Software Exploitation

Assignment 1

Timo Lehosvuo, M3426
TTV18S1

Harjoitustyö
Software Exploitation, Mikko Neijonen
8.4.2021
Tekniikan ala
Tieto- ja viestintätekniikka

1. Initial setup

I started the assignment by installing the needed tools to my kali linux and downloading the needed files from moodle. I used the commands: *"apt-get install build-essential gcc-multilib"* and *"apt-get install gdb nasm"* for downloading the needed tools. After this, I navigated to the directories "src" and "asm" and compiled the programs using "make":

```
root@kali:~/Desktop/src# ls
array                   environ    hello     Makefile   types.c
array.c                 environ.c  hello.c   pointer
disassembly_required    greeter    inputs    pointer.c
disassembly_required.c  greeter.c  inputs.c  types
root@kali:~/Desktop/src# ls ../asm/
hello  hello.asm  Makefile
root@kali:~/Desktop/src#
```

Figure 1: Compiled files

2. Linux toolchain

The assignment was to get to know some tools used in software exploitation and to examine some of the files with the tools. Tools used in this assignment are: file, readelf, objdump, Makefile, env and gdb. I started investigating the files using the "file" command:

```
root@kali:~/Desktop/asm# file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
root@kali:~/Desktop/asm# file ../src/array
../src/array: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, inter
preter /lib/ld-linux.so.2, BuildID[sha1]=104d2505ccfef830174d3a1d8a78c77b2a4bc861, for GNU/Linux 3.2
.0, with debug_info, not stripped
```

Figure 2: Results from file command

Depending on the type of the executable the result shows what file format the executable is using, byte order, architecture, identifies the operating systems ABI (version 1 (SYSV)), if the libraries the program uses are statically or dynamically linked, the id for the build (compilation), what version of linux the executable was compiled and if its stripped or not (stripped files don’t contain debug info or other information that executable does not need to be able to run to reduce the size of the file). I didn’t include picture of the other files since the result were the same for all of the files. After this switched to “readelf” and checked what can I find from the results. I used the command “readelf -a hello”:

```
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8049000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          8452 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         6
  Section header string table index: 5

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        08049000 001000 000022 00  AX  0   0 16
  [ 2] .data             PROGBITS        0804a000 002000 00000e 00  WA  0   0  4
  [ 3] .symtab           SYMTAB          00000000 002010 0000a0 10      4   6  4
  [ 4] .strtab           STRTAB          00000000 0020b0 00002b 00      0   0  1
  [ 5] .shstrtab         STRTAB          00000000 0020db 000027 00      0   0  1
```

Figure 3: Readelf result

A lot of the information is the same as using the command “file” but in a more understandable form. Also “readelf” shows information about the file’s headers, sections, starting addresses of headers and sections, memory addresses, sizes, flags, and some function names. I moved on and used “objdump” on the executables with different parameters and the information is similar to the former tools:

```
root@kali:~/Desktop/asm# objdump -x hello

hello:     file format elf32-i386
hello
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08049000

Program Header:
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000094 memsz 0x00000094 flags r--
    LOAD off    0x00001000 vaddr 0x08049000 paddr 0x08049000 align 2**12
         filesz 0x00000022 memsz 0x00000022 flags r-x
    LOAD off    0x00002000 vaddr 0x0804a000 paddr 0x0804a000 align 2**12
         filesz 0x0000000e memsz 0x0000000e flags rw-

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000022  08049000  08049000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         0000000e  0804a000  0804a000  00002000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
SYMBOL TABLE:
08049000 l    d  .text  00000000 .text
0804a000 l    d  .data  00000000 .data
00000000 l    df *ABS*  00000000 hello.asm
0804a000 l       .data  00000000 msg
0000000e l       *ABS*  00000000 len
08049000 g       .text  00000000 _start
0804a00e g       .data  00000000 __bss_start
0804a00e g       .data  00000000 _edata
0804a010 g       .data  00000000 _end
```

Figure 4: objdump result

You can find the file format, architecture, start address etc, but “objdump” also supports disassembly and even has parameters for different architectures:

```
root@kali:~/Desktop/asm# objdump -D hello

hello:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:       b8 04 00 00 00          mov    $0x4,%eax
 8049005:       bb 01 00 00 00          mov    $0x1,%ebx
 804900a:       b9 00 a0 04 08          mov    $0x804a000,%ecx
 804900f:       ba 0e 00 00 00          mov    $0xe,%edx
 8049014:       cd 80                   int    $0x80
 8049016:       b8 01 00 00 00          mov    $0x1,%eax
 804901b:       bb 00 00 00 00          mov    $0x0,%ebx
 8049020:       cd 80                   int    $0x80

Disassembly of section .data:

0804a000 <msg>:
 804a000:       48                      dec    %eax
 804a001:       65 6c                   gs insb (%dx),%es:(%edi)
 804a003:       6c                      insb   (%dx),%es:(%edi)
 804a004:       6f                      outsl  %ds:(%esi),(%dx)
 804a005:       2c 20                   sub    $0x20,%al
 804a007:       77 6f                   ja     804a078 <_end+0x68>
 804a009:       72 6c                   jb     804a077 <_end+0x67>
 804a00b:       64                      fs
 804a00c:       2e                      cs
 804a00d:       0a                      .byte 0xa
```

