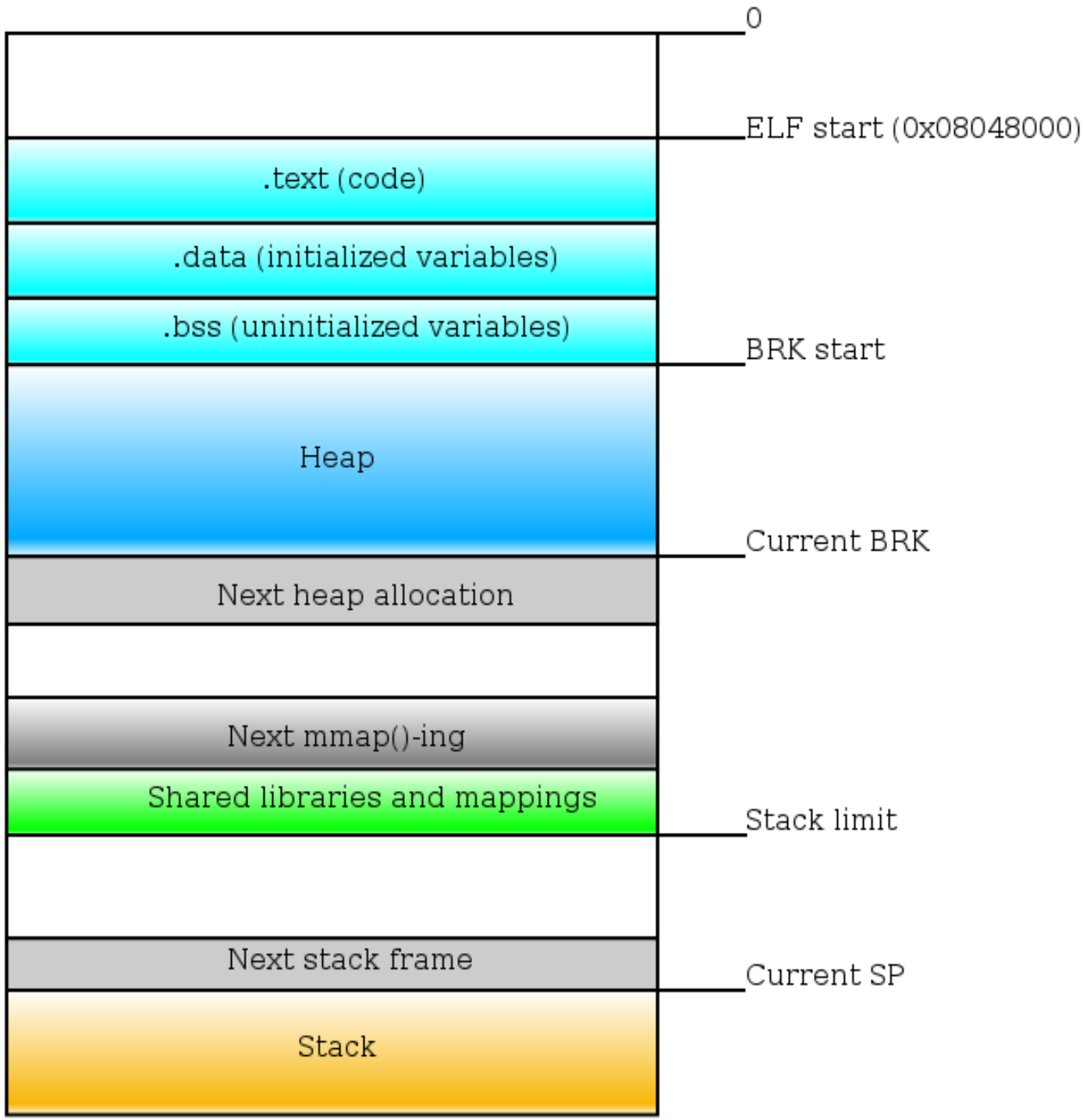
Intro to memory corruption mitigation

Radu Caragea
(radu.caragea@orangemail.ro)
11th July 2013
ROSEdu Summer Workshops

Overview


0
ELF start (0x08048000)
.text (code)
.data (initialized variables)
.bss (uninitialized variables)
BRK start
Heap
Current BRK
Next heap allocation
Next mmap()-ing
Shared libraries and mappings
Stack limit
Next stack frame
Current SP
Stack

