x86 Software Reverse-Engineering, Cracking, and Counter-Measures

Lab: Shark Sim 3000

Environment Needed:

- Linux Virtual Machine (Recommended Ubuntu)
- Nasm (`sudo apt-get install nasm`)

Links for any tools are available in the github under 'Tools'

Overview

- For your project, you've spent thousands of hours crafting the most terrifying, realistic shark simulation the world has ever seen – all in the x86 assembly program on the right! There's just one problem – the project is due in an hour, but the program is seg faulting!!! You'll need to debug the code to save your reputation and unleash the greatest game of all time – Shark Sim 3000!
- For this lab, follow the given instructions *exactly*.

Caution

- This lab will have you enter gdb commands to watch the program as it executes. There is no "undo" for many of these commands. If you make a mistake, you will likely have to start over.
- It is critical that all steps are followed, in order. Do not skip any steps; do not rush; do not make mistakes or typos. Read each instruction, and follow it carefully.

```
; shark sim 3000

USE32
section .data
s: times 20 db 7eh
   db 5eh, 7eh, 0dh
.L equ $-s

section .text
global _start
_start:
    mov esi, s.L-1
.p:
    mov ebx, 1
    mov ecx, s+s.L
    sub ecx, esi
    mov edx, esi
    mov eax, 4
    int 80

    mov ecx, [0xfffffff]
    loop $

    dec esi
    jnz .p

    mov ebx, 0
    mov eax, 1
    int 80
```

X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
ISBN: **1394199880**

x86 Software Reverse-Engineering, Cracking, and Counter-Measures

Reference

- Remember, GDB is the GNU Debugger; it is a powerful debugger that can be used for analyzing both C and x86.
- The following GDB commands are used in this lab:
    - breakpoint (b) – set a breakpoint
    - run – run the program
    - continue (c) – continue executing a program after it paused at a breakpoint
    - info reg – examine the processor registers
    - info file – examine file details
    - x – examine memory
    - quit (q) – quit GDB
- If at any time you cannot remember a command, or do not understand a command's syntax, you can use
    - help – list overview for all commands
    - help <command> – get help on specific command
- In GDB, you can usually use a shorthand form of a command, if it does not conflict with any other commands. For example, "b" is short for "breakpoint"; "disas" can be used instead of "disassemble".

Building

- Locate the file called "shark_sim_3000.asm" in your directory.
    - The "asm" extension is used to denote *assembly* source code files.
    - It is important the code is *exactly* as it appears in this document.
        - Seriously. *Exactly*. Don't modify it. Be careful.
- Spend a few minutes looking over this code. Try to understand the basics of what the code is doing.
    - eax, ebx, ecx, edx, and esi are *registers* – they hold 32 bit values, stored and manipulated by electronic circuitry built directly into the processor.
    - "mov", "sub" and "dec" manipulate the registers.
    - "jnz", "loop", and "int" jump to different locations.
- *Assemble* this code into an object file with the following command:
    - nasm –f elf shark_sim_3000.asm
    - This uses the nasm assembler to translate the assembly code into machine code, with additional information that can later be used to create an executable file.

X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
ISBN: **1394199880**

    - The output is in the “elf” format, the standard object format on UNIX and Linux.
    - This will create a file called “shark_sim_3000.o” in your directory.
- *Link* the object into an executable file, with the following command:
    - ld –o shark_sim_3000 –melf_i386 –s shark_sim_3000.o
    - This creates an executable file called “shark_sim_3000” in your directory.
- Examine the raw assembly code from the executable to see what it will do:
    - objdump –Mintel –d shark_sim_3000
    - This will *disassemble* the file and show you its contents
    - You could do this with any executable, from any language, on any architecture, to see exactly what that program will do when it is run.
- *Run* the executable:
    - ./shark_sim_3000
- The execution seg faults! We need to fix the code, so we will use GDB to analyze the program as it runs.

Debugging

- Load the executable into GDB:
    - gdb shark_sim_3000
- Type the following command to change to Intel x86 syntax:
    - (gdb) set disassembly-flavor intel
- Type the following command to automatically display the next line of assembly:
    - (gdb) set disassemble-next-line on
- The code for your program sits in memory at a certain address. Find out what address your code starts at:
    - (gdb) info file
    - Look for “Entry point:”, followed by a hexadecimal address, like “0x8048080”
- Set a *breakpoint* on the program entry point, using the address you observed in the previous step:
    - (gdb) break *0x8048080
    - This will pause the program when the program begins.
- Start running the program:
    - (gdb) run
- Due to the breakpoint, the program will pause immediately when it starts to run.

