ABSTRACT

BOOKHOLT, CHRISTOPHER G. Address Space Layout Permutation:
Increasing Resistance to Memory Corruption Attacks. (Under the direction of Jun Xu).

A key problem with current address randomization techniques is their use of randomly sized pads that shift the location of critical memory regions. Pads limits the potential of existing techniques because pads are unused space. To increase protection, pad size needs to increase, thereby wasting additional address space. The relationship between protection and pad size forces system designers to choose between security and address space conservation.

This thesis proposes Address Space Layout Permutation (ASLP), which improves upon existing address randomization techniques with a novel approach to increase probabilistic protection at performance overhead comparable to contemporary techniques *without* using large pads. ASLP randomly places the user stack, heap, and shared libraries throughout the entire 3 gigabyte user address space. In doing so, ASLP randomizes 28 bits of the user stack base address, 29 bits of the heap base address, and 20 bits of each shared library allocation. Our approach randomizes 4 bits more than any other techinque, for a 16-fold increase in region placement options. Further, both SPEC CPU2000 and Apache Benchmarks indicate ASLP incurs less than 1% runtime overhead.

We also present a validation of address space randomization by showing that randomization limits the propagation speed of worms reliant on the absolute location of either the user stack, heap, or a shared library function. The increased randomization leads to increased exploit time needed for such worms. Compared to kernel-based related work, we increase average exploit time by a factor of 16. This means the fastest conceivable memory corruption worm could not infect a single host in the same amount of time the Slammer Worm took to infect approximately 500,000 hosts – or 90% of its vulnerable population.

Our analysis provides an in depth discussion of the probabilistic protection provided by ASLP. The performance results offer detailed information regarding the expected performance impact in three critical computing environments: scientific, network server, and general operating system micro-operations. We conclude that ASLP is capable of providing greater probabilistic protection than existing techniques at a comparable performance cost.

Address Space Layout Permutation:
Increasing Resistance to Memory Corruption Attacks

by

Christopher G. Bookholt

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh

2005

Approved By:

Dr. Peng Ning	Dr. Laurie Williams

Dr. Jun Xu
Chair of Advisory Committee

Biography

Chris Bookholt is from Charlotte, North Carolina. He received his Bachelor of Science in Computer Science attending NC State University from 2000 to 2004. In 2002 Chris began taking graduate classes and in 2003 he enrolled in the Accelerated Bachelors/Masters program, seeking a Master of Science degree in Computer Science. He will receive his Master of Science degree in December, 2005.

Acknowledgements

First, my parents deserve great deal of thanks for putting me through school. Without their help staying in school would have been impossible. Simply put, I owe it all to them.

Carolyn, my girlfriend, was a tremendous source of support and comfort. In addition to making meals and letting me vent frustration, she listened to my incessant babbling about address space randomization far, far more than any one person should have to endure.

Fellow members of the NC State University Systems Security research group – particularly Emre (John) Sezer, Chongkyung Kil, and Prachi Gauriar – were an invaluable source of second opinions, reality checks, and thoughtful discussions. It was a pleasure and honor to work closely with such obviously talented company.

I would also like to thank the members of the PaX community for their constructive criticism. Specifically, on several occasions Alexander (Pappy) Gabert volunteered hours of his time to explain Linux virtual memory and general kernel development concepts to me, a total stranger. Pappy, you have my respect and admiration.

Credit for the initial idea of ASLP goes to Dr. Jun Xu, my adviser and the chair of my thesis committee. His direction, motivation, and assistance were crucial to the completion of this project. Many thanks also go to my committee members, Dr. Peng Ning and Dr. Laurie Willians. In addition to finding time for me in their busy schedules, their input helped define a more logical organization of this paper.

Last but not least, my friends provided much needed encouragement and diversion during long hours spent working at home. They selflessly and tirelessly made good on their offers to trounce me in round after round of video games, disc golf, or both. Always with my health in mind, they also made sure I got enough fresh air by escorting me on a weekend hiking trip. Their thoughtfulness knows no bounds.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Memory corruption attacks are the most common means by which attackers take control of computers: of all US-CERT Cyber Security Alerts between mid-2005 and 2004, at least 56% of them have components of memory corruption [42]. With a high concentration of memory corruption attack vectors, there is reason to believe that mitigating attacks of this kind would give attackers significantly fewer ways to exploit their targets, thereby reducing the threat they pose.

Memory corruption is a mechanism to locate and alter critical portions of memory in a target program. Examples of critical memory regions are buffers, function return addresses, and function pointers [44]. In an attack scenario, the attacker attempts to alter program memory with the goal of causing that program to behave in a certain way. When an attacker is able to modify the contents of memory at will, the results can range from system instability to execution of arbitrary code. Examples of common memory corruption attacks are buffer overflows [29], format string exploits [30], and double-free attacks [1].

Vulnerability to this type of attack is typically caused by a lack of input validation. Programs written in the C language are particularly susceptible because they grant the programmer the freedom to decide when and how to handle input validation. This flexibility often results in improved application performance. However, the number of vulnerabilities caused by failure of input validation indicates that programming errors of this type are easy to make and difficult for programmers to fix.

Furthermore, software products do not always receive adequate testing for vulner-

ability to memory corruption attacks. Testing can be expensive and time consuming. Tools like CCured [27] offer a way to guarantee that a program is free from memory corruption vulnerabilities. However, such tools must be run directly against the source code, which is not always available. Further, when a bug is detected, the program must be manually corrected and tested again. The correction process requires a specially trained developer and also takes manpower away from other areas of development. Other products like AccMon [45] run with program in the precense of a memory monitor that identifies likely attacks on the fly based on a statistical model of memory accesses. However, in addition to relatively high overhead of 24 to 288%, this latter type of tool does not provide a sound analysis of the program, meaning some vulnerabilities may be missed.

Although bug prevention techniques attempt to address the problem at the source, the continued presence of memory corruption vulnerabilities indicates that developers are either not commonly using such techniques when it is feasible to do so or the tools are missing vulnerabilities. Instead, we believe it is effective to deter the successful exploitation of memory corruption vulnerabilities by altering the operating system on which vulnerable code is executed. Providing protection at the operating system kernel level provides numerous benefits: only one system component needs to be changed to provide effective protection for the entire system; no application source code needed; no developer time required; low performance overhead.

The kernel level protection against memory corruption attacks discussed in this thesis comes in the form of address randomization. It has been observed that attackers use absolute memory addresses during memory corruption attacks. Without randomization, the location of a process' memory regions will be the same every time. This allows an attacker to run a local copy of the target application on his own machine to determine the location of critical memory regions. This information can then be hard-coded into the exploit code. The general solution to this attack is to randomize the layout of process memory, thereby making the needed memory addresses unpredictable and breaking the hard-coded address assumption. The implication is that to attack randomized systems, an attacker must guess the needed memory address(es) with a low probability of success. The benefit is that when an attacker makes an incorrect guess, the program crashes. Although this may cause some services to become unavailable, we argue that an unavailable service is more desirable than a compromised system.

This thesis investigates the potential to improve upon existing kernel-based ap-

proaches to address randomization for protection against memory corruption vulnerabilities. Our approach is implemented by making modifications to the memory management subsystems of the Linux kernel. We randomize three user land memory regions: the stack, the `brk` managed heap, and `mmap` allocations. These three regions were chosen because control information exploited by memory corruption attacks resides within these areas. Compared to existing kernel-based techniques, our approach provides 16 times better protection against derandomization attacks described in [31]. To validate its practicality, we demonstrate the low performance overhead of our approach in computing environments such as scientific computing applications, basic system operations, and high-load server applications.

Other existing software is capable of detecting and mitigating specific types of memory corruption attacks. To name a few, StackGuard [10] addresses stack smashing buffer overflows; FormatGuard [11] addresses format string attacks; PointGuard [12] addresses pointer overwriting; LibSafe [2] addresses more generic buffer overflows. Although they may successfully mitigate their target vulnerabilities, the problem with these techniques is that they cannot readily be merged to provide complete protection against all types of memory corruption attacks.

Randomization provides a generic, low-overhead, runtime mechanism to mitigate memory corruption vulnerabilities. Exploiting such vulnerabilities requires precise knowledge of the memory layout of the target program. By dynamically randomizing the memory layout during process creation, the information needed to conduct a successful exploit must be guessed.

1.1 The Case for Increasing Randomness

Existing randomization techniques randomize the location of three critical memory regions: the user stack, heap, and shared libraries. These regions were chosen because their contents are frequently abused by memory corruption attacks. The techniques accomplish randomization by effectively inserting a randomly sized pad before each region during process startup, thereby dynamically shifting the regions and their contents by a random amount. The result is that two sequential process instances for the same program will have different memory layouts.

In [31], Shacham *et al.* develop a "derandomizing attack" capable of converting

a buffer overflow into a brute-force guessing attack against address randomization. Their attack works by repeatedly guessing the memory address of the standard C library. Once the library is located, they use the `system()` function to cause the target computer to execute arbitrary commands. This attack is feasible for two reasons. First, the target application in this case was the Apache web server, which uses the worker-supervisor design where a crashed worker processes is immediately replaced by forking a child process from a supervisor processes. Although the supervisor process was randomized, the replacement process is forked from an existing process, so the memory regions are copied and not re-randomized. This allows the attacker to keep making guesses without the process memory layout changing after each guess. The second reason for their derandomizing attack success is that the shared libraries are traditionally the least randomized region, meaning they are the weakest point of protection.

Of the existing kernel-based techniques, PaX [34] Address Space Layout Randomization [35] provides the most protection. However, the derandomization attack can defeat PaX ASLR in an average of 216 seconds [31]. This is a clear indication that stronger randomization is needed if address randomization is to effectively deter a determined attacker from exploiting a specific vulnerable target. Their paper also formally identifies a limitation of address randomization on x86 memory architectures: although each region's address may be composed of 32 bits, due to x86 memory architecture design regions must be page-aligned, meaning they are placed at 4 KB (1 page) intervals. This restriction reduces the maximum randomness from 32 bits down to 20 bits for page-aligned allocations.

Note that the repeated guessing of a brute-force derandomizing attack should be easy for Network Intrusion Prevention Systems to detect and block. However, even if the attack is allowed to continue, we demonstrate that a significant increase of the time needed to exploit a single target has a profound impact on the propagation speed of Internet worms. Our ASLP implementation is the only kernel-based address randomization technique to reach the maximum 20-bit randomization on the x86 architecture for page-aligned allocations of the user stack, heap, and `mmap` allocations. We also employ sub-page randomization for both the heap and user stack to reclaim some low-order bits lost to page alignment in these regions.

1.2 ASLP Contributions

ASLP fills a gap in contemporary kernel land address randomization techniques: page-aligned memory allocations allow 20 out of each 32-bit memory region address to be randomized, but even the best existing technique only randomizes 16. ASLP randomizes all 20 bits for each page-aligned allocation. It does so by randomly permuting regions throughout the entire 3 GB user address space. Then ASLP applies a sub-page random shift to both the user stack and the heap to reclaim some low-order bits lost to randomness. This increases the randomized bits to 28 for the user stack and 29 for the heap, which means ASLP is capable of generating process memory layouts that are an order of magnitude more unpredictable than any other kernel-based address randomization technique, making it the most resistant to brute-force derandomization attacks.

ASLP also has comparable performance overhead to contemporary techniques. Our results show that overhead occurs during process startup and runtime performance overhead is less than 1%.

A traditional argument against address randomization is that it wastes too much space. The wasted is space is caused by an approach shared by all contemporary address randomization techniques: in effect, they insert a randomly sized pad at the beginning of each memory region to be randomized. The pad shifts all address in the region by its random size, creating an unpredictable addresses for allocations in the padded regions. The pad is wasted space. A further limitation of these approaches is that the only way to increase the amount randomization is by increasing the size of the pad, thereby wasting more space.

For illustration, the diagram in Figure 1.1 depicts the memory layout for a process randomized with either PaX ASLR or Exec Shield. Note that (1) the relative order of memory regions is the same as an unrandomized layout and (2) the use of three pad allocations that consume space.

ASLP provides increased protection *without* the use of large pads. With ASLP, pads are only used in the sub-page randomization with a random size of 0 to 4 KB. Since this pad is only applied to the user stack and heap, a maximum of 8 KB out of the 3 GB user address space is consumed by pads. Figure 1.2 depicts the memory layout of a process randomized with ASLP. Note (1) the random relative order of memory regions and (2) the lack of large pad allocations.

With an unrandomized address space, attackers can make several important as-

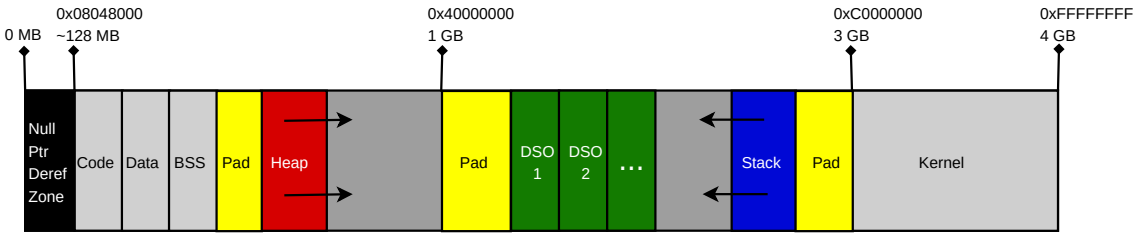

Figure 1.1: Example PaX ASLR Process Memory Layout

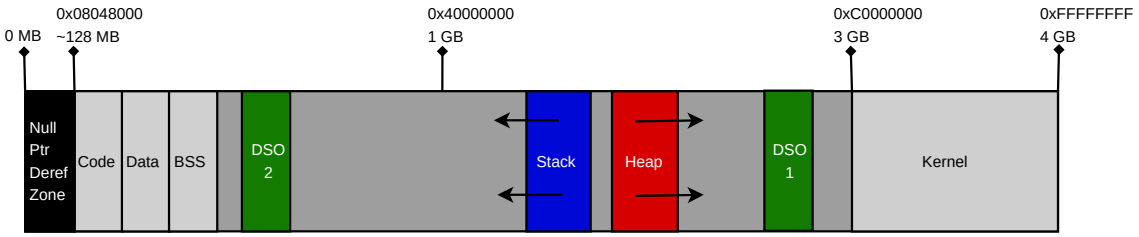

Figure 1.2: Example ASLP Process Memory Layout

sumptions about the memory layout of any process before they attack: the heap is allocated following uninitialized data near the bottom of the address space, shared libraries are allocated contiguously, and the stack is the memory region highest in the address space. Even with contemporary address randomization techniques, attackers can still make the same assumptions. These assumptions allow attackers to craft more efficient derandomization attacks.

In summary, the ASLP approach contributes in three ways:

1. ASLP provides probabilistic protection an order of magnitude stronger than related kernel-based techniques.

2. A negligible amount of address space is consumed by pads.

3. The performance overhead is generally less than 1% and is comparable to contemporary techniques that provide less protection.

These contributions simultaneously debase the traditional space consumption argument while increasing protection against derandomizing attacks.

The rest of this thesis is organized as follows:

Chapter 2 discusses related work in the address randomization area. Chapter 3 describes the theory and implementation of ASLP. Chapter 5 proposes two techniques to introduce randomness into kernel address space. Chapter 6 contains our concluding remarks and a brief overview of future work.

