

Address Space Layout Randomization

Vitaly Shmatikov

was at Cornell U
2016

notes by JRM

Reading Assignment

- ◆ Shacham et al. "On the effectiveness of address-space randomization" (CCS 2004).
- ◆ Optional:
 - PaX documentation (<http://pax.grsecurity.net/docs/>)
 - Bhatkar, Sekar, DuVarney. "Efficient techniques for comprehensive protection from memory error exploits" (Usenix Security 2005).

Problem: Lack of Diversity [?] *Randomness*

- ◆ Buffer overflow and **return-to-libc** exploits need to know the (virtual) address to hijack 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

one jump to next

ASLR

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

PaX

pax.arssecurity.net

- ◆ 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 unsafe changes to protection state at runtime
- ◆ Randomize address space layout

Non-Executable Pages in PaX

- ◆ ~~In older 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 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:
 - 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

syscall

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 the 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
 - Executable, mapped, stack
- ◆ 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
 - Why are only 16 bits of randomness used?
 - Stack: 24-bit random shift
 - Main user stack

PaX's view

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
- ◆ In addition to base address, PaX randomizes the range of allocated 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

- ◆ Linux heap allocation: `do_mmap()` starts at the base of the process's unmapped memory and looks for the first unallocated chunk which is large enough
- ◆ PaX: add 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 produces a page fault
 - Page handler redirects to the mirror “if safe”
 - Looks for “signatures” of return-to-libc attacks and may result in false positives

Base-Address Randomization

- ◆ Only the 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) overflows with different values for addr of known library function – how long does it take?
 - Shacham et al. attacked Apache with return-to-libc
 - usleep() is used (why?)
 - If address is wrong, target will simply crash

ASLR in Windows

optional

- ◆ Vista and Server 2008
- ◆ Stack randomization
 - Find N^{th} hole of suitable size (N is a 5-bit random value), then random word-aligned offset (9 bits of randomness)
- ◆ Heap randomization: 5 bits
 - Linear search for base + random 64K-aligned offset
- ◆ EXE randomization: 8 bits
 - Preferred base + random 64K-aligned offset
- ◆ DLL randomization: 8 bits
 - Random offset in DLL area; random loading order

Bypassing Windows ASLR

optional

- ◆ Implementation uses randomness improperly, thus distribution of heap bases is biased
 - Ollie Whitehouse's paper (Black Hat 2007)
 - Makes guessing a valid heap address easier
- ◆ When attacking browsers, may be able to insert arbitrary objects into the victim's heap
 - Executable JavaScript code, plugins, Flash, Java applets, ActiveX and .NET controls...
- ◆ **Heap spraying**
 - Stuff heap with large objects and multiple copies of attack code (how does this work?)

Example: Java Heap Spraying

[See Sotirov & Dowd]

- ◆ JVM makes all of its allocated memory RWX: readable, writeable, executable (why?)
 - Yay! DEP now goes out the window...
- ◆ 100MB applet heap, randomized base in a predictable range
 - 0x20000000 through 0x25000000
- ◆ Use a Java applet to fill the heap with (almost) 100MB of NOP sleds + attack code
- ◆ Use your favorite memory exploit to transfer control to 0x25A00000 (why does this work?)

Information Leaks Break ASLR

 [See Sotirov & Dowd]

