Address-Space Randomization

Andy Podgurski
EECS Dept.
Case Western Reserve University

Sources

- *On the Effectiveness of Address-Space Randomization*, H. Shacham et al, 2004.
- *Address-Space Randomization for Windows Systems*, L. Li et al, 2006.
- *An Analysis of Address Space Layout Randomization on Windows Vista*, O. Whitehouse, Symantec Corp, 2007.

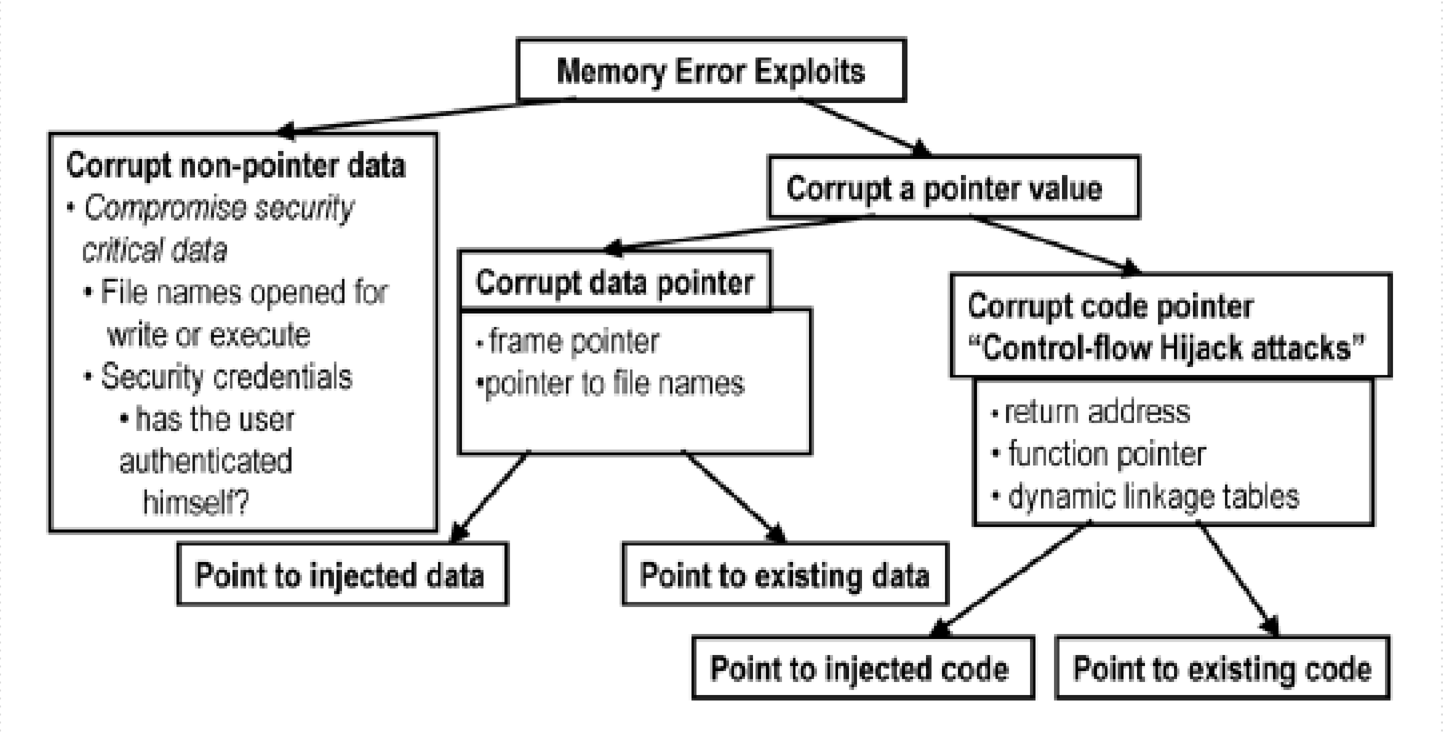
Sources continued

- *Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization* by Giuffrida et al., USENIX Security Symposium. 2012.

Overview

- Address space (layout) randomization (ASR, ASLR) is a defense against memory corruption attacks.
- It makes exploitable memory addresses harder for an attacker to locate.
- The address space layout is randomized each time a program is restarted.
- ASR has been integrated into Windows, Linux, and OpenBSD.