Appendix A gives background information on virtual memory management in the standard Linux kernel for those that may not already be familiar. Although the material is very straightforward, it is crucial to first understand the basic operation of an unrandomized Linux virtual memory system in order to appreciate the contributions made by this and related work.

For full disclosure, Appendices B, C, and D contain the raw benchmark data from CPU2000, LMBench, and Apache Benchmark, respectively; Chapter E contains the raw results of the PaXtest experiments.

Chapter 2

Related Work

There is a large quantity of related research in the area of mitigating memory corruption attacks. Included in this section are a subset of those works that mitigate memory corruption attacks via address randomization.

The seminal work in the field by Forrest *et al.* illustrates the value of diversity in ecosystems and proposes similar benefits for diverse computing environments [15]. In short, their case for diversity is that if a vulnerability is found it one computer system, that same vulnerability is likely to be effective against all identical systems. By introducing diversity into a population, resistance to vulnerabilities is increased. Address randomization achieves diversity by making the virtual memory layout of every process unique and unpredictable.

We separate related work into two categories: kernel land and user land. Kernel land techniques achieve randomization with modification to the kernel; user land approaches achieve randomization with modification to user space applications.

2.1 User Land

The primary advantage of user land address randomization techniques is their ability to reorder elements of code and data. Modifications of this granularity are not readily possible from kernel land. This means that user land randomization techniques are capable of providing randomization at finer granularity than kernel-based techniques.

An important weakness of most user land address randomization techniques is their need to recompile every application to allow it to have address randomization. It is unreasonable to assume that users will be able and willing to recompile every program to be randomized; recompilation assumes an unacceptably high level of technical knowledge for end users. Further, recompilation is impossible without access to the source code of the application. Thus, the remaining option is to assume that binary distributors will recompile application binaries with the corresponding randomization options. However, it seems unrealistic to assume that the myriad (occasionally competing) distributors will willingly align under a common standard set of configuration options for application compilation. Further, this model neglects legacy binaries that are no longer supported by any distributors. Finally, consider management of software upgrades from system administrators' perspective: as newer versions of the user land address randomization techniques are released, software must again be recompiled to transition to the latest version. This requirement means any technique requiring software recompilation is unattractive in large scale deployments.

Address Obfuscation The techniques presented by Bhatkar *et al.* in [3] and [4] are significant because they implement a mechanism for not only randomly shifting the placement of the three critical regions, but also randomly re-ordering code and data *within* their respective segments, which addresses offset-based and data attacks. This allows for a very fine-grained randomization. For shared library randomization, they statically link a randomly sized dummy library to each executable, effectively padding the start of the shared library area by the size of the dummy library. However, this means that the shared library randomization is then static because it does not change on a per process basis.

Note that *all* of the major drawbacks listed at the beginning of this section apply. Further, the base address for both the stack and heap are randomized by shifting each region by up to approximately 95 megabytes (MB). This means nearly 200 MB of address space can be wasted in just these two regions. Although this method provides the strongest probabilistic protection of any existing address randomization technique – and neglecting the typical user land drawbacks stated earlier – it wastes too much memory to be considered practical.

Transparent Runtime Randomization (TRR) [44] TRR is different than other user land approaches because it modifies only the dynamic loader and does *not* require applica-

tion recompilation or access to source code. Although this is a highly desirable feature from a user land technique, when compared to PaX ASLR or our own ASLP, Transparent Runtime Randomization does not offer as much randomness in the user stack, heap, or shared library regions. Therefore, TRR is most adept in computing environments where maximum availability (*i.e.* avoiding reboots) is valued more than providing strong resistance to memory corruption attacks.

2.2 Kernel Land

Kernel land techniques strike an attractive compromise between ease of management and probabilistic protection. Software upgrades are a common task as updated versions are released. The same applies to the Linux kernel: the release rate for new kernels depends on the vendor, but once every two months is a fair estimate. So, although updating the kernel requires a reboot, address randomization installation and upgrades can transparently accompany normal system maintenance as updated kernels are installed.

In contrast to the user land approaches described in Section 2.1, kernel land address randomization techniques are being actively used in major Linux distributions: Exec-Shield can be found in both Red Hat Enterprise Linux [22] and Fedora Core [8] from Red Hat Inc. [21]; PaX ASLR can be found in Hardened Gentoo [5], Adamantix (Trusted Debian) [6], and OpenBSD [13].

All kernel land projects in this section share a fundamental approach to address randomization: insert a randomly sized pad before the stack, heap, and `mmap` area. Although they share a general methodology, they differ greatly in their priorities. Since padding consumes address space, the few programs that need all available address space will observe a reduction in available memory. This extreme case forces developers to prioritize the protection provided by large pads and conservation of available address space.

Red Hat Exec-Shield [24] Red Hat Exec-Shield developers chose to favor the conservation of address space. As such, their pad regions are fairly small. Of the tested techniques, Exec-Shield provides the lowest amount of randomization in every region. However, according to comments made by Ingo Molnar [25], the Exec-Shield primary author, the lack of randomization is by design and is considered a feature. From a security perspective this is

plainly backwards, but the target audience is not security experts. Instead, they target a very broad range of server and scientific computing environments that don't normally allow security features to encroach upon application performance. Molnar argues that increasing space devoted to shifting regions consumes too much space to be practical because it reduces the amount of memory an application can allocate.

Although the probabilistic protection provided by this method suffers from this ideology, a major deterrent remains: attackers must still spend extra development time to make an attack that can succeeded against a randomized target program. For more information on Exec-Shield see [14].

PaX Address Space Layout Randomization [35] In contradiction, PaX ASLR favors improved protection at the expense of those few applications by making the pads up to 256 MB. Predictably, PaX ASLR provides considerably better protection than Exec-Shield with its pads of only a few megabytes. By some standards, PaX ASLR is considered impractical due to the large amount of address space consumed by the padding that shifts each region's base address. Even so, several security-oriented distributions promote PaX ASLR for use in computing environments where increased protection against memory corruption vulnerabilities is valued more than increased memory utilization. For more information on PaX see [37, 39, 38, 40].

Chapter 3

ASLP Randomization

This chapter presents Address Space Layout Permutation (ASLP) in detail. We begin with a discussion of supporting theory underlying our approach. This is followed by an overview of the ASLP implementation. Then a concrete demonstration of ASLP permutation is presented using the memory allocations of the `cat` executable.

3.1 Supporting Theory

The protection afforded by all address randomization techniques is probabilistic. Assuming no other countermeasures are in place, a derandomizing attack against a vulnerable application will eventually succeed. We forego the argument that additional countermeasures could effectively prevent derandomizing attacks in order to focus on the probabilistic protection provided solely by the address randomization in ASLP.

In a derandomizing attack, the attacker must repeatedly make guesses for an address. For example, a derandomizing stack smashing attack guesses the absolute address of the attacker's injected code, which resides in the stack. Intuitively, the larger the range in which the stack can be randomly placed, the more guesses the attack must make to successfully guess the correct address. To make addresses harder to guess, ASLP uses nearly all 3 GB of user address space in which to randomly place memory allocations.

In a 32-bit address space there are 2^{32} addresses. However, with page alignment,

allocations must be made at the beginning of a page. With a page size of 4 kilobytes (KB) on IA32 systems, only 1 of every 2^{12} addresses can be used as the starting location for region allocations. This leaves a maximum of 2^{20} possible locations for each randomized page-aligned region. The consequence of this memory architecture design is that even though ASLP can randomly allocate a region anywhere in 32-bit user space, only 20 bits of page-aligned allocations can be random.

The mechanism for circumventing this limitation is to find ways to insert sub-page randomization after the initial page-aligned allocation is made. As discussed in Section 3.2, ASLP uses a two phase randomization for the heap and stack regions that regains some of the low-order bits lost to page alignment. However, no sub-page randomization method for `mmap` allocations has yet been implemented because doing so would violate the POSIX `mmap` specification [19]. Thus, in all kernel-based techniques, `mmap` allocations are page-aligned and limited to at most 20 bits of randomness.

3.2 Implementation

We build our proof of concept ASLP kernel for the popular 32-bit x86 CPU. Although 64-bit processors are slowly entering the market, 32-bit processors are still in production and will remain online for several years to come. Recall the standard memory layout of a process in 32-bit virtual address space with the Linux 2.4.31 kernel[1]. The addresses at which the code and data segments are loaded is determined by the ELF executable standard. Standard ELF executables do not include relocation information in the binary file. Therefore relocating code and data segments would break placement assumptions and is not done. The other region that remains untouched is the top 1 gigabyte (GB) of address space, which is reserved for kernel data structures. Kernel address space is handled very differently than user address space. For example, user processes cannot directly access kernel space virtual address. Further, access control checks are frequently done based on the assumption that access to virtual addresses higher than 3 GB is strictly reserved for kernel-mode execution. Thus, moving this region would require a more complex access control check on each memory access, introducing additional performance overhead. As a result, ASLP does not permute the top 1 GB of virtual address space. This leaves three regions for permutation:

[1] A refresher on standard process memory layout is available in Appendix A.

the user-mode stack, `brk()`-managed heap, and `mmap` allocations.

We employ a two-phase randomization process for both the user stack and the `brk()`-managed heap. The primary phase is a major, page-aligned permutation that places the region on any page boundary between approximately 128 MB (following the data segment) and 3 GB (the top of user address space). The address of such regions are randomized in 20 bits. To reclaim the low-order bits, the secondary minor, sub-page sift is used. This shift only moves a region by 0 to 4 KB, but has the effect of de-aligning these regions.

We need a two-phase process because all regions must be page aligned due to the design of the hardware memory management unit (MMU). According to the MMU, the atomic unit of memory is the page. So with the two-phase sub-page randomization, although the actual region allocations are page-aligned, the contents of both the user stack and `brk()`-managed heap are shifted within their respective regions by up to 4 KB. Although the secondary phase is conceptually similar to the large pads applied by other techniques, this sub-page shift consumes at most 8 KB of space, which is negligible in a 3 GB user address space.

The User Stack. The location of the user stack is determined and randomized during process creation. In the early states of process creation the kernel builds a data structure which holds process arguments and environment variables. This data structure is not yet allocated in process memory, but rather it is prepared for the process in kernel space memory. In this structure, the stack pointer is defined. This pointer is merely an offset into the first page of the soon-to-be stack region. We subtract a random amount between 0 and 4 KB from the stack pointer, thereby introducing randomization in low-order bits [2].

In the later stages of process creation, the same data structure is copied into the process address space. During this phase we introduce the large scale randomization. A random amount is subtracted from the standard 3 GB stack base location so that the region starts anywhere between approximately 128 MB and 3 GB.

To ensure the stack has room to grow, ASLP prevents subsequent allocations immediately below the stack. This feature prevents the stack from being allocated so close to another region that it cannot expand. The exact amount of reserved area is configurable, but empirical evidence suggests that 8 MB is sufficient for most applications.

[2]The idea and implementation for sub-page randomization of the user stack was taken from PaX ASLR.

The `brk()`-managed Heap. Similar to the stack, the heap location is set during process creation. In an unmodified Linux kernel, the heap is allocated along with the BSS region, which is conceptually part of the data segment. We modify the allocation code for the BSS and heap so they occur in two independent steps. Separation allows the heap location to be defined independently of the data segment. The amount of space to be allocated for the heap is then augmented by 4 KB (1 page). Then a random, page-aligned virtual address between 0 and 3GB is generated for the start of the heap. Finally, a random value between 0 and 4 KB is added to this address to achieve sub-page randomization. Since the initial heap allocation was given an extra page, the sub-page shift will not push it beyond the original allocation.

The heap is also similar to the stack in that it can grow to fulfill dynamic memory requirements as the corresponding process runs. As with the stack, a comparable solution is used for the heap in which an unused region of configurable size is maintained following the heap. This prevents the heap from being placed too near other regions and ensures that it has enough room to grow. Empirical evidence suggests a reservation of 128 MB is needed for the heap.

`mmap()` Allocations. The `mmap` system call is used to map objects into memory. Such objects include shared libraries as well as any other files the application may wish to bring into memory. Allocations made by `mmap` are randomized using a one-phase, major randomization that is nearly identical to the primary phase used for the stack and heap. A secondary, sub-page shift is *not* used for `mmap` allocations because doing so would violate the POSIX `mmap` specification. From the specification:

> The *off* argument is constrained to be aligned and sized according to the value returned by *sysconf()* when passed `_SC_PAGESIZE` or `_SC_PAGE_SIZE`. When `MAP_FIXED` is specified, the application shall ensure that the argument *addr* also meets these constraints. The implementation performs mapping operations over whole pages. [19]

From this excerpt, the straightforward approach of applying a similar 0 to 4 KB random shift for `mmap` allocations is forbidden.

Since there can be multiple independent `mmap` allocation per process (such as for two different shared libraries) each allocation is made randomly throughout the entire available user level address space. This means that allocations for multiple shared libraries do

not necessarily occupy a contiguous, sequential set of virtual memory addresses as they do in all related techniques and unrandomized kernels. This is beneficial because the location of one library will be of no use to determine the location of another library.

Although the `mmap` system call allows a user process to request a specific virtual address at which to store the mapping, there is no guarantee the request will be honored even in the vanilla kernel. In the ASLP kernel, if a specific address is requested it is simply disregarded and replaced by a random address. The random, page-aligned addresses are issued between 0 and 3GB. Therefore, `mmap` allocations use a one-phase major randomization rather than the two-phase approach used for the stack and heap. An exception to overriding supplied addresses exists for fixed regions, such as the code and data segments. These regions are also brought into memory via `mmap`, but because they are flagged as fixed the supplied address is honored without modification.

Randomization Mechanism. A critical part of the probabilistic protection is the randomization function responsible for introducing unpredictability into process memory layout. The Linux kernel has a built-in non-deterministic random number generator (RNG) based on both hardware events and user activity sources. These individual sources are mixed together into an entropy pool on every hardware interrupt. When a request for random bytes is issued, the pool is hashed using the SHA1 algorithm and the resulting hash is given to the requester. Thus, even if there were a lack of entropy from all sources, an attacker would still need to be able to derive useful information from the hash to determine the values used to generate randomness.

However, in the interests of reducing overhead and preserving the entropy pool, we use a combination of the built-in RNG and an adapted pseudo random number generator (PRNG). Our PRNG adaptation is based on a traditional linear congruential of the form in equation 3.1 from the IEEE POSIX specification document example [20]

$$x_{n+1} = (A \times x_n + C) \bmod M \tag{3.1}$$

where x_0 is the seed, x_n is the nth member of the sequence, and A, C, and M can be adjusted to modify the sequence as desired [43]. However, the implementation needs differ from this originating function in two key ways. First, there is no need for a modulus operator because we want values to span the full 32-bit range. Second, user space addresses can only range from 0 to 3 GB because the kernel occupies the top 3 to 4GB.

This requirement means we must incorporate a fractional modifier to scale down the range of possible values. We do not simply use a modulus operator to limit addresses between 0 to 3 GB because this would skew the distribution toward the lower 1 GB of memory. The resulting formula is equation 3.2.

$$x_{n+1} = \frac{(1103515245 \times x_n + 12345)}{4} \times 3 \tag{3.2}$$

Early during process creation, the built-in RNG is used to establish the initial seed for our PRNG. This approach introduces non-determinism into a computationally simple PRNG without the need to tap the system entropy pool for subsequent randomization operations.