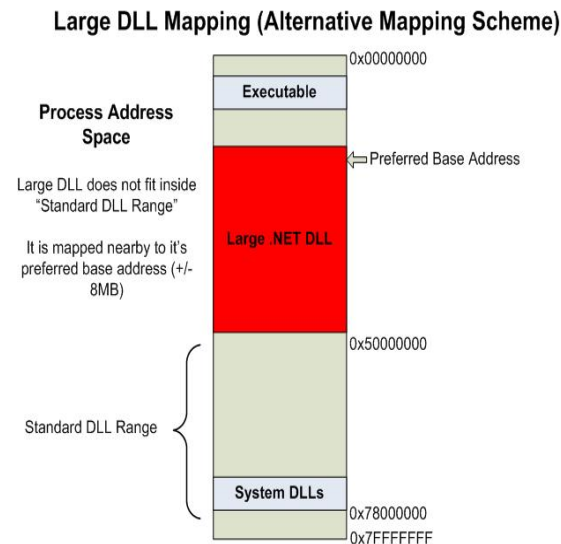
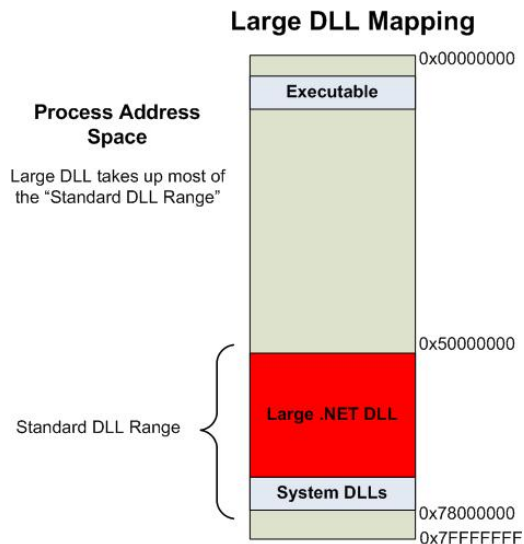
- ◆ User-controlled .NET objects are not RWX
- ◆ But JIT compiler generates code in RWX memory
 - Can overwrite this code or “return” to it out of context
 - But ASLR hides location of generated code stubs...
 - Call `MethodHandle.GetFunctionPointer()`NET itself will tell you where the generated code lives!
- ◆ ASLR is often defeated by information leaks
 - Pointer betrays an object’s location in memory
 - For example, a pointer to a static variable reveals DLL’s location... for all processes on the system! (why?)
 - Pointer to a frame object betrays the entire stack

.NET Address Space Spraying

skip

[See Sotirov & Dowd]

- ◆ Webpage may embed .NET DLLs
 - No native code, only IL bytecode
 - Run in sandbox, thus no user warning (unlike ActiveX)
 - Mandatory base randomization when loaded
- ◆ Attack webpage include a large (>100MB) DLL



Dealing with Large Attack DLLs

 [See Sotirov & Dowd]

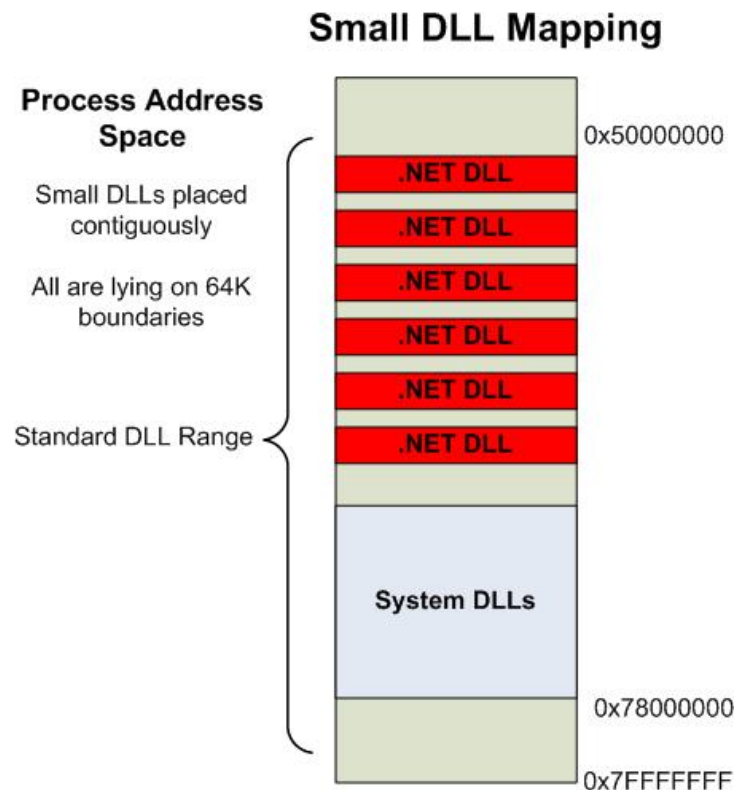
- ◆ 100MB is a lot for the victim to download!
- ◆ Solution 1: binary padding
 - Specify a section with a very large VirtualSize and very small SizeOfRawData – will be 0-padded when mapped
 - On x86, equivalent to `add byte ptr [eax], al` - NOP sled!
 - Only works if EAX points to a valid, writeable address
- ◆ Solution 2: compression
 - gzip content encoding
 - Great compression ratio, since content is mostly NOPs
 - Browser will unzip on the fly



