Popa & Wagner Spring 2016 — CS 161 Computer Security — Homework 2

Due: Monday, February 22nd, at 11:59pm

Instructions. This homework is due Monday, February 22nd, at 11:59pm. It *must* be submitted electronically via Gradescope (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 optional feedback ("problem 5")

If your solution to problems 3 and 4 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: first page of your solution to problem 3
page 4: second page of your solution to problem 3
page 5: first page of your solution to problem 4
page 6: second page of your solution to problem 4
page 7: your optional feedback ("problem 5")

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

Problem 1 *Web Security* (30 points)

(a) In class we learned about Same Origin Policy for cookies and the DOM and how it protects different sites from each other. Your friend says that you should be careful of visiting any unfamiliar website, because their owners can read cookies from *any* other websites they want. Is your friend right? Explain in 1–2 sentences why or why not.

(b) Google has a website builder service at `sites.google.com/[NAME]`. On this service, you can choose your own NAME and upload any script or html that you desire. Why is this a better design than putting user sites on `google.com/sites/[NAME]`?

Modified 2/17: We removed the hint, as it was confusing, and reworded the question.

(Not for credit: Even this design is not perfect. Once you solve this question, you might enjoy thinking about what the limitations of this design are and how it could be further improved. But we're not asking you to write about this in your answer; that's just a thought exercise for your own understanding.)

(c) You are the developer for a new fancy payments startup, and you have been tasked with developing the web-based payment form. You have set up a simple form with two fields, the amount to be paid and the recipient of the payment. When a user clicks submit, the following request is made:

```
https://www.cashbo.com/payment?amount=<dollar amount>&recipient=<friend's username>
```

You show this to your friend Eve, and she thinks there is a problem. She later sends you this message:

```
Hey, check out this funny cat picture.  tinyurl.com/as3fsjg
```

You click on this link, and later find out that you have paid Eve 1 dollar. (TinyURL is a url redirection service, and whoever creates the link can choose whatever url it redirects to.)

How did Eve steal one dollar from you? What did the tinyurl redirect to? Write the link in your solution.

(d) Continuing from part (c), how could you defend your form from the sort of attack listed in part (c)? Explain in 1–2 sentences.

Problem 2 *XSS: The Game* (15 points)

Visit https://xss-game.appspot.com/ and complete the first 4 levels. This game is similar to Project 1, except you'll be exploiting XSS vulnerabilities instead of buffer overflows. You may use the hints provided by the game.

For each level, describe the vulnerability and how you exploited it in 2–3 sentences. Show the code that you used or what you typed into the input fields.

We recommend using the Chrome browser for this. (We had problems getting past level 3 in Firefox.)

Problem 3 *Biometrics and Passwords* (25 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 4 ***Fuzz testing*** **(30 points)**

This question will teach you about *fuzz testing*, a method for finding (some) memory-safety security vulnerabilities.

(a) First, you are going to fuzz-test a simple C program that has a vulnerability in it, using the American Fuzzy Lop (AFL) fuzzer—a fuzzer that is used in industry. We've set up a virtual machine for you. Grab the VM from `/home/tmp/daw/FuzzingVM.ova` on instructional machines and import it into VirtualBox. Log in with SSH to username `neo` and port 2222, with password `cgciivf9`, like so:

```
ssh -p2222 neo@127.0.0.1
```

Change into the `part1/` directory, where you will find `imgtype`, a simple program that inspects an image file and guesses what type of image it is.

AFL works by starting from one or more *seed files*: files that contain valid inputs for the program being tested. We've chosen a seed file for you: a minimal JPEG image (`in/jpeg.jpg`). AFL works by making random changes to this seed file, running the program on each variant of the file, and seeing if any of them cause the program to crash. The idea is that inputs that cause the program to crash are often an indicator of an underlying memory-safety bug or vulnerability.

Use AFL to find the vulnerability in `imgtype`. You can use a command like

```
timeout -s INT 30s afl-fuzz -i in -o out ./imgtype @@
```

This will run AFL for 30 seconds, using the seed files in the directory `in`, and storing various output to the directory `out`. The status screen gives you some indication

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

of progress. The most helpful field is the part in the upper-right that says `uniq crashes`—if this has a non-zero number, then AFL has found at least one input that triggers a crash. AFL will the input files that it has discovered cause a crash in `out/crashes`.

