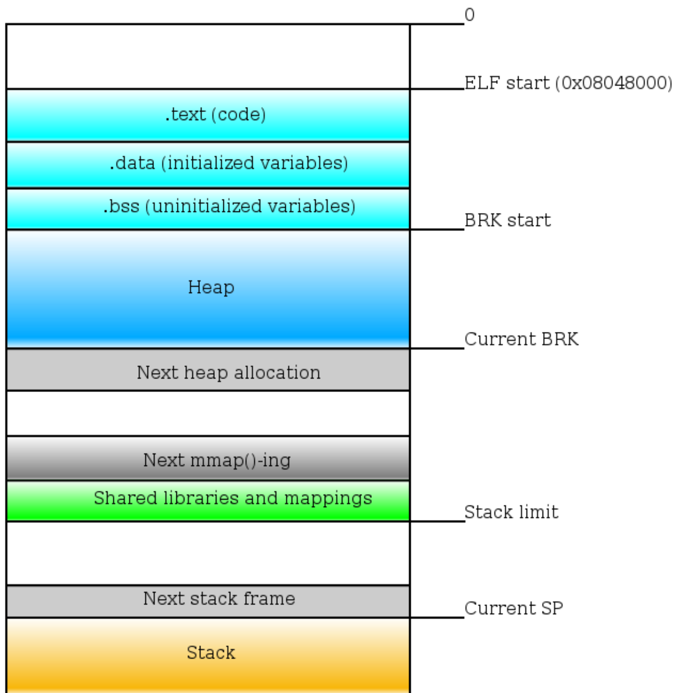
Intro to memory corruption mitigation

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

Overview

1. Address space recap
2. Memory corruption mitigation techniques


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

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
  - envp[0], envp[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: functions and non-stack variables

```
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 (for disassembly purposes)

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

Overview

1. Address space recap
2. Memory corruption mitigation techniques

Who cares?

- If the software does what it's supposed to on the test cases why bother?

Microsoft

- http://arstechnica.com/security/2013/06/microsoft-will-pay-up-to-100k-for-new-windows-exploit-techniques

Microsoft will pay up to $100K for new Windows exploit techniques

Redmond finally joins Google, Mozilla, by offering cash rewards for security flaws.

by **Peter Bright** - June 19 2013, 10:20pm EEST

HP

- http://dvlabs.tippingpoint.com/blog/2013/01/17/pwn2own-2013

Rules & Prizes

HP ZDI is offering more than half a million dollars (USD) in cash and prizes during the competition for vulnerabilities and exploitation techniques in the below categories. The first contestant to successfully compromise a selected target will win the prizes for the category.

- Web Browser
  - Google Chrome on Windows 7 ($100,000)
  - Microsoft Internet Explorer, either
    - IE 10 on Windows 8 ($100,000), or
    - IE 9 on Windows 7 ($75,000)
  - Mozilla Firefox on Windows 7 ($60,000)
  - Apple Safari on OS X Mountain Lion ($65,000)
- Web Browser Plug-ins using Internet Explorer 9 on Windows 7
  - Adobe Reader XI ($70,000)
  - Adobe Flash ($70,000)
  - Oracle Java ($20,000)

Google Chrome

- http://blog.chromium.org/2013/03/pwnium-3-and-pwn2own-results.html

Today we're announcing our third Pwnium competition—Pwnium 3. Google Chrome is already featured in the Pwn2Own competition this year, so Pwnium 3 will have a new focus: Chrome OS.

We'll issue Pwnium 3 rewards for Chrome OS at the following levels, up to a total of $3.14159 million USD:

- $110,000: browser or system level compromise in guest mode or as a logged-in user, delivered via a web page.
- $150,000: compromise with device persistence -- guest to guest with interim reboot, delivered via a web page.

Classical stack overflow exploit

- An application processes data given by a user and doesn't check for input length
- The user input is stored on the stack

Classical stack overflow exploit

- An application processes data given by a user and doesn't check for input length
- The user input is stored on the stack
- A malicious user can send really long inputs and overwrite the return address of the stack frame

Classical stack overflow exploit

- An application processes data given by a user and doesn't check for input length
- The user input is stored on the stack
- A malicious user can send really long inputs and overwrite the return address of the stack frame
- When the function ends, the program will jump to the address that gets overwritten.
- If you set the return address to the beginning of the buffer it will jump there

Classical stack overflow exploit

- An application processes data given by a user and doesn't check for input length
- The user input is stored on the stack
- A malicious user can send really long inputs and overwrite the return address of the stack frame
- When the function ends, the program will jump to the address that gets overwritten.
- If you set the return address to the beginning of the buffer it will jump there
- What if the buffer contains executable code?

NX (Never eXecute)

- 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?

Is NX enough?

- Short answer: no

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-*

