LiveOverflow BinExp Notes

0x04

8-32 fixed sized global variables called registers, depending on architecture

- 32bit architecture = 32 bits of information, 2,147,483,647 biggest number
- 64bit arch = 64 bits of information, 18446744073709551615 biggest number

Special registers

- Program counter / PC / IP / EIP / Instruction pointer - tells us which instructions to execute next, every time an instruction is ran, it gets incremented.
  - If the value of the IP is 5, we would say the register "points to line 5"
- Stack Pointer / ESP - points to the top of the stack (a region of memory). The stack is at the top of RAM and as it is pushed to, **it grows downwards** in addresses

Most operations on registers, like subtraction and addition, have the side effect of altering status flags. The status flags live in a special register. Eg if the operation is `sub eax, ebx`, both registers having the value 8. then the `zeroflag` will be set to 1, since the output is 0.

If you run out of registers, you use memory, ram.

- Memory is accessed either with loads and stores at addresses, as if it were a big array. Or through PUSH and POP operations on a stack
- loads and stores are when (in 32 bit assembly) you `mov <register>, [address]`, eg `mov eax, [14]`

Control flow is done via GOTOs — jumps, branches or calls. These change the program counter directly

- A jump is an uncondtitional GOTO. Eg `jmp 5` is the same as `mov eip, 5`
- a jump if equal to zero, `je`, checks the zero flag
- a call is an unconditional GOTO that pushes the next address on the stack, and when the function you called is finished, it will hit a `RET` statement, which will pop the address value off the stack into the PC, which continues the program.

If you optimise your code, and only use cache and registers, it will be signifinantly faster than using ram or non volatile memory

0x05

GNU debugger, **GDB**

- `disassemble main` will show you the assmebly of main
- `set disassembly-flavor intel` will pretty the assembly
- `break *main` will set a breakpoint at main
- `run` will start the program
- `info registers` will tell you the values of registers
- `si` will skip to the next instruction
- `ni` will skip to the next instruction **without** going into functions that are calle d
- `continue` will make the program continue untill it hits the next break point or ends
- `set $<register>=<value>`, eg `set $eax=0` will set the value of a register

EAX refers to the first 32 bits of the 64 bit register, RAX

0x06

Looking for strings in the binary can reveal file types, libraries used, predefined strings.

`objdump -d file` will dump all of the assembly in a binary

`objdump -x file` will dump all of the headers in a binary

- The `.text` section will hold the code, and will tell where in memory it is, and how long it is
- The `.rodata` is the read only data, this is where predefined strings will be

The `strace` tool will trace system calls and signals, it tells you how a program interacts with the kernal

The `ltrace` tool will trace library functions, useful for when it calls basic c libraries

The `radare2` tool is another debugger

- `aaa` auto analyse
- `afl` print all functions used
- `s sym.main` seek (s) the main function, navigate to it
- `pdf` to print the assembly
- `VV` enter visual mode with a control graph
- `r2- d file` to debug a file
- `db address` to make a break point

0x07

One way of making a binary harder to crack is to remove the key string

- This can be done by encrypting the string
- This can be done by scrambling the string

All of these methods can be bypassed by opening the file in a debugger and working out the "win condition"

0x09

Syscalls are the interface between an application and then linux kernel

Apps use wrapper functions via glibc to syscall

Low level syscall functions such as read() and write() are located in predifined, *unchangable* level 0 spaces, when the computer is booted.

MMU, Memory management unit translates the address we see (virtual addresses) to an actual memory address (physical addresses).

- This is useful as when we debug a program we will always see the same addresses even if we open multiple programs.

0x0A

Each hexadecimal symbol refers to one nible

python converts:

- `bin(123) = 0b1111011`
- `hex(123) = 0x7b`
- `int('0b111102', 2) = 123`
- `hex(int('0b1111011',2)) = 0x7b`
- `hex(struct.unpacl("I", "AbCD")[0])`

Big endian and little endian determine the oritnetaiton of the data

A "word" on a 32 bit machine refers to all 32 bits

A "word" on a 64 bit machine refers to all 64 bits

A "double word" on a 32 bit machine refers to 64 bits

0x0B

if the file has `---s------` than this means the file has elevates its priviledge to the owner of the file, even if it is run by someone with less permission

GDB cannot debug a process whilst it has root permisisons, even if the setuid bit is enabled, it will force it to be ran with lower permissions

0x0C

`lea` moves the address into a variable. Eg. `lea eax, [esp+0x1c]` is like eax=esp+0×1c

Education dot Exploit Phoenix notes