Memory Error Exploits



[Li et al]

Effectiveness of ASR

- This depends on several factors:
  - how predictable the random memory layout is
  - how tolerant an attack is to variations in memory layout
  - how many exploitation attacks an attacker can make

Absolute Address Randomization (AAR)

- Randomizes the absolute memory addresses of code and data objects.
- Relative distances aren't randomized.
- AAR blocks pointer corruption attacks (e.g. stack smashing).
  - Attacker can't predict the objects that will be referenced by a corrupted pointer.

Limitations of AAR

- It's hard to protect the randomization key from local users.
- Doesn't protect against some memory attacks, e.g.:
  - Relative-address attacks don't rely on absolute locations of data.
    - Ex.: data corruption attacks
  - Information leakage attack reads pointer, uses it to compute location of other objects.
  - Brute-force attacks repeatedly guess the value to be used for corrupting a pointer.

Need to Relocate All Memory Regions

- Locations of some memory objects aren’t randomized in some AAR implementations.
- If a code region *S* is not randomized, an attacker can execute a return-to-existing-code attack into *S*.
  - In Windows **`jmp esp`** is common.
  - In many attacks the top of the stack contains attacker-provided data.
- Any unrandomized writable section *W* is vulnerable to a 2-step attack:
  1. Attacker injects short opcode sequence into *W*.
  2. Control is transferred to this code.

Relative Address Randomization (RAR)

- Randomizes inter-object distances as well as absolute addresses.
- Can defeat non-pointer attacks.

ASR in Windows Vista

- Any executable image (.exe, .dll) can participate in ASR by setting a bit in the PE header.
- When loading the image, the OS uses a random global image offset.
  - This is selected pseudorandomly once per reboot from 256 values.
  - Thus, the locations of the code, data, and libraries change only between reboots.
- Each execution, the process memory layout is further randomized by placing the thread stack and process heaps randomly.
  - The stack region is selected from 32 locations.
  - The initial stack pointer is further randomized by a random decrement (16,384 choices on IA23 system).
  - Each heap is allocated from 32 locations.
  - The address of the Process Environment Block (PEB) is also selected randomly.

DAWSON Approach to AAR in Windows [Li et al]

- A randomization DLL is injected into target process.
  - It is loaded early in process creation.
  - It “hooks” standard API functions for memory allocation and randomizes base addresses of all memory regions.
- A customized loader is used to randomize memory allocated before the randomization DLL is loaded.
- A kernel driver is used to randomize base addresses of DLLs loaded very early in the boot process.

Types of Virtual Memory Regions in Windows Process

Type	Description	Protection	Granularity of Rebasing
Free	Free space	Inaccessible	Not rebased
Code	Executable or DLL code	Read-only	15 bits
Static data	Within executable or DLL	Read-Write	15 bits
Stack	Process and thread stacks	Read-Write	29 bits
Heap	Main and other heaps	Read-Write	20 bits
TEB	Thread Environment Block	Read-Write	19 bits
PEB	Process Environment Block	Read-Write	19 bits
Parameters	Command-line and Environment variables	Read-Write	19 bits
VAD	Returned by virtual memory allocation routines	Read-Write	15 bits
VAD	Shared Info for kernel and user mode	Unwritable	Not rebased

[Li, et al]

More on DLLs in Windows

- DLLs contain absolute references to addresses in themselves.
- Hence, they aren’t position independent.
- IF a DLL is not loaded at its default location, it must be rebased.
- This precludes loading each library at a random address.
  - It would require a copy of each library for every process.
- DAWSON rebases a library the first time it is loaded after a reboot.
- The randomization is somewhat coarse because DLLs must be aligned on 64K boundaries.

Whitehouse’s Study of Windows Vista

Table 1. Comparison between the number of unique values expected and observed in each data set.

