Wagner
Spring 2014

CS 161 Computer Security

Homework 1

Due: Tuesday, February 18, at 11:59pm

Instructions. This homework is due Tuesday, February 18, at 11:59pm. It *must* be submitted electronically via Pandagrader (and not in person, in the drop box, by email, or any other method). This assignment must be done on your own.

Please put your answer to each problem on its own page, in the order that the problems appear. For instance, if your answer to every problem fits on a single page, your solution will be organized as follows:

page 1: your solution to problem 1
page 2: your solution to problem 2
page 3: your solution to problem 3
page 4: your solution to problem 4
page 5: your solution to problem 5
page 6: your optional feedback ("problem 6")

If your solution to problems 4 and 5 both take up two pages, your solution would be organized as follows:

page 1: your solution to problem 1
page 2: your solution to problem 2
page 3: your solution to problem 3
page 4: first page of your solution to problem 4
page 5: second page of your solution to problem 4
page 6: first page of your solution to problem 5
page 7: second page of your solution to problem 5
page 8: your optional feedback ("problem 6")

Scan your solution to a PDF—or, write it electronically and save it as a PDF. Then, upload it to Pandagrader.

Problem 1 *Reasoning about code* (15 points)

Consider the following C code:

```
void delescapes(char *s, int n) {
    int i=0, j=0;
    while (j < n) {
        if (s[j] == '%') {
            j=j+3;
        } else {
            s[i] = s[j];
            i=i+1; j=j+1;
        }
    }
}
```

Your job is to figure out the conditions under which `delescapes()` is memory-safe, and then prove it. Fill in the blanks in the four spots shown below with a precondition and three invariants, so that: (a) each invariant is guaranteed to be true whenever that point in the code is reached, assuming that all of `delescapes()`'s callers respect the precondition that you identified, and (b) your invariants suffice to prove that `delescapes()` is memory-safe, assuming that it is called with an argument that satisfies the precondition that you identified.

```
/* Requires: ______(1) */
void delescapes(char *s, int n) {
    int i=0, j=0;
    while (j < n) {
        /* ________(2) */
        if (s[j] == '%') {
            j=j+3;
            /* __________(3) */
        } else {
            s[i] = s[j];
            i=i+1; j=j+1;
            /* __________(4) */
        }
    }
}
```

On your solution, write your answer to each of the parts below.

(a) What did you write for the precondition, at the spot marked (1)?

(b) What invariant did you fill in, at the spot marked (2)?

(c) What invariant did you fill in, at the spot marked (3)?

(d) What invariant did you fill in, at the spot marked (4)?

Problem 2 ***Is it vulnerable?*** **(20 points)**

For each code snippet below, determine whether it is contains a memory-safety bug or not, then write either "buggy" or "not buggy" on your answer depending on whether it contains a memory-safety bug or not. If you wrote "buggy", explain the bug, state which line number(s) contain the bug, and describe an input that would trigger the bug. Assume the inputs to these functions satisfy their preconditions (as documented by the `Requires:` clause) but are otherwise under adversarial control.

(a)

```
    /* Escapes all newlines in the input string, replacing them with "\n". */
    /* Requires: p != NULL; p is a valid '\0'-terminated string */
1:  void escape(char *p) {
2:      while (*p != '\0')
3:          switch (*p) {
4:              case '\n':
5:                  memmove(p+2, p+1, strlen(p));
6:                  *p++ = '\\'; *p++ = 'n';
7:                  break;
8:              default:
9:                  p++;
10:         }
11: }
```

(b)

```
    /* Reverses a string. */
    /* Requires: p != NULL; p is a valid '\0'-terminated string */
1:  int strrev(char *p) {
2:      char *q = p;
3:      if (*p == '\0')
4:          return;
5:      while (*q != '\0')
6:          q++;
7:      q--;
8:      while (p < q) {
9:          char t;
10:         t = *p; *p = *q; *q = t;
11:         p++; q--;
12:     }
13: }
```

(c)

```
1:  struct triple { unsigned int a;
2:                  unsigned int b;
3:                  int c;       };
    /* Requires: if p != NULL, then size(p) >= sizeof(unsigned int)*160 */
4:  int f(int i, char *p, int data) {
5:      struct triple arr[20];
6:      char buf[160];
7:      if (((i<0) || (i>20)) && p != NULL) {
```

```
8:          memcpy(buf, p, sizeof(unsigned int)*160);
9:      } else {
10:         arr[i].b  = data;
11:         arr[i].c  = data;
12:     }
13: }
```

(d)
```
1:  /* Information about the current CD. */
2:  struct cd {
3:      int numtracks; /* The number of tracks on this disc. */
4:      int tracklen[16]; /* The length of each track on the disc, in seconds. */
5:      void (*notify)(struct cd *); /* Call this whenever the CD info changes. *
6:  };
7:
8:  struct cd *curcd = makestructcd();
9:
10: /* Update the length of track number 'track'. */
11: void update_cdinfo(int track, int newtracklen) {
12:     if (track > 16)
13:         return;
14:     curcd->tracklen[track] = newtracklen;
15:     (curcd->notify)(curcd);
16: }
```

Don't worry about `makestructcd()`; it just allocates and initializes a `struct cd`. Assume the adversary can arrange for `update_cdinfo()` to be called with whatever values of `track` and `newtracklen` he likes (those values may have been read directly off the CD, for instance).

Problem 3 *Name the security principle* (20 points)

For each part, name the security principle that was violated, and give a one- or two-sentence justification of how the security principle applies. If more than one of the security principles we discussed are applicable, pick the best answer (the principle that is most directly applicable).