```
void f()
{
        printf("How did you do that?\n");
}
int main()
{
        char name[10];
        scanf("%s", name);
        printf("Hello %s\n", name);
        return 0;
}
```

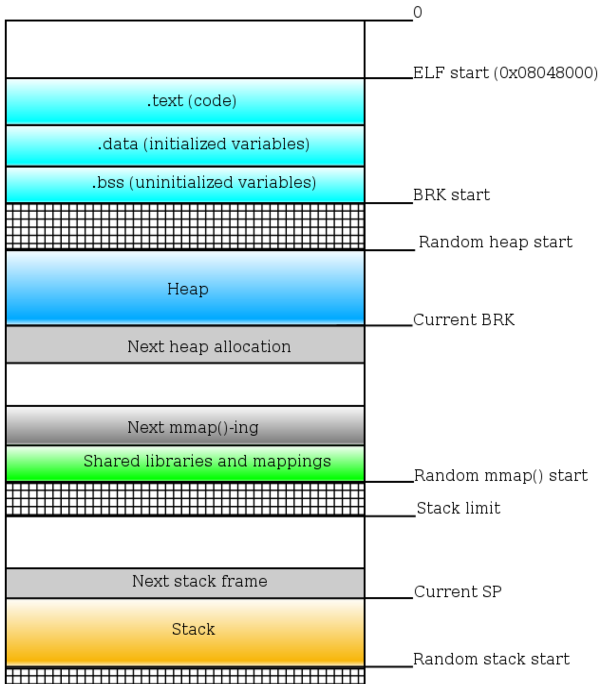
```
root@dmns [example] # ./ret
user
Hello user
root@dmns [example] # echo -e "RADU" | ./ret
Hello RADU
root@dmns [example] # echo -e "\x72\x61\x64\x75" | ./ret
Hello radu
root@dmns [example] # echo -e "AAAAAAAAAAAAAAAAAAAAAA" | ./ret
Hello AAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
root@dmns [example] # echo -e "AAAAAAAAAAAAAAAAAAAAAA\xac\x84\04\x08"  | ./ret
Hello AAAAAAAAAAAAAAAAAAAAAA¬
How did you do that?
Segmentation fault
root@dmns [example] #
```

ASLR

- Problem: attackers can still use return-to-* if they know the address of a desired function

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?

Is ASLR enough?

- Short answer: maybe

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?

Is ASLR enough?

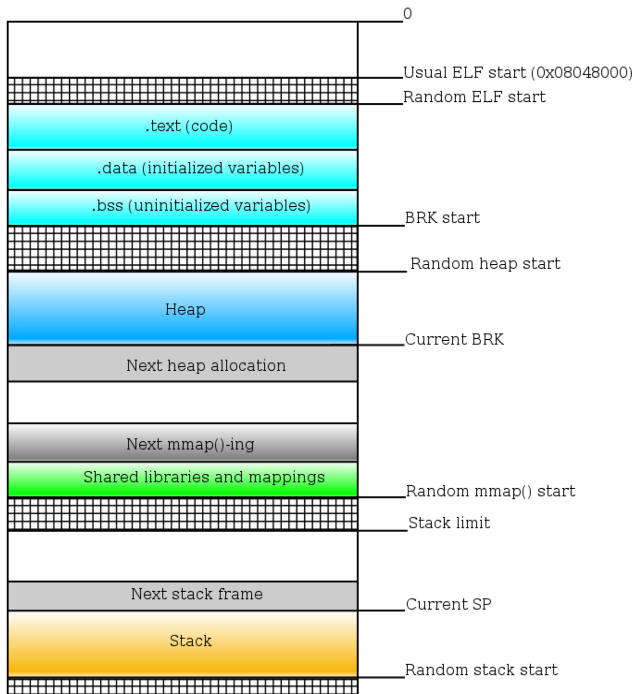
- Short answer: maybe
- Long answer: depends on the OS features and the CPU.
- On 32 bit Linux unprivileged users (with shell access) can deactivate mmap randomization and then use:
  - Return-to-text
  - Return-to-plt
  - Return-to-libc
  - Return-to-*
- If shell access is disabled, brute force guessing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

PIE

- Problem: the .text segment is at a fixed position everytime

PIE

- Problem: 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?

Is PIE enough?

- Short answer: mostly yes

Is PIE enough?

- Short answer: mostly 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

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

```
root@dmns [protection] # checksec.sh --file randmin_01
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Full RELRO      No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   randmin_01
root@dmns [protection] # checksec.sh --file randmin_07
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   randmin_07
root@dmns [protection] #
```