Program Analysis

Sicurezza (CT0539) 2024-25
Università Ca' Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Exercises

Exercise 1: change the branch. Modify the `/home/rookie/Assembly/count` executable file so that the loop ends at 8 instead of 9

Hint: try to change **`jle`** into **`jl`**, this summary of x86 opcodes will help you!

Exercise 2: change the value. Modify the `/home/rookie/Assembly/count` executable file so that it steps of 2 instead of 1 (only prints even values)

Hint: In this case you need to change data

Exercise 3: skip a branch. Modify the `/home/rookie/Assembly/checkPassword` so that it skips the password check

Hint: nop is your friend! (opcode `0x90`)

Dynamic analysis

Static vs. dynamic analysis

Programs are analysed in two ways:

Static analysis: by **inspecting** the assembly we try to understand program **logic** (tools can infer program control flow effectively)

Dynamic analysis: program is run with **debuggers** in order to observe its dynamic **behaviour** (for example, malware executed in *sandboxes*)

Usually the two techniques **complement** each other

The GNU project debugger (gdb) allows for executing programs **step-by-step**, inspecting memory and registers

It can be used both for debugging and (dynamically) **analyzing** programs

gdb

Consider again program `count.c`:

```c
#include <stdio.h>

int main()
{
    int i;
    for (i=0; i<10; i++)
        printf("%d ",i);
    printf("\n");
}
```

If we compile it with the `-g` option we have additional information on the **source code**, directly in gdb

```
$ gcc -g count.c -o count
$ ./count
0 1 2 3 4 5 6 7 8 9
```

Program works as before but source code is **included** in the executable, for debugging

gdb in docker

To run gdb in docker you need a customized *secure computing mode* (**seccomp**) profile

Download it here (or from secgroup)

Run docker as:

```
docker run  -it --security-opt seccomp=./gdb.json secunive/sec:testbed
```

NOTE: for Mac M1 run `gdb` with `gdb-m1` and let me know if you have issues.
The program will start automatically and stop at the beginning of `main`.

Starting gdb

```
$ gcc -g count.c -o count
$ gdb -q count
Reading symbols from count...done.
(gdb) list
1    #include <stdio.h>
2
3    int main()
4    {
5        int i;
6        for (i=0; i<10; i++)
7            printf("%d ",i);
8        printf("\n");
9    }
(gdb)
```

We now disassemble the program. We can set **Intel syntax** as follows:

```
(gdb) set disassembly intel
```

NOTE: This can be made the **default syntax** by executing the following (in the home directory):

```
$ echo "set disassembly intel" > ~/.gdbinit
```

Already set in the docker image!

Disassembling

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000000068a <+0>:     push   rbp
   0x000000000000068b <+1>:     mov    rbp,rsp
   0x000000000000068e <+4>:     sub    rsp,0x10
   0x0000000000000692 <+8>:     mov    DWORD PTR [rbp-0x4],0x0
   0x0000000000000699 <+15>:    jmp    0x6b5 <main+43>
   0x000000000000069b <+17>:    mov    eax,DWORD PTR [rbp-0x4]
   0x000000000000069e <+20>:    mov    esi,eax
   0x00000000000006a0 <+22>:    lea    rdi,[rip+0xad]        # 0x754
   0x00000000000006a7 <+29>:    mov    eax,0x0
   0x00000000000006ac <+34>:    call   0x560 <printf@plt>
   0x00000000000006b1 <+39>:    add    DWORD PTR [rbp-0x4],0x1
   0x00000000000006b5 <+43>:    cmp    DWORD PTR [rbp-0x4],0x9
   0x00000000000006b9 <+47>:    jle    0x69b <main+17>
   0x00000000000006bb <+49>:    mov    edi,0xa
   0x00000000000006c0 <+54>:    call   0x550 <putchar@plt>
   0x00000000000006c5 <+59>:    mov    eax,0x0
   0x00000000000006ca <+64>:    leave
   0x00000000000006cb <+65>:    ret
End of assembler dump.
```

A note on short names

In gdb it is possible to specify a command by **any prefix**, as soon as there is no ambiguity

Tab **autocompletes**, so it is possible to check for matching commands

```
(gdb) disa                ← ambiguous
disable      disassemble

(gdb) disas main          ← OK!
```

Example:

```
(gdb) set disassembly intel
```

is, in fact:

```
(gdb) set disassembly-flavor intel
```

Often short names are used without even noticing!

⇒ **use tab** to check the full name

Breakpoints

Breakpoints allow us to stop the execution at a particular address

⇒ State inspection is possible!

```
(gdb) break main
Breakpoint 1 at 0x692: file
count.c, line 6.

