

Address Space Randomization

Vitaly Shmatikov

Problem: Lack of Diversity

- ◆ Buffer overflow and **return-to-libc** exploits need to know the (virtual) address to which pass control
 - Address of attack code in the buffer
 - Address of a standard kernel library routine
- ◆ Same address is used on many machines
 - Slammer infected 75,000 MS-SQL servers using same code on every machine
- ◆ Idea: introduce **artificial diversity**
 - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

Address Space Randomization

- ◆ Randomly choose base address of stack, heap, code segment
- ◆ Randomly pad stack frames and malloc() calls
- ◆ Randomize location of Global Offset Table
- ◆ Randomization can be done at compile- or link-time, or by rewriting existing binaries
 - Threat: attack repeatedly probes randomized binary
- ◆ Several implementations available

PaX

- ◆ Linux kernel patch
- ◆ Goal: prevent execution of arbitrary code in an existing process's memory space
- ◆ Enable executable/non-executable memory pages
- ◆ Any section not marked as executable in ELF binary is non-executable by default
 - Stack, heap, anonymous memory regions
- ◆ Access control in `mmap()`, `mprotect()` prevents changes to protection state during execution
- ◆ Randomize address space

Non-Executable Pages in PaX

- ◆ In x86, pages cannot be directly marked as non-executable
- ◆ PaX marks each page as “non-present” or “supervisor level access”
 - This raises a page fault on every access
- ◆ Page fault handler determines if the page fault occurred on a data access or instruction fetch
 - Instruction fetch: log and terminate process
 - Data access: unprotect temporarily and continue

mprotect() in PaX

- ◆ mprotect() is a Linux kernel routine for specifying desired protections for memory pages
- ◆ PaX modifies mprotect() to prevent the following attacks:
 - Creation of executable anonymous memory mappings
 - Creation of executable and writable file mappings
 - Making executable, read-only file mapping writable
 - Except when relocating the binary
 - Conversion of non-executable mapping to executable

Access Control in PaX mprotect()

- ◆ In standard Linux kernel, each memory mapping is associated with permission bits
 - `VM_WRITE`, `VM_EXEC`, `VM_MAYWRITE`, `VM_MAYEXEC`
 - Stored in the `vm_flags` field of the `vma` kernel data structure
 - 16 possible write/execute states for each memory page
- ◆ PaX makes sure that the same page cannot be writable AND executable at the same time
 - Ensures that the page is in one of only 4 “good” states
 - `VM_MAYWRITE`, `VM_MAYEXEC`, `VM_WRITE | VM_MAYWRITE`, `VM_EXEC | VM_MAYEXEC`
 - Also need to ensure that attacker cannot make a region executable when mapping it using `mmap()`

PaX ASLR

- ◆ User address space consists of three areas
- ◆ Base of each area shifted by a random “delta”
 - Executable: 16-bit random shift (on x86)
 - Program code, uninitialized data, initialized data
 - Mapped: 16-bit random shift
 - Heap, dynamic libraries, thread stacks, shared memory
 - Stack: 24-bit random shift
 - Main user stack
- ◆ Only 16 bits of randomness are used to determine the random shift (why?)

PaX RANDUSTACK

- ◆ Responsible for randomizing userspace stack
- ◆ Userspace stack is created by the kernel upon each `execve()` system call
 - Allocates appropriate number of pages
 - Maps pages to process's virtual address space
 - Userspace stack is usually mapped at `0xbfffffff`, but PaX chooses a random base address
- ◆ PaX randomizes not only the address at which the stack is mapped, but also the range of allocated kernel memory

PaX RANDKSTACK

- ◆ Linux assigns two pages of kernel memory for each process to be used during the execution of system calls, interrupts, and exceptions
- ◆ PaX randomizes each process's kernel stack pointer before returning from kernel to userspace
 - 5 bits of randomness
- ◆ Each system call is randomized differently
 - By contrast, user stack is randomized once when the user process is invoked for the first time

