory Error Exploits

Danilo Bruschi Lorenzo Cavallaro* 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

Abstract

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 but has some "structural" diversity from it. This makes possible to detect memory corruption attacks in a* deterministic *way. In our solution, p and p_r differ in their address space which is properly diversified, thus defeating* absolute *and* partial overwriting *memory error exploits.*

We also give a characterization and a preliminary solution for shared memory management, one of the biggest practical issue introduced by this approach. Speculation on how to deal with synchronous signals delivery is faced as well.

A user space proof-of-concept prototype has been implemented. Experimental results show a 68.93% throughput slowdown on a worst-case, while experiencing only a 1.20% slowdown on a 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 [14, 18, 7, 17, 10, 16, 21]. Researchers in the computer security field started to apply different kinds of software transformations such as address space layout randomization [21, 17], instruction set randomization [7, 10], and several forms of more general program transformation techniques [18] in order to defeat or at least strongly thwart memory error exploits. By memory error exploits we mean those techniques that can be used to exploit a particular memory error vulnerability (see for example [8, 15, 19]) by overwriting, and thus corrupting, particular memory locations. The final attack purpose is usually 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 [19]) or are not so effective in protecting a process or, again, cannot protect from all the existing memory corruption attacks [1, 12].

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, its replica p_r is a process 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] the two processes were diversified by two factors, namely the address space and the instruction set. 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.

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

The former attacks class refers to all those exploitation techniques an attacker may use to corrupt a particular memory object with an absolute memory address value with the final intent to hijack the process execution control flow. The latter attacks class, instead, permits an attacker to partially overwrite a memory object (usually the least significant byte(s)), thus allowing a relative execution flow hijacking. This latter class of attacks, generally known as *Impossible Path Execution* (IPE), can permit an attacker to bypass critical application-based security checks. Even if at first glance it might be argued that IPE attacks are not so realistic, as pointed out in [9], this class of attacks can become a serious real security threat;

2. protection is obtained by using only one *diversity*, namely non-overlapped processes address spaces, no matter what memory error exploitation technique is used. This has the advantage of making the whole framework simple while still defeating a broader range of memory corruption attacks;

3. we give a complete characterization and we propose a preliminary solution about 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, we also speculate on how to deal with synchronous signals delivery between a process and its replica;

4. we developed a prototype proof-of-concept running in user-space 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.

The paper is organized as follows. § 2 shows some related works while § 3 outlines the idea of diversified process replicæ, the framework we devised, the diversification approach, and the replication mechanism we adopted. Practical issues, such as shared memory, signals and non-determinism, are faced in § 4. Experimental results show (§ 5) that the process replication with diversification approach give out a 68.93% throughput slowdown on a testbed web server application on a worst case, while exhibiting only a 1.20% throughout slowdown on a best case. Conclusions and fewer considerations about future works are given in § 6.

2. Related Works

Forrest *et al.* suggested preliminary ideas for building diverse computer systems [16]. 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, as a direct consequence of this observation, 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 [21], 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. 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 [12] can still defeat or thwart these protection mechanisms.

Other improved address obfuscation techniques have been proposed in [18, 17] 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 [21] 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 might be seen as 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 [18], 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 (for instance, information leakage attacks

pose a serious threat to the ASLR approach).

Recently, Cox *et al.* faced in [2] the concept of process replication with diversification. 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 corruption 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/overwriting of absolute addresses (either partial or not) if *properly enhanced* (see § 3.2). 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 preliminary solution that can represent a first step towards a more realistic model usage.

3. 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 diversification approaches (§ 3.2) and the replication actions (§ 3.3) both p and p_r will exhibit the same behavior as long as we guarantee they are executing 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 usually use an attack pattern comprising a given absolute memory address a. Since p and p_r are artificially diversified in their address space (AS) and properly 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 only non-overlapping address spaces, which only modify the base addresses of p and p_r memory segments, were ensured. 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.

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

3.1. 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 the 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.