(a) BankOBits is a local bank that offers its customers access to a number of conveniently located ATMs. Normally, when a customer inserts his/her ATM card into a BankOBits ATM, the ATM will contact the BankOBits central server to validate the ATM card inserted into it and check that the corresponding account has sufficient funds before allowing the user to withdraw money. However, if the server does not respond, the network connection is down, or something else goes wrong with this query, the BankOBits ATM will assume all is well, allow the customer to withdraw up to $300, keep a record of the transaction, and upload that information to the BankOBits server whenever connectivity is restored. As a result of this design decision, a gang of criminals are able to steal from the bank by cutting the network connection on BankOBits ATMs and withdrawing $300 from them using a fake ATM card.

(b) David Wagner once heard about a kiosk at one airport that let you access the web, for a fee. To use the kiosk, you had to enter your credit card information at a welcome screen before the kiosk would give you access to a web browser. However, some hacker discovered that if you press F1 to invoke the “help” screen, the Windows help subsystem would pop up a window with generic help information about the login screen. The help text happened to contain a link to an external web site with more help information, and if you click on that link, the kiosk would open the Internet Explorer web browser to display that web page. At that point, one could change the URL in the Internet Explorer address bar and gain full access to the web, without paying.

(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, sets the duress code 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. Criminals discover this standard duress code, rendering the duress code ineffective.

(d) SuperFlashlight 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 requires the app to declare what permissions it needs. SuperFlashlight asks for the following permissions: storage, system tools (to prevent the phone from sleeping), location (GPS), phone call state, send SMS messages, read your contacts, and full network access. It is later discovered that one of the ad libraries used by SuperFlashlight is vulnerable. Criminals exploit the vulnerability to take control of users’ SuperFlashlight app and send premium SMS messages, making a load of money.

Problem 4 *Biometrics and Passwords* (15 points)

Biometric authentication schemes often produce a “confidence” value that trades off between “false positive” and “false negative” errors. A “false positive” is when the system accepts someone when it should not have; a “false negative” is when the system doesn’t accept someone it should have. A false negative prevents an authorized user from logging in; a false positive allows an unauthorized user to access the system.

Password authentication tends to be much more “black and white”. If you mis-type even a single letter when entering your password, your login will be rejected.

(a) How might you modify standard password authentication to afford a sort of “confidence” level, in light of the potential for users to inadvertently mis-type part of their password?

(b) What effects would your modification (in part (a)) have on the security of password authentication?

(c) One simplistic model for how users select passwords is that there is some universal dictionary of 2^{20} possible passwords, and each user randomly picks a password by choosing uniformly at random from this dictionary.[1] Assume that all of the passwords in this dictionary are 10 characters long, and that people have a 1% error rate per character they type, i.e., each character they type independently has a 0.01 probability of being mis-typed. Suppose that we want a false negative rate that is below 0.5%, i.e., below 0.005. Describe what specific parameters your scheme should use, and list the false positive rate your scheme will have at this parameter setting, assuming the attacker gets to make one try at guessing the password. To simplify your calculation, assume that every pair of passwords in the dictionary differ in at least 3 positions.

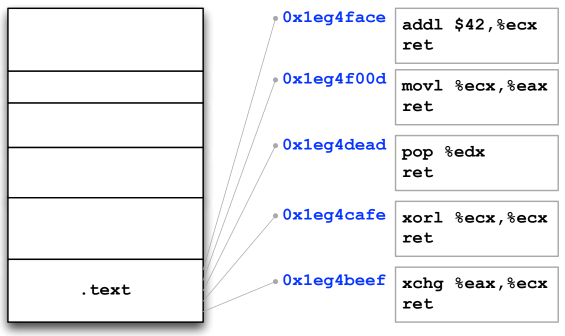
Problem 5 *Return-Oriented Programming (ROP)* (30 points)

In discussion 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:

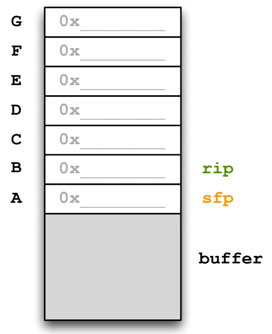


[1]This model is pretty crude, but let's run with it, for purposes of this homework question.

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):

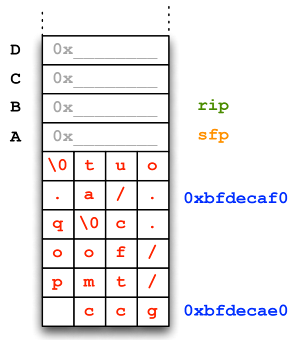
```
D: 1eg4f00d
C: 1eg4beef
B: 1eg4cafe
A: 1eg4f00d
```

(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
```

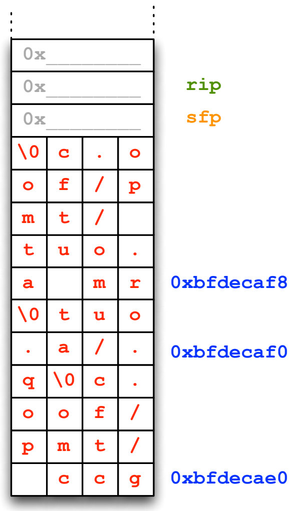


Assume that `system` does not modify the stack and only executes the shell command passed to it as a parameter. Write down the cell values as in part (b), again assuming cells available from `A` to `Z`.

(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 a.out /tmp/foo.c")`

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.

Problem 6 ***Feedback*** **(0 points)**

Optionally, feel free to include feedback. What's the single thing we could do to make the class better? Or, what did you find most difficult or confusing from lectures or the rest of class, and what would you like to see explained better?