Initial layout

- The heap is empty

Initial layout

- The heap is empty
- The shared libraries are mapped into the address space

Initial layout

- The heap is empty
- The shared libraries are mapped into the address space
- The stack is not empty
  - argc
  - argv[0], argv[1], ... , NULL
  - env[0], env[1], ... , NULL

Recon tools: ldd

- Shows dynamic link dependencies
- Also shows their order and load addresses

```
root@dmns [protection] # ldd randmin_01
        linux-gate.so.1 (0xf7fdd000)
        libc.so.6 => /lib32/libc.so.6 (0xf7e13000)
        /lib/ld-linux.so.2 (0xf7fde000)
root@dmns [protection] #
```

Recon tools: nm

- Shows symbols: function and static variable addresses

```
root@dmns [protection] # nm randmin_01
08049f0c d _DYNAMIC
0804a000 d _GLOBAL_OFFSET_TABLE_
080487f0 R _IO_stdin_used
         w _ITM_deregisterTMCloneTable
         w _ITM_registerTMCloneTable
         w _Jv_RegisterClasses
0804a04c D __TMC_END__
0804a04c B __bss_start
0804a03c D __data_start
0804a040 D __dso_handle
         w __gmon_start__
08048560 T __i686.get_pc_thunk.bx
08049f04 t __init_array_end
08049f00 t __init_array_start
         U __isoc99_scanf@@GLIBC_2.7
080487d0 T __libc_csu_fini
08048770 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
0804a04c D _edata
0804a05c B _end
080487d8 T _fini
080487ec R _fp_hw
0804843c T _init
08048530 T _start
08048692 T check_passwd
0804a044 D cmd
0804a03c W data_start
         U geteuid@@GLIBC_2.0
080486fc T main
         U memcmp@@GLIBC_2.0
0804a050 B passwd
         U puts@@GLIBC_2.0
         U rand@@GLIBC_2.0
0804862c T set_passwd
         U setuid@@GLIBC_2.0
         U sleep@@GLIBC_2.0
         U srand@@GLIBC_2.0
         U system@@GLIBC_2.0
         U time@@GLIBC_2.0
```

Recon tools: objdump

- Advanced tool. Use if you can dive into assembly code

Overview

NX

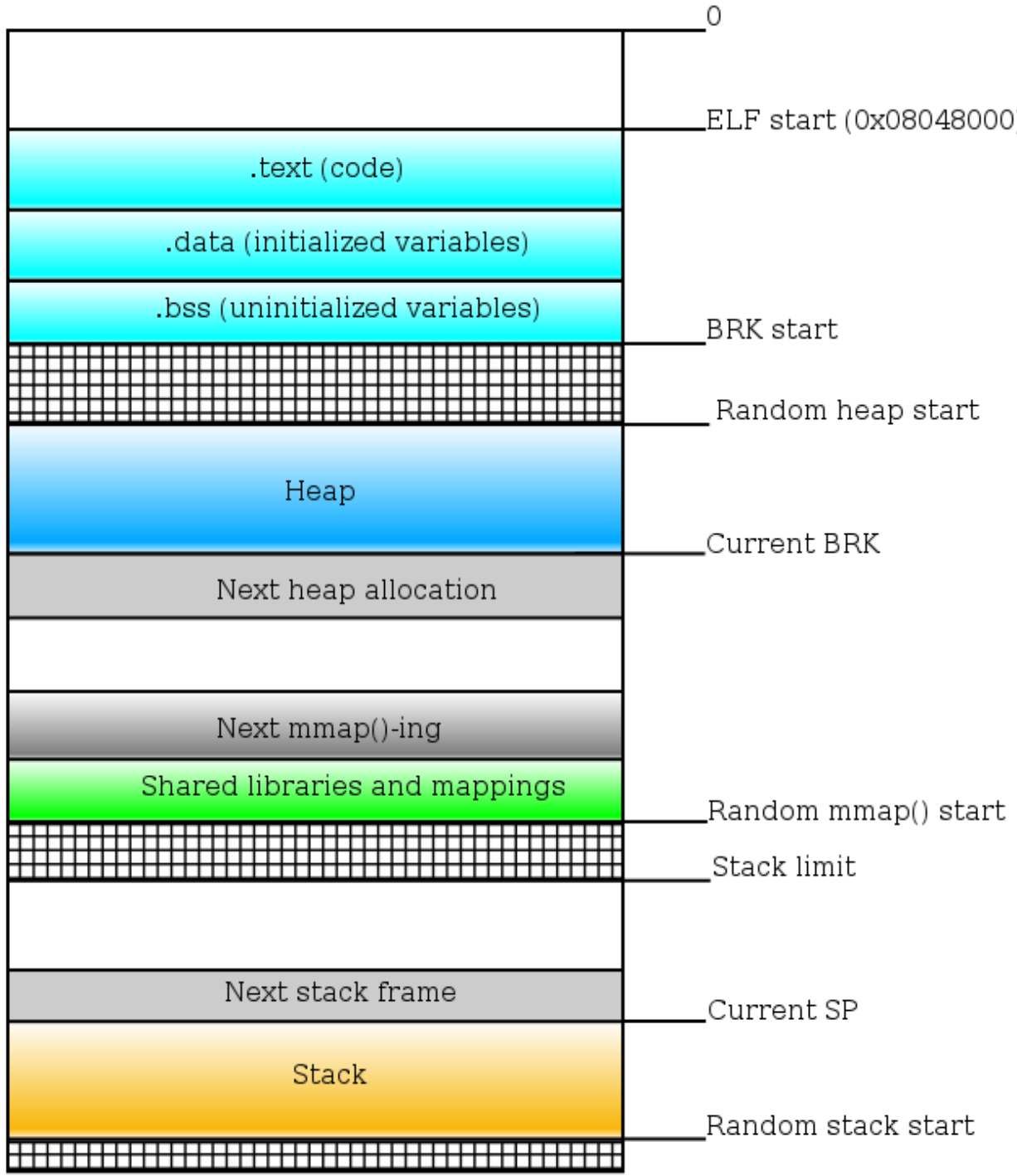
- Older CPUs only had R/W permission bits for memory pages
- Modern CPUs (since 2001) also include an execute bit. Why?
- W$\oplus$X

