note @143 stop following 146 views

Actions

BOMBLAB: Tips & stuff I've learned about

Hello all! Bomblab is my favorite assignment btw. Hope you all have enjoyed working on it as much as I've enjoyed helping. (haha).

Assignment goals

I see a few main goals for this assignment.
1- Get better at using GDB. This is going to be your main tool for completing the assignment (rather than trying to understand what the code is doing statically), and GDB will be super useful for future assignments. You'll find that you can be very specific in the information that you're requesting when using GDB.
2- Understanding programs in assembly. Practically, this is good practice with assembly commands, and interpreting them quickly will be on exams. Getting a feel for the flow of an assembly dump, how it interacts with registers and moves data around; these are a few aspects of gaining expertise in low-level programming/concepts, which is what this class is all about. (Yeah it will be cool to see how some concepts apply when you're doing C/C++ coding in the future!)
3- General problem solving skills. Really, these are puzzles!

Before you get going

It can be useful to have all the assembly in a text file you can open in your editor and easily search it or just read it separately from gdb.

To dump out the assembly, use the objdump program:

```
objdump -d bomb
```

This just prints the assembly to the screen, to capture it to a file use I/O redirection (as discussed in HelloLab).

```
objdump -d bomb > bomb.s
```

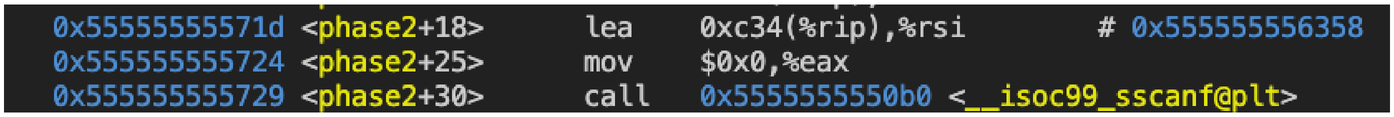
Now to the specifics of solving the bomblab.

Getting started with a phase & what if you're stuck

Let's go over some rule of thumb things. Here's how I'd recommend getting started with a phase.

Step 1: Run the program, set a breakpoint at <phase[x]> and <explode_bomb>

Step 2: See what your input format should be. Towards the top you'll see a call to this function: <__isoc99_sscanf@plt>
This function examines your input format and compares it with the expected input format. Example format: "%d %d %[^\n]" means integer (decimal aka base 10), integer, string.
How do we find the expected format string? Let's see what happens right before.



```
0x55555555571d <phase2+18>      lea     0xc34(%rip),%rsi        # 0x555555556358
0x555555555724 <phase2+25>      mov     $0x0,%eax
0x555555555729 <phase2+30>      call    0x5555555550b0 <__isoc99_sscanf@plt>
```

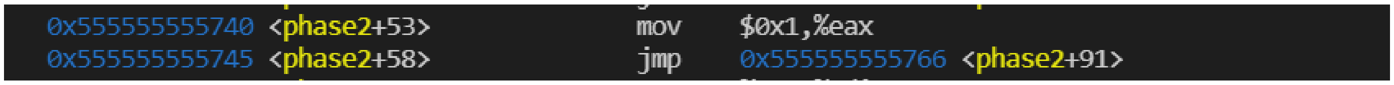
Any time we call a function, the first argument is the value stored in %rdi, the second argument is the value stored in %rsi, (...). The function's return value is always immediately stored in %rax.

We notice that something gets moved right into %rsi right before the function call, and we know from lecture that *%rsi holds the second argument* that gets passed into a function, any time we call a function. So, if we print the contents of %rsi right before we call the function, that will give us the expected input string. Note, that %rsi just has a pointer. To print the string it points at see below for printing memory contents. (What do we think is in %rdi right before the function call?)

Step 3: Once we know what data types we need to input, we get to the meat of the program. Now you employ your problem solving skills.

