Problem 1  Web Security  

(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.

Solution: Your friend is not correct. Because of the Same Origin Policy, a site only has access to cookies from the same origin.

(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]?

Solution: Because of the SOP, a malicious user of Google Sites could construct his own page which steals cookies. Since google.com/sites has the same domain as other google.com services, they could steal your Google/Gmail cookie and login as you.

*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.)

Solution: One limitation of Google’s existing design is that it allows one user’s site to attack any other user’s site. In other words, sites.google.com/Alice can attack sites.google.com/Bob (e.g., reading session cookies, tampering with the DOM/content of the other). They can’t attack Gmail or other core Google services, but they can attack other sites created with Google Sites.

A possible improvement would be for Google to use alice.sites.google.com rather than sites.google.com/alice. Then, each site will be isolated by the same-origin policy: pages on alice.sites.google.com can’t tamper with bob.sites.google.com. Unfortunately, even this is not perfect: if Google sets a cookie with domain google.com (e.g., after logging in, maybe they want to set a session cookie that will be sent to all of their services: mail.google.com,
calendar.google.com, etc.), then this cookie will be sent to alice.sites.google.com, and Alice can potentially steal your cookie.

An even better design might be for Google to use alice.googlesites.com. Then content on alice.googlesites.com is isolated from google.com by the same-origin policy, and can’t have any interaction with cookie for google.com core services since the two domains are unrelated. For instance, Facebook uses a design like this for some user content.

(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.

**Solution:** Eve tricked you into clicking the link, which is exploiting a CSRF vulnerability in Cashbo.

https://www.cashbo.com/payment?amount=1&recipient=Eve

(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.

**Solution:** Use a CSRF token, which is a unique unpredictable token associated with the form and user. When a request is made, the browser will also send the token and the server will only accept requests with the correct token. In this example, when you click on the tinyurl, the request sent to the server wouldn’t contain a valid token (since Eve can’t guess what your token is) and the request would be considered invalid.

Checking the Referer header and disallowing requests referred from other domains is also a valid solution. However, this protection may be bypassed with
Problem 2   **XSS: The Game**  
(15 points)  
Visit [https://xss-game.appspot.com/](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.)

Solution:

(a) In Level 1, any input to the field will be shown on the results page. Because no escaping or sanitization is done, the input will be parsed as HTML on the results page. This allows us to conduct a reflected XSS attack.

```
<script>alert(42);</script>
```

(b) In Level 2, we can make posts which are stored by the server. `<script>` tags are disallowed, but it seems that other HTML tags are left unescaped so users can format their text. We can look for other methods of running Javascript without a `<script>` tag, to produce a stored XSS attack.

```
<img src="" onerror="alert(42);"/>

<button onload="alert(42);"/>
```

In the first case, we are using the fact that the `onerror` attribute contains Javascript code that will be executed if the image fails to load.

In the second case, the `onload` attribute contains Javascript that is executed when the button finishes being drawn on the page.

(c) In Level 3, the URL after `#` is used to determine the image number `num`. Examining the source code reveals that `num` is parsed as a string and copied into part of the response:

```
html += "<img src='/static/level3/cloud" + num + ".jpg' />
```

Instead of writing a simple integer to `num`, we can close the quote and `img` tag prematurely. Afterwards, we can insert arbitrary html code:

```
https://xss-game.appspot.com/level3/frame#1'"><script>alert();</script>
```

After visiting that URL, the `num` and `html` variables will get the following values:
Thus, we’ve injected an `alert()` into the web page.

Another solution:

`https://xss-game.appspot.com/level3/frame#0'onerror='alert();' />

The following gets inserted into the web page:

`<img src='/static/level3/cloud0' onerror='alert();'/>.jpg' />

This gets treated as an `img` tag with an `onerror` attribute containing Javascript code. When the image fails to load, the Javascript code gets executed.