Look at one of them, and use it to identify the line of code in `imgtype.c` that contains the vulnerability. You might try running the program under gdb with that input file and then generate a backtrace. Or, you can use valgrind, which outputs a handy backtrace:

```
valgrind ./imgtype out/crashes/whatever
```

Each line of the backtrace represents one stack frame, and indicates a corresponding line of code (e.g., `imgtype.c:67` represents line 67 of `imgtype.c`). Generally, the top-most line in the backtrace that is in `imgtype.c` is the best place to start looking for the bug. Look at that line of code in `imgtype.c` and the surrounding lines to see if you can spot what the bug in the code is.

In your answer, write down: (a) the line of code in `imgtype.c` where the vulnerability occurs, (b) an English description of what the vulnerability is, and (c) a description of what conditions the input file must satisfy to trigger the memory-safety failure.

(b) *Generation-based fuzzing* works as follows: in each iteration, it generates a random input file, runs the program on that input, and checks whether the program crashes. Suppose we implement a particularly naive form of generation-based fuzzing, where in each iteration we generate every byte of the file uniformly and independently at random. Then, we apply this to the `imgtype` program from part (a).

About how many iterations would we need to perform, to find the vulnerability in `imgtype`? You can estimate this by computing the expected number of iterations until we find the vulnerability. You can assume that the program crashes whenever we feed it any input that writes out of bounds of any buffer or array. If we can perform 1000 iterations per second, about how long would it take for this naive generation-based fuzzer to find the vulnerability?

(c) *Mutation-based fuzzing* is a little different. We start with a *seed file*, a valid input file. In each iteration, the fuzzer randomly makes a small change to the seed file, runs the program on the result, and checks whether the program crashes.

Suppose we implement a naive mutation-based fuzzer where each iteration works as follows: for each byte in the seed file, with probability 0.99 we leave that byte unchanged; with probability 0.01, we change to some other random byte (with all possible values equally likely). Suppose we apply this to the `imgtype` program from part (a). About how many iterations would we need to perform, to find the vulnerability in `imgtype`? If we can perform 1000 iterations per second, about how long would it take for this naive mutation-based fuzzer to find the vulnerability?

(d) Now, we'll have you fuzz a real, large program: in this case, you'll be fuzzing an

older version of ImageMagick's `convert` program, which converts between image formats. Your task is to find one or more vulnerabilities in `convert`'s GIF parser—i.e., to find a malicious input `evil.gif` such that the command `convert evil.gif whatever.pnm` triggers a crash.

Log into the VM and switch to the `part4` directory. Put one or more seed GIF files in the `in/` directory. Then, fuzz for a few minutes, or until you find a vulnerability, by running

```
./start-fuzzing
```

You should be able to find at least one vulnerability.

In your writeup, describe (a) how you selected your seed files, and (b) include a stack backtrace corresponding to each vulnerability you found.

Hints: Feel free to get creative in your choice of seed file(s). It's to your advantage to choose seed files(s) that are as small as possible, as the fuzzer will be able to try more iterations per second. One good heuristic is to take an ordinary GIF file and truncate it:

```
dd if=bigfile.gif of=in/small.gif bs=1 count=128
```

If you pick your seed file(s) well, you should be able to find a vulnerability or two within a few minutes of fuzzing. You can always stop AFL by pressing Ctrl-C.

Beware that not all crashes reported by AFL are real. It seems that AFL sometimes gets confused and reports an input file as triggering a crash, when it didn't actually cause a serious problem. Therefore, make sure you run `convert` by hand on each candidate crasher-file to see if it does indeed cause a memory-safety vulnerability. For this part, we've helpfully compiled `convert` to use Address Sanitizer (ASAN), so if a memory-safety error occurs, you'll see an output message (`ERROR`) and a backtrace when you run `convert` on that input file. Due to incompatibilities between Valgrind and ASAN, you won't be able to use Valgrind with `convert`, but you shouldn't need to, as ASAN already does everything Valgrind does.

You can run the fuzzing VM on one of the `hiveNN.cs.berkeley.edu` instructional machines, but please check first that no one is actively using the machine: log into the machine and run `top` first. Fuzzing is CPU-intensive, so if you see someone else actively using the machine, pick a different machine.

Problem 5 ***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?