Rootcamp

Advanced stack based buffer overflow

Summary

Red line: vulnerable app

Protections and how to defeat them

- DEP
- ASCII Armor
- ASLR
- Stack Smashing Protector

Conclusion

Vulnerable app

See code in serv.c

→ 32 bits

→ Server listens on a port

→ reads user input and call a vulnerable function with it

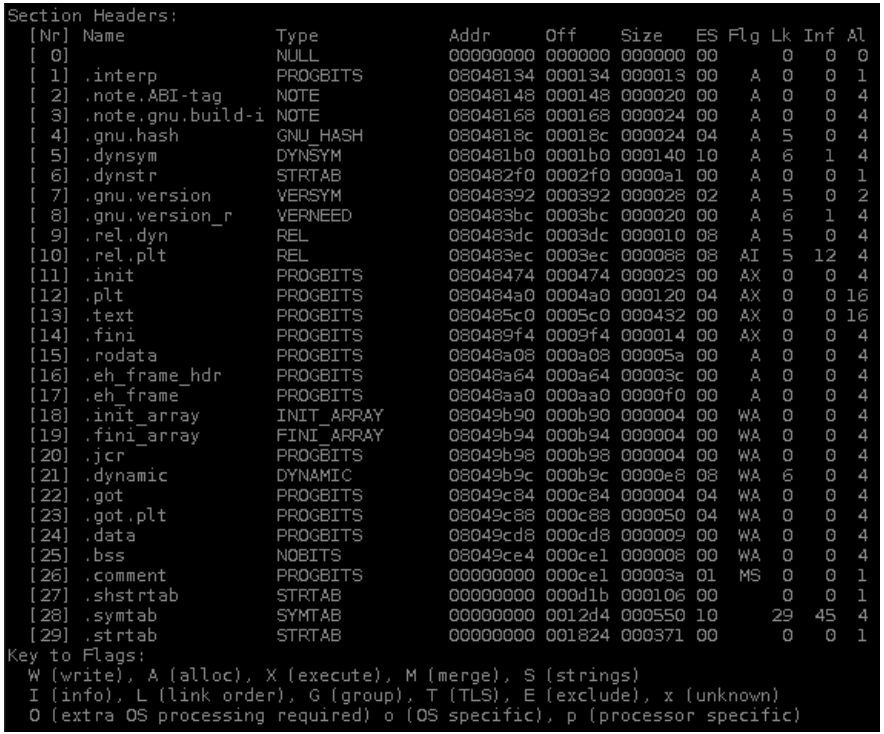
DEP (aka. NX bit)

Data Execution Protection (No eXecution bit)

In linux kernel since 2004

Set data segments and stack to NX space

DEP (aka. NX bit)



```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048168 000168 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        0804818c 00018c 000024 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481b0 0001b0 000140 10   A  6   1  4
  [ 6] .dynstr           STRTAB          080482f0 0002f0 0000a1 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          08048392 000392 000028 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         080483bc 0003bc 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             080483dc 0003dc 000010 08   A  5   0  4
  [10] .rel.plt          REL             080483ec 0003ec 000088 08  AI  5  12  4
  [11] .init             PROGBITS        08048474 000474 000023 00  AX  0   0  4
  [12] .plt              PROGBITS        080484a0 0004a0 000120 04  AX  0   0 16
  [13] .text             PROGBITS        080485c0 0005c0 000432 00  AX  0   0 16
  [14] .fini             PROGBITS        080489f4 0009f4 000014 00  AX  0   0  4
  [15] .rodata           PROGBITS        08048a08 000a08 00005a 00   A  0   0  4
  [16] .eh_frame_hdr     PROGBITS        08048a64 000a64 00003c 00   A  0   0  4
  [17] .eh_frame         PROGBITS        08048aa0 000aa0 0000f0 00   A  0   0  4
  [18] .init_array       INIT_ARRAY      08049b90 000b90 000004 00  WA  0   0  4
  [19] .fini_array       FINI_ARRAY      08049b94 000b94 000004 00  WA  0   0  4
  [20] .jcr              PROGBITS        08049b98 000b98 000004 00  WA  0   0  4
  [21] .dynamic          DYNAMIC         08049b9c 000b9c 0000e8 08  WA  6   0  4
  [22] .got              PROGBITS        08049c84 000c84 000004 04  WA  0   0  4
  [23] .got.plt          PROGBITS        08049c88 000c88 000050 04  WA  0   0  4
  [24] .data             PROGBITS        08049cd8 000cd8 000009 00  WA  0   0  4
  [25] .bss              NOBITS          08049ce4 000ce1 000008 00  WA  0   0  4
  [26] .comment          PROGBITS        00000000 000ce1 00003a 01  MS  0   0  1
  [27] .shstrtab         STRTAB          00000000 000d1b 000106 00      0   0  1
  [28] .symtab           SYMTAB          00000000 0012d4 000550 10     29  45  4
  [29] .strtab           STRTAB          00000000 001824 000371 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

DEP (aka. NX bit)

Bypass:

Don’t use old ways… No shellcode allowed.

ASCII Armor

Load libraries in the 16 first megabyte of the address space.

As a result, all code and data of these shared libraries are located at addresses beginning with a NULL byte.

Well known bypass technique: return to .plt

In our case, it’s not a problem.

ASLR

Address Space Layout Randomization

```
cat /proc/sys/kernel/randomize_va_space
```

0: Disable ASLR. This setting is applied if the kernel is booted with the norandmaps boot parameter.

1: Randomize the positions of the stack, virtual dynamic shared object (VDSO) page, and shared memory regions.

The base address of the data segment is located immediately after the end of the executable code segment.

2: Randomize the positions of the stack, VDSO page, shared memory regions, and the data segment. This is the default setting.

SSP

Stack Smashing Protection

gcc since version 4.x

3 Main different protection :

- Stack reordering
- Padding
- Canary bytes

SSP: Stack reordering

Pointers are moved at the bottom

Buffers are moved at the top

Example:

```
int func(char *arg1, char *arg2) {
        int a;
        int *b;
        char c[10];
        char d[3];

        memcpy(c,arg1,strlen(arg1));
        *b = 5;
        memcpy(d,arg2,strlen(arg2));
        return *b;
}
```

SSP: Padding

Prevent Off-by-one attacks

Gcc from version 3.3.x and 3.4.x

- 20 bytes length

SSP: Canary bytes

- Random

  masked with 0x00ffffff

- Not so random

  0x000AFF0D

SSP: Random canary

The fork() case

- Shared memory for each process
- In our case,

Execve() ?

Not so random after all … :-)

Demo time

Canary bypass

Overwriting RIP.

Conclusion

Security has been improved but, in some cases, we can still bypass it.

Never EVER trust user input!!

http://phrack.org/issues/67/13.html

Questions?

Contact: quentin.roland-gosselin@epitech.eu

Twitter: @Ark_444

GitHub: Ark444

IRC: Server: irc.rezosup.org

Channel: #cq-toulouse

User: Ark