Item	Expected	Observed	Difference
Stack	16,384 (2^{14})	8,568	−48%
Malloc[2]	>= 32 (>= 2^5)	192	+500%
HeapAlloc[3]	>= 32 (>= 2^5)	95	+200%
CreateHeap[4]	>= 32 (>= 2^5)	209	+550%
Image	256 (2^8)	255	−0.4%
PEB	16 (2^4)	13	−19%

Whitehouse's Study cont. (2)

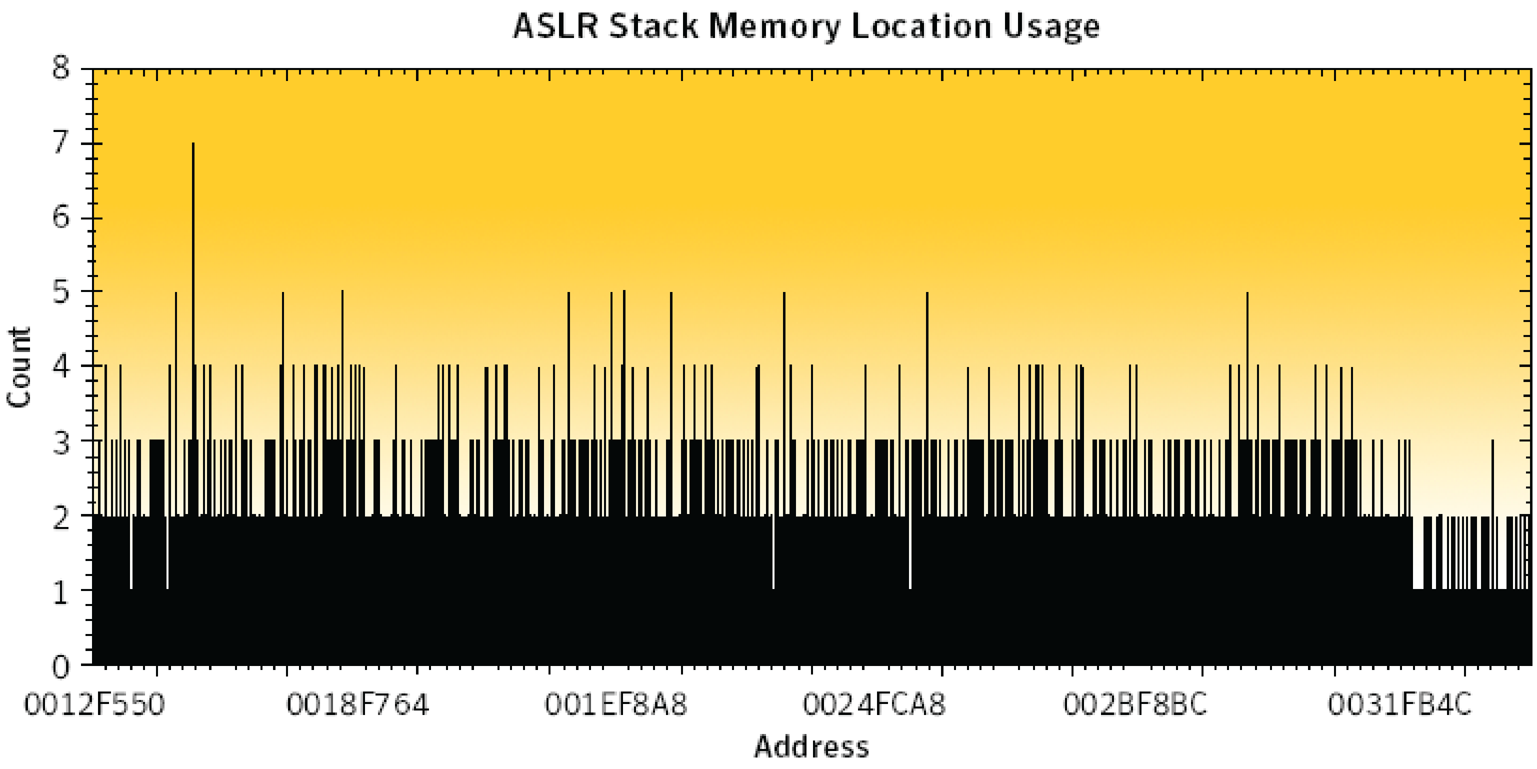


Figure 2. Distribution of stack addresses.