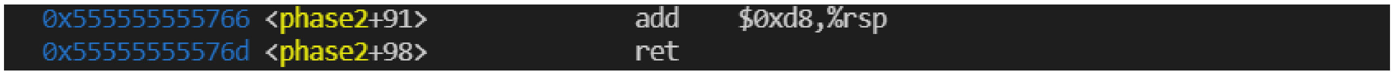
- What do we need for sure? As a reminder, if %eax holds 0x0 when you return, you've passed, and if it holds 0x1 when you've returned, you've failed. Any time we see "mov $0x1, %eax" followed by "ret" without any more modifications to %eax in between, you know you've failed. So, we know we need to avoid this sequence of instructions.
- What's happening to our inputs? Try to track your inputs and see how they're being modified. What registers do they end up in?
- It may actually be most time efficient to just go back to the assembly (apart from gdb) and try to convert it to a more C-like (high level language) format with if, while, for, etc. At the very least some key chunks of the assembly in a phase may be most easily understood through this kind of analysis rather than blindly stepping through the code.
- Set more breakpoints. To set a breakpoint at <phase2+18>, do "b *(phase2+18)". Do "c" (continue) to go from one breakpoint to the next. This will be useful in later phases where there's loops, and you want to get past a loop without doing "ni" a thousand times.
- If you're stuck, try writing it out step by step on a piece of paper. Even if we don't really know what's happening with these instructions, it's going to be easier to keep track of what's going on if it's on paper in your own words.
- If you're still stuck, try working backwards. Like bullet (2), any time we see "mov $0x0, %eax" followed by "ret" without any more modifications to %eax in between, you know you've passed. Try to figure out what you need to get to that point.
- If you're still stuck, just **try a new input.**

For example, if we get to this point:



```
0x555555555740 <phase2+53>          mov     $0x1,%eax
0x555555555745 <phase2+58>          jmp     0x555555555766 <phase2+91>
```

And <phase2+91> looks like this:



```
0x555555555766 <phase2+91>          add     $0xd8,%rsp
0x55555555576d <phase2+98>          ret
```

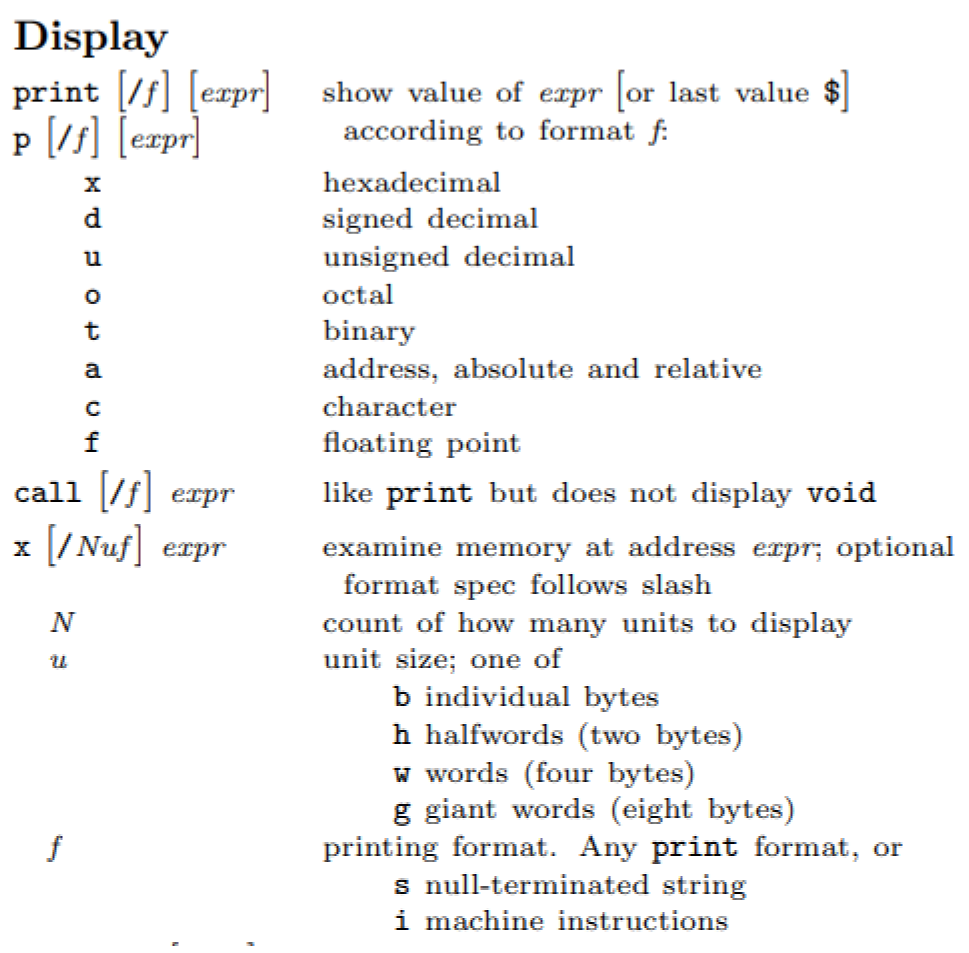
We know we've failed so we need to avoid running <phase2+53>. If we instead moved $0x0 into %eax on <phase2+53>, we would pass and we should figure out how to get there.