Is NX enough?

- Short answer: no
- Long answer: maybe. Attackers can write/overwrite buffers but anything written cannot be executed.
- Attackers can still overwrite the return address and then use:
  - Return-to-text
  - Return-to-plt
  - Return-to-libc
  - Return-to-*

ASLR

- Problem: attackers can still use return-to-* if they know the address of a desired function
- Solution: Randomize the address space layout
  - The stack start
  - The heap start
  - The mmap() base
- Kernel responsibility


0
ELF start (0x08048000)
.text (code)
.data (initialized variables)
.bss (uninitialized variables)
BRK start
Random heap start
Heap
Current BRK
Next heap allocation
Next mmap()-ing
Shared libraries and mappings
Random mmap() start
Stack limit
Next stack frame
Current SP
Stack
Random stack start

ldd under ASLR

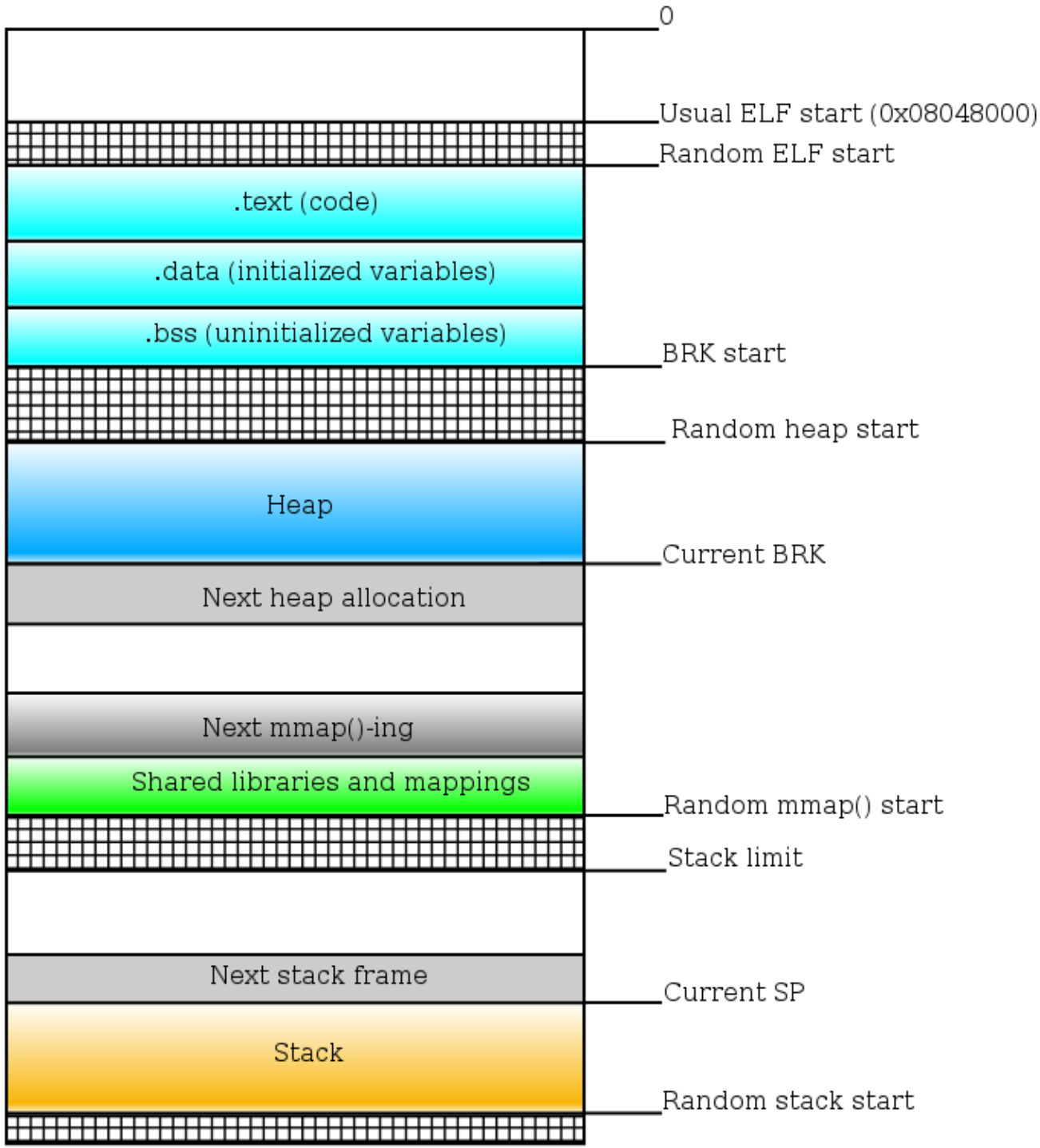
```
root@dmns [protection] # ldd randmin_01
        linux-gate.so.1 (0xf779f000)
        libc.so.6 => /lib32/libc.so.6 (0xf75d5000)
        /lib/ld-linux.so.2 (0xf77a0000)
root@dmns [protection] # ldd randmin_01
        linux-gate.so.1 (0xf76e0000)
        libc.so.6 => /lib32/libc.so.6 (0xf7516000)
        /lib/ld-linux.so.2 (0xf76e1000)
root@dmns [protection] #
```

Is ASLR enough?

- Short answer: maybe
- Long answer: depends on the OS features and the CPU.
- On 32 bit Linux unprivileged users (with shell access) can deactivate mmap randomization and then use:
  - Return-to-text
  - Return-to-plt
  - Return-to-libc
  - Return-to-*
- If shell access is disabled, brute force guessing can be applied. Is it feasible?