3.2.1 Demonstration of Permutation

As a demonstration of the region permutation provided by ASLP, Table 3.1 provides a side-by-side comparison of two memory layouts of `cat` process instances on a vanilla kernel (left) and our ASLP kernel (right).

Vanilla Kernel

Addresses	File	Description
08048000-0804c000	cat	cat code
0804c000-0804d000	cat	cat data
0804d000-0804e000	-	heap
40000000-40015000	ld-2.3.2.so	loader code
40015000-40016000	ld-2.3.2.so	loader data
40016000-40017000	-	anon map A
40021000-40154000	libc-2.3.2.so	libc code
40154000-40158000	libc-2.3.2.so	libc data
40158000-4015a000	-	anon map B
4015a000-4035a000	locale-archive	mmap
40154000-40158000	-	user stack

ASLP Kernel

Addresses	File	Description
08048000-0804c000	cat	cat code
0804c000-0804d000	cat	cat data
0bc29000-0bd5c000	libc-2.3.2.so	libc code
0bd5c000-0bd60000	libc-2.3.2.so	libc data
0bd60000-0bd62000	-	anon map B
1be74000-1be89000	ld-2.3.2.so	loader code
1be89000-1be8a000	ld-2.3.2.so	loader data
23452000-23454000	-	heap
30d9f000-30f9f000	locale-archive	mmap
589e8000-589ea000	-	user stack
69996000-69997000	-	anon map A

Table 3.1: Comparing Process Memory Layout on Standard and ASLP Kernels

There are several important characteristics that demonstrate the random permutation of memory regions. Only the code and data segments of the executable remain in

their original location. Even these segments can be randomized by using a combination of PIE and ASLP. Also, the heap is allocated independently of the data segment, which allows it to be relocated. Finally, the user stack is not the highest allocation in user space. In short, the result is that the stack, heap, and `mmap` allocations occur randomly throughout the 3 GB user address space.

Chapter 4

ASLP Evaluation

This chapter evaluates ASLP randomization from both security and performance perspectives.

4.1 Security Evaluation

This section provides a security evaluation of ASLP compared to both Exec-Shield and PaX ASLR. Presented first is an evaluation of security in the context of a single randomized target against a determined attacker. The results for PaXtest evaluate the randomness in each randomized region for each technique. Next, a chart of the estimated average protection provided by each technique is presented. Finally, security is evaluated in the context of Internet worm propagation in the presence of the different kernel-based randomization techniques.

4.1.1 Micro-Security

As previously discussed, although every address in the x86 memory architecture is represented by 32-bits, not all of those bits can be randomized. To assess the bits of randomness in each region and for each technique, we use a third-party application called PaXtest [7], which performs an attack on the address randomization to determine the virtual address for the stack, heap, and shared library base. By running locally, PaXtest

is able to spawn simple helper programs that merely output the virtual addresses of their variables, each allocated in a randomized region. In doing so, it can determine the number of randomized bits for such variables based on where they would normally reside in an unrandomized system. The application provides unbiased verification of randomization effectiveness for PaX ASLR as well as Red Hat Exec-Shield and our own ASLP.

Table 4.1 provides a summary of the PaXtest results for each of the tested kernels. The raw results are included in Appendix E.

Region	Vanilla	Exec-Shield	PaX ASLR	ASLP
User Stack	0 bits	17	24	28
Heap	0 bits	13	13	29
Mmap	0 bits	12	16	20

Table 4.1: PaXtest Results Summary

The vanilla kernel has no randomization, so each region has zero bits of randomness. Exec-Shield is the most susceptible to derandomization attacks, providing the lowest amount of randomization in all three regions. PaX ASLR comes in the middle, with significantly better stack and `mmap` randomization. Our ASLP approach comes out ahead by at least 4 bits in all three regions for an 1600% increase in probabilistic protection. Figure 4.1 shows a graphical representation of the PaXtest results summary.

The results from PaXtest are used to make estimates regarding the probabilistic protection provided by each technique. For example, PaX ASLR randomizes 16 bits in the address of the shared library base. This means that there are 2^{16} or 65,536 possible locations for placement for the shared libraries. Assuming a random distribution, the address can be guessed in a number of guesses equal to half of the possible locations. Knowing both the reported attack duration of 216 seconds to brute-force guess the address of shared library region [31] and the number of possible locations in this region from Table 4.1, we can estimate the average guess rate. Equation 4.1 shows the calculation of guess rate.

$$\frac{65536 \text{ possible locations}}{2} \rightarrow \frac{32768 \text{ average guesses}}{216 \text{ seconds}} = 151.7 \text{ seconds to guess on average} \tag{4.1}$$

Using the guess rate of 151.7 guesses per second derived from equation 4.1 and the

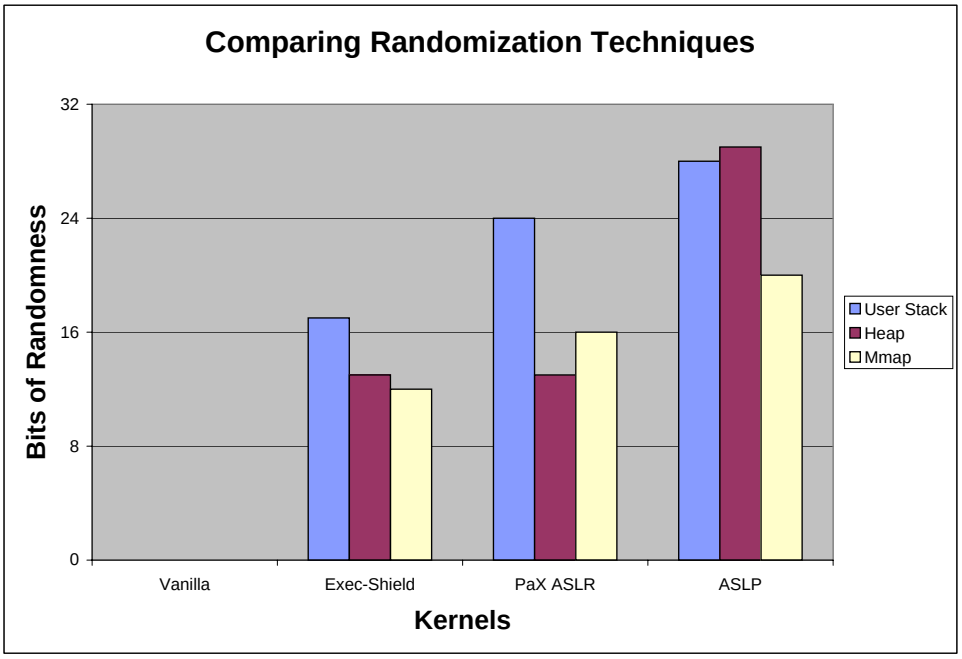

Figure 4.1: Comparing Randomness Provided by Related Techniques

bits of randomness returned from PaXtest in Table 4.1, we can calculate the amount of time required to brute force the randomization in each memory region for ASLP, PaX ASLR and Exec-Shield. Table 4.2 illustrates how we derive the probabilistic protection provided by our implementation of ASLP.

Region	Random Bits	Num. Locations	Avg. Guesses	Avg. Protection (s)
stack	28	268435456	134217728	884757.60
heap	29	536870912	268435456	1769515.20
mmap	20	1048576	524288	3456.08

Table 4.2: ASLP Protection Evaluation

Against ASLP, a derandomizing attack for either the stack or heap would take weeks of constant guessing. The heap randomization requires an average brute-forge attack time of nearly an hour. Given a vulnerability in which only the address of the `mmap` area is needed, we do *not* claim that ASLP is an effective deterrent to prevent a determined attacker from penetrating a single randomized target. Instead, we refer the reader back to

earlier discussion of macro- and micro-security and Figure 4.2, which presents the effect of address space randomization on worm propagation speed. We argue from a macro-security perspective that address space randomization – specifically ASLP – provides a mechanism by which the memory corruption Warhol worm can be slowed to a rate that allows intrusion detection systems and system administrators to respond.

Table 4.3 estimates the amount of protection provided by Red Hat Exec-Shield using the same methodology.

Region	Random Bits	Num. Locations	Avg. Guesses	Avg. Protection (s)
stack	17	131072	65536	432.01
heap	13	8192	4096	27.0
mmap	12	4096	2048	13.0

Table 4.3: Red Hat Exec-Shield Protection Evaluation

Similarly, Table 4.4 estimates the amount of protection provided by PaX ASLR.

Region	Random Bits	Num. Locations	Avg. Guesses	Avg. Protection (s)
stack	23	8388608	4194304	27648.68
heap	13	8192	4096	27.0
mmap	16	65536	32768	216.0

Table 4.4: PaX ASLR Protection Evaluation

In PaX ASLR, all three regions have a placement range of 256 MB, however the stack has randomization both above and below the page level, which is how it retains more possible placement locations within the same placement range.

Considered at the micro-security level – in the sense of protecting a single computer against a single determined attacker – 20-bit randomization is still insufficient. Shacham *et al.* defeat the 16-bit shared library randomization in PaX ASLR in an average of 216 seconds [31]. Although 20-bit randomization provides more deterrence, the same derandomization attack can still succeed in approximately an hour, which is firmly in the realm of feasibility for the micro-security scenario.

4.1.2 Macro-Security

Consider the effect of address randomization at the macro-security level: Internet worm propagation on a large scale. With ASLP, exploiting a single host via memory corruption attacks takes longer than worms like Sapphire/Slammer took to infect their entire vulnerable population [26]. With address space randomization, the speed at which worms can spread using memory corruption attacks is bounded not by how fast they can find vulnerable hosts, but by how fast they they can derandomize the randomized address space of each target.

Using the derived attack speed and average probabilistic protection of 57 minutes (the lowest protection value from Table 4.2, consider Figure 4.2 showing the exponential growth rate of a hypothetical hit list-based worm in a population of more than 500,000 vulnerable hosts when it must first brute-force through ASLP address randomization.

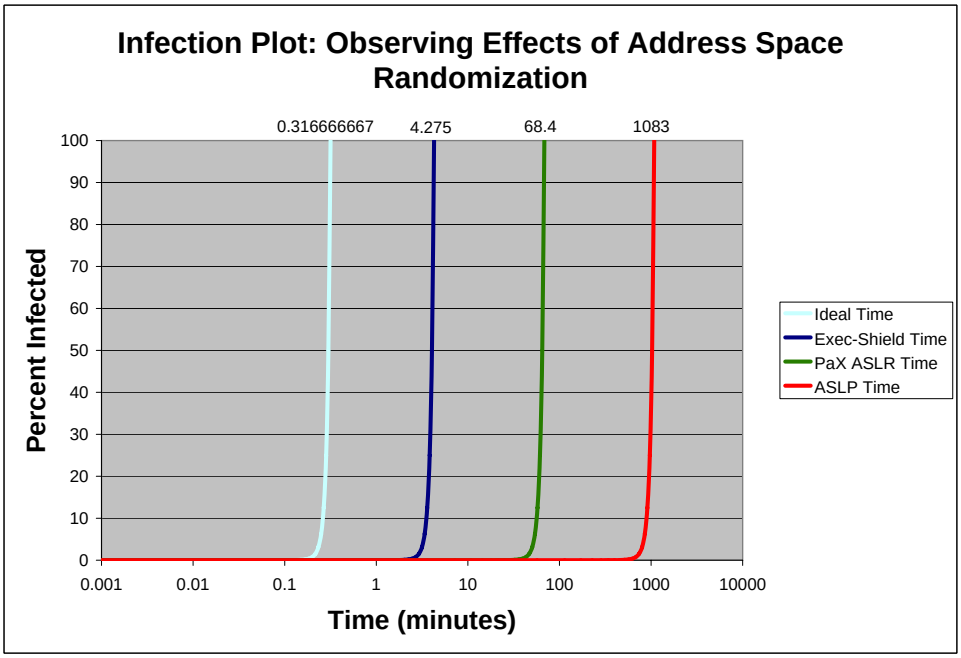

Figure 4.2: Infection Plot: With ASLP vs. Without ASLP

Note that the worm propagation model presented here makes several unrealistic assumptions in favor of the rapid worm spread:

1. The entire vulnerable population of over 500,000 hosts is known in advance and included in the hit list, so no scanning is required to find the next target.

2. The worm propagation code is configured such that there is no overlap in population coverage.

3. Network congestion caused by thousands of hosts flooding the network with address guesses is assumed to have no affect on propagation speed.

The result of this model is a wost-case worm outbreak on a hypothetical, ideal network. The reason for these pessimistic assumptions is to assert that ASLP is still able to limit worm propagation speed such that total infection is achievable in hours instead of minutes. Therefore, the propagation speed limit enforced by ASLP pushes the infection time well above the 15 minute limit for so called Warhol worms [32]. This is crucial because the extra time gives system administrators additional time to react.

Specifically, with no randomization the hypothetical worst-case worm is able to infect 100% of vulnerable hosts in less than one minute by doubling in size every one second. With randomization in place, the worm must first brute-force guess the location of the least protected region: the shared libraries. For Exec-Shield, 100% infection occurs in just over four minutes; for PaX ASLR the time is just over one hour. Our ASLP method is able to delay 100% infection for over eighteen hours.

This extension of infection time illustrates the benefit of having effective address space layout protection because fast worms that exploits memory corruption vulnerabilities must first get through address randomization before they can compromise their target. The Sapphire/Slammer worm [33] is currently the fastest know worm, which infected 90% of vulnerable hosts in less than 10 minutes [26]. Similar or even faster worms could not achieve such speeds if exploiting a vulnerable target requires an average of more than 57 minutes.

Further, the barrage of attempts to guess the correct address should be visible by intrusion detection and prevention systems. Increasing probabilistic protection means forcing attackers to make more guesses, effectively giving intrusion detection systems a bigger target.

4.2 Performance Evaluation

This section contains the performance analysis of the ASLP kernel in terms of overhead caused by address randomization. To get an understanding of the performance

impact of our randomization technique, we compare our ASLP kernel to the configurations listed in Table 4.5.

Test Configuration	**Description**
Vanilla Kernel	Acts as control
PaX ASLR	Randomizes user stack, heap, & shared libraries
Red Hat Exec-Shield	Randomizes user stack, heap, & shared libraries
Position Independent Executable	PIEs allows relocation of code & data segments

Table 4.5: Configurations Compared to ASLP

The various kernels are tested by installing each of them on the a single test computer and selecting the desired kernel at boot time. The test computer runs Red Hat Linux 9.0 with a 2.66 GHz Intel Pentium 4 CPU, 512 MB of RAM, and a 7200 RPM ATA IDE hard disk drive. All benchmark suites were compiled using GCC version 3.4.4.

The vanilla kernel establishes the baseline by which the absolute performance impact of other configurations can be measured. Both Exec-Shield and PaX ASLR are closely related works that provide kernel-based address layout randomization. Performance of the two closely related projects provides a metric to determine if ASLP does or does not have comparable performance. PIE is included in this evaluation because, although it doesn't provide any randomization by itself, it does allow address randomization mechanisms to move the code and data segments from their traditionally fixed position. This is valuable to see how much overhead would be introduced by randomizing the location of the code and data segments. Since PIE is a compilation option and not a kernel modification, we test it using a vanilla kernel and compile the benchmark suite with the PIE options; in this configuration we isolate the performance impact of PIE without the additional overhead of address randomization techniques.