Whitehouse’s Study cont. (3)

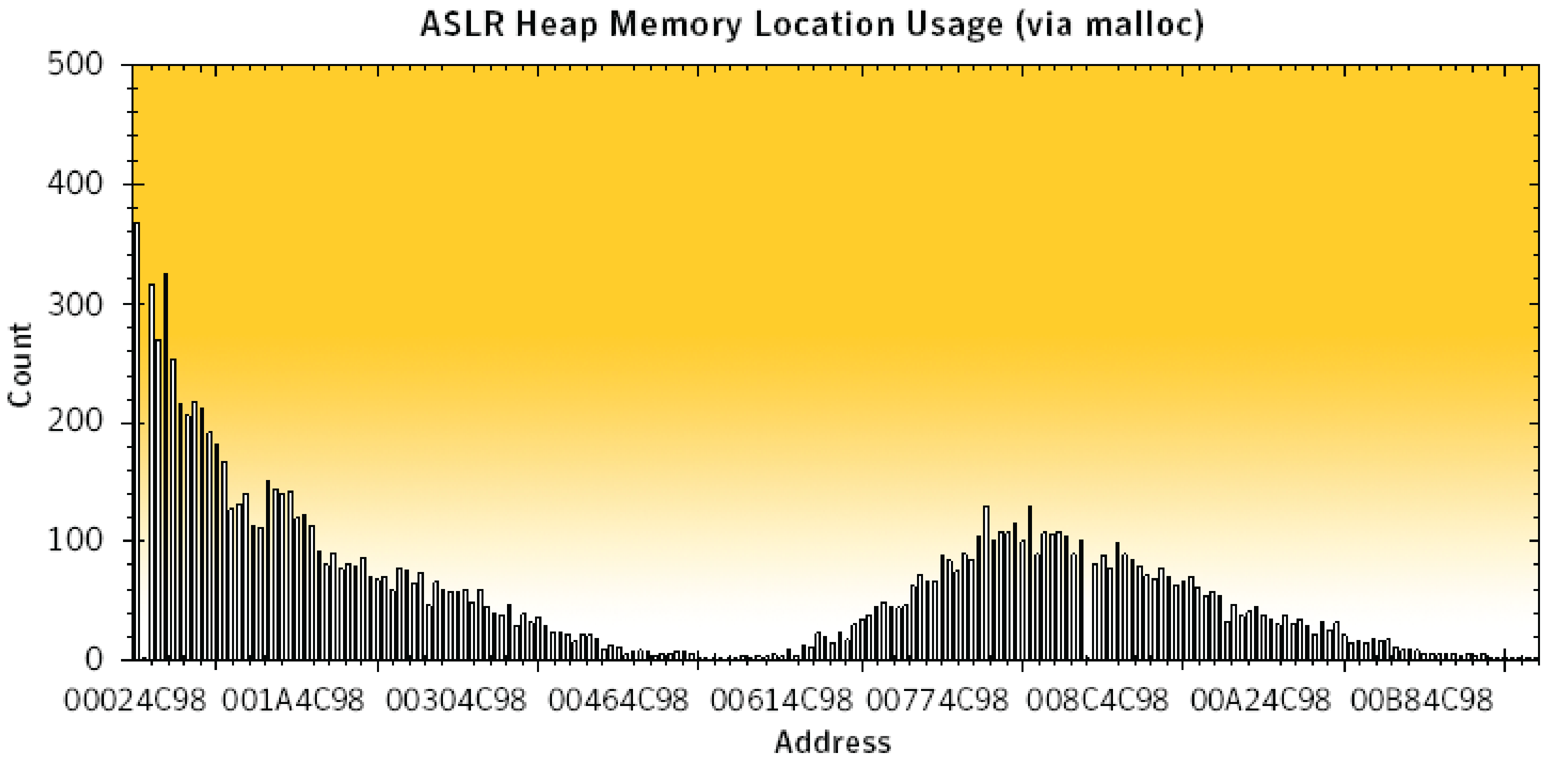


Figure 3a. Distribution of heap addresses using malloc, HeapAlloc, and HeapAlloc with CreateHeap.