- On 64 bit Linux unprivileged users can only use:
  - Return-to-text
  - Return-to-plt

PIE

- Problem: In real-life applications the .text segment is big enough to create a working exploit without using shared libs.
- But the .text segment is at a fixed position everytime
- Solution: Position independent executables with a random start (basically ASLR on .text)
- Joint kernel-compiler responsibility


0
Usual ELF start (0x08048000)
Random ELF start
.text (code)
.data (initialized variables)
.bss (uninitialized variables)
BRK start
Random heap start
Heap
Current BRK
Next heap allocation
Next mmap()-ing
Shared libraries and mappings
Random mmap() start
Stack limit
Next stack frame
Current SP
Stack
Random stack start

Is PIE enough?

- Short answer: mainly yes
- Long answer: still depends on OS and CPU.
- Attackers on 32 bits can still use brute force guessing (takes a few minutes)
- Attackers on 64 bits can still use brute force guessing (takes 6-8 hours)
- Why isn't PIE used by default in compilations? up to 20% slow down

SSP

- Problem: even with ASLR + PIE, return addresses can still be overwritten
- Solution: Set a random stack 'cookie'/'canary' between buffers and return address and check for corruption before returning
- Compiler responsibility

How hard is it to defeat SSP?

- (Usually) next to impossible, even with ASLR and PIE turned off.
- Logic bugs can still happen
- Bad coding can lead to attackers overwriting the stack check function and bypassing SSP

Recon tools: checksec

- Shell script that checks for security features in apps, shared libraries, in the kernel and system-wide