note @296 stop following 143 views

Actions

ATTACKLAB: Tips & stuff I've learned about

Hello all, hope the assignment is going well. This one can be pretty tricky. Before I wrote this post, it was the one I understood the least. Hope this post can be of help

Intro

This assignment is cool because you are actually hacking a program. Here's the outline of the program we're attacking, and why it's "hacking."

```
void test();
void target_f1();

int main(){

        //stuff that gives you your cookie number

        //inside test we use getbuf() to read your input
        test();

        //stuff that prints the result

        return 0;

}

void test(){
  int val = getbuf();
  printf("No exploit.  Getbuf returned 0x%x\n", val);
  fail();
}

void target_f1() {
    level = 1;
    printf("SUCCESS: You called target_f1()\n");
    validate();
}
```

Looking at this C code, it **should be impossible to enter target_f1.** But, as we've discussed, the `getbuf()` function is vulnerable to code injection or ROP attacks, namely **if we input a string longer than the size of the buffer array.**

When we enter `getbuf()`, **the first line will tell us how large the buffer array is**. We will see a `sub` instruction acting on rsp, which holds a pointer to the stack. **This allocates space on the stack for your input string,** and once we subtract the buffer size **rsp points to the start of the your input.** Once `Gets` finishes copying your input string to the stack, we add the size of the buffer back to rsp, so **the stack pointer goes back to where it was.**

But, the question is... now what is the stack pointer pointing to? **The return address!** When we `call getbuf`, **the address of the next instruction in test gets pushed onto the stack.** This is just what the `call` command in x86 does, it acts as a `push` onto the stack followed by a `jump` to whichever function we're calling. Then, when we `return` from `getbuf`, we **go back to the address we just stored, which *should* be what rsp is pointing to**. For the sake of completeness, this is what the `ret` instruction in x86 does, it acts as a `pop` of some free register, followed by a jump to the address stored in that register (as a reminder, the `pop` instruction copies the data from the top of the stack, or the data to which rsp is pointing, and puts it in the specified register).

So here's where it all comes together. Since `getbuf` doesn't check whether you inputted more than what it allocated, **we can overwrite other data on the stack because the stack stores temporary variables, like the buffer array.** And like we just explained, after the stack goes back to where it was before we allocated space for the temporary variables, **the stack is pointing to the return address**, namely the address of the instruction immediately after the call to getbuf. So if we overwrite that address with a different address, **instructions will start being executed at the address we want, rather than where we left off in test().**

Ok woah. That was a lot. Very dense. But, I tried to make it as complete as possible, so for those of you who are curious, you can get a real good grasp of what's going on. If you're one of those curious people, it's going to be hard to fully understand from just reading words. I'd recommend following along in gdb and checking the registers to see where everything is going. Tip: you can use `x/20i [address]` to examine 20 instructions at the specified address. The big ones are `test` and `getbuf`, but I also looked at other functions. Don't be afraid to look at a lot of functions. You can always go deeper and try to understand more. If you want to take it another step further, try figuring out what's going on with rbp (and let me know because I couldn't).

Let's look at some pictures to help understand all this.

```
0x401709 <getbuf>       sub    $0x18,%rsp
0x40170d <getbuf+4>     mov    %rsp,%rdi
0x401710 <getbuf+7>     call   0x401777 <Gets>
0x401715 <getbuf+12>    mov    $0x1,%eax
0x40171a <getbuf+17>    add    $0x18,%rsp
0x40171e <getbuf+21>    ret
```

Here's my `getbuf` function. How big is the buffer array? How many bytes do I need in my attack string before I can start overwriting data on the stack?

```
B+  0x401709 <getbuf>       sub    $0x18,%rsp
    0x40170d <getbuf+4>     mov    %rsp,%rdi
    0x401710 <getbuf+7>     call   0x401777 <Gets>
    0x401715 <getbuf+12>    mov    $0x1,%eax
    0x40171a <getbuf+17>    add    $0x18,%rsp
  > 0x40171e <getbuf+21>    ret
    0x40171f <save_char>    mov    0x465f(%rip),%eax        # 0x405d84 <gets_cnt>
    0x401725 <save_char+6>  cmp    $0x3ff,%eax
    0x40172a <save_char+11> jg     0x401776 <save_char+87>
    0x40172c <save_char+13> mov    %edi,%ecx
multi-thre Thread 0x7ffff6dbd9 In: getbuf
(gdb) x/20x $rsp
0x54fea010:     0x24    0x19    0x40    0x00    0x00    0x00    0x00    0x00
0x54fea018:     0x09    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x54fea020:     0xe1    0x1b    0x40    0x00
```

I printed the contents of the stack when I was about to return from `getbuf`. What's the first thing on the stack (Hint: little-endian storage so read it backwards)? It may or may not be an address... BUT if it is an address what's at that address? Does that make sense with the previous explanation? Does that give any ideas for our attack string?