Whitehouse's Study cont. (4)

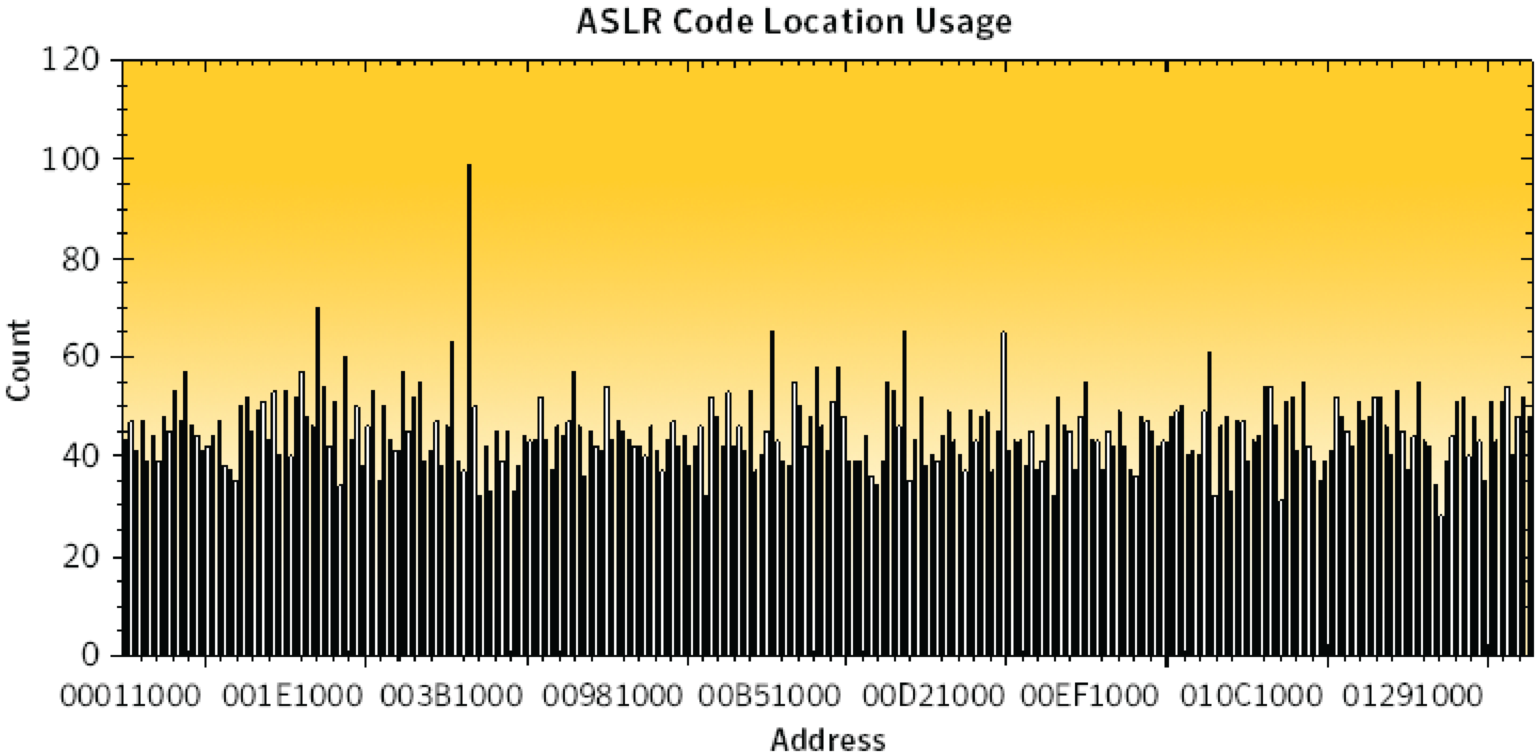


Figure 4. Distribution of code addresses.

Whitehouse's Study cont. (5)