Although minimizing performance overhead was not the primary goal of the ASLP implementation, our target is to incur comparable performance overhead to both PaX ASLR and Exec-Shield. It should be noted that both PaX ASLR and Exec-Shield can be configured to do more than address randomization. Where possible, we disable their extra features to make the related projects as similar to ASLP as possible.

We employ three popular benchmarks to measure the performance of each configuration: SPEC CPU2000 [9], LMBench micro-benchmark [23], and Apache Benchmark

[16]. Each benchmark suite is chosen to represent a popular area of computing: scientific, general desktop, and server, respectively.

The SPEC CPU2000 Integer Benchmark shows our ASLP technique has an average runtime overhead of 0.27%. The LMBench micro-benchmark indicates start-up performance less than 13%. Finally, Apache Benchmark shows an average runtime performance for server applications of 0.71%. Each of the three benchmarks was run nine times on every configuration. Having nine runs is less than ideal, but was as many as possible given the time constraints of the project. Further, these performance results are not intended to have high precision, but rather to present the general performance characteristics of the ASLP approach and its similarity to closely related work. The remainder of this section discusses these results in more detail.

4.2.1 SPEC CPU2000 Integer Benchmark

The SPEC CPU2000 Integer benchmark suite is a set of computationally intensive integer applications that simulate the workload of scientific computation. Each benchmark in the CPU2000 suite measures the time required to complete a different integer computation. For this reason, we use the CPU2000 benchmark to measure the impact of address randomization on computational efficiency.

Figure 4.3 gives the normalized performance overheads of each benchmark and configuration, relative to the vanilla kernel configuration while Appendix B presents the absolute runtime information for each of the 11 integer benchmarks. [1] The CPU2000 benchmark suite was run nine times on each configuration; the resulting times for each benchmark are the geometric mean.

According to the CPU2000 Integer benchmark results, ASLP has an average performance overhead of 0.27%. All of the tested address randomization techniques have an average computational overhead of less than 0.6%. The average overhead for all address randomization techniques differ by less than 0.5%. With a range this small we can safely say ASLP has computational performance overhead comparable to closely related works. In High Performance Computing environments it seems unlikely that *any* address ran-

[1] The full CPU2000 Integer benchmark suite contains 12 benchmarks. One of the 12, the "eon" benchmark, did not compile on GCC version 3.4.4. Although this GCC version was newer than that recommended by the benchmark documentation, a GCC compiler of at least version 3.4.4 was needed to support PIE linking. We therefore elected to exclude the eon benchmark from the final results.

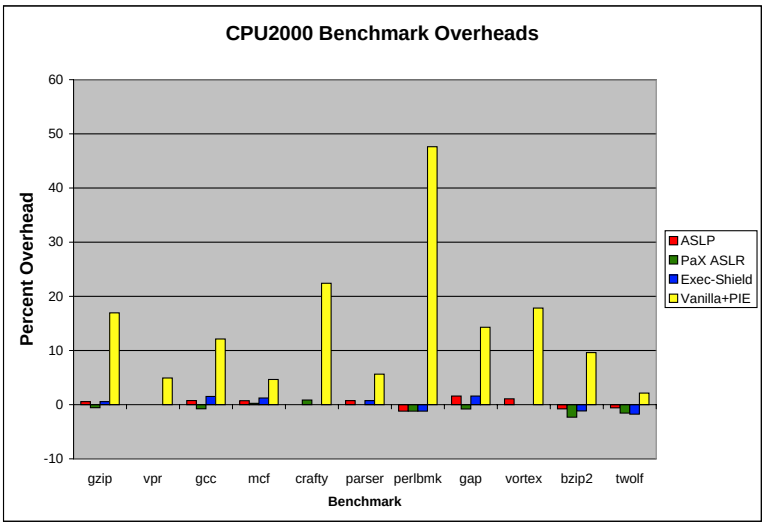

Figure 4.3: CPU2000 Integer Benchmark Overheads

domization features would be used because they measurably – albeit minimally – impact performance. However, should they be used, any of the tested techniques will be roughly equivalent in terms of performance impact.

A second notable aspect of the performance data is that PIE linking incurs an average of nearly 15% computational overhead. To gain an understanding of why PIE performance is consistently degraded during runtime, consider the following simple C program:

Program 1 An Example Two-Function Simple C Program

```
int my_global_var = 0;

int sim()
{
    return my_global_var + 5;
}

int main()
{
    sim();
    return 0;
}
```

Table 4.6 shows the assembly language translation of the `sim()` function in `simple.c`. The assembly code on the left was generated using standard compilation options while the code on the right is the same function, but is part of a position independent executable. Other than the code in the `sim()` function that references a global variable, the assembly code representation is the same for both versions of the executable.

```
1 sim:
2 pushl %ebp
3 movl %esp, %ebp
4 movl my_global_var, %eax
5 addl $5, %eax
6 leave
7 ret
```

```
1 sim:
2 pushl %ebp
3 movl %esp, %ebp
4 call .L3
5 .L3:
6 popl %ecx
7 addl $_GLOBAL_OFFSET_TABLE_+[.-.L3], %ecx
8 movl my_global_var@GOTOFF(%ecx), %eax
9 addl $5, %eax
10 leave
11 ret
```

Table 4.6: Understanding PIE Overhead

Since PIE code is relocatable, its location in virtual memory must be determined at run time. To determine the location of data structures, the position independent code is automatically augmented by the compiler with instructions used to determine the needed location information. Note that on line 4 the PIE version deviates by going to `L3`, which was automatically inserted by the compiler. In line 6 the function's address is popped into the `ecx` register. This address is then used in line 7 to calculate an offset into the global offset table (GOT). In line 8, the PIE instructions again match those in the non-PIE version, by moving the variable value into the `eax` register.

These additional instructions are the source of the PIE runtime performance overhead. Therefore code compiled with PIE options has more instructions and takes longer to execute than its non-PIE counterpart. For example, consider Table 4.7 that contains details about the executable size and running time increases associated with PIE and the CPU2000 integer benchmarks.

In the CPU2000 integer benchmark suite, the PIE versions of each benchmark are larger by 9–21%. Of interest is that the average code size increase caused by the PIE location instructions is roughly equivalent to the average performance overhead, which is a

Bench-mark	Base Code Segment Size (bytes)	Base GOT Size (bytes)	PIE Code Segment Size (bytes)	PIE GOT Size (bytes)	Code Size Increase (%)	GOT Size Increase (%)	Benchmark Performance Overhead (%)
gzip	34,900	116	40,192	352	15.16	203.45	16.95
vpr	119284	148	135376	328	13.49	121.62	4.93
gcc	1337860	236	1462224	3008	9.3	1174.58	12.12
mcf	8852	84	9648	112	8.99	33.33	4.66
crafty	192004	176	216608	1808	12.81	927.27	22.41
parser	118228	148	134272	376	13.57	154.05	5.64
perlbmk	536548	392	596176	1884	11.11	380.61	47.62
gap	435620	144	482560	1076	10.78	647.22	14.29
vortex	526260	164	576016	1388	9.45	746.34	17.84
bzip2	32420	92	39408	312	21.55	239.13	9.62
twolf	188532	112	220928	1244	17.18	1010.71	2.14
Average	-	-	-	-	13.04	512.58	14.38

Table 4.7: CPU2000 PIE Overhead Analysis

reasonable outcome.

4.2.2 LMBench Micro-Benchmark

The LMBench benchmark suite differs from CPU2000 because it strives to benchmark general operating system performance rather than the computational performance of a set of applications. It provides a meaningful way to measure the performance impact of address space randomization on everyday computing. The LMBench operating system benchmark suite consists of five sets of micro-benchmarks, each of which is designed to focus on a specific aspect of operating system performance: system calls & process operations, context switching, file & virtual memory system latency, local communication latency, and local communication bandwidth. The series of figures in this section show the overhead for each set of benchmarks; Appendix C contains the results for absolute run times. All LMBench benchmarks were run nine times. Results and analysis use the arithmetic mean time for each benchmark.

Figure 4.4 shows the performance overhead for process creation and system calls. A noteworthy detail is that ASLP slowed `fork()` and `exec()` operations by 6.86% and

12.53% respectively. Process creation overhead is the primary location of expected overhead from address space randomization techniques like PaX ASLR, Exec-Shield, and our ASLP because additional operations are inserted into the process creation functions. The process creation functions are augmented with randomization hooks that change the location of the stack, heap, and `mmap` allocations. Such code is not present in the vanilla kernel. However, observe that both PaX ASLR and Exec-Shield have consistently higher overheads for the same tasks: 13.83–21.96% and 12.63–32.18% respectively. ASLP incurs less overhead for these operations because both Exec-Shield and PaX ASLR are doing additional operations that could not be disabled without modification to their respective source codes. However, even though those extra operations prevent a direct comparison, we are confident stating that ASLP has comparable overhead to these two closely related projects because ASLP is roughly twice as fast at both operations, which leaves a comfortable margin of error. This estimation is good enough because we are only interested in demonstrating that ASLP has reasonably similar performance when compared to closely related techniques.

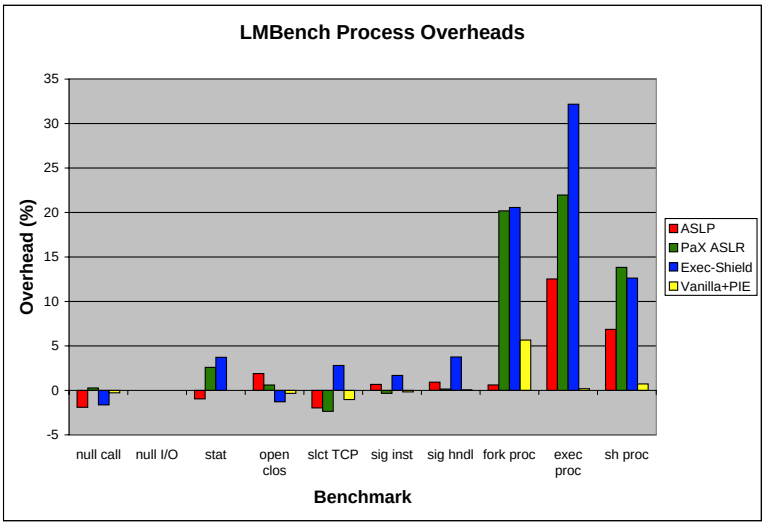

Figure 4.4: LMBench Process Operation Overheads

The context switching overhead results in Figure 4.5 give a similar story to process operation results. We incur approximately 10% overhead in two tests: 2 process 0 KB and 16 process 16 KB context switch. However, in both cases we incur less overhead than closely related projects. See Figure C.2 for a numeric representation of the context switching data

set.

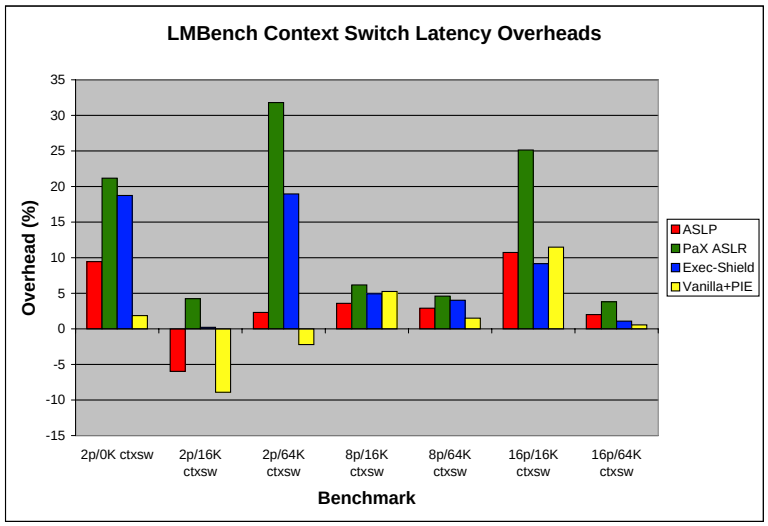

Figure 4.5: LMBench Context Switch Overheads

File and virtual memory (VM) system latency results are shown in Figure 4.6. This area of operations is the second location where we expect to observe performance overhead because each `mmap` operation has been instrumented with additional randomization instructions. Randomization instructions are executed every time an allocation is made via `mmap` and the Mmap Latency test does exactly that. More specifically, every time `mmap` is called, the location of the allocation is randomly generated, verified against consistency checks, and finally approved. We therefore expect an increase in `mmap` latency because we are increasing the number of instructions to complete `mmap` allocations. Although ASLP incurs an average of 12.51% overhead for `mmap` latency, this added latency is still reasonably similar to closely related techniques. Again our goal is not to claim a speed gain over related works, but rather to achieve similar performance. Also notice that although ASLP provides more randomized bits for `mmap` allocations than both PaX ASLR and Exec-Shield, the performance is comparable. See Table C.8 for a numeric representation of the file and virtual memory system data set.

According to the results shown in Figure 4.7, local communication latency initially appears to be severely affected by our ASLP technique. Specifically, according to LMBench, ASLP incurs more than 3377% overhead for TCP connection latency. This is unexpected

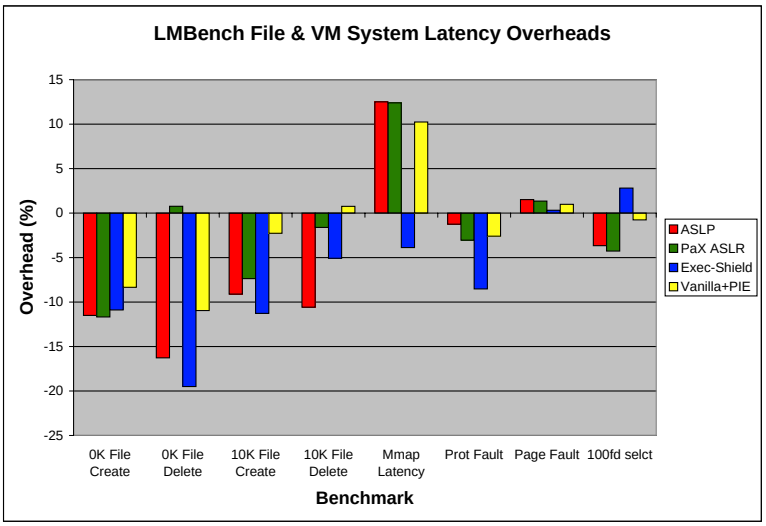

Figure 4.6: LMBench File and VM System Latency Overheads

since changes were made only to the user space memory management subsystems; the Linux kernel networking code and the memory management code are quite separate.

Figure 4.8 shows another apparently significant performance feature: local communication bandwidth is *improved* by more than 32%. Again this behavior is unexpected because there were no changes to the networking code. In fact, our modifications should only decrease performance because of the additional instructions added to memory allocation routines.

