# Address Space Layout Permutation

Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, and Jun Xu
Department of Computer Science
North Carolina State University
{ckil, jjun2, cbookholt, junxu}@ncsu.edu

## 1. Introduction

Address space randomization is an emerging and promising method for stopping a broad range of memory corruption attacks. By randomly shifting critical memory regions at process initialization time, a memory layout randomization scheme converts an otherwise successful malicious attack into a benign process crash. Recent work[7] has shown that current randomization schemes can be brute-forced attacked. Such attacks are possible because of the lack of sufficient randomness in current schemes. We propose Address Space Layout Permutation (ASLP) that introduces higher entropy with minimal performance overhead. Our evaluation result shows that the randomized applications by ASLP incur less than 1% performance overhead with orders of magnitude improvement in randomness up to 29 bits of randomness on a 32-bit architecture.

## 2. Address Space Layout Permutation

ASLP permutes all sections (including code and static data) in the program address space without source code modification. ASLP provides address permutation in two ways: user level and kernel level.

### 2.1. User Level Address Permutation

The goal of user level address permutation is to randomly relocate static code and data segments and re-arrange functions and data objects within the code and data segment. Therefore, the program will have different memory layout each time it is loaded for execution by the runtime system loader. Although this goal has been achieved by using the linker script with a binary editing tool[1], source code transformation[2], or kernel patch[8], it has limitations such as requirements of kernel modifications, source code recompilation, or additional overhead. Our work fills a gap in current state-of-art techniques by achieving the same goal with different implementation. We have developed a novel binary rewriting tool that permutes static code and data segments and re-orders its elements. Since our tool operates directly on compiled program executables, it does not require source code modifications or kernel patch. We only require a linker option to access the relocation information in the program. Currently our tool works on Executable and Linking Format (ELF). Rewriting ELF executable file for randomization, and to make it run exactly as before is non-trivial. It requires understanding on how ELF file is generated and how the loader creates the program memory layout from the ELF file. Moreover, since we change the virtual addresses of functions and data objects within the code and data segments, we also need to modify all cross-references among them to make the program runs without crash. We found that total of 12 sections in the ELF file need to be changed for the randomization. Those sections include ELF header, symbol table, and other sections related to program execution. Rewriting process comprises two major phases: coarse-grained and fine-grained permutation. In the coarse-grained permutation, the starting addresses of static code and data segments are changed. Fine-grained permutation randomly re-arranges the functions and data objects within the code and data segments. ASLP also allows users to change the relative orders of sections in the program. For example, data segment comes first before the code segment. This gives benefit that breaks an attacker's assumption of general section order such that code segment comes first, data segment and bss segment follow. We believe that this makes attackers more difficult to craft successful attacks.

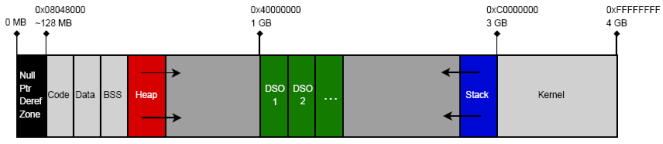
### 2.2. Kernel Level Address Permutation

The goal of kernel level address permutation is to randomly relocate three regions: the user stack, brk()-ed heap, and mmap()-ed area. These regions are frequent target of attacks and have been exploited in many real attacks. We have built the ASLP kernel for 32-bit x86 CPU with Linux 2.4.31 kernel to randomize these regions. The stack location is randomized during the process creation when the kernel builds a data structure that holds process arguments and en-

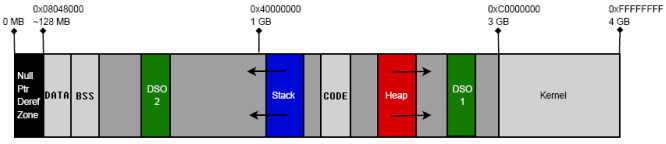
vironment variables. We subtract a random amount between 0 and 4 KB from the stack pointer, thereby introducing randomization in low-order bits 2. Another random amount is subtracted from the standard 3 GB stack base location in later stage of process creation so that ASLP can provide 29 bits randomness in user stack area. The brk()-ed heap and mmap()-ed location is determined as similar to stack. We modify the brk() and mmap() allocation code to generate random, page-aligned virtual address between 0 and 3 GB for heap and mmap() regions.

### 2.3. Demonstration of Permutation

After both the rewriting and kernel level randomization, all critical memory regions including static code and data segments can be placed in different locations throughout the user memory space. Figure 1 shows a possible permutation of the normal process memory layout. Note (1) the random relative order of memory regions, (2) lack of large pad allocations for randomization that is required in closely related works.




(a) Normal Process Layout




(b) Randomized Process Layout

**Figure 1. Example Memory Layout**

## 3. Evaluation

To validate the practicality and effectiveness of ASLP, we have evaluated ASLP using security and performance benchmarks. For security benchmark, we use PaXtest[3] to evaluate the randomness in each randomized region. For performance benchmark, we use SPEC CPU2000 [4], LMBench micro-benchmark[5], and Apache Benchmark. We compare benchmark results of ASLP with two other popular techniques: Exec-Shield[6] and PaX ASLR[8].

Table 1 shows the PaXtest result. The vanilla kernel has no randomization, so each region has zero bits of randomness. Exec-shield shows up to 17 bits randomness and PaX ASLR provides up to 24 bits randomness. ASLP comes out

| 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 1. PaXtest Results Summary**

ahead by at least 4 bits in all regions and provide maximum 29 bits in heap region. Performance benchmark is done with different techniques on the test computer that 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. All three techniques including ASLP shows less than 1% average overhead from SPEC CPU2000 benchmark. The LMBench benchmark shows that ASLP provide better performance on fork() and exec() operations. ASLP slowed fork() and exec() operations by 6.86% and 12.53% respectively. Both PaX ASLR and Exec-Shield have consistently higher overheads for the same tasks: 13.83-21.96% and 12.63-32.18% respectively. The context switching overhead results give a similar story to process operation results. The Apache Benchmark result shows that all techniques including ASLP incur low overhead less than 1%.

## 4. Limitations and Future Work

Although ASLP can mitigate many types of memory corruption attacks, current implementation of ASLP does not support stack frame randomization. This can be achieved by adding pads among elements [1] in the stack frame, but their work does not completely randomize the stack frame. We leave this problem as future work.

## References


[1] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an effcient approach to combat a broad range of memory error exploits. In *Proc. 12th USENIX Security Symposium*, 2003.

[2] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. 14th USENIX Security Symposium*, 2005.

[3] P. Busser. Paxtest.

[4] S. P. E. Corporation. Spec cpu2000 v1.2.

[5] L. McVoy and C. Staelin. Lmbench: Tools for performance analysis.

[6] I. Molnar. Exec-shield.

[7] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. 11th ACM Conference on Computer and Communications Security*, 2004.

[8] T. P. Team. Address space layout randomization, 2003.