← Project 3 details

Reference Material

An **ASCII table** can be handy for knowing what hex values mean what characters.

Manpages are your friend. `man some_function` will be very helpful. **Set up your manpages if you haven't already.**

Really look at **the slides on calling conventions**. *Most* programs are mostly function calls, and understanding the function calling mechanism can make it much easier to identify what code is doing. Remember:

- Function calls look like one or more `mov [esp+...], ...` followed by a `call`.
- The function prologues and epilogues can usually be ignored.
- Function parameters are usually accessed as `[ebp+...]`.
- Local variables may be accessed as either `[ebp-...]` or `[esp+...]`.

Reverse-engineering basics

When reverse-engineering anything, you don't jump into the middle and try to figure out where you are. You **start with the information that you *do* know.**

Your programs are these mysterious black boxes, but **all** programs that run on UNIX follow certain conventions. Basically, you have this:

```
       call to main
            |
            v
-------------------------
|     ???????????       | program to debug
-------------------------
  |         |          |
  v         v          v
  calls to the C stdlib
```

You have two ways to attack the unknown code: you can **start from the top** with `main` and you can **build up from the bottom** by looking at what

standard library functions it calls. Some programs are easier to start from one side or the other.

Static analysis is when you look at and analyze the executable file without running the program. You can learn a lot about a program this way.

Dynamic analysis is when you *run* the program and inspect it while it runs, to figure out what it does. `gdb` is perfect for this.

This project can be done entirely with one method or the other, but it's often useful to try both out. Run the program, inspect some stuff, see what it does when you enter wrong things; that will give you insight into what to look for in the executable file.

Finding information

The more you know, the easier it will be. Here are some helpful tools.

- `mystrings` might be helpful.
- `nm` shows the symbol table. Remember that it shows *imported* (undefined) symbols as well…
- `hexdump -C somefile | less` will show a hexadecimal/ASCII view of a file. You can scroll up and down with arrow keys/page up/down. `q` to quit.
- `objdump` is another really useful tool.
  - `objdump -x somefile` gets all the information available.
  - `objdump -d -Mintel somefile` lets you get a disassembly without the debugger.
    - This might be easier than the debugger in some cases!
- The `man` pages and search engines are useful for looking up what standard library functions do.
  - Whenever you come across a function, **google it.** Don't waste your time trying to reverse-engineer `printf`!

In addition: use your brain! Think about how *you* would write a program like this. How would *you* write a program to ask the user for a line of text, do something to it, and then make a decision? These programs were written by humans in C too.

Finding the stdlib function calls

If you're looking at a bunch of unreadable code, but you see this:

```
call 0x8040040 <printf>
```

Heyyyy, you know what `printf` is!

Since you know (or can look up) what the parameters to stdlib functions are, you can work *backwards* to figure out what stack locations or registers hold which values. If you see:

```
mov  [esp+4], ebx
mov  [esp],   0x806c004
call 0x8040040 <printf>
```

What do you think it's doing? Remember the parameter passing order in cdecl? Which `printf` argument does `0x806c004` represent? Which argument does `ebx` represent? Inspect, print out, work backwards. (See the gdb section below for tips on inspecting things.)

---

x86 tips

`repz cmps`

`repz cmps` is a weird-ass complicated instruction. `cmp` implies it has something to do with comparison. What do you think the `s` means? The `repz` is an *instruction prefix* which modifies the operation of the `cmps` instruction. Be sure to look up what both of these instructions mean and also google "repz cmps" or "repz cmpsb".

`DWORD PTR`

When you see `DWORD PTR`, `WORD PTR`, `BYTE PTR` in `mov`s that access memory, this specifies the data size to load/store. `DWORD` = 32-bit, `WORD` = 16-bit, `BYTE` = 8-bit. It's just like how MIPS has `lw/sw` for 32-bit, `lh/sh` for 16-bit, and `lb/sb` for 8-bit.

`lea`

The `lea` instruction *looks* like it's loading memory, but don't let the `[brackets]` fool you. It doesn't access memory. It only **calculates** an address and puts it in the destination.

If you see:

`lea eax, [ebp-0x20]`

that means, "put the address of the stack variable at address `ebp-0x20` into `eax`." In this case `lea` is acting like C's address-of operator (like `&x`).

To complicate things further, the compiler likes to use `lea` to do **addition and multiplication** as well. If you see something like...

```
lea eax, [ebx+ecx*1]
```

it's probably using it to do addition. Think of this like:

```
add eax, ebx, ecx
```

but x86 doesn't let you write `add` like this, so the compiler uses `lea` instead!