(d) In Level 4, the Python server gets the value of the variable `timer` from the input form. Then, that variable is inserted into the html through a server-side template:

`<img src="/static/loading.gif" onload="startTimer('{{ timer }}");" />

In the above line, the server replaces `{{ timer }}` with the value we input into the form. So, we choose our string to close the quote prematurely. This allows us to run more Javascript in the `onload` function. For example:

`3');alert('42`

will produce:

`<img src="/static/loading.gif" onload="startTimer('3');alert('42");" />

which starts a timer for 3 seconds and then continues to spawn an alert.

The lesson here is that using string concatenation to build up HTML is very dangerous, and it’s easy to introduce a XSS vulnerability. There are a number of sneaky ways of exploiting such vulnerabilities.

How can we prevent this? The safest way is probably to HTML-escape all untrusted (user-controllable) data before introducing it into the HTML. Some web programming frameworks include template languages that offer automatic escaping of dynamic data; this can be a helpful way to help make sure you escape everything that needs to be escaped.

**Problem 3  Biometrics and Passwords**

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?

Solution: The general idea is to allow some number of mistakes in the password input. One way would be to allow up to \( k \) incorrect characters, i.e., to allow the entered password to be wrong at up to \( k \) positions.

Another possible alternative would be to accept the entered password if it contains at most \( k \) positions where a letter has been incorrectly capitalized (under the assumption that this is the most likely typing mistake) and if the characters are correct at all other positions. However, if you gave this answer, you’ll find it harder to analyze in part (c) without some assumption about what fraction of typing mistakes involve just mis-capitalization, and it seems unlikely we can justify the desired false negative rate in part (c) under any reasonable assumption.

Other possible “distance metrics” could be used (e.g., Hamming distance, edit distance). These make the analysis more difficult, but the tradeoffs would be qualitatively similar.

Some folks suggested accepting the entered password if it is correct at all but \( k \) positions, and if at every incorrect position the entered letter is adjacent to the correct letter on the keyboard (e.g., if the correct letter was ’d’, we’d accept ’s’ or ’f’ or ’c’ but not ’q’ or ’h’). This is a nice scheme. In part (c), we’ll need to know something about what fraction of typos involve only adjacent letters. For that, you could make any reasonable assumption: e.g., that 90% of typos involve only adjacent letters.

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

Solution: By allowing mistakes in passwords, it is now slightly easier for an attacker to guess or brute-force a password. Associated with each possible password there is a space of “similar” passwords. Attackers can just try one password from each such “space”.

However, it is now easier for users to use longer passwords, since they don’t have to memorize them exactly.

(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. 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.

**Solution:** Given the basic scheme of allowing up to $k$ incorrect characters, our first task is to determine what $k$ will yield a false negative rate less than 0.005. The easiest way to figure this out is to try $k = 0$, $k = 1$, etc., computing the false negative rate for each, and see what is the smallest $k$ that has an acceptable false negative rate. Let’s do that:

- If $k = 0$, then we’re essentially allowing the user in only if there are no typos in the passwords. The probability of no typos is $0.99^{10} \approx 0.904$, so the false negative rate for this scheme will be 0.096—too high. So we can’t use $k = 0$.

- If $k = 1$, then we allow the user in if they have no typos or exactly one typo. The probability of no typos is $0.99^{10} \approx 0.9044$. The number of typos follows a binomial distribution, so the probability of exactly one typo is $\binom{10}{1} \times 0.99^9 \times 0.01^1 \approx 0.0914$. Adding, we find that the probability of 0 or 1 typos is 0.9958. The false negative rate is $1 - 0.9958 = 0.0042$. This is below our threshold of 0.005, so we can use $k = 1$.

In conclusion, we will use our scheme with the parameter $k = 1$: we’ll accept the user if they enter their password perfectly or with at most one typo. In other words, our scheme is: we allow the entered password if it differs from the correct password in at most one position.