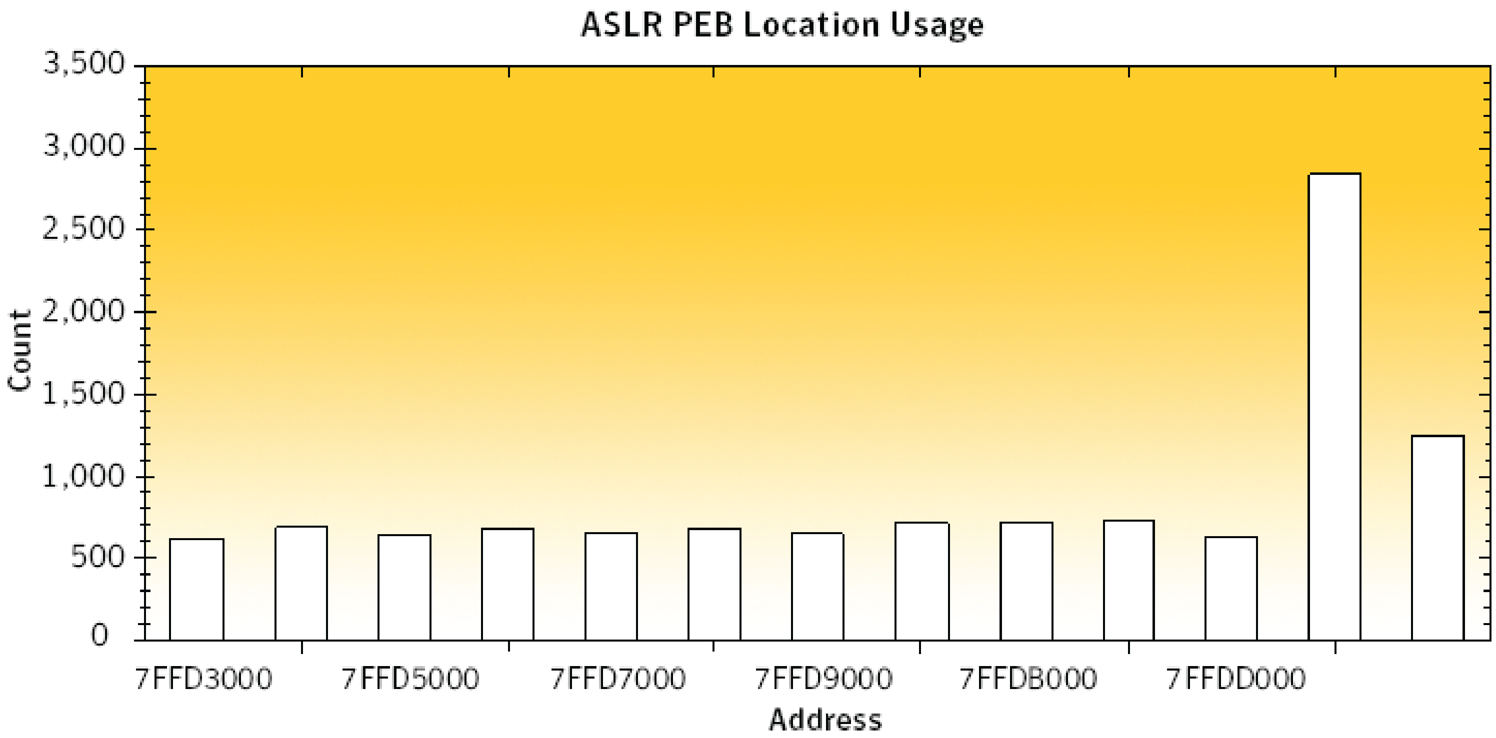


Figure 5. Distribution of PEB addresses.

ASR Inside the OS

- *Kernel-level exploitation* is increasingly popular among attackers
- Existing OS-level countermeasures are insufficient against generic memory error exploits
  - e.g., tampering with *non-pointer data* to elevate privilege
- Giufridda et al. proposed a design for *fine-grained ASR inside the OS*
- It is based on *runtime state migration* and can *rerandomize* code and data

Challenges in OS-Level ASR

- Enforcing $W \oplus X$ protection for kernel pages causes *unacceptable overhead*
- Same is true for instrumentation
- Some parts of OS are particularly *difficult to randomize*
- Many attack strategies become *more effective inside the OS*
  - e.g., non-control data attacks
- *Information leakage attacks* are prevalent

Giufridda et al.’s OS-Level ASR Design

- Confines OS components into *hardware-isolated event-driven processes*
  - Enables *selective* randomization and re-randomization
  - Simplifies synchronization and state management at rerandomization
  - Helps prevent direct intercomponent control tranfer
- Implemented by *microkernel*-based OS architecture
  - Microkernel provides only IPC and *low-level resource management*
  - Microkernel and OS processes are randomized at *link-time*
- Randomization manager *periodically re-randomizes* every OS process
- *Entire execution state* is *transferred* to new randomized process variant

Link-Time ASR Transformations

- Goal: randomize all code and data for every OS component
- Fine-grained randomization of
  - *Relative distance/alignment* between any two memory objects
  - *Internal layout* of memory objects and functions
- *Randomly permutes functions*
- Introduces *randomized padding* before and between objects and functions
- Randomizes static data, stack data, and heap data