The opposing nature of these two TCP measurements seems contradictory. Particularly, the roughly 30% improvement in performance seems out of place because although our modifications were only to the memory management portion of the kernel and not to the network code, we only add instructions that should lead to a decrease in performance. Upon inspection of the raw data, several entries were missing for both the TCP latency and TCP bandwidth tests run on our ASLP kernel, the PaX ASLR kernel, and the vanilla kernel running a PIE executable. Missing entries indicate that the corresponding runs failed. Because the test does not fail on the vanilla kernel this indicates a bug exists in either the TCP benchmark or all of the three independently developed kernel modifications. Further, of the tests that completed, the variance in the data sets were much higher than for the vanilla kernel. The resulting data indicates that some test runs had better performance

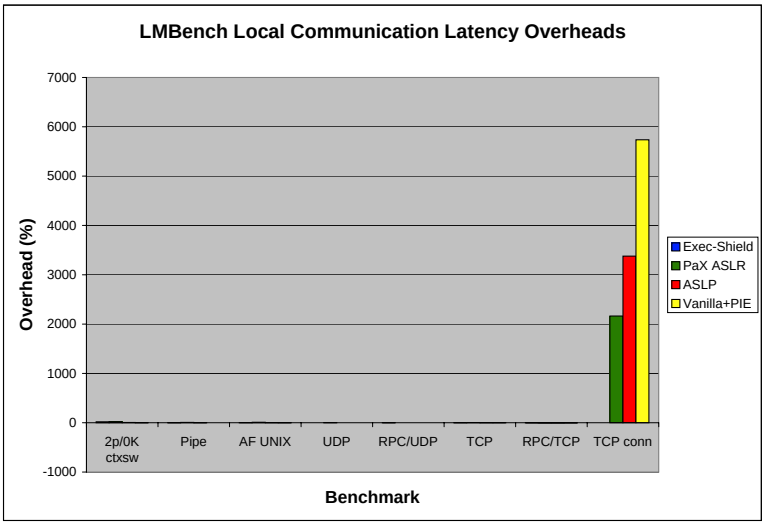

Figure 4.7: LMBench Communication Latency Overheads

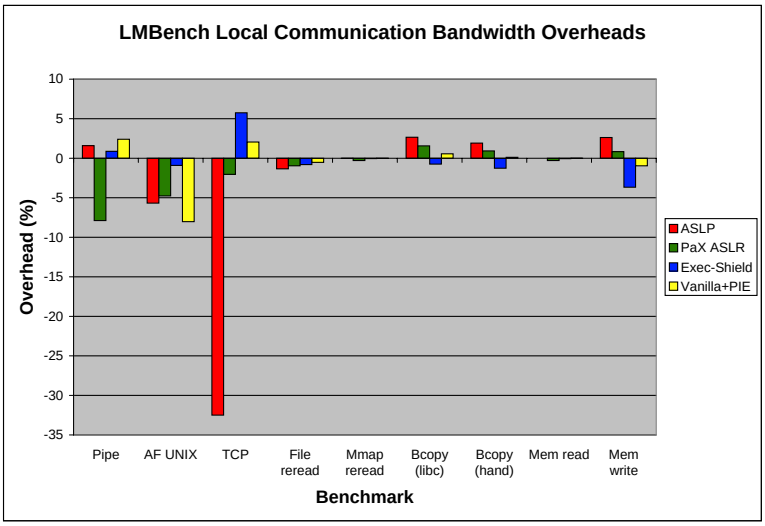

Figure 4.8: LMBench Communication Bandwidth Overheads

than vanilla kernel and some had worse. The fact that this occurs on several different kernels suggests either a bug in the LMBench TCP benchmarks or buggy behavior shared by several randomization methods. Although it seems most likely that the bug resides within the benchmark suite, without considerable code analysis of all affected kernels and benchmarks we do not have enough information to decide. Such analysis is outside the scope of this thesis and additional robustness testing is left for future work.

However, the Apache Benchmark results in Section 4.2.3 put the significance of the LMBench TCP micro-benchmarks into perspective. The LMBench TCP latency benchmark is a synthetic benchmark that measures the speed at which a specially written client program can establish a TCP connection to a specially written local server. When compared to a performance evaluation using the Apache web server, a real world TCP-based client-server application, in which ASLP has no test run failures and less than 1% performance overhead, the importance of the LMBench TCP benchmarks is greatly diminished in the context of this performance assessment.

4.2.3 Apache Benchmark

The Apache Benchmark [16] measures the performance of an Apache HTTP server [17] in terms of how quickly it can serve a given number of HTML pages via TCP/IP. The test environment places two identical test computers on a 100 megabit per second switched network where one computer acts as HTTP client and the other acts as HTTP server. The client system configuration remains constant throughout all tests. To test the various kernels we boot the server using the appropriate kernel before running the benchmark. To test PIE, we recompile the Apache HTTP server with PIE options and boot from the vanilla kernel.

Our Apache Benchmark configuration makes 1 million requests, in simultaneous batches of 100, for a static HTML page of 1881 bytes, which includes 425 bytes of images. The Apache HTTP server configuration was modified from the default configuration to preemptively spawn 100 worker processes to handle incoming requests. This modification removes the previously documented affect of process startup overhead on this benchmark.

Figure D.1 shows the overhead results of each configuration. The results indicate the absence of major overhead induced by our ASLP kernel for this benchmark. Further, the ASLP overhead is again comparable to both PaX ASLR and Exec-Shield.

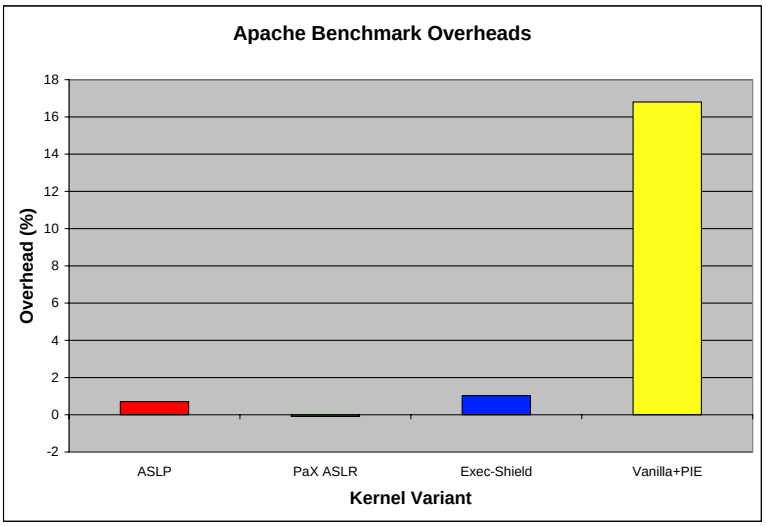

Figure 4.9: Apache Benchmark Overheads

4.2.4 Summary of Performance Evaluation

The CPU2000 results indicate that ASLP incurs 0.27% performance overhead in scientific computing environments, which is comparable to other address space randomization techniques. Although scientific computing is not the primary target environment, the small performance overhead indicates that deployment of ASLP is feasible in mixed environments concerned with providing protection against remote memory corruption exploits.

The LMBench results show a more granular perspective on the types of operating system functions that incur performance overhead. Of highest significance, process creation incurs 7-13% overhead, context switching incurs less than 11% overhead, and `mmap` incurs less than 13% latency overhead.

The Apache Benchmark results indicate that ASLP incurs less than 1% performance overhead. The results also show ASLP performance is reasonably similar to closely related techniques. We favor the Apache Benchmark over the LMBench TCP measurements because the Apache Benchmark evaluation is based on widely used, real world code.

In summary, the majority of ASLP performance overhead occurs during process startup. Applications that frequently `mmap` will notice an additional latency because additional randomization instructions are executed on every `mmap` call. However, ASLP overhead

is comparable to than other kernel-based address randomization techniques while randomizing more bits of virtual memory allocations.

Chapter 5

Kernel Space Randomization Proposal

ASLP and other related work focuses primarily on randomizing allocations made in user space. That is, the techniques all randomize the user stack, heap, and `mmap` in some way, which reside in the user space virtual address space of 0 to 3 GB. However, with the exception of PaX kernel stack randomization [36], the kernel address space between 3 to 4 GB in virtual memory is left untouched. This section proposes two novel techniques for randomizing kernel space allocations.

Figure 5.1 depicts kernel space address layout for the default configuration on x86 architecture based on the survey [18] by Mel Gorman.

There are two straightforward options to randomize the layout of kernel space memory: either permute the regions as we did in user space or add a random amount to each gap region thereby shifting the contents of each region. However, the kernel address space differs from user address space in several important ways.

In user space, processes allocate relatively small (typically KB) regions in their 3 GB address space. These regions last only as long as their corresponding process. In kernel space, the regions are much larger and the address space is one third as large. All kernel space memory regions are defined during the boot process. Allocations may be made within

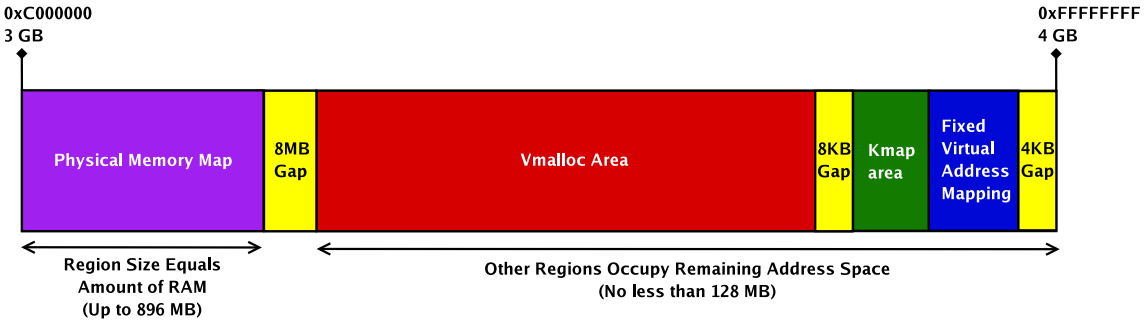

Figure 5.1: Kernel Address Space Memory Layout

each region as hardware and software interrupts occur, but region boundaries do not move, meaning that once the system is booted, the location of each region does not change even as different processes are switched in and out.

The Physical Memory Map region size depends on the amount of RAM installed in the computer. This region will match the RAM size up to 896 MB. Further, the location of the Fixed Virtual Address Mapping region is determined during kernel compilation. Although the presence of this region depends on the configuration options during kernel compilation, if present, it cannot be moved at all. Finally, notice that only slightly more than 8 MB of the kernel address space is unused, which means there is little space left to move the large regions.

We first consider permuting the regions as we did in user space. With three movable regions, we have only six (3!) possible permutations, as Table 5.1 shows:

Permutation	Movable Kernel Space Regions		
1.	map|8 MB	vmalloc|2 pg	kmap
2.	map|8 MB	kmap	vmalloc|2 pg
3.	vmalloc|2 pg	map|8 MB	kmap
4.	vmalloc|2 pg	kmap	map|8 MB
5.	kmap	map|8 MB	vmalloc|2 pg
6.	kmap	vmalloc|2 pg	map|8 MB

Table 5.1: Possible Permutations for Kernel Space Regions

Having only 8 MB of page-aligned address space in which to move the large regions means there are only 2048 possible placements for any given region. Using the previously

derived estimated guess rate of 151.7 guesses per second, this randomization is clearly insufficient. It also adds complexity to the kernel space memory management code. We wish to make modifications only if the benefits outweigh the added complexity and in this case, they do not. So, we eliminate the permutation option and investigate the padding option.

The size of the small gaps between regions may be padded by a random amount during each boot. With this approach, the amount of randomization is proportional to the amount of wasted space (as are all pad-based randomization techniques). This method is unappealing in an already cramped address space. Achieving substantial page-aligned randomization requires a many MB and preferably several GB. Pads of such large magnitude would waste large chunks of limited space in the 1 GB kernel address space. Thus we conclude that the padding approach is also not worthy of pursuit.

A further weakness to both of these approaches is that once the regions are randomized during the boot process, no further randomizations are made until the computer is rebooted. Although the two randomization techniques are technically dynamic since they change during every boot, the address space layout of the kernel is relatively long lived, so an attacker would conceivably have enough time to brute-force guess the needed information.

With the elimination of both options, permutation and padding of the kernel address space are not promising avenues for kernel address space randomization. However, we present two novel options that randomize allocations made in kernel space.

5.1 Randomizing Kernel Data Structures

The size of the physical memory map is equal to the amount of physical memory installed in the computer, up to 896 MB. Physical memory address 0 is mapped to virtual address 3 GB (0xC0000000), with the rest of physical memory mapped to contiguous subsequent virtual addresses. The contents of the region correspond to the contents of that physical memory one to one. This mapping allows translation between virtual and physical addresses by simply adding or subtracting 0xC0000000. The purpose of this region is to allow efficient allocation and management of pages in physical memory using two complimentary algorithms: the buddy allocator and the slab allocator. Although physical memory map region is technically redundant because page tables could be used for such translations,

using page table translation requires several times more calculations compared to a simple one step addition or subtraction. This redundancy is tolerated because of the importance of fast memory allocation.

Low level memory allocations are principally controlled by the buddy allocator, which is responsible for the allocation of physical memory pages. The slab allocator is logically positioned on top of the buddy allocator and is responsible for efficiently handling relatively small allocations of frequently used sizes. This two tiered approach is desirable because finding, allocating, and initializing memory is relatively expensive. The slab allocator can avoid initialization steps by using pre-allocated memory of the correct size. For example, when a process is spawned, several kernel data structures are created to track the process and its resources. Those data structures, which are always the same size, are retrieved from the slab allocator's cache of correctly sized memory slabs. These memory slabs are typically fractions of contiguous physical pages, which the buddy allocator previously issued to the slab allocator.

It is from this interconnection that the possibility exists for randomizing kernel data structures. We propose to modify or replace the buddy allocator, slab allocator, or both to introduce unpredictability in their allocations.

The technicalities of randomizing allocations in the physical memory map exist in much greater numbers than those of other regions. This is primarily because allocations in this region correspond directly to physical memory. In effect, we propose the randomization of physical memory.

Both the buddy allocator and slab allocator are written for the express purpose of efficiently finding, allocating, and otherwise managing physical memory while minimizing internal and external fragmentation. The slab allocator is also written to organize allocations in such a way that optimizes the use of hardware caches. Introducing randomness into this carefully balanced system will almost certainly be accompanied by a performance penalty, an increase in unusable memory due to fragmentation, or both. Due to the sensitive balance of the existing system it is likely that new randomization-friendly memory allocation mechanisms will be required.

An efficient, portable, and sufficiently random low level memory allocation implementation would likely require years of development and testing. Although the scale of this proposal is outside the scope of this work, the potential to randomize the layout of physical memory makes the pursuit of this goal attractive enough to warrant further research de-

spite technical challenges. Future attacks may use assumptions about physical memory and kernel memory layout. Introducing unpredictability would potentially thwart those attacks.

5.2 Randomizing Vmalloc

Due to its limited size, the Vmalloc area is used sparingly. Its space is reserved for mapping loadable kernel modules and swap maps into virtual memory. To randomize these allocations, we propose a method similar to that done by our ASLP `mmap` randomization, where each `vmalloc()` call would return an allocation with a random virtual address in the Vmalloc region. Since the Vmalloc area can be no smaller than approximately 100 MB, there is a relatively large amount of room for random placement of each allocation without wasting any address space.