Conditional jumps (`je`, `jne`, `ja` etc.) and control structures

There's a whole family of conditional jump instructions in x86 which all start with `j`, such as `je`, `jne`, `jg`, `jlt`, `ja` etc. Look up what these conditions mean.

Conditional jumps make their decision based on the `eflags` register. Because of that, they're virtually always preceded immediately by an instruction which does some kind of comparison. You'll see `cmp` for integer comparison a lot. `test` is another instruction that's often used to see if a number is 0. Page 111 of Dr. Misurda's book explains this in more detail.

An important thing to keep in mind is that **this is compiled C code.** It's not hand-written assembly. When there are conditional jumps, they were generated from conditional control structures, like `if`, `else`, `while`, `for`, and `switch`. The conditional jumps will always follow certain patterns.

Don't rack your brain trying to figure out what a huge clump of conditional jumps do. **It's not spaghetti code.**

Instead, learn the patterns by writing your own little test programs with if-elses, loops, and switches. Compile them **with the -m32 and -g flags** and inspect their disassembly. Once you see what the compiler will produce for each kind of control structure, you'll start to notice the patterns and be able to think "oh, this is an if-else."

`gdb` tips

Pausing programs on the command line

While running `gdb`, you might want to go look up a man page or something. **You don't have to quit!**

Hit `Ctrl+Z` (mac users, use the `control` key, not `command`). This will pause the currently-running program and bring you back to the shell. It's kinda like minimizing it.

Once in the shell, you can use the `jobs` command to see a list of paused programs. You can pause multiple programs, and each one will have a number next to it, like:

`[1]+ Stopped gdb jfb42_3`

That `[1]` is the program's number.

Then to resume `gdb`, I can type `fg %1` since the program's number is `1`. `fg` brings the program back to the **f**ore**g**round. (There is also `bg` to run a program in the **b**ack**g**round while you continue to work in the shell, but that wouldn't be too useful for `gdb`...)

Showing assembly instructions as you step

Use the `set disassemble-next-line on` command in `gdb`. (You can put this in your `.gdbinit` file too, if you like.)

Now when you use the `si` or `ni` commands, you'll see the line of assembly code you're on:

```
(gdb) si
0x080482d5 in main ()
=> 0x080482d5 <main+6>:  57     push   edi
(gdb)
```

Repeat last command

If you hit enter without typing anything, `gdb` will just repeat the last command you typed. This works great with `si` and `ni` to just keep stepping through:

```
(gdb) si
0x080482d5 in main ()
```

```
=> 0x080482d5 <main+6>:  57     push   edi
(gdb)
0x080482d6 in main ()
=> 0x080482d6 <main+7>:  56     push   esi
(gdb)
0x080482d7 in main ()
=> 0x080482d7 <main+8>:  81 ec 88 00 00 00      sub    esp,0x8
(gdb)
0x080482dd in main ()
=> 0x080482dd <main+14>:         a1 1c 69 0d 08 mov    eax,ds:
(gdb)
```

Breaking on unnamed functions

If you see something like

`call 0x803320`

and there's no name after it, don't worry! You can still break at the start of that function by using an asterisk before the address:

`b *0x803320`

You'll also need to use this if your executable has no `main` function...

`disas` doesn't work!

It's OK! You just have to help it out a little.

To disassemble where you currently are, you can do:

```
disas $pc, +100
```

This will disassemble 100 bytes of code starting from the current program counter. You can use whatever number you like.. some functions are short, and some are long.

Using `disas` without breaking inside a function

If you see a call to a function that has no name, and you want to see what it does, you can look at it without setting a breakpoint. Just do:

```
disas 0x803320, +100
```

Similarly to before.

Inspecting the contents of registers

If you want to see what's in a register currently, use

```
print $eax
```

or any other register name. You have to use the $, it's weird.

Inspecting the contents of the stack

The stack is where local variables and function parameters live, so it's very useful to be able to see what values are in them. The `x` command lets you examine the contents of a memory address.

```
x $esp
```

will print out the 32-bit integer that is pointed to by `esp`. You can also add offsets:

```
x $esp+0xc
x $ebp-0x1d
```

This way you can look directly at what is in variables that are referenced by `mov` instructions.

Inspecting strings

If you have what you think is the address of a string, you can do this:

```
x/s 0x0800808
```

`x` is a really flexible command with a lot of options. `/s` is just one, and it treats the memory address you give it as a 0-terminated string.