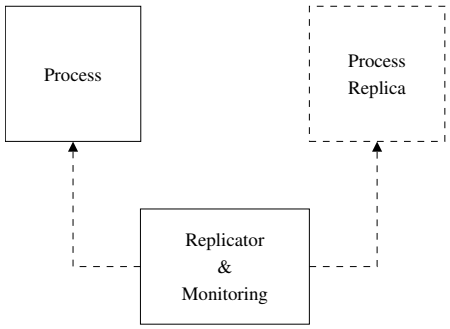

Figure 1. Model Framework

3.2. Non Overlapping Processes Address Spaces

The diversity approach we adopted aims at providing non-overlapping address spaces 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. 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, partial address overwrite attacks can still be successful even when adopting non-overlapping address spaces. 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" p_r's memory segments 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 order to successfully diversify `ET_EXEC` ELF objects [23] we modified the default `ld` linker script for p_r (i) to load p_r starting at a custom address different from the one define in the ELF ABI and used for p, and (ii) to insert "junk" data right at the beginning of the `.text` segment description in the linker script, using the `LONG(`k`)` linker script keyword. This permit to "shift" p_r's memory segment

by k bytes diversifying relative distances between p and p_r address spaces. The same approach has been used for the `.data` segment as well.

Other process memory areas (e.g., stack, heap, `mmap` area) as well as ELF `ET_DYN` shared objects are diversified as well at run time to take full advantage of the diversification approach. Due to space constraints, further details are given in the extended version of this paper [6].

3.3. 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 both processes upon anomalous events (attacks) detection.

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 (basically, some system call have to be simulated by t on behalf of p_r).

An accurate and detailed approach description as well as proof on the effectiveness of our model are given in the extended version of this paper [6].

4. Practical Issues

Unfortunately, even if the idea of diversified process replication is simple and 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.

4.1. 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 and extensively described in the extended version of this paper ([6]), one of t's task is to synchronize p and p_r at each system call (rendez-vouz point). Afterwards, t can perform some sanity checks on the system call p and p_r want to invoke (e.g., system call number, its argument), and eventually perform the replication task, if any, depending on the examined system call. However, no system calls are invoked when shared memory is involved[1]. 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 § 4.1.1, 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 simplicity and due to space constraint, we consider only writable non-anonymous shared memory here and we try to give the main idea behind a prospective solution. A complete characterization of the whole issue and solution are given in [6].

4.1.1 Data Inconsistency and Behavioral Divergence

For this example, let suppose that p creates a readable and writable non-anonymous shared memory segment (read-only shared memory is not an issue), that is a memory segment that maps a file system (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. 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* (IPC).

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. If not properly handled this could lead to *data inconsistency* and *processes behavioral divergence*, as shown in the following code snippet.

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.

```
1       if (*ptr == 'A')
2          *ptr = 'B';
3       else
4          *ptr = 'C';
5       ...
6       // execute something based on *ptr
```

   Suppose the kernel schedules-in p[2] and 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,

[1] We refer here to the *operation performed on* a shared memory segment. Obviously, shared memory segment *creation* requires the use of system calls.

[2] Indeed, 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 of view. Actually, the kernel is in charge of performing the real process scheduling task and all the processes, even p, p_r and t, are involved.

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 but, as shown, it is not. 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.

As shown in the following, in the presence of unrelated processes e that interact with the shared resource r, things can even be worst because we cannot control anyhow e behavior.

To propose a possible solution to this 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" [20]*

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.

4.1.2 A Possible Solution

The aforementioned example does not make use of any synchronization because, without the replication approach, there are no other processes involved. However, as soon as p is replicated into p_r a form of unwanted IPC is established between them. In that example, 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 (§ 3.3).

However, this condition is necessary but not sufficient in presence of an unrelated process e, because e might modify the shared resource r. As a direct consequence, while p will see the modification, p_r will never and this might again lead to a behavioral divergence between the involved processes.

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 § 4.1.1 (must hold) which, as a consequence, (i) permits to define a new rendez-vouz point that makes possible to decide *when* to perform the refresh operation (updates r), and (ii) permits to wait until p "acquire a lock" for r. This avoids data and processes behavioral inconsistency while it permits p_r to have an up-to-date mapping with respect to the current status of r.