The primary limitation of this approach is that the Vmalloc area may be so small that external fragmentation caused by random allocation may considerably reduce the usable address space. That is, even though enough free pages may be available in the virtual address space of the Vmalloc area, if the virtual addresses are not contiguous they cannot be used for the same allocation.

The affect of the external fragmentation will vary depending on the size of the Vmalloc area, with a smaller Vmalloc area being affected most noticeably. To test the affects of various region sizes, the randomized system needs to be booted with various amount of RAM installed, from 128 MB to 4 GB. With only 128 MB of RAM installed, the Vmalloc area is at its biggest; with 4 GB of RAM installed, the Vmalloc area is at its smallest.

5.3 Summary of Kernel Space Randomization Proposal

Future attacks may use assumptions about physical memory and kernel memory layout. Introducing unpredictability would potentially thwart those attacks. It is therefore important to consider kernel space randomization possibilities. The two options proposed in this chapter would dynamically randomize both kernel address space and physical memory without padding, a key advantage in the relatively small address space of the kernel.

Chapter 6

Conclusions & Future Work

This thesis investigates methods for improving kernel-based address space randomization techniques for the purpose of increasing resistance to memory corruption attacks. ASLP randomizes the location of the user stack, heap, and shared libraries with an order of magnitude greater unpredictability than other kernel-based approaches without the use of large pads. Using various benchmarks we found that it is possible to achieve better randomization for process virtual memory layout without incurring obtrusive performance overhead. With ASLP, runtime overhead is less than 1%, as indicated by both the SPEC CPU2000 Integer Benchmark as well as the Apache Benchmark. These performance figures are comparable to other kernel-based address space randomization techniques that provide less randomization.

This thesis also validates the use of address space randomization by demonstrating that the randomization provided by ASLP dramatically reduces the speed at which worms can propagate throughout the Internet. The increase in time needed to exploit targets introduced by randomization means that the fastest infection time for worms relying on the absolute location of either the user stack, heap, or an `mmap` allocation is on the order of hours, not minutes.

Kernel-based address layout permutation provides blanket protection against remote memory corruption exploits for all system components without need to patch or recompile user-level software. Address randomization is only effective against network-based attacks; local attacks are outside the scope of this technique. We acknowledge the

drawbacks of kernel-based address randomization from the perspective of systems availability and micro-security, but argue that the ease of management, scope of protection, and macro-security effectiveness are more than sufficient to justify its limitations.

6.1 Future Work

Using the exiting ASLP kernel as a foundation, several interesting avenues for future work exist. The remainder of this section is devoted to the discussion of future work.

Additional Reliability Testing. We automated the testing of dozens of common applications and, in some cases, ran them millions of times to ensure their reliability. To reach general acceptance, kernel features must not break otherwise working applications [41]. Although we have made efforts to test ASLP with a wide range of common user and server applications, more testing remains necessary to convincingly establish a degree of reliability comparable to the vanilla Linux kernel.

Network Intrusion Prevention System to Block Derandomizing Attacks. Derandomization attacks against ASLP must make thousands guess to identify the randomized address(es) needed to exploit a vulnerable program. These guesses traverse the network and can therefore be intercepted by Network Intrusion Prevention Systems (NIPS). Further investigation is needed to study and model derandomizing traffic patterns to create filters capable of detecting and blocking the attacks. With the protection of such an NIPS, systems employing address randomization should be much more difficult to successfully attack.

Combination of User Land and Kernel Land Address Randomization. A lucrative avenue to improve randomization is to merge the benefits of user land and kernel land techniques. The incorporation of portions of user level techniques like [4] into kernel level techniques like ASLP would yield an unprecedented level of protection. Specifically, randomly permuting the layout of code and data from executables and shared libraries could be combined with coarse-grained kernel level techniques capable of efficiently reordering large portions of memory.

Non-invasive Crash Monitoring. When an attacker incorrectly guesses an address needed to exploit a vulnerable program, the target program begins executing instructions stored at an arbitrary memory location, which typically results in a crash. The brute-force nature of a derandomization attack means the target program will repeatedly crash as incorrect guesses are made. Tools like SegvGuard [28] detect these repetitive crashes and prevent the target program from respawing, legitimately judging that an unavailable system is better than an available, but compromised system.

However, other options may exist that are less invasive to legitimate users. Many networks services, such as HTTP, are based on TCP, which requires two way communication to complete a connection handshake before any data is exchanged. With this information one can assume that active connections to TCP services are not spoofed. Using this assumption and the work done by the author of SegvGuard, it should be possible to identify and block the IP address associated with connections that cause a TCP service to repeatedly crash. The ideal result is that derandomizing attacks could be detected and blocked at the host level. Further study and testing are needed to assess the feasibility and effectiveness of this and other potential methods to block derandomizing attacks at the host level without degrading legitimate user access.

Bibliography

[1] Anonymous. Once upon a free(). *Phrack Magazine*, 11(57), August 2001. Available from URL `http://www.phrack.org/phrack/57/p57-0x09`.

[2] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference*, June 2000. Available from URL `http://www.research.avayalabs.com/project/libsafe/doc/usenix00.ps`.

[3] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In V. Paxson, editor, *Proceedings of the 12th USENIX Security Symposium*, pages 10–20, August 2003. Available from URL `http://www.seclab.cs.sunysb.edu/seclab/pubs/papers/ao.pdf`.

[4] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD*, pages 271–286, July 2005. Available from URL `http://www.seclab.cs.sunysb.edu/seclab/pubs/papers/usenix_sec05.pdf`.

[5] Joshua Brindle. Hardened gentoo. Available from URL `http://www.gentoo.org/proj/en/hardened/`.

[6] Peter Busser. Adamantix. Available from URL `http://www.adamantix.org/`.

[7] Peter Busser. Paxtest. Available from URL `http://www.adamantix.org/paxtest/`.

[8] The Fedora Community. The fedora project. Available from URL `http://fedora.redhat.com/`.

[9] Standard Performance Evaluation Corporation. Spec cpu2000 v1.2. Available from URL `http://www.spec.org/cpu2000/`.

[10] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium, San Antonio, Texas*, January 1998. Available from URL `http://wirex.com/~crispin/usenixsc98.pdf`.

[11] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *10th USENIX Security Symposium, Washington, D.C.*, August 2001. Available from `http://wirex.com/~crispin/formatguard.pdf`.

[12] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium*, pages 91–104, August 2003. Available from URL `http://wirex.com/~crispin/pointguard_usenix_security2003.pdf`.

[13] Theo de Raadt. Openbsd. Available from URL `http://www.openbsd.org/`.

[14] Ulrich Drepper. Security enhancements in red hat enterprise linux (besides selinux). Available from URL `http://people.redhat.com/drepper/nonselsec.pdf`, June 2004.

[15] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems, Los Alamitos, CA*, pages 62–72. IEEE Computer Society Press, 1997. Available from URL `http://www.cs.unm.edu/~immsec/publications/hotos-97.pdf`.

[16] Apache Software Foundation. Apache benchmark. Available from URL `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[17] Apache Software Foundation. Apache http server project. Available from URL `http://httpd.apache.org`.

[18] Mel Gorman. Understanding the linux virtual memory manager. Master's thesis, University of Limerick, 2004. Available from URL http://www.csn.ul.ie/~mel/projects/vm/guide/pdf/understand.pdf.

[19] The IEEE and The Open Group. *The Open Group Base Specifications: mmap*, 6 edition, 2004. Availale from URL http://www.opengroup.org/onlinepubs/009695399/functions/mmap.html.

[20] The IEEE and The Open Group. *The Open Group Base Specifications: rand*, 6 edition, 2004. Availale from URL http://www.opengroup.org/onlinepubs/009695399/functions/rand.htm.

[21] Red Hat Inc. Red hat. Available from URL http://www.redhat.com/.

[22] Red Hat Inc. Red hat enterprise linux. Available from URL http://www.redhat.com/software/rhel/.

[23] Larry McVoy and Carl Staelin. Lmbench: Tools for performance analysis. Available from URL http://www.bitmover.com/lmbench/.

[24] Ingo Molnar. Exec-shield. Available from URL http://people.redhat.com/mingo/exec-shield/.

[25] Ingo Molnar. Re: Patch 4/6 randomize the stack pointer, January 2005. Available from URL http://lwn.net/Articles/121848/.

[26] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. The spread of the sapphire/slammer worm, 2003. Available from URL http://www.cs.berkeley.edu/~nweaver/sapphire/.

[27] George Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Portland, Oregon*, pages 128–139, January 2002. Available from URL http://www.cs.berkeley.edu/~smcpeak/papers/ccured_popl02.pdf.

[28] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack Magazine*, 11(58), December 2001. Available from URL http://www.phrack.org/phrack/58/p58-0x04.

[29] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), November 1996. Available from URL `http://www.phrack.org/phrack/49/P49-14`.

[30] Scut. Exploiting format string vulnerabilities, March 2001. Available from URL `http://julianor.tripod.com/teso-fs1-1.pdf`.

[31] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In V. Atluri, B. Pfitzmann, and P. McDaniel, editors, *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, D.C.* ACM, October 2004. Available from URL `http://www.stanford.edu/~blp/papers/asrandom.pdf`.

[32] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, August 2002. Available from URL `http://www.cs.berkeley.edu/~nweaver/cdc.web/cdc.pdf`.

[33] Computer Emergency Response Team. Cert advisory ca-2003-04 ms-sql server worm, January 2003. Available from URL `http://www.cert.org/advisories/CA-2003-04.html`.

[34] The PaX Team. The pax project, 2001. Available from URL `http://pax.grsecurity.net/`.

[35] The PaX Team. Address space layout randomization, March 2003. Available from URL `http://pax.grsecurity.net/docs/aslr.txt`.

[36] The PaX Team. Kernel stack randomization, January 2003. Available from URL `http://pax.grsecurity.net/docs/randkstack.txt`.

[37] The PaX Team. The main pax document: Overall description, November 2003. Available from URL `http://pax.grsecurity.net/docs/pax.txt`.

[38] The PaX Team. mmap() randomization, January 2003. Available from URL `http://pax.grsecurity.net/docs/randmmap.txt`.

[39] The PaX Team. Userland stack randomization, February 2003. Available from URL `http://pax.grsecurity.net/docs/randustack.txt`.

[40] The PaX Team. Vma mirroring: The core of segmexec and randexe, October 2003. Available from URL `http://pax.grsecurity.net/docs/vmmirror.txt`.

[41] Linus Torvalds. Re: [patch] futex requeueing feature, futex-requeue-2.5.69-d3, May 2003. Available from URL `http://lwn.net/Articles/32980/`.

[42] United States Computer Emergency Readiness Team (US-CERT). Technical cyber security alerts. Available from URL `http://www.us-cert.gov/cas/techalerts/`.

[43] Tony Warnock. Random-number generators. *Los Alamos Science*, 1987. Available from URL `http://www.fas.org/sgp/othergov/doe/lanl/pubs/00418729.pdf`.

[44] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In A. Fantechi, editor, *Proceedings of the 22nd Sympomsium on Reliable Distributed Systems*, pages 260–269. IEEE Computer Society, October 2003. Available from URL `http://www.csc.ncsu.edu/faculty/junxu/Papers/SRDS2003_final_trr.pdf`.

[45] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. Accmon: Automatically detecting memory-related bugs. In *Proceedings of the 37th annual International Symposium on Microarchitecture, Portland, Oregon*, pages 269–280, December 2004. Availble from URL `http://opera.cs.uiuc.edu/paper/Micro04-AccMon.pdf`.

Appendix A

GNU/Linux User Address Space

To understand the methods presented in this thesis, it is first necessary to understand the handling and organization of memory by the operating system and memory architecture. The memory layout produced by different operating systems will be different, even on the same hardware. The same is also true for different versions of the same operating system. However, the fundamental memory layout concepts are still comparable. For the purpose of this thesis, the GNU/Linux kernel version 2.4.31 was chosen for its source availability, popularity, stability, and the relative abundance of documentation. The focus is further directed toward the Intel i386 and compatible processors due to their current dominance in the PC market. Unless stated otherwise, all subsequent discussion is relevant to the chosen kernel on the 32-bit Intel architectures (IA32).

Each process has it's own virtual 32-bit address space ranging sequentially from 0 bytes to 4 gigabytes (GB). This address space is divided into two fundamental areas: user space and kernel space. User space exists from 0 bytes to 3 GB, while kernel space exists from 3 GB to 4 GB. Using this model of virtual address space, from the perspective of a user process there are 3 GB of usable memory in which allocation can be made, regardless of any other concurrent processes in the system.

An executable file is divided into two primary regions: code and data. The data segment is further divided into three regions: a section for initialized data, the BSS for uninitialized data, and the heap for dynamically allocated data. As defined by the Executable and Linking Format (ELF) specification, a program code segment is mapped at

virtual address 0x08048000, which is approximately 128 MB from the beginning of the address space. This leaves a large gap of unused space at the beginning of virtual memory, which is dedicated for dereferencing null pointers to prevent collision with other allocated data. Immediately following the code segment is the data segment. Following the BSS segment is the heap. The heap expands and contracts to match the runtime requests for memory. Following the heap is another large gap to the the `mmap` base, which is traditionally located at 1 GB. The gap grants the heap several hundred megabyte of allocatable virtual address space. The `mmap` base is where shared libraries, otherwise known as dynamically shared objects (DSO), are mapped into memory. Then there is another large gap into which many files may be mapped into memory. Finally, at 3 GB in virtual address space, the user mode stack begins. The stack grows down in the direction of the shared libraries.

Figure A.1 depicts the standard memory layout of a typical process with two shared libraries. Note the three colored regions: heap, stack, and the dynamically shared objects (DSO) area.

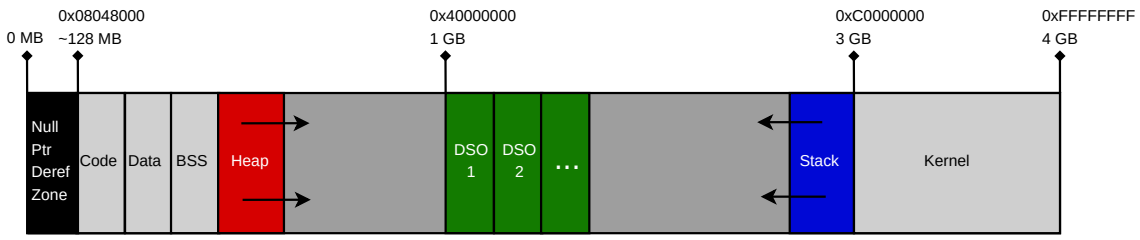

Figure A.1: Example Normal Process Memory Layout

This virtual address space is mapped to the physical addresses corresponding to physical memory by the memory management unit at the hardware level. However, Linux processes only directly interact with virtual addresses, so lower levels of memory management will not be covered further.

Appendix B

CPU2000 Benchmark Data

Results from the SPEC CPU2000 Integer Benchmark Suite. This appendix contains the raw CPU2000 integer benchmark data for the following configurations: vanilla kernel, PaX ASLR kernel, Red Hat Exec-Shield kernel, ASLP kernel, and PIE compilation with vanilla kernel. Table B.1 offers a brief description of each benchmark provided in the CPU2000 documentation.