PaX RANDMMAP

- ◆ When Linux allocates heap space, it starts at the base of the process's unmapped memory and finds the nearest chunk of unallocated space which is large enough
 - This is done in `do_mmap()` routine
- ◆ PaX modifies `do_mmap()` so that it adds a random `delta_mmap` to the base address before looking for new memory
 - 16 bits of randomness

PaX RANDEXEC

- ◆ Randomizes location of ELF binaries in memory
 - Problem if the binary was created by a linker which assumed that it will be loaded at a fixed address and omitted relocation information
- ◆ PaX maps the binary to its normal location, but makes it non-executable; creates an executable mirror copy at a random location
 - Access to the normal location will result in a page fault; page handler determines whether it is safe to redirect to the randomized mirror
 - Looks for “signatures” of return-to-libc attacks and may result in false positives

Base-Address Randomization

- ◆ Note that only **base address** is randomized
 - **Layouts** of stack and library table remain the same
 - Relative distances between memory objects are not changed by base address randomization
- ◆ To attack, it's enough to guess the base shift
- ◆ A 16-bit value can be guessed by brute force
 - Try 2^{15} (on average) different overflows with different values for the address of a known library function
 - In Shacham et al. paper, `usleep()` is used (**why?**)
 - How long does this take?
 - If address is wrong, target will simply crash

Ideas for Better Randomization (1)

◆ 64-bit addresses

- At least 40 bits available for randomization
 - Memory pages are usually between 4K and 4M in size
- Brute-force attack on 40 bits is not feasible

◆ Does more frequent randomization help?

- ASLR randomizes when a process is created
- Alternative: re-randomize address space while brute-force attack is still in progress
 - E.g., re-randomize non-forking process after each crash (recall that unsuccessful guesses result in target's crashing)
- This does not help much (why?)
 - See Shacham et al. paper for probability calculations

Ideas for Better Randomization (2)

- ◆ Randomly re-order entry points of library functions
 - Finding address of one function is no longer enough to compute addresses of other functions
 - What if attacker finds address of `system()`?
- ◆ ... at compile-time
 - No virtual mem constraints (can use more randomness)
 - What are the disadvantages??
- ◆ ... or at run-time
 - How are library functions shared among processes?
 - How does normal code find library functions?

Comprehensive Randomization [Bhatkar et al.]

- ◆ Idea: randomize relative distances as well as absolute locations
- ◆ Locations of stack-allocated objects randomized continuously during execution
 - Create a separate shadow stack for arrays
 - Each array surrounded by inaccessible memory regions
 - Allocation order is randomized for each function call
 - Random frame bases and random inter-frame gaps
- ◆ Indirect access to all static variables and functions
 - Accessed only via pointers stored in read-only memory
 - Addresses chosen randomly at execution start

Code Transformations

[Bhatkar et al.]

- ◆ Make all accesses to static variables indirect
 - Modify source code so that, before `main()` is entered, “shadow” memory is allocated for all static variables
 - Use `mmap()` to allocate memory in random location at runtime
 - Replace all variable accesses with dereferences of pointers pointing to the variables’ “shadow” locations
 - Use `mprotect()` to make pointers write-protected
- ◆ Make all function calls indirect
 - Modify code to store all function pointers in an array
 - Surrounded by special markers (easy to recognize in binary)
 - Reorder functions within the binary
 - Corresponding pointers are easy to recognize and modify

Run-Time Transformations

[Bhatkar et al.]

- ◆ Randomize base of stack at program start
- ◆ Insert random stack gap when a function is called
 - Can be done right before a function is called, or at the beginning of the called function (what's the difference?)
- ◆ setjmp/longjmp require special handling
 - Must keep track of context (e.g., shadow stack location)
- ◆ Read “Efficient Techniques for Comprehensive Protection from Memory Error Exploits” (USENIX Security 2005) for more details

Reading Assignment

- ◆ Shacham et al. “On the Effectiveness of Address-Space Randomization” (CCS 2004)
- ◆ Several other papers on the website