Figure 5: objdump disassembly

Next, I needed to figure out what flags I needed to compile a 32-bit program with “gcc” and the answer is “-m32” and for 64-bit program you can use “-m64” or just leave it out since it is the default for “gcc” nowadays (usually). After this the assignment was to study the “environ.c” located in the “src” directory. It seemed that the program prints all the environment variables:

```
root@kali:~/Desktop/src# ./environ
argv[0] @ 0xffbc14e4 = ./environ
environ[0] @ 0xffbc14ee = SHELL=/bin/bash
environ[1] @ 0xffbc14fe = SESSION_MANAGER=local/kali:@/tmp/.ICE-unix/1293,unix/k
ali:/tmp/.ICE-unix/1293
environ[2] @ 0xffbc154c = QT_ACCESSIBILITY=1
environ[3] @ 0xffbc155f = COLORTERM=truecolor
environ[4] @ 0xffbc1573 = XDG_MENU_PREFIX=gnome-
environ[5] @ 0xffbc158a = GNOME_DESKTOP_SESSION_ID=this-is-deprecated
environ[6] @ 0xffbc15b6 = POWERSHELL_TELEMETRY_OPTOUT=1
environ[7] @ 0xffbc15d4 = SSH_AUTH_SOCK=/run/user/0/keyring/ssh
environ[8] @ 0xffbc15fa = XMODIFIERS=@im=ibus
environ[9] @ 0xffbc160e = DESKTOP_SESSION=gnome
environ[10] @ 0xffbc1624 = SSH_AGENT_PID=1259
environ[11] @ 0xffbc1637 = GTK_MODULES=gail:atk-bridge
environ[12] @ 0xffbc1653 = PWD=/root/Desktop/src
environ[13] @ 0xffbc1669 = LOGNAME=root
environ[14] @ 0xffbc1676 = XDG_SESSION_DESKTOP=gnome
environ[15] @ 0xffbc1690 = QT_QPA_PLATFORMTHEME=qt5ct
environ[16] @ 0xffbc16ab = XDG_SESSION_TYPE=x11
environ[17] @ 0xffbc16c0 = GPG_AGENT_INFO=/run/user/0/gnupg/S.gpg-agent:0:1
environ[18] @ 0xffbc16f1 = XAUTHORITY=/run/user/0/gdm/Xauthority
environ[19] @ 0xffbc1717 = GJS_DEBUG_TOPICS=JS ERROR;JS LOG
```

Figure 6: Environ script result

I wanted to be sure and created a new one with command "export testi=testailen" and ran the program again:

```
environ[42] @ 0xffb22f3e = GDMSESSION=gnome
environ[43] @ 0xffb22f4f = testi=testailen
environ[44] @ 0xffb22f5f = DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/0/bus
environ[45] @ 0xffb22f92 = _JAVA_OPTIONS=-Dawt.useSystemAAFontSettings=on -Dswin
g.aatext=true
environ[46] @ 0xffb22fd5 = OLDPWD=/root
environ[47] @ 0xffb22fe2 = _=./environ
foo @ 0xffb21d14 = 1
```

Figure 7: New environmental variable

appears it works. Tested what it does with arguments:

```
root@kali:~/Desktop/src# ./environ 5 testi
argv[0] @ 0xffa394cc = ./environ
argv[1] @ 0xffa394d6 = 5
argv[2] @ 0xffa394d8 = testi
```

Figure 8: Environ program with arguments

it prints the arguments. Also, the memory address of “foo” changes every time the program is ran:

```
foo @ 0xffd61724 = 1
root@kali:~/Desktop/src#
```

Figure 9: Address of foo

I switched to using “env” and ran it with no parameters and with the parameter “-i”:

```
root@kali:~/Desktop/src# env ./environ
argv[0] @ 0xffda74d1 = ./environ
environ[0] @ 0xffda74db = SHELL=/bin/bash
environ[1] @ 0xffda74eb = SESSION_MANAGER=local
```

Figure 10: Env without parameters

```
root@kali:~/Desktop/src# env -i ./environ
argv[0] @ 0xffbc7fe4 = ./environ
foo @ 0xffbc6e14 = 1
root@kali:~/Desktop/src#
```

Figure 11: Env with the parameter "-i"

and it indeed does not show a single environment variable with the parameter “-i”. Then I disabled address space randomization with the command “sysctl -w kernel.randomize_va_space=0” and ran the program again:

```
environ[47] @ 0xffffdfe2 = _=./environ
foo @ 0xffffd264 = 1
root@kali:~/Desktop/src#
```

