LINGI2144: Secured System Engineering
Complement on GDB

Academic year : 2020 - 2021

Teacher : Legay Axel
Course : LINGI2144
Collaborators :
Crochet Christophe

Contents

1 Introduction

You can use whatever VM you want, just download the file and compile with:

- The `-g` indicates that the compiled files contains all information for debugging with `gdb`
- The `-fno-stack-protector` indicates that no stack protection are present
- The `-z execstack` forces gcc to compile code with non-executable instruction on the stack

We also disable randomization of the memory in the admin session:

```
sudo cat /proc/sys/kernel/randomize_va_space
0
```

2 Exercise

This tutorial aims at help you to familiarize more with the `gdb` tools which is very important to master. It is based on the excellent book "Hacking: The Art Of exploitation, 2nd" by Jon Erickson.

2.1 Complement on register

Some notion on the register such that you understand more when and why they are used in the case of x86 architecture.

The first four registers are known as general purpose registers:

1. `EAX`: Accumulator
2. `ECX`: Counter
3. `EDX`: Data
4. `EBX`: Base

They are used for a variety of purposes, but they mainly act as temporary variables for the CPU when it is executing machine instructions.

Then we have the pointers and indexes registers:

1. `ESP`: Stack Pointer
2. `EBP`: Base Pointer
3. `ESI`: Source Index
4. `EDI`: Destination Index

The first two registers are called pointers because they store 32-bit addresses, which essentially point to that location in memory. The last two registers are also technically pointers, which are commonly used to point to the source and destination when data needs to be read from or written to. There are load and store instructions that use these registers, but for the most part, these registers can be thought of as just simple general-purpose registers.

Finally the `EIP` register is the Instruction Pointer register, which points to the current instruction the processor is reading. Like a child pointing his finger at each word as he reads, the processor reads each instruction using the `EIP` register as its finger.

2.2 Complement on gdb

First, you have probably remark that the basic version of gdb use the ATX representation of the x86 instructions. Since the Intel one is the most used now, you probably want to use this version. To do so use the command:

```
set disassembly intel
```

```
(gdb) disass main
Dump of assembler code for function main:
   0x00401199 <+0>:     lea    0x4(%esp),%ecx
   0x0040119d <+4>:     and    $0xfffffff0,%esp
   0x004011a0 <+7>:     pushl  -0x4(%ecx)
   0x004011a3 <+10>:    push   %ebp
   0x004011a4 <+11>:    mov    %esp,%ebp
   0x004011a6 <+13>:    push   %ebx
   0x004011a7 <+14>:    push   %ecx
   0x004011a8 <+15>:    sub    $0x10,%esp
   0x004011ab <+18>:    call   0x4010a0 <__x86.get_pc_thunk.bx>
   0x004011b0 <+23>:    add    $0x2e50,%ebx
   0x004011b6 <+29>:    movl   $0x0,-0xc(%ebp)
   0x004011bd <+36>:    jmp    0x4011d5 <main+60>
   0x004011bf <+38>:    sub    $0xc,%esp
   0x004011c2 <+41>:    lea    -0x1ff8(%ebx),%eax
   0x004011c8 <+47>:    push   %eax
   0x004011c9 <+48>:    call   0x401030 <puts@plt>
   0x004011ce <+53>:    add    $0x10,%esp
   0x004011d1 <+56>:    addl   $0x1,-0xc(%ebp)
   0x004011d5 <+60>:    cmpl   $0x9,-0xc(%ebp)
   0x004011d9 <+64>:    jle    0x4011bf <main+38>
   0x004011db <+66>:    mov    $0x0,%eax
   0x004011e0 <+71>:    lea    -0x8(%ebp),%esp
   0x004011e3 <+74>:    pop    %ecx
   0x004011e4 <+75>:    pop    %ebx
   0x004011e5 <+76>:    pop    %ebp
   0x004011e6 <+77>:    lea    -0x4(%ecx),%esp
   0x004011e9 <+80>:    ret
End of assembler dump.
(gdb) set disassembly intel
(gdb) disass main
Dump of assembler code for function main:
   0x00401199 <+0>:     lea    ecx,[esp+0x4]
   0x0040119d <+4>:     and    esp,0xfffffff0
   0x004011a0 <+7>:     push   DWORD PTR [ecx-0x4]
   0x004011a3 <+10>:    push   ebp
   0x004011a4 <+11>:    mov    ebp,esp
   0x004011a6 <+13>:    push   ebx
   0x004011a7 <+14>:    push   ecx
   0x004011a8 <+15>:    sub    esp,0x10
   0x004011ab <+18>:    call   0x4010a0 <__x86.get_pc_thunk.bx>
   0x004011b0 <+23>:    add    ebx,0x2e50
   0x004011b6 <+29>:    mov    DWORD PTR [ebp-0xc],0x0
   0x004011bd <+36>:    jmp    0x4011d5 <main+60>
   0x004011bf <+38>:    sub    esp,0xc
```

```
   0x004011c2 <+41>:    lea    eax,[ebx-0x1ff8]
   0x004011c8 <+47>:    push   eax
   0x004011c9 <+48>:    call   0x401030 <puts@plt>
   0x004011ce <+53>:    add    esp,0x10
   0x004011d1 <+56>:    add    DWORD PTR [ebp-0xc],0x1
   0x004011d5 <+60>:    cmp    DWORD PTR [ebp-0xc],0x9
   0x004011d9 <+64>:    jle    0x4011bf <main+38>
   0x004011db <+66>:    mov    eax,0x0
   0x004011e0 <+71>:    lea    esp,[ebp-0x8]
   0x004011e3 <+74>:    pop    ecx
   0x004011e4 <+75>:    pop    ebx
   0x004011e5 <+76>:    pop    ebp
   0x004011e6 <+77>:    lea    esp,[ecx-0x4]
   0x004011e9 <+80>:    ret
End of assembler dump.
```

As note, `DWORD PTR [ecx-0x4]` simply mean that use the value located at `ECX - 0x4`.