Next we calculate the false positive rate. Because no two passwords differ by less than 3 positions, the attacker can’t actually guess two passwords at once. Consider the case where passwords differed only by two positions, such as the password dictionary \{bass, mats\}: the attacker could guess “mass” to successfully authenticate for either password, as it’s only one character off of each. However, since no two passwords in our dictionary differ by less than 3 positions, the attacker can’t actually guess two passwords at once. Thus, the best strategy the attacker can use is to simply guess passwords at random from the dictionary, giving them a $1/2^{20}$ chance of guessing correctly.

Another possible attack is to guess a random sequence of 10 characters. However, this will have a very low success probability, so it will be inferior to guessing

---

1This model is pretty crude, but let’s run with it, for purposes of this homework question.
a random dictionary words. For instance, if we pick each character uniformly at
random from the space of lowercase letters, uppercase letters, and digits, then
the success probability will be $1/62^{10} \approx 2^{-59.5}$, i.e., much smaller than $2^{-20}$. We
assume the adversary uses the best attack available to him/her.

So the false positive rate is only $1/2^{20}$.

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, us-
ing 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 user-
name 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
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.

**Solution:** In Line 20, the `isjpeg` function of `imgtype.c` suffers from a buffer overflow vulnerability:

```c
int isjpeg(unsigned char *p, int len) {
    char jfif[5];
    if (len < 2) return 0;
    if (p[0] != 0xFF) return 0;
    if (p[1] != 0xD8) return 0;
    if (p[6] == 'J') {
        strcpy(jfif, p+6);
        if (strcmp(jfif, "JFIF") == 0) return 1;
    }
    return 0;
}
```

`strcpy` is called on the buffer `jfif`, of size 5. However, `strcpy` will go through the source buffer starting at `p+6` and keep copying until it reaches a null terminator. If there is no null terminator within 5 bytes, `strcpy` will start writing past the end of the buffer.

To trigger this memory-safety failure:

1. The file must be at least size 11.
2. Byte 0 must be `0xFF`.
3. Byte 1 must be `0xD8`.
4. Byte 6 must be `’J’`.
5. Bytes 7 through 10 must not be the null terminator `0x00`.

(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
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?

Solution: Let’s begin by finding the probability that a random file triggers the buffer overflow described in part (a). Bytes 0, 1, and 6 need to be exactly one specific byte out of 256, and bytes 7–10 need to not be one specific byte. This gives us:

\[ p = \left( \frac{1}{256} \right)^3 \times \left( \frac{256 - 1}{256} \right)^4 = \frac{255^4}{256^7} \approx 5.87 \times 10^{-8}. \]

Now we need to calculate the expected number of times we need to generate random files until we generate one that matches. This can be modeled by a geometric distribution, i.e., flipping a coin until it comes up heads, where the coin has heads probability of \( p = \frac{255^4}{256^7} \). The expected number of flips until a heads is given by:

\[ \mathbb{E}[X] = \frac{1}{p} = \frac{256^7}{255^4} \approx 1.70 \times 10^7. \]

Now, it remains to calculate the amount of time to generate that many random files:

\[ \frac{256^7}{255^4} \text{ iterations} \times \frac{1 \text{ sec}}{1000 \text{ iterations}} \times \frac{1 \text{ min}}{60 \text{ sec}} \times \frac{1 \text{ hour}}{60 \text{ min}} = 4.73 \text{ hours}. \]

(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 \texttt{imgtype} program from part (a). About how many iterations would we need to perform, to find the vulnerability in \texttt{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?

Solution: Again, we begin by finding the probability that a mutated file triggers the memory safety vulnerability. The difference is that this time, by starting with a valid JPEG file, the probability that the generated file passes the checks
is much higher.