CPU2000 Integer Benchmark Descriptions	
Name	Description
165.gzip	Data compression utility
175.vpr	FPGA circuit placement and routing
176.gcc	C compiler
181.mcf	Minimum cost network flow solver
186.crafty	Chess program
197.parser	Natural language processing
253.perlbmk	Perl
254.gap	Computational group theory
255.vortex	Object Oriented Database
256.bzip2	Data compression utility
300.twolf	Place and route simulator

Table B.1: Descriptions of CPU2000 Integer Benchmarks

Figure B.1 contains the CPU2000 integer benchmark runtime results in chart form.

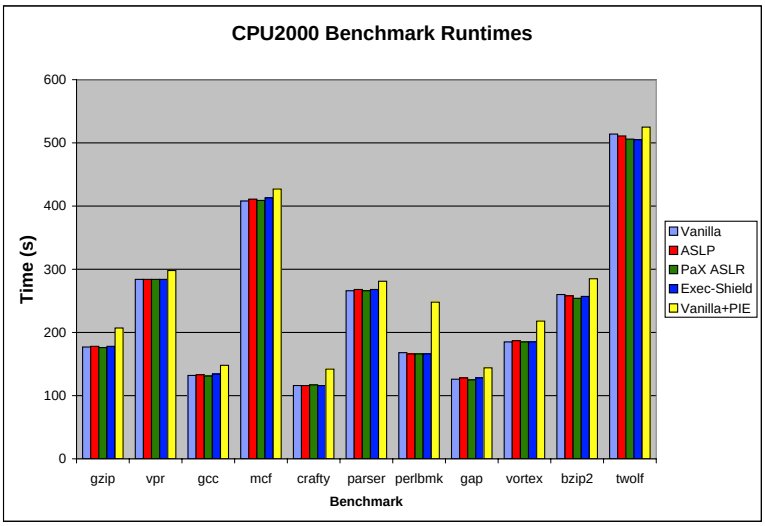

Figure B.1: CPU2000 Integer Benchmark Runtimes

Table B.2 contains the CPU2000 integer benchmark runtime results in numeric form.

Table B.3 contains the CPU2000 integer performance overhead measurements in numeric form.

CPU2000 Integer Benchmark Run Times (seconds)					
Benchmark	Vanilla	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
gzip	177	178	176	178	207
vpr	284	284	284	284	298
gcc	132	134	131	133	148
mcf	408	413	409	411	427
crafty	116	116	117	116	142
parser	266	268	266	268	281
perlbmk	168	166	166	166	248
gap	126	128	125	128	144
vortex	185	185	185	187	218
bzip2	260	257	254	258	285
twolf	514	505	506	511	525
Total	2636	2634	2619	2640	2923

Table B.2: SPEC CPU2000 Benchmark Run times (seconds)

CPU2000 Integer Benchmark Overheads (%)				
Benchmark	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
gzip	0.56	-0.56	0.56	16.95
vpr	0	0	0	4.93
gcc	1.52	-0.76	0.76	12.12
mcf	1.23	0.25	0.74	4.66
crafty	0	0.86	0	22.41
parser	0.75	0	0.75	5.64
perlbmk	-1.19	-1.19	-1.19	47.62
gap	1.59	-0.79	1.59	14.29
vortex	0	0	1.08	17.84
bzip2	-1.15	-2.31	-0.77	9.62
twolf	-1.75	-1.56	-0.58	2.14
Average	0.14	-0.55	0.27	14.38

Table B.3: SPEC CPU2000 Benchmark Overhead (%)

Appendix C

LMBench Benchmark Data

Results from the LMBench Benchmark Suite. This appendix contains the raw LMBench benchmark data for the following configurations: vanilla kernel, PaX ASLR kernel, Red Hat Exec-Shield kernel, ASLP kernel, and PIE compilation with vanilla kernel.

C.1 LMBench: Process Operations

Table C.1 offers a brief description of each process operation benchmark in the LMBench suite.

Figure C.1 contains the LMBench process operation runtime results in chart form.

Table C.2 contains the process benchmark runtime results in numeric form.

Table C.3 contains the process performance overhead measurements in numeric form.

C.2 LMBench: Context Switching

Table C.4 offers a brief description of each context switching benchmark in the LMBench suite.

Figure C.2 contains the LMBench context switching latency results in chart form.

Table C.5 contains the context switching latency results in numeric form.

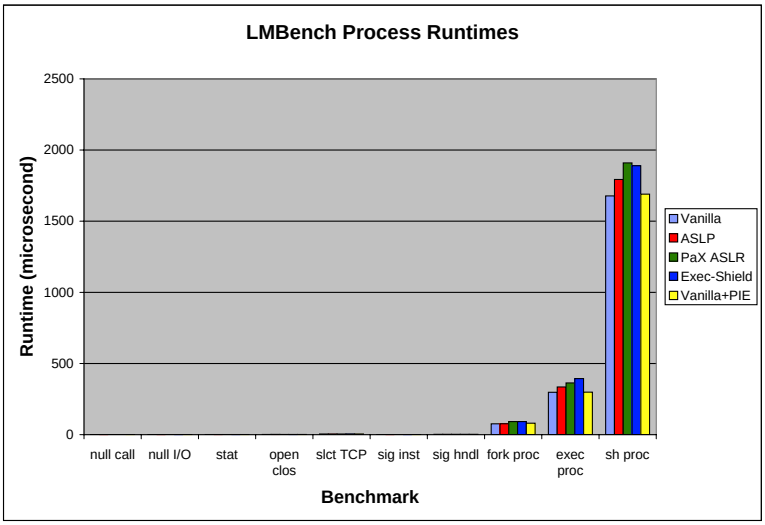

Figure C.1: LMBench Process Operation Runtimes

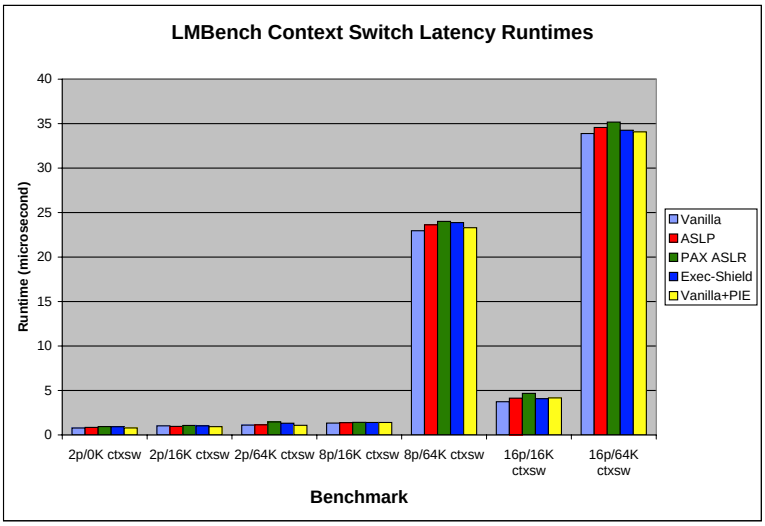

Figure C.2: LMBench Context Switch Runtimes

Process Operations Benchmark Descriptions	
Name	Description
Null Call	Times a `getpid()` system call.
Null I/O	Times a write to `/dev/null`.
Stat	Times `stat()` operation a file whose inode is already cached.
Open Clos	Measures time to `open()` and then `close()` a file.
Slct TCP	Measures time to do a select on a TCP file descriptor.
Sig Inst	Times a signal installation.
Sig Hndl	Times a signal catch.
Fork Proc	Times forking an identical process copy and have one exit.
Exec Proc	Measures time to create a process and have that process run a program.
Sh Proc	Measures time to create a process and have that process run a program by asking the system shell to find that program and run it.

Table C.1: Descriptions of LMBench Process Operations Benchmarks

Table C.6 contains the context switching latency overhead measurements in numeric form.

C.3 LMBench: File and VM System

Table C.7 offers a brief description of each file and VM system benchmark in the LMBench suite.

Figure C.3 contains the LMBench file and virtual memory latency results in chart form.

Table C.8 contains the file and VM system benchmark run time results in numeric form.

Table C.9 contains the file and VM system performance overhead measurements in numeric form.

C.4 LMBench: Local Communication Latency

Table C.10 offers a brief description of each communication latency benchmark in the LMBench suite.

LMBench Process Operations Benchmark Run Times (microseconds)					
Benchmark	Vanilla	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
null call	0.41	0.4	0.41	0.4	0.41
null I/O	0.47	0.47	0.47	0.47	0.47
stat	1.05	1.09	1.07	1.04	1.05
open clos	1.64	1.62	1.65	1.68	1.64
slct TCP	4.96	5.1	4.84	4.86	4.91
sig inst	0.66	0.67	0.66	0.67	0.66
sig hndl	2.16	2.24	2.16	2.18	2.16
fork proc	76.24	91.92	91.63	76.71	80.56
exec proc	298	393.89	363.44	335.33	298.56
sh proc	1677.89	1889.78	1909.89	1793	1690.11

Table C.2: LMBench Process Operations Benchmark Run times

Figure C.4 contains the LMBench local communication latency runtime results in chart form.

Table C.11 contains the communication latency benchmark runtime results in numeric form.

Table C.12 contains the communication latency performance overhead measurements in numeric form.

C.5 LMBench: Communication Bandwidth

Table C.13 offers a brief description of each communication bandwidth benchmark in the LMBench suite.

Figure C.5 contains the LMBench local communication bandwidth results in chart form.

Table C.15 contains the communication bandwidth performance overhead measurements in numeric form.

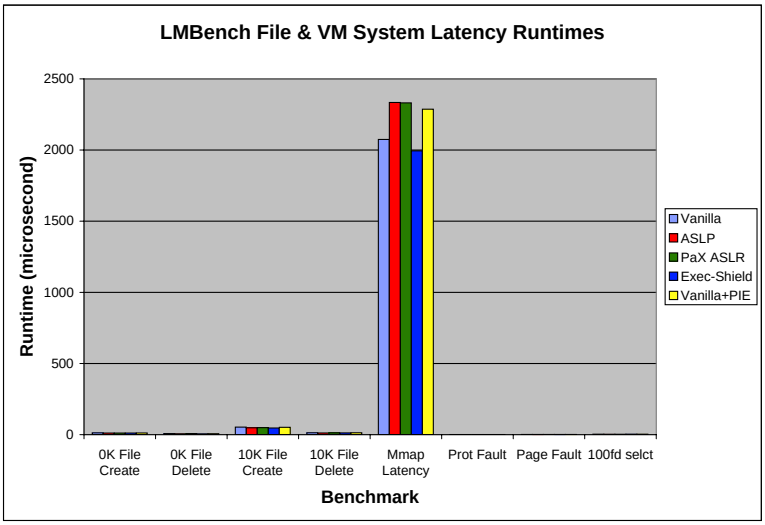

Figure C.3: LMBench File & VM Runtimes

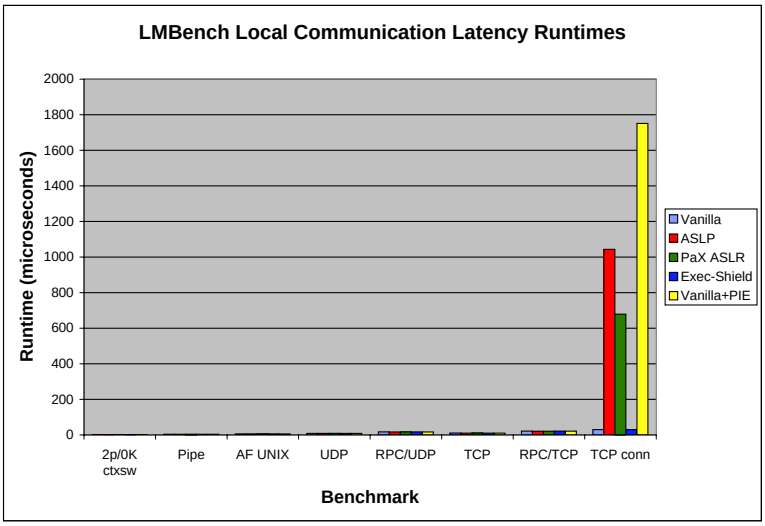

Figure C.4: LMBench Communication Latency Runtimes

LMBench Process Performance Overheads (%)				
Benchmark	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
null call	-1.63	0.27	-1.9	-0.27
null I/O	0	0	0	0
stat	3.72	2.59	-0.96	0
open clos	-1.28	0.61	1.89	-0.34
slct TCP	2.8	-2.35	-1.97	-1.03
sig inst	1.68	-0.34	0.67	-0.17
sig hndl	3.75	0.15	0.93	0.05
fork proc	20.56	20.18	0.61	5.65
exec proc	32.18	21.96	12.53	0.19
sh proc	12.63	13.83	6.86	0.73

Table C.3: LMBench Process Benchmark Overheads

Context Switching Benchmark Descriptions	
Name	Description
2p/0K	Measures context switching time for 2 processes of 0K in size.
2p/16K	Measures context switching time for 2 processes of 16K in size.
2p/64K	Measures context switching time for 2 processes of 64K in size.
8p/16K	Measures context switching time for 8 processes of 16K in size.
8p/64K	Measures context switching time for 8 processes of 64K in size.
16p/16K	Measures context switching time for 16 processes of 16K in size.
16p/64K	Measures context switching time for 16 processes of 64K in size.

Table C.4: Descriptions of LMBench Context Switching Benchmarks

LMBench Context Switching Benchmark Run Times (microseconds)					
Benchmark	Vanilla	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
2p/0K	0.78	0.92	0.94	0.85	0.79
2p/16K	1.02	1.02	1.07	0.96	0.93
2p/64K	1.11	1.32	1.46	1.13	1.08
8p/16K	1.33	1.4	1.41	1.38	1.4
8p/64K	22.96	23.88	24.01	23.62	23.3
6p/16K	3.74	4.08	4.67	4.14	4.16
16p/64K	33.89	34.26	35.18	34.57	34.08

Table C.5: LMBench Context Switch Benchmark Run times

LMBench Context Switching Performance Overheads (%)				
Benchmark	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
2p/0K	18.74	21.17	9.44	1.86
2p/16K	0.22	4.24	-5.98	-8.91
2p/64K	18.96	31.8	2.31	-2.21
8p/16K	4.92	6.17	3.59	5.25
8p/64K	4.02	4.6	2.9	1.5
16p/16K	9.16	25.13	10.74	11.48
16p/64K	1.08	3.8	2	0.56

Table C.6: LMBench Context Switch Benchmark Overheads

File and Virtual Memory Benchmark Descriptions	
Name	Description
0K File Create	Measures file system create performance.
0K File Delete	Measures file system delete performance.
10K File Create	Measures file system create performance.
10K File Delete	Measures file system delete performance.
Mmap Latency	Measures time to mmap and unmmap files of various size.
Prot Fault	Measures time to catch a protection fault.
Page Fault	Measures time to page fault from a file.
100fd Selct	Measures time to do a select on 100 file descriptors.