GDB Tips:

Whenever someone doesn't fully understand the GDB commands they're using, I always direct them to this page.

https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

Specifically, the **display** section:



Display

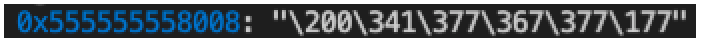
`print` [/*f*] [*expr*] `p` [/*f*] [*expr*]	show value of *expr* [or last value `$`] according to format *f*:
`x`	hexadecimal
`d`	signed decimal
`u`	unsigned decimal
`o`	octal
`t`	binary
`a`	address, absolute and relative
`c`	character
`f`	floating point
`call` [/*f*] *expr*	like `print` but does not display `void`
`x` [/*Nuf*] *expr*	examine memory at address *expr*; optional format spec follows slash
N	count of how many units to display
u	unit size; one of `b` individual bytes `h` halfwords (two bytes) `w` words (four bytes) `g` giant words (eight bytes)
f	printing format. Any `print` format, or `s` null-terminated string `i` machine instructions

Many of you probably don't quite understand the difference between "p/..." and "x/..."

Use "p/... *expr*" when you want to print the value of *expr*. Use "x/..." when you want to print the value at memory address *expr*. In other words, *expr* is a memory address, **for you guys the memory addresses start with 0x555555....**, and you want to print the data stored at that memory address. If you use "p/... *expr*" when *expr* is a memory address, that won't be very helpful. You'll just see 0x555555... .

We also see that we can be more specific with the format *f*. Here's a few examples of types of calls (not these specific ones) you may have seen so far:

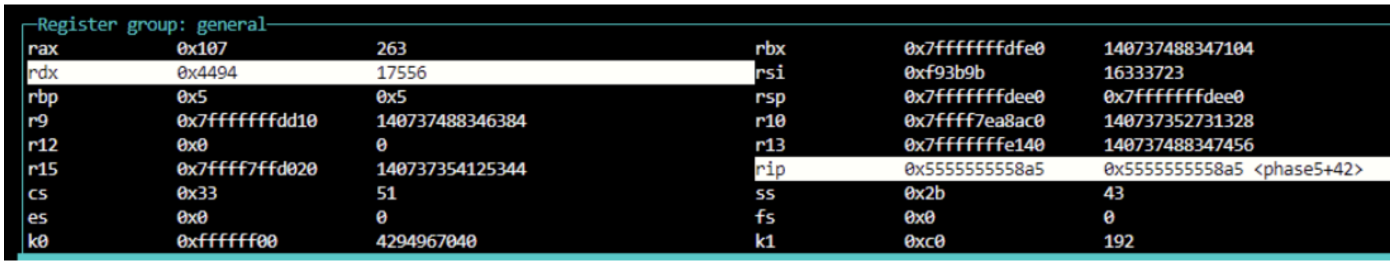
- p/x $rax -- print the integer in %rax in hexadecimal format (to follow the structure from the pic, the format *f* is *x*)
- p/d $0x12345 -- print the integer 0x12345 in decimal (base 10) format, aka converting 0x12345 from hex to decimal
- x/x $rcx -- print the integer stored at memory address in %rcx. Here, %rcx should contain 0x5555555..., otherwise it's going to give you an error.
- x/s $rcx + 4*$rax + 0xcc -- compute $rcx + 4*$rax + 0xcc, and return the string stored at that address
    - If you do "p/x $rcx" and get 0x55555..., $rcx contains a memory address. try doing "x/x $rcx"
    - If you do "x/x $rcx" and get something like "**<error: Cannot access memory at address 0x6b>**", that means $rcx contained 0x6b, which is not a valid memory address.
    - If you do "x/s $rcx" and get something like this:



```
0x555555558008: "\200\341\377\367\377\177"
```

then you you're interpreting the data wrong. Try interpreting it as an integer using "x/x..." or "x/d ..." instead.

layout regs will provide the values stored in all registers at the top of your screen, updating after each instruction:



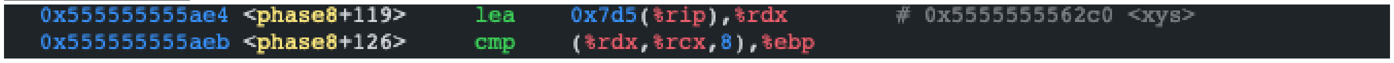
```
┌Register group: general─────────────────────────────────────────────────────────────────────
rax            0x107               263                    rbx            0x7fffffffdfe0      140737488347104
rdx            0x4494              17556                  rsi            0xf93b9b            16333723
rbp            0x5                 0x5                    rsp            0x7fffffffdee0      0x7fffffffdee0
r9             0x7fffffffdd10      140737488346384        r10            0x7ffff7ea8ac0      140737352731328
r12            0x0                 0                      r13            0x7fffffffe140      140737488347456
r15            0x7ffff7ffd020      140737354125344        rip            0x5555555558a5      0x5555555558a5 <phase5+42>
cs             0x33                51                     ss             0x2b                43
es             0x0                 0                      fs             0x0                 0
k0             0xffffff00          4294967040             k1             0xc0                192
```

One more gdb tip is included in the section below.

Addressing mode with offset (may help with later phases)

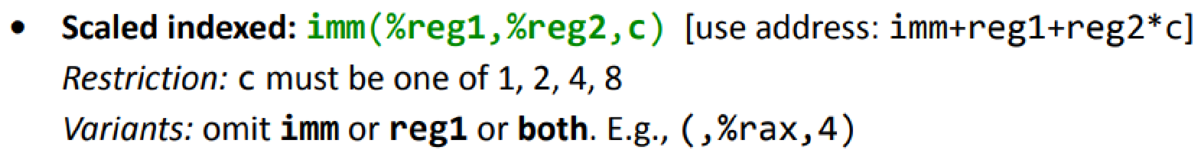
Occasionally you will come across a line that looks like this:

```
(%rdx,%rcx,8)
```



```
   0x555555555ae4 <phase8+119>      lea     0x7d5(%rip),%rdx        # 0x5555555562c0 <xys>
   0x555555555aeb <phase8+126>      cmp     (%rdx,%rcx,8),%ebp
```

This is one of your addressing modes. (Taken from Unit 5, slide 14):



- **Scaled indexed:** `imm(%reg1,%reg2,c)` [use address: `imm+reg1+reg2*c`]
  Restriction: c must be one of 1, 2, 4, 8
  Variants: omit **imm** or **reg1** or **both**. E.g., `(,%rax,4)`

In this case, "imm" is omitted. Notice how the resulting arithmetic is treated as a **memory address**. Cool, but what's actually going on here? In some cases, we can think of this as an **array**.

Well, we notice that a memory address is moved into %rdx immediately before this instruction, we can see the # 0x55555.... commented out, so that's what %rdx is going to be each time we run <phase8+126>, so the value of %rdx can be thought of as the **starting address of the array**. If we notice that %rcx contains small values (ie something like 0 to 20), %rcx can be thought of as the **index** of the array. In this case, the %rcx is being multiplied by 8, so we are reading the data at addresses %rdx + 0, %rdx + 8, %rdx + 16, etc. Since the offsets are 8 bytes, we can consider it to be an **array of longs** (8 byte integer data type).

If we look at this as an array, we likely need to ensure that we have the right index. How do we know the right index? Well, it might be helpful to look at the elements of the array, and see if any of those elements is the one we need. We can use GDB to do that!

Looking at the "**Display**" image from the previous section, we see we can also specify the number of units to display *N* and the unit size *u.*

Since we're dealing with 8 bytes at a time in this example, it may be useful to specify the size parameter so we know for sure that we're getting the relevant info. Suppose we have some specific index %rdx for example. If we want to print 8 bytes (in gdb terms, a "giant word"), in hex format, at the correct address, we can run

- x/gx $rdx + 8*%rdx -- reads 8 bytes starting at address $rdx + 8*%rdx

What if we get to line <phase8+126> and we think, hm... we're comparing it with %ebp, so let's see if there's any entries that match %edp. We can print many long elements of this "array" at once. If we want to print 20 longs, we can run

- x/20gx $rdx -- prints 20 longs starting at address $rdx

Maybe one of those longs is the one you need!

Note: Pay attention to the right hand side! We are reading in offsets of 8 bytes, but we are comparing the data with %ebp, a 4-byte register. In this case, we are comparing the least significant 4 bytes of the left hand side with the 4 bytes stored in %ebp.

Thank you to the staff and students for being awesome. Comments and/or critiques are encouraged, and I'd love to see you in my OHs!
Written by Anthony Winney.

bomblab

~ An instructor (Marco Paolieri) endorsed this note ~

Edit good note 9 Updated 1 month ago by Anthony Winney

followup discussions *for lingering questions and comments*

Start a new followup discussion

Compose a new followup discussion