Paxson
Spring 2013

CS 161
Computer Security

Homework 1

Due: Friday, February 15, at 10PM

Instructions. You must submit this homework electronically. To submit, put a single solution file `hw1.pdf` in a directory and run `submit hw1` from that directory. Print your name, your class account name (e.g., `cs161-xy`), your TA's name, the discussion section where you want to pick up your graded homework, and "HW1" prominently on the first page.

Problem 1 ***Basic Memory Safety*** **(20 points)**

The new social mashup *Hipstagram* allows aspiring hackers to share their exploits and brag about their pwnage stories. One lonely night, it so happened that an eager hipster wrote up his story, got bored, and played with the Hipstagram URLs. Accidentally the hipster landed on a website that displayed part of the Hipstagram code that enables user comments. What a find! Having grown up in the social networks era, the hipster by reflex immediately dumped the code on pastebin (http://pastebin.com/m9FZgLsR):

```
1  int debug;
2
3  struct user
4  {
5    char name[32];
6    int is_hip;
7  };
8
9  int send_if_hip(struct user sender, const char* msg, const char* dst)
10 {
11   int len = strlen(sender.name) + strlen(msg) + strlen(dst) + 20 + 1;
12   char* buf = (char*) malloc(len);
13   /* create a shell command to transmit the name and message to dst */
14   snprintf(buf, len, "echo \"%s: %s\" | nc %s 1337", sender.name, msg, dst);
15   if (debug)
16     fprintf(stderr, buf);
17   /* run the command - but only if the user is hip, of course */
18   return sender.is_hip ? system(buf) : -1;
19 }
20
21 int publish()
22 {
23   struct user publisher;
24   char content[256];
25   char to[32];
26   scanf("%s: %s -> %s", publisher.name, content, to);
27   return send_if_hip(publisher, content, to);
28 }
```

Unfortunately, none of the developers of Hipstagram took CS 161 seriously. As a result, the above code snippet contains numerous security flaws.

Describe each flaw you find and suggest an appropriate fix. If a flaw may lead to multiple security vulnerabilities, enumerate each.

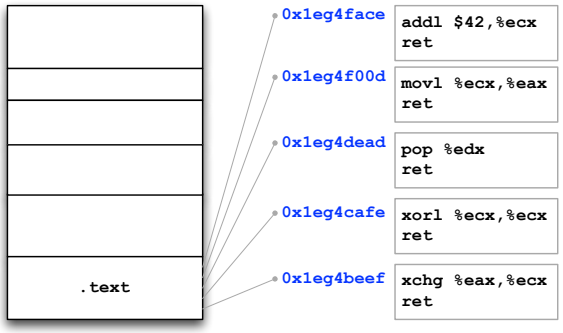
Problem 2 ***Return-Oriented Programming (ROP)*** **(40 points)**

In section you learned how to bypass a non-executable stack via *arc injection*, an attack which introduces new data that existing instructions operate on. In other words, compared to a classic buffer overflow where the attacker provides the malicious instructions (typically shellcode), arc injection "recycles" existing ones.

Return-oriented programming (ROP) takes arc injection to the extreme. In a nutshell, a ROP exploit is built out of many pre-existing *gadgets*, which are small instruction sequences ending in a `ret` instruction. The attacker looks for gadgets in executable segments (e.g., `.text`) and stitches them together into a working program—much like a ransom note.

Recall that `ret` semantically behaves like `popl %eip`, which takes the top-most stack element and writes it into the program counter. After the attacker has transferred control flow into a specific gadget, the execution will eventually end in another `ret`. Due to the previous `ret`, `%esp` now points to one element higher in the stack. The current gadget's `ret` will thus cause that element to be the next address to load into `%eip`. This style of executing can continue for quite a while and depends on how the attacker prepared the stack.

(a) We begin by chaining gadgets to perform a desired computation. To do so, suppose that earlier efforts have already done the grunt work of finding usable gadgets:

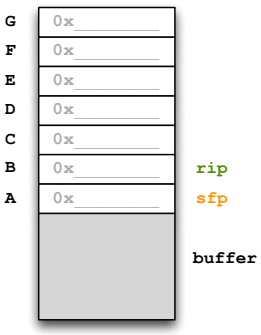


To get the hang of how ROP exploits work, the first step is for you to glue the gadgets together into a working program that assigns the register `%eax` the value 42. Ignoring the `ret` instruction, write down the sequence of instructions that can achieve this task by cherry-picking from the gadgets above.

(b) The next step involves performing the computation in a return-oriented way. To this end, consider a program vulnerable to a stack-based buffer overflow that is just about to execute the function epilogue, that is, right before the `leave` instruction. Assume that you can overflow the entire local buffer plus everything higher up

the stack, i.e., saved frame pointer, return instruction pointer, and arbitrarily far beyond.

How would you set up the stack to implement the computation from part (a)? Use the template below as orientation to figure out how to insert the gadget addresses such that the program executes (in the correct order) the gadgets you identified in part (a). You can assume cells labeled from `A` to `Z`, although you will not need nearly that many.