(gdb) run
Starting program: /tmp/r1x/count
Breakpoint 1, main () at count.c:6
6        for (i=0; i<10; i++)
```

```
(gdb) disass main
Dump of assembler code for function main:
   0x000055555555468a <+0>:    push   rbp
   0x000055555555468b <+1>:    mov    rbp,rsp
   0x000055555555468e <+4>:    sub    rsp,0x10
=> 0x0000555555554692 <+8>:    mov    DWORD PTR [rbp-0x4],0x0
   0x0000555555554699 <+15>:   jmp    0x5555555546b5 <main+43>
   0x000055555555469b <+17>:   mov    eax,DWORD PTR [rbp-0x4]
   ...
```

Breakpoints can be set to a particular address using the syntax `break *addr`

```
(gdb) break *0x000055555555469b

Breakpoint 2 at 0x55555555469b: file
count.c, line 7.
```

Inspecting registers

Registers can be inspected with `info registers`

```
(gdb) info registers
rax            0x55555555468a 93824992233098
rbx            0x0    0
...
rbp            0x7fffffffe5d0 0x7fffffffe5d0
rsp            0x7fffffffe5c0 0x7fffffffe5c0
r8             0x7ffff7dd0d80 140737351847296
...
rip            0x555555554692 0x555555554692 <main+8>
eflags         0x206  [ PF IF ]

(gdb) info registers rsi
rsi            0x7fffffffe6b8 140737488348856
```

Examining memory (x)

Command **x** allows for memory inspection

`x/<num><format><size> <addr>`

num: the number of elements to inspect

format: the format that should be used to display values (1 char)

size: the size of elements: **b** byte, **h** (halfword) 2b, **w** 4b, **g** 8b

Formats:

o octal

x hexadecimal

u decimal (unsigned)

t binary

i instruction

c character

s string

Example: first parameter of `printf`

```
(gdb) x/i main+22
… lea rdi,[rip+0xad]  # 0x555555554754

(gdb) x/o 0x555555554754
0x555555554754:   010062045

(gdb) x/x 0x555555554754
0x555555554754:   0x00206425

(gdb) x/u 0x555555554754
0x555555554754:   2122789

(gdb) x/t 0x555555554754
...00000000001000000110010000100101
```

```
(gdb) x/i 0x555555554754
   … and    eax,0x1002064

(gdb) x/c 0x555555554754
0x555555554754:   37 '%'

(gdb) x/s 0x555555554754
0x555555554754:   "%d "
```

little-endian byte order
(least significant first) string is sequential in memory but integers are represented in little-endian

Example: number of elements and size

```
(gdb) x/4c 0x555555554754
0x555555554754:  37 '%'   100 'd'  32 ' '   0 '\000'

(gdb) x/4i $rip
=> 0x555555554692 <main+8>:    mov    DWORD PTR [rbp-0x4],0x0
   0x555555554699 <main+15>:   jmp    0x5555555546b5 <main+43>
   0x55555555469b <main+17>:   mov    eax,DWORD PTR [rbp-0x4]
   0x55555555469e <main+20>:   mov    esi,eax

(gdb) x/4xb 0x555555554754
0x555555554754:  0x25 0x64 0x20 0x00
```

prints 4 bytes hexadecimal

```
(gdb) x/4x 0x555555554754
0x555555554754:  0x25 0x64 0x20 0x00
```

remembers the last size used!

Step-by-step execution

```
(gdb) nexti
0x0000555555554699    6        for (i=0; i<10; i++)
(gdb)
0x00005555555546b5    6        for (i=0; i<10; i++)
...
(gdb) x/5i $rip
=> 0x55555555469b <main+17>:    mov    eax,DWORD PTR [rbp-0x4]
   0x55555555469e <main+20>:    mov    esi,eax
   0x5555555546a0 <main+22>:    lea    rdi,[rip+0xad]        # 0x555555554754
   0x5555555546a7 <main+29>:    mov    eax,0x0
   0x5555555546ac <main+34>:    call   0x555555554560 <printf@plt>
(gdb) nexti 4
0x00005555555546ac    7            printf("%d ",i);
(gdb) info registers esi rdi
esi            0x0     0
rdi            0x555555554754    93824992233300
(gdb) x/s $rdi
0x555555554754:  "%d "
```

repeats last command

executes next 4 commands

printf parameters in rsi and rdi

Continuing to next breakpoint

```
(gdb) c
Continuing.
Breakpoint 2, main () at count.c:7
7               printf("%d ",i);
(gdb) x/x $rbp-0x4
0x7fffffffe5cc: 0x01
(gdb) c
Continuing.
Breakpoint 2, main () at count.c:7
7               printf("%d ",i);
(gdb) x/x $rbp-0x4
0x7fffffffe5cc: 0x02

(gdb) delete 2
(gdb) c
Continuing.
0 1 2 3 4 5 6 7 8 9
[Inferior 1 (process 61) exited normally]
(gdb)
```

continues execution

inspects variable i on the stack

inspects variable i on the stack

deletes second breakpoint

Useful links

gdb home page : with complete documentation

gdb cheat sheet : a summary of most frequently used commands

PEDA : Python Exploit Development Assistance for GDB. Available in `/home/rookie/GDB/peda`.

```
(gdb) source /home/rookie/GDB/peda/peda.py
gdb-peda$
```

Exercise

Find the **password** of `/home/rookie/GDB/checkPasswordEasy`

Hint 1: Find the point where the program rejects the wrong password

Hint 2: You can script gdb. For example, this script executes the program step-by-step showing the executed instruction:

```
(gdb) define mystep
Type commands for definition of "mystep". End with a line saying just "end".
>while(1)
 >nexti
 >x/1i $rip
 >end
>end
```