Figure 12: Address of foo stays the same

The address of “foo” stays the same as long as the parameters stay the same. Now that I knew what the “environ” program does I needed to know how the “src/inputs.c” works so I can get the cookie out. I used nano to check the code from the file “inputs.c” and started testing:

```
int main(int argc, char **argv) {
    // isatty() tests whether a file decriptor (in this case stdin)
    // refers to a terminal.

    // You can use pipe or redirection to change standard input
    // from terminal to file.

    if (isatty(STDIN_FILENO)) {
        fprintf(stderr, "Nope.\n");
    } else {
        fprintf(stderr, "Nicely done, input is not a terminal.\n");
    }

    // You can use pipe or redirection to change standard output
    // from terminal to file.

    if (!isatty(STDOUT_FILENO)) {
        fprintf(stderr, "Output is not a terminal.\n");
    }

    // You can use pipe or redirection to change standard output
    // from terminal to file.
```

Figure 13: inputs.c code

```
root@kali:~/Desktop/src# ./inputs 2>testi | ./inputs 2>testi | ./inputs
Nicely done, input is not a terminal.
root@kali:~/Desktop/src# ls
array                  disassembly_required.c  greeter.c  inputs.c   testi
array.c                environ                 hello      Makefile   types
cookie                 environ.c               hello.c    pointer    types.c
disassembly_required   greeter                 inputs     pointer.c
root@kali:~/Desktop/src# cat cookie
Here, have a cookie.
```

Figure 14: Cookie

Had to learn how to redirection works but managed to get the cookie out. The final task for the assignment 1 was to use the debugger “gdb” and to debug “src/disassembly_required.c”. Started the debuggerd with “gdb disassembly_required” and then I started the program with “run”:

```
(gdb) run
Starting program: /root/Desktop/src/disassembly_required
'"foo" is a red magical unicorn (len=30)
"bar" is a green magical unicorn (len=32)
"baz" is a blue magical unicorn (len=31)
"qux" is a shiny magical unicorn (len=32)
[Inferior 1 (process 3070) exited normally]
(gdb)
```

Figure 15: gdb run

Then I set a breakpoint using “break 10”

```
(gdb) break 10
Breakpoint 1 at 0x565561a6: file disassembly_required.c, line 24.
(gdb) run
Starting program: /root/Desktop/src/disassembly_required

Breakpoint 1, strcolor (color=RED) at disassembly_required.c:24
24        char *strcolor(enum color color) { return colornames[color]; }
(gdb)
```

Figure 16: Breakpoint

and step through the code with commands:

s = single line step

n = single line instruction

c = continue execution

```
(gdb) s
magical_unicorn (name=0x56557056 "qux", color=SHINY)
    at disassembly_required.c:31
31            return len;
(gdb) n
32        }
(gdb) c
Continuing.
"qux" is a shiny magical unicorn (len=32)
[Inferior 1 (process 3445) exited normally]
(gdb)
```

Figure 17: Stepping through the code

After this I backtraced with ”bt”:

```
(gdb) bt
#0  magical_unicorn (name=0x5655704e "bar", color=GREEN)
    at disassembly_required.c:31
#1  0x5655626d in main (argc=1, argv=0xffffd2f4) at disassembly_required.c:40
(gdb)
```

Figure 18: Backtrace

and printed the value of BLUE with “p /xBLUE”:

```
(gdb) p /xBLUE
$5 = 0x2
(gdb)
```

Figure 19: Value of BLUE

Then I had to figure out addresses for the program’s functions. I found the addresse for main with “info address main”:

```
(gdb) info address main
Symbol "main" is a function at address 0x56556218.
```

Figure 20: Address of main

and for the addresses of the other functions, I printed them out with “info functions” and then used “info address” command for the functions to find their address:

```
(gdb) info functions
All defined functions:

File disassembly_required.c:
26:     int magical_unicorn(char *, enum color);
34:     int magical_unicorn_struct(struct unicorn *);
38:     int main(int, char **);
24:     char *strcolor(enum color);

Non-debugging symbols:
0x56556000  _init
0x56556030  printf@plt
0x56556040  __libc_start_main@plt
0x56556050  __cxa_finalize@plt
0x56556060  _start
0x565560a0  __x86.get_pc_thunk.bx
```

Figure 21: Functions

```
(gdb) info address magical_unicorn
Symbol "magical_unicorn" is a function at address 0x565561b2.
(gdb) info address magical_unicorn_struct
Symbol "magical_unicorn_struct" is a function at address 0x565561ee.
(gdb) info address strcolor
Symbol "strcolor" is a function at address 0x56556199.
(gdb)
```

Figure 22: Addresses of the functions

Lastly I figured out the address for variable “u” and for “u.name” and “u.color” using print with “&”:

```
(gdb) p &u
$9 = (struct unicorn *) 0xffffd228
(gdb) p &u.name
$10 = (char **) 0xffffd228
(gdb) p &u.color
$11 = (enum color *) 0xffffd22c
(gdb)
```

Figure 23: Print with &

I compiled the files using “Makefile” and the platform I used to test the tools and programs was:

```
root@kali:~# uname -a
Linux kali 5.8.0-kali2-amd64 #1 SMP Debian 5.8.10-1kali1 (2020-09-22) x86_64 GNU
/Linux
```