We can have information about the registers of a running program with the command:

`info registers`

```
(gdb) b *0x004011c9
Breakpoint 1 at 0x4011c9: file program.c, line 7.
(gdb) r
Starting program: /home/user/SecurityClass/GDB-complement/program

Breakpoint 1, 0x004011c9 in main () at program.c:7
7           puts("Hello, world!\n"); // put the string to the output.
(gdb) i registers
eax            0x402008            4202504
ecx            0xbffff320          -1073745120
edx            0xbffff344          -1073745084
ebx            0x404000            4210688
esp            0xbffff2e0          0xbffff2e0
ebp            0xbffff308          0xbffff308
esi            0xb7fb8000          -1208254464
edi            0xb7fb8000          -1208254464
eip            0x4011c9            0x4011c9 <main+48>
eflags         0x296               [ PF AF SF IF ]
cs             0x73                115
ss             0x7b                123
ds             0x7b                123
es             0x7b                123
fs             0x0                 0
gs             0x33                51
```

You can also display information in many ways, lets check that with:

- o in octal

```
(gdb) x/o 0x4011c9
0x4011c9 <main+48>:     037777461350
```

- h in hexadecimal

```
(gdb) x/h 0x4011c9
0x4011c9 <main+48>:     0x62e8
```

- u in unsigned

```
(gdb) x/u $eip
0x4011c9 <main+48>:     232
```

- t in binary

```
(gdb) x/t $eip
0x4011c9 <main+48>:     11101000
```

- A number can also be prepended to the format of the examine command to examine multiple units at the target address

```
(gdb) x/h $eip
0x4011c9 <main+48>:     0x62e8
(gdb) x/2h $eip
0x4011c9 <main+48>:     0x62e8 0xfffe
```

You might also need to know that the default size for a *word* in x86/32 bits architecture is 4 bytes. We can also tell to gdb the default size of words that we want to display. Don't forget the little-endian convention:

- b single byte

```
(gdb) x/2bx $eip
0x4011c9 <main+48>:     0xe8    0x62
```

- h a halfword of 2 bytes

```
(gdb) x/2hx $eip
0x4011c9 <main+48>:     0x62e8  0xfffe
```

- w a word of 4 bytes, DWORD even if meaning "double word" refer to a 4 bytes value.

```
gdb) x/2wx $eip
0x4011c9 <main+48>:     0xfffe62e8      0x10c483ff
```

- g a giant word of 8 bytes

```
(gdb) x/2gx $eip
0x4011c9 <main+48>:     0x10c483fffffe62e8      0x09f47d8301f44583
```

You notice that gdb take care of the order alone and display them in the correct order.

Another interesting thing to know is that you can also print "instruction" directly with the `i` option. This can be useful for `EIP` for example:

```
(gdb) x/i $eip
=> 0x4011c9 <main+48>:  call   0x401030 <puts@plt>
(gdb) x/10i $eip
=> 0x4011c9 <main+48>:  call   0x401030 <puts@plt>
   0x4011ce <main+53>:  add    esp,0x10
   0x4011d1 <main+56>:  add    DWORD PTR [ebp-0xc],0x1
   0x4011d5 <main+60>:  cmp    DWORD PTR [ebp-0xc],0x9
   0x4011d9 <main+64>:  jle    0x4011bf <main+38>
   0x4011db <main+66>:  mov    eax,0x0
   0x4011e0 <main+71>:  lea    esp,[ebp-0x8]
   0x4011e3 <main+74>:  pop    ecx
   0x4011e4 <main+75>:  pop    ebx
   0x4011e5 <main+76>:  pop    ebp
```

Let's move to the next instruction with `nexti`:

```
(gdb) b* 0x004011c8
(gdb) c
Breakpoint 2, 0x004011c8 in main () at program.c:7
7           printf("Hello, world!\n"); // put the string to the output.
(gdb) x/i $eip
=> 0x4011c8 <main+47>:  push   eax
(gdb) nexti
7           printf("Hello, world!\n"); // put the string to the output.
(gdb) x/i $eip
=> 0x4011c9 <main+48>:  call   0x401030 <puts@plt>
```

As a last things, sometimes you can find some very interesting information in the code. For example here take the instruction:

```
0x004011c2 <+41>:    lea    eax,[ebx-0x1ff8] #load content of [ebx-0x1ff8] in eax
```

We know that the result of a `printf` is store in `EAX`. So by inspecting `ebx-0x1ff8` we should be able to see what is print.

```
(gdb) x/2xw $ebx - 0x1ff8
0x402008:       0x6c6c6548      0x77202c6f
(gdb) x/6xb $ebx - 0x1ff8
0x402008:       0x48    0x65    0x6c    0x6c    0x6f    0x2c
```

```
(gdb) x/6ub $ebx - 0x1ff8
0x402008:       72      101     108     108     111     44
```

Does these value means something for you ? Try to think about ASCII encoding. Let try with `gdb`:

```
(gdb) x/s $ebx - 0x1ff8
0x402008:       "Hello, world!"
```

As you has seen, `gdb` is very versatile and we can do lot of things with this tools. Hoping that you will try by yourself to play with.