Spraying with Small DLLs

[See Sotirov & Dowd]

- ◆ Attack webpage includes many small DLL binaries
- ◆ Large chunk of address space will be sprayed with attack code



Turning Off ASLR Entirely

skip

[See Sotirov & Dowd]

- ◆ Any DLL may “opt out” of ASLR
 - Choose your own ImageBase, unset IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE flag
- ◆ Unfortunately, ASLR is enforced on IL-only DLL
- ◆ How does the loader know a binary is IL-only?

```
if( ( (pCORHeader->MajorRuntimeVersion > 2) ||  
      (pCORHeader->MajorRuntimeVersion == 2 && pCORHeader->MinorRuntimeVersion >= 5) ) &&  
      (pCORHeader->Flags & COMIMAGE_FLAGS_ILONLY) )  
{  
    pImageControlArea->pBinaryInfo->pHeaderInfo->bFlags |= PINFO_IL_ONLY_IMAGE;  
    ...  
}
```

Set version in the header to anything below 2.5
ASLR will be disabled for this binary!

Bypassing IL Protections

Intermediate
language

[Dowd & Sotirov, PacSec 2008]

- ◆ Embedded .NET DLLs are expected to contain IL bytecode only - many protection features
 - Verified prior to JIT compilation and at runtime, DEP
 - Makes it difficult to write effective shellcode
- ◆ ... enabled by a single global variable
 - `mcorwks!s_eSecurityState` must be set to 0 or 2
 - Does `mcorwks` participate in ASLR? **No!**
- ◆ Similar: disable Java bytecode verification
 - JVM does not participate in ASLR, either
 - To disable runtime verification, traverse the stack and set NULL protection domain for current method

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?)

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
 - Access to source, thus no virtual memory constraints; can use more randomness (any disadvantages?)
- ◆ ... or at run-time
 - How are library functions shared among processes?
 - How does normal code find library functions?

Comprehensive Randomization (1)

[Bhatkar et al.]

◆ Function calls

- Convert all functions to function pointers and store them in an array
- Reorder functions within the binary
- Allocation order of arguments is randomized for each function call

◆ Indirect access to all static variables

- Accessed only via pointers stored in read-only memory
- Addresses chosen randomly at execution start

Comprehensive Randomization (2)

[Bhatkar et al.]

- ◆ Locations of stack-allocated objects randomized continuously during execution
 - Separate shadow stack for arrays
 - Each array surrounded by inaccessible memory regions
- ◆ 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?)
- ◆ Randomize heap-allocated objects
 - Intercepts malloc() calls and requests random amount of additional space

Comprehensive Randomization (3)

~~CONFIDENTIAL~~ [Bhatkar et al.]

- ◆ Randomize base of stack at program start
- ◆ Shared DLLs (see any immediate issues?)
- ◆ Procedure Linkage Table/Global Offset Table
- ◆ setjmp/longjmp require special handling
 - Must keep track of context (e.g., shadow stack location)

Summary

- ◆ Randomness is a potential defense mechanism
- ◆ Many issues for proper implementation
- ◆ Serious limitations on 32-bit architecture
 - "Thus, on 32-bit systems, runtime randomization cannot provide more than 16-20 bits of entropy"
 - Shacham et al.