Knowing the synchronization mechanisms a process p can use for gaining "mutual" access to the shared resource r can help in finding a solution. We see two main different kinds of methods to obtain synchronization between processes accessing shared memory, that is, *shared memory-based* and *system call-based*. The latter method is handled by t because it resembles an "old-style" rendez-vouz point (§ 3.3) and will be faced in § 4.1.3. Moreover, it makes use of the same solution we propose for shared memory-based synchronization method as well to update r (for further details see the extended paper version [6]).

Memory-based Sync and Fault Interpretation

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[3] 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 some assembly instruction (usually `cmpxchg`) preceded by the `lock` prefix to basically turn the instruction into an atomic one. The instruction used for acquiring a lock for r can give us information on whether the lock is successfully acquired or not ([6]). If it is, then it is possible to update p_r's memory area which refers to r (private mapping). To reach such a goal, we propose an approach similar to *fault interpretation* [5].

The idea is simple: we exploit the CPU page fault (PF) exception to know whenever p is writing into a given memory page m[4] 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 intercepts p and p_r system calls whenever the mapping is created (see [6] for further technical details).

In particular, whenever p or p_r wants to write to their shared mapping m, they acquire a lock in m (actually, this is a simplification which we try to get rid of in 4.1.3). Since m is read-only, the CPU will raise a PF exception which causes a segmentation violation signal to be delivered to the faulty process (caught by t which can distinguish between at

[3]Note that `sem_wait` is actually a C library function that make use of the `futex` system call as well as atomic assembly instruction like `cmpxchg` on the systems of the authors. Moreover, other synchronization primitives like System V semaphores are implemented as system call. System call-based synchronization already gives out rendez-vouz point. However, it is still necessary to correctly update the shared resource r (see 4.1.3).

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

attack attempt or a memory protection violation). 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* in a data structure used by t to describe the mappings. Moreover, t marks a *lock* meta-data information associated to these mappings (m_p and m_{p_r}) to true (actually, we should 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 § 4.1.3). Or, alternatively
   (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 § 4.1.1, 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 (§ 4.1.3).

   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.

4.1.3 Towards a Generic Solution

It might be argued that, since we are under the Assumption given in § 4.1.1, the aforementioned steps 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 § 4.1.1: 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 an l pattern it stores information about the type of synchronization l represent. Moreover, t associates every shared memory area obtained by p with a shared memory meta-data, such as a boolean *lock* variable and a set representing *active* l patterns (that is, l pattern that are not balanced by/paired with 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.

Currently, we are investigating on the viability of the approach that, however, requires a more precise characterization of the synchronization primitives.

4.2. 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.

Actually, since t catches every signal sent to p and p_r, it could delay the signal delivery for a while and it can arrange 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 calls could probably not benefit from this delayed action, but even in this case, the signal can be delivered at a given time chosen by t anyway.

Alternatively, when necessary, as shown in previous works ([11, 22]), we can leverage on CPU specific counters (`branch_retired`) and on the adopted diversification approach (§ 3.2) to turn an asynchronous event, like

a signal delivery, to a synchronous one even in absence of rendez-vouz points.

We are currently studying the feasibility of the idea and 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 ([6]).

5. 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, we developed a user-space `ptrace` proof of concept. The prototype 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` [13], a small and fast web server. Moreover, `httperf` [4], 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, 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 (e.g., `read`) was 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 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 FS 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 FS object "parts" onto the process address space. Moreover, the web server makes use of a cache system to avoid duplicate mapping or reading of FS objects which gave out 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 on or off. Table 1 reports the combination we obtained and, as we expected, a throughput slowdown of 43.78% till 68.93% for non caching read operations on a 7.5MB and 98MB web site, respectively, was obtained.

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.

6. Conclusions and 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 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

#	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

model can also form a basis for other security-related applications, such as malware collector and memory error-free training sets learnt in production environment ("in the wild") for anomaly-based Host Intrusion Detection System (HIDS).

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. 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.

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

[6] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified Process Replicæ for Defeating Memory Error Exploits. Technical Report RT 14-06, Università degli Studi di Milano, 2006.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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`.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

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

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

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

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

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

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