Table C.7: Descriptions of LMBench Virtual Memory and File System Benchmarks

LMBench File and VM System Benchmark Run Times (microseconds)					
Benchmark	Vanilla	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
0K File Create	12.66	11.28	11.18	11.2	11.6
0K File Delete	7.35	5.92	7.41	6.15	6.54
10K File Create	52.92	46.96	49.02	48.1	51.72
10K File Delete	13.13	12.47	12.92	11.74	13.23
Mmap Latency	2074.56	1994.22	2331.67	2334.11	2287.11
Prot Fault	1	0.91	0.97	0.99	0.97
Page Fault	1.54	1.54	1.56	1.56	1.55
100fd selct	3.13	3.22	2.99	3.01	3.1

Table C.8: LMBench File and VM System Latency Benchmark Run times

LMBench File and VM System Performance Overheads (%)				
Benchmark	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
0K Create	-10.89	-11.68	-11.5	-8.34
0K File Delete	-19.5	0.77	-16.28	-10.96
10K File Create	-11.27	-7.37	-9.11	-2.27
10K File Delete	-5.08	-1.61	-10.58	0.76
Mmap Latency	-3.87	12.39	12.51	10.25
Prot Fault	-8.53	-3.04	-1.25	-2.59
Page Fault	0.32	1.35	1.52	0.98
100fd selct	2.81	-4.27	-3.66	-0.76

Table C.9: LMBench File and VM System Latency Benchmark Overheads

Communication Latency Benchmark Descriptions	
Name	Description
2p/0K ctxsw	Measures context switching latency for 2 processes of 0K in size.
Pipe	Measures interprocess communication latency through pipes.
AF UNIX	Measures interprocess communication latency via UNIX socket streams.
UDP	Measures interprocess communication latency via UDP/IP.
RCP/UDP	Measures interprocess communication latency via Sun RPC over UDP/IP.
TCP	Measures interprocess communication latency via TCP/IP.
RPC/TCP	Measures interprocess communication latency via Sun RPC over TCP/IP.
TCP conn	Measures interprocess connection latency via TCP/IP.

Table C.10: Descriptions of LMBench Communication Latency Benchmarks

LMBench Communication Latency Benchmark Run Times (microseconds)					
Benchmark	Vanilla	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
2p/0K ctxsw	0.78	0.92	0.94	0.82	0.79
Pipe	3.85	3.91	4.09	3.88	3.86
AF UNIX	5.77	5.91	6.32	5.88	5.85
UDP	8.64	8.66	8.92	8.65	8.59
RPC/UDP	16.9	17.01	16.77	16.77	16.83
TCP	10.2	10.29	10.57	10.28	10.27
RPC/TCP	21.77	21.54	20.93	20.98	21.24
TCP conn	30	30	678.89	1043.22	1751

Table C.11: LMBench Local Communication Latency Benchmark Run times

LMBench Communication Latency Performance Overheads (%)				
Benchmark	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
2p/0K ctxsw	18.74	21.17	5.01	1.86
Pipe	1.64	6.34	0.74	0.28
AF UNIX	2.39	9.63	1.89	1.44
UDP	0.17	3.27	0.08	-0.56
RPC/UDP	0.66	-0.79	-0.79	-0.39
TCP	0.87	3.59	0.76	0.65
RPC/TCP	-1.02	-3.83	-3.62	-2.4
TCP conn	0	2162.96	3377.41	5736.67

Table C.12: LMBench Local Communication Latency Benchmark Overheads

Communication Bandwidth Benchmark Descriptions	
Name	Description
Pipe	Times data movement through pipes.
AF UNIX	Times data movements through UNIX socket streams.
TCP	Times data movement through TCP/IP sockets.
File Reread	Times the reading and summing of a file.
Mmap Reread	Times the reading and summing of a file.
Bcopy (libc)	Times memory copy speeds.
Bcopy (hand)	Times memory copy speeds.
Mem Read	Times memory read rate (with overhead).
Mem Write	Time memory write rate (with overhead).

Table C.13: Descriptions of LMBench Communication Bandwidth Benchmarks

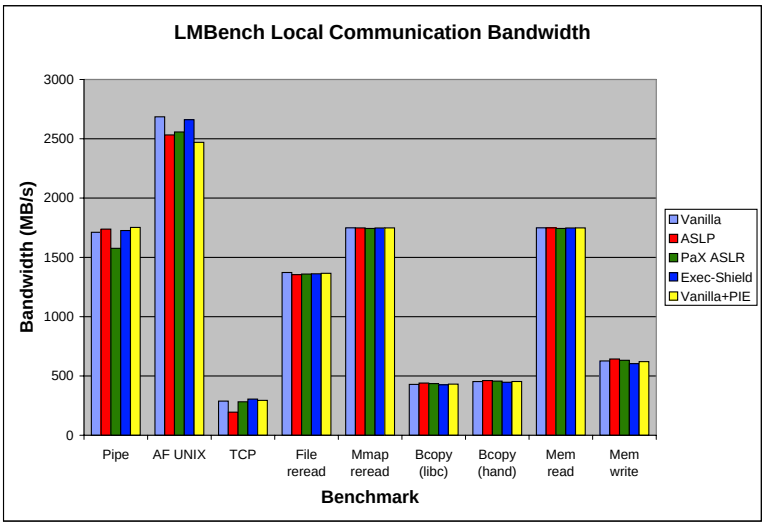

Figure C.5: LMBench Communication Bandwidth Runtimes

LMBench Communication Bandwidth Benchmark Run Times (microseconds)					
Benchmark	Vanilla	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
Pipe	1711.33	1726.33	1576.33	1738.56	1752.44
AF UNIX	2685.22	2660.78	2557.22	2532.67	2469.78
TCP	288.22	304.78	282.33	194.57	294.13
File reread	1372.63	1361.63	1359.48	1354.18	1365.49
Mmap reread	1748.86	1748.11	1743.72	1748.53	1748.42
Bcopy (libc)	428.83	425.63	435.54	440.21	431.23
Bcopy (hand)	452.72	447.01	456.91	461.37	453.26
Mem read	1749	1748.11	1743.89	1749	1748.78
Mem write	626.87	603.9	632.1	643.29	620.86

Table C.14: LMBench Local Communication Bandwidth Benchmark Run times

LMBench Communication Bandwidth Performance Overheads (%)				
Benchmark	Exec-Shield	PaX ASLR	ASLP	Vanilla+PIE
Pipe	0.88	-7.89	1.59	2.4
AF UNIX	-0.91	-4.77	-5.68	-8.02
TCP	5.74	-2.04	-32.49	2.05
File reread	-0.8	-0.96	-1.34	-0.52
Mmap reread	-0.04	-0.29	-0.02	-0.02
Bcopy (libc)	-0.75	1.56	2.65	0.56
Bcopy (hand)	-1.26	0.93	1.91	0.12
Mem read	-0.05	-0.29	0	-0.01
Mem write	-3.66	0.83	2.62	-0.96

Table C.15: LMBench Local Communication Bandwidth Benchmark Overheads

Appendix D

Apache Benchmark Data

Results from the Apache Benchmark. This appendix contains the raw Apache Benchmark data for the following configurations: vanilla kernel, PaX ASLR kernel, Red Hat Exec-Shield kernel, ASLP kernel, and PIE compilation with vanilla kernel.

The Apache Benchmark configuration using in this setting makes 1 million requests, in simultaneous batches of 100, for a static HTML page of 1881 bytes, which includes 425 bytes of images. The Apache HTTP server configuration was modified from the default to preemptively spawn 100 worker processes to handle incoming requests.

Figure D.1 contains the Apache Benchmark runtime results in chart form.

Table D.1 contains the Apache benchmark runtime and performance overhead results in numeric form.

Apache Benchmark Run Times & Overheads		
Kernel	Runtime (seconds)	Overhead (%)
Vanilla	736.92	–
Exec-Shield	744.46	1.02
PaX ASLR	736.26	-0.09
ASLP	742.12	0.71
PIE	860.73	16.8

Table D.1: Apache Benchmark Run times (seconds) & Overheads (%)

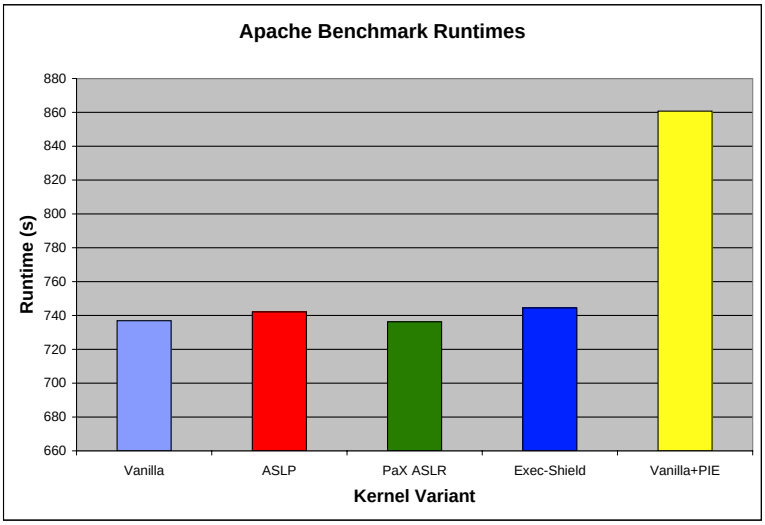

Figure D.1: Apache Benchmark Runtimes

Appendix E

PaXtest Data

Results from the PaXtest Experiment. This appendix contains the raw results from PaXtest, an evaluation tool to determine the protection provided by address randomization techniques. The tool was run against the following kernels: vanilla, Red Hat Exec-Shield, PaX ASLR, and ASLP.

Vanilla Kernel PaXtest Results

```
Executable anonymous mapping             : Vulnerable
Executable bss                           : Vulnerable
Executable data                          : Vulnerable
Executable heap                          : Vulnerable
Executable stack                         : Vulnerable
Executable anonymous mapping (mprotect)  : Vulnerable
Executable bss (mprotect)                : Vulnerable
Executable data (mprotect)               : Vulnerable
Executable heap (mprotect)               : Vulnerable
Executable shared library bss (mprotect) : Vulnerable
Executable shared library data (mprotect): Vulnerable
Executable stack (mprotect)              : Vulnerable
Anonymous mapping randomisation test     : No randomisation
Heap randomisation test (ET_EXEC)        : No randomisation
Heap randomisation test (ET_DYN)         : No randomisation
Main executable randomisation (ET_EXEC)  : No randomisation
Main executable randomisation (ET_DYN)   : No randomisation
```

```
Shared library randomisation test        : No randomisation
Stack randomisation test (SEGMEXEC)      : No randomisation
Stack randomisation test (PAGEEXEC)      : No randomisation
Return to function (strcpy)              : Vulnerable
Return to function (strcpy, RANDEXEC)    : Vulnerable
Return to function (memcpy)              : Vulnerable
Return to function (memcpy, RANDEXEC)    : Vulnerable
Executable shared library bss            : Vulnerable
Executable shared library data           : Vulnerable
Writable text segments                   : Vulnerable
```

Red Hat Exec-Shield Kernel PaXtest Results

```
Executable anonymous mapping             : Killed
Executable bss                           : Killed
Executable data                          : Killed
Executable heap                          : Killed
Executable stack                         : Killed
Executable anonymous mapping (mprotect)  : Vulnerable
Executable bss (mprotect)                : Vulnerable
Executable data (mprotect)               : Vulnerable
Executable heap (mprotect)               : Vulnerable
Executable shared library bss (mprotect) : Vulnerable
Executable shared library data (mprotect): Vulnerable
Executable stack (mprotect)              : Vulnerable
Anonymous mapping randomisation test     : 8 bits (guessed)
Heap randomisation test (ET_EXEC)        : 13 bits (guessed)
Heap randomisation test (ET_DYN)         : 13 bits (guessed)
Main executable randomisation (ET_EXEC)  : No randomisation
Main executable randomisation (ET_DYN)   : 12 bits (guessed)
Shared library randomisation test        : 12 bits (guessed)
Stack randomisation test (SEGMEXEC)      : 17 bits (guessed)
Stack randomisation test (PAGEEXEC)      : 17 bits (guessed)
Return to function (strcpy)              : Vulnerable
Return to function (strcpy, RANDEXEC)    : Vulnerable
Return to function (memcpy)              : Vulnerable
Return to function (memcpy, RANDEXEC)    : Vulnerable
Executable shared library bss            : Vulnerable
Executable shared library data           : Vulnerable
Writable text segments                   : Vulnerable
```

PaX Kernel PaXtest Results

```
Executable anonymous mapping             : Killed
```

```
Executable bss                            : Killed
Executable data                           : Killed
Executable heap                           : Killed
Executable stack                          : Killed
Executable anonymous mapping (mprotect)   : Killed
Executable bss (mprotect)                 : Killed
Executable data (mprotect)                : Killed
Executable heap (mprotect)                : Killed
Executable shared library bss (mprotect) : Killed
Executable shared library data (mprotect): Killed
Executable stack (mprotect)               : Killed
Anonymous mapping randomisation test      : 16 bits (guessed)
Heap randomisation test (ET_EXEC)         : 13 bits (guessed)
Heap randomisation test (ET_DYN)          : 25 bits (guessed)
Main executable randomisation (ET_EXEC)   : No randomisation
Main executable randomisation (ET_DYN)    : 17 bits (guessed)
Shared library randomisation test         : 16 bits (guessed)
Stack randomisation test (SEGMEXEC)       : 23 bits (guessed)
Stack randomisation test (PAGEEXEC)       : 24 bits (guessed)
Return to function (strcpy)               : Vulnerable
Return to function (strcpy, RANDEXEC)     : Vulnerable
Return to function (memcpy)               : Vulnerable
Return to function (memcpy, RANDEXEC)     : Vulnerable
Executable shared library bss             : Killed
Executable shared library data            : Killed
Writable text segments                    : Killed
```

ASLP Kernel PaXtest Results

```
Executable anonymous mapping              : Vulnerable
Executable bss                            : Vulnerable
Executable data                           : Vulnerable
Executable heap                           : Vulnerable
Executable stack                          : Vulnerable
Executable anonymous mapping (mprotect)   : Vulnerable
Executable bss (mprotect)                 : Vulnerable
Executable data (mprotect)                : Vulnerable
Executable heap (mprotect)                : Vulnerable
Executable shared library bss (mprotect) : Vulnerable
Executable shared library data (mprotect): Vulnerable
Executable stack (mprotect)               : Vulnerable
Anonymous mapping randomisation test      : 20 bits (guessed)
Heap randomisation test (ET_EXEC)         : 29 bits (guessed)
Heap randomisation test (ET_DYN)          : 29 bits (guessed)
Main executable randomisation (ET_EXEC)   : No randomisation
```

```
Main executable randomisation (ET_DYN)   : 20 bits (guessed)
Shared library randomisation test        : 20 bits (guessed)
Stack randomisation test (SEGMEXEC)      : 28 bits (guessed)
Stack randomisation test (PAGEEXEC)      : 28 bits (guessed)
Return to function (strcpy)              : Vulnerable
Return to function (strcpy, RANDEXEC)    : Vulnerable
Return to function (memcpy)              : Vulnerable
Return to function (memcpy, RANDEXEC)    : Vulnerable
Executable shared library bss            : Vulnerable
Executable shared library data           : Vulnerable
Writable text segments                   : Vulnerable
```