X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
ISBN: **1394199880**

- Now we want to know what instruction the processor is about to execute. The “eip” register (the extended *instruction pointer*) holds an address of (“points to”) the machine code for the upcoming instruction. Determine the contents of the eip register:
  - (gdb) info reg eip
  - GDB will report the value in the eip register
    - eip 0x8048080 0x8048080
- Let’s examine the memory pointed to by eip. The “x” command lets us examine memory. Learn a little more about “x”:
  - (gdb) help x
- Use x to examine the machine code the processor is about to run:
  - (gdb) x/8xb $eip
  - This will show you the actual stream of bytes that will flow through the processor, to cause it to execute a machine instruction.
- As humans, we’re not much good at reading machine code. Let’s look at the same thing as assembly instructions instead:
  - (gdb) x/20i $eip
  - You should recognize the code from the lab. The first lines initialize and manipulate some registers.
  - The first line should look like
    - => 0x08048080: mov esi, 0x16
    - The arrow indicates that address 0x08048080 is where the instruction pointer is pointing; this is the *next line* of assembly that the processor will execute.
- Let exactly one assembly instruction execute:
  - (gdb) stepi
  - GDB should print:
    - => 0x08048085: bb 01 00 00 00 mov ebx, 0x1
    - The previous instruction (mov esi, 0x16) has executed.
    - The printed instruction is the *next instruction* that will execute.
- Press enter to run this instruction.
  - (gdb) <enter>
    - => 0x0804808a: b9 cb 90 04 08 mov ecx,0x80490cb

X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
ISBN: **1394199880**

Error 1

- Continue pressing enter; this will rerun the last gdb command (in this case, stepi); this is called *single stepping* the program. Stop at address 0x08048098.
    - => 0x08048080: be 16 00 00 00 mov esi,0x16
    - => 0x08048085: bb 01 00 00 00 mov ebx,0x1
    - => 0x0804808a: b9 cb 90 04 08 mov ecx,0x80490cb
    - => 0x0804808f: 29 f1 sub ecx,esi
    - => 0x08048091: 89 f2 mov edx,esi
    - => 0x08048093: b8 04 00 00 00 mov eax,0x4
    - => 0x08048098: cd 50 int 0x50
- The “int” instruction will call into the *operating system* to do something for us. In this case, we are asking the OS to print the string pointed to by the ecx register. Examine the memory pointed to by the ecx register, as a string:
    - (gdb) x/s $ecx
- Allow the “int” instruction to execute:
    - (gdb) si
- The instruction will cause a segmentation fault, so the operating system will terminate your program.
- Check the problematic instruction in the original source code:
    - int 80
- Check the problematic instruction from gdb:
    - int 0x50
- Although these look different, they are the same. 80 is in decimal, and 0x50 is in hexadecimal – we are looking at the same number in two different bases. Skilled programmers know that “int eighty” calls into the operating system on Linux, but when they say “eighty”, they mean “hexadecimal eighty”. By writing “int 80” in our source code, we used the wrong base. This might be the cause of our segfault!
- Exit gdb:
    - (gdb) quit
- Fix the “int 80” lines in the assembly code, by changing them to “int 0x80”. There are *two* lines that must be fixed.
- Reassemble and relink your program.
- Run the program:
    - ./shark_sim_3000

X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
ISBN: **1394199880**

- We *still* have a seg fault.

Error 2

- Reload your program into gdb, set gdb to use Intel syntax, and turn on next line disassembly.
    - If you add these commands to your ~/.gdbinit file, you will not have to retype them every time you launch gdb.
- We want to pick up where we left off, so set a breakpoint on the last instruction we reached:
    - (gdb) b *0x08048098
- Run the program.
- It will pause when it hits the breakpoint. Notice that the instruction has changed since last time:
    - => 0x08048098: cd 80 int 0x80
- Clear the breakpoint:
    - (gdb) delete 1
- Step one instruction.
- The “int 0x80” instruction successfully runs.
- Continue execution (allow the program to run from the current location) to determine where the next seg fault is:
    - (gdb) continue
    - => 0x0804809a: 8b 0d ff ff ff 0f mov ecx,DWORD PTR ds:0xffffffff
- This instruction copies data from the memory at address 0xffffffff into register ecx.
- Check the data at address 0xffffffff to determine what is being copied into ecx:
    - (gdb) x *0xffffffff
- gdb will tell you that it cannot access this memory. That’s exactly what causes a seg fault! Trying to access memory that does not belong to your code is a security violation, and usually a programming error – that’s why the operating system terminates your program when it catches your mistake.
- The programmer didn’t mean to use the *memory* at 0xffffffff, they meant to use the *value* 0xffffffff. In C, this is the difference between “*ptr” and “ptr” – “ptr” is a value, “*ptr” is the memory at that value. In assembly, we write “[x]” to access the memory at address x, and “x” to access the value x.
- Exit gdb:

X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
ISBN: **1394199880**

    - (gdb) q
- Fix the code by changing the seg faulting line so that it moves the value 0xffffffff into the ecx register, instead of moving the memory at address 0xffffffff into the ecx register.

Conclusion

- Reassemble and relink your program.
- Run the finished program to experience the revolutionary ultra-realistic shark sim:
    - ./shark_sim_3000
- Now that it's working, shark_sim_3000 is destined to be a smashing commercial success. But to protect your intellectual property (and the untold millions you are bound to make off of this), you should refresh yourself on the program internals, one last time.
- **Use the techniques from this lab to determine the following information, it will help in your mastery of x86.**
- Single step through the "loop $" instruction, and use the "info reg" command to evaluate its execution. What does the "loop $" instruction do? What do you think its purpose is in this situation?
- What is the value of esi the first time the "jnz .p" instruction is executed?
- What is the value of esi the second time the "jnz .p" instruction is executed?
- What is the value of esi immediately before the final int 0x80 instruction is executed?
- What do you think esi is being used for in this program?
- Learning to quickly make *inferences* like this regarding the purposes of different instructions and registers is the heart of reverse engineering and software cracking!

Source: iusedtobescaredofcats.blogspot.com

X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
ISBN: **1394199880**