If you want to take it a step further, try to figure out what's going on with the second address being stored on the stack (ie 0x401be1).

```
0x401916 <test>             sub    $0x8,%rsp
0x40191a <test+4>           mov    $0x0,%eax
0x40191f <test+9>           call   0x401709 <getbuf>
0x401924 <test+14>          mov    %eax,%esi
0x401926 <test+16>          lea    0x17c3(%rip),%rdi
0x40192d <test+23>          mov    $0x0,%eax
0x401932 <test+28>          call   0x401030 <printf@plt>
0x401937 <test+33>          call   0x401af0 <fail>
0x40193c <test+38>          add    $0x8,%rsp
0x401940 <test+42>          ret
```

Here's my test function. Notice where we call `getbuf`. Notice anything familiar here? Anything that may or may not have something to do with the addresses surrounding the call to getbuf and/or the contents of the stack from the previous picture?

Code injection attacks: target_f2 and target_f3 inside ctarget

The intro gives all the information necessary to complete target_f1 inside ctarget (and target_f1 inside rtarget), but we didn't use code injection for that. The difference now is that `target_f2(unsigned val)` and `target_f3(char* sval)` take inputs. And they need to be the correct inputs, so **which register do we need to make sure has the correct value** when we call the functions?

The trick here is **writing our own assembly instructions, storing them on the stack, and overwriting the return address with the address of our written instructions.** Since you only have control of your input, and your input is

stored on the stack, look at rsp to find the address where you're writing your instructions. Your input should contain assembly instructions. But inputting "mov $1, %rdi" won't do what you want, that's assembly but the machine reads binary. So make sure you read the section titled **Generating Binary Instructions.** Also, make sure you fully understand what's going on on **Unit 8, Slide 14**.

Hint: When we subtract the buffer size from rsp, rsp is pointing to the start of your input string. You can verify this in gdb.

FAQ: Why do I have to fill the buffer with my machine code? Why can't I do like 50 lines of filler and then put my machine code, and just make sure I'm pointing there?
Answer: I just tried it and apparently you can. Go nuts. My only guess would be that if you go too crazy you might overwrite other addresses that get called after `target_f2` which are required for the program to terminate.

FAQ: What's going on with the `hexmatch` function inside `target_f3` ?
Answer: It randomizes the position of the check string. Meaning, the check string, or the string which you need to match, is being stored in a temporary variable, `char* s` , but if the address of `char* s` is the same every time, you could just set rdi to that address rather than the address of the string you inputted. Don't think many of you would've tried that. It did seem like a good idea though.

ROP attacks: target_f2 and target_f3 inside rtarget

`rtarget` uses two features to thwart code injection attacks: marking the area of the stack before the return address as nonexecutable (Unit 8 Slide 22) and ASLR (Unit 8 Slides 19), but NOT canary values. Canary values protect against both code injection and ROP -- this was a test question for me so see if you can understand why. So, we cannot do the same trick of injecting code onto the stack and hardcoding an address found via rsp to execute that code, because rsp will be different every time you run the program.

Instead, we have a "gadget farm." These are a bunch of functions that we could call **just like we can call target_f[x],** and if we call the right ones, we can achieve the same effect of passing the correct input into `target_f[x].` Of course, we'd need to still call `target_f[x]` after the gadgets. The assignment page and slides both have everything you need. Make sure you understand what's happening on Unit 8, Slides 23+ for ROP.

Tip 1: When I solved this, I didn't use `grep` to find the opcodes. Instead, I dumped the disassembled instructions into a text file using `objdump -d rtarget > rdump.txt,` then I deleted everything above `<start_farm>` and everything below `<end_farm>` so I could just use CTRL+F to search for opcodes in a txt file.

Tip 2: To put our cookie in a register, we need to use a `pop` instruction. Remember that `pop` copies the top of the stack to the specified register. But, once we return from the pop command, rsp is conveniently pointing to the next line of your attack string.

Tip 3: The trick for `target_f3` is figuring out how to move an address to rdi, which is what `target_f3` requries for its input, when rsp is different each time. See if you can find gadgets that deal with rsp directly, that way even if rsp is random, we still get the **same address relative to our attack string.** Also, there's a function in the farm called `<add_xy>` which you will need to use. Notice how it's the only function whose name suggests it could do something useful rather than just random nonsense.

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.

attacklab

~ An instructor (Marco Paolieri) endorsed this note ~

Edit good note 10 Updated 3 weeks ago by Anthony Winney

followup discussions *for lingering questions and comments*