A solution might look like (though this particular solution is not correct):

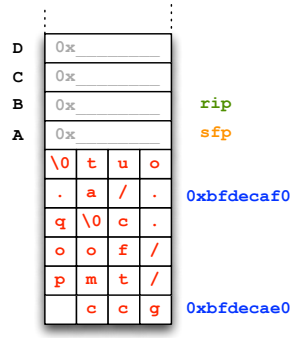
```
A: 1eg4f00d
B: 1eg4beef
D: 1eg4cafe
```

(c) Having developed a basic feel for return-oriented programming, it is time to perform a real attack. Your task is to invoke the `libc` function `system`, which takes a single string as an argument and executes it as a shell command. You scan the program and discover that `system` maps to the location `0x1ffa4b28` — sweet! Now you inject data that `system` should eventually operate on. Your goal is to execute the two statements in order:

1. `system("gcc /tmp/foo.c")`
2. `system("./a.out")`

You have filled a local buffer with the necessary data and know their addresses:

```
0x1ffa4b28 system:  pushl %ebp          ; Prologue
                    movl %esp, %ebp
                    …
                    movl 8(%ebp), %eax  ; Use first argument
                                        ; string and execute it
                    …
                    leave               ; Epilogue
                    ret
```

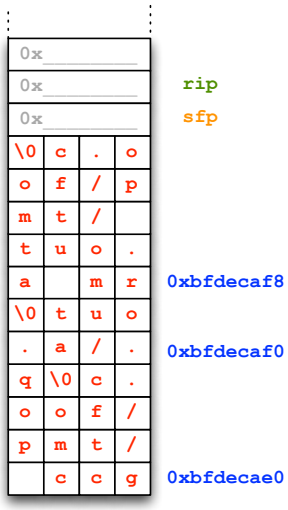


Write down the cell values as in part (b), again assuming cells available from `A` to `Z`.

(d) You found that your exploit worked well, but now you realize that you should cover your tracks after exploitation, and decide to delete the source and generated executable.

1. `system("gcc /tmp/foo.c")`
2. `system("./a.out")`
3. `system("rm /tmp/foo.c a.out")`

Your prepared buffer now looks as follows:



Use the same stack template as in the previous part, but this time with the goal to execute three times `system` with the arguments as shown above. Also explain why you have to use a gadget from part (a) to properly chain the gadgets together.

(e) How can you defend against ROP exploits in buggy C code that you cannot change? Briefly describe three solutions (other than the example below), analyzing their cost and limitations.

Here's an example of what a description of a solution might look like:

Bounds Checking. Modify the compiler to generate code that performs a bounds check on each memory access.

- **Cost**: Proportional to the number of array accesses.
- **Limitations**: All code in the address space, including shared libraries, must be compiled with such a compiler.

Problem 3 ***Security Principles*** **(20 points)**

Identify one or more security principles relevant to each of the following scenarios, giving a one or two sentence justification for your answer:

(a) A popular online stock brokerage has decided to protect its users from having their passwords recorded by keystroke loggers. Now, to log in, users must enter their password using an on-screen keyboard, clicking one letter at a time. As on a physical keyboard, uppercase characters and symbols require users to hit the on-screen "shift" key first. Many users report that they have changed their passwords to be shorter and contain fewer uppercase letters and symbols. Others are concerned that they cannot log in without coworkers or people around them easily spotting the password as they slowly and visibly click it in.

(b) A major airline offers inflight wireless Internet access for a fee. On a recent flight, a curious flyer connects to the network but doesn't pay for access. He notices that even without paying, you still have access to a small number of sites, including the ISP's site (where full access can be purchased), the airline's website, and some 3rd-party shopping sites like Amazon. Requests to any other websites (port 80) result in a redirect to the ISP's landing page. The ISP also blocks attempts to ping other servers and use external DNS servers. However, the flyer notices that for some reason, an HTTPS connection to google.com is allowed (possibly to enable statistics to be sent to Google Analytics), while secure requests to other sites are blocked. Realizing that this means that Google's cloud hosting platform, App Engine, is available too, the user finds a public web-based proxy that someone has hosted there, effectively allowing him to access any external web site.

(c) One of the most widely-deployed home security systems in the country contains a feature designed to help homeowners who are caught by an attacker or robber and are forced to disarm the system. The homeowner can enter a *duress code*, a special combination that appears to disable the alarm, while actually sending an alert to the alarm company's monitoring station. The monitoring station then contacts law enforcement, though it can take some time for them to arrive. As a convenience to its customers, one major national alarm company, responsible for installing and configuring security systems, routinely sets the duress code on installation to be 2-5-8-0, the four numbers that run down the middle of the keyboard. It does not, however, document this policy in any publicly-available user manuals, nor does it give it to customers unless they specifically ask for it. Incidentally, customers receive a discount if they display signs with the alarm company's logo, alerting potential intruders that the home is protected—but also letting a savvy intruder pinpoint houses that might use this preset code.

(d) Flashlight is an app for Android that lets a user turn her phone into a rudimentary light source by displaying a blank white screen at maximum brightness. Android places limits on what an app can do and requires it to request additional permissions from the user on installation. Flashlight asks for the following permissions: storage, system tools (to prevent phone from sleeping), location (GPS), phone call state, and full network access.

Problem 4 ***Reasoning About Code*** **(20 points)**

Consider the following C code:

```
void zeroOut (char s[], char sep, int n)
{
    int j = 0;
    int k;

    while ( s[j] != sep )
    {
        j++;
    }

    for ( k = j+1; k < n; k++ )
    {
        s[k] = '0';
    }

    s[k] = '\0';

}
```

Is the above code memory safe? If yes, prove it by writing the precondition and invariants. If not, describe the modifications required and prove that the modified code is memory safe.

Problem 5 ***Feedback*** **(0 points)**

The feedback we received from Homework #0 was highly helpful, and further feedback would be great to have. So, optionally, feel free to include comments about the course, such as *What's the single thing we could to make the class better?*, or *What did you find most difficult or confusing from lectures or the rest of class, and would like to see explained better?*