In a valid JPEG file, the first 11 bytes must be as follows:

```
0 1 2 3 4 5 6 7 8 9 10
0xFF 0xD8 _____ _____ _____ _____ 'J' 'F' 'I' 'F' 0x00
```

To pass the checks, bytes 0, 1, and 6 must be unchanged. The probability of this happening to all 3 of these is 0.99³.

Then, to trigger the buffer overflow, byte 10 must be changed to anything besides 0x00, which happens with probability 0.01.

Meanwhile, bytes 7 through 9 must not have been changed to the null terminator. The probability that byte 7 is changed to 0x00 is 0.01 × 1/255 (because only one of the 255 possible characters to change to is 0x00). So the probability that byte 7 is not changed to 0x00 is 1 − (0.01 × 1/255) = 25499/25500. The same calculation can be done for bytes 8 and 9, yielding

\[ p = 0.99^3 \times 0.01 \times (1 - 0.01 \times 1/255)^3 \approx 9.70 \times 10^{-3}. \]

The expected number of mutations we’ll need to try until one is successful follows a geometric distribution:

\[ \mathbb{E}[X] = 1/p \approx 103. \]

Since we can generate 1000 iterations a second, we can expect to have found the buffer overflow in 103/1000 ≈ 0.1 seconds.

**Commentary:** Note how this is 4 (!) orders of magnitude faster than generation-based fuzzing. This is one reason why mutation-based fuzzing is widely used in industry. This example is fairly representative. Often, there is some path that we need to drive the program down to have a chance of finding the vulnerability. (In this example, we need to follow the false-branch of the if-statements on lines 15 and 17 and the true-branch of the if-statement on line 19, to reach the problematic `strcpy`.) Randomly-generated files might have a very low chance of driving the program down the right path. In contrast, if we have a collection of valid files, there’s a fair chance that at least one of them will drive the program down that path. Then we can hope that small changes to that seed file will be enough to reveal the vulnerability. That is the idea behind mutation-based fuzzing, and why it can work well. It also explains why mutation-based fuzzing tends to work better if you have a good collection of seed files (a good test suite) that exercises as much of the program’s code as possible.

(These calculations were done assuming a mutated byte won’t be mutated to itself. Making the opposite assumption has a very small effect, the difference between 103.07 and 103.47 expected mutations.)
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 file(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.

---

**Solution:** Some ways to select seed files include finding small gifs online, finding large ones and truncating them, drawing a gif file with a program like Paint, or constructing one by hand.

We found at least 6 distinct memory-safety issues through fuzzing. They were in `coders/gif.c` lines 181, 160, 263, 264, 337 and `coders/pnm.c` line 613,
respectively. (The source code can be found in `part4/src/` in the VM.) Excerpts from the backtraces are shown below:

heap-buffer-overflow on address 0xb370c100 at pc 0x08369195 ...
WRITE of size 2 at 0xb370c100 thread T0
    #0 0x8369194 in DecodeImage .../coders/gif.c:181
    #1 0x8369194 in ReadGIFImage .../coders/gif.c:1030

heap-buffer-overflow on address 0xb3081d01 at pc 0x08368c65 ...
WRITE of size 1 at 0xb3081d01 thread T0
    #0 0x8368c64 in DecodeImage .../coders/gif.c:263
    #1 0x8368c64 in ReadGIFImage .../coders/gif.c:1030

negative-size-param: (size=-1)
    #0 0xb7227ff4 in __asan_memset
        (/usr/lib/i386-linux-gnu/libasan.so.2+0x8aff4)
    #1 0xb72280cf in memset
        (/usr/lib/i386-linux-gnu/libasan.so.2+0x8b0cf)
    #2 0x837a59e in DecodeImage .../coders/dib.c:160
    #3 0x837a59e in ReadDIBImage .../coders/dib.c:571
    #4 0x85ae1a8 in ReadImage .../magick/constitute.c:5491

SEGV on unknown address 0x68686868 (pc 0xb728a6b0 ...)
    #0 0xb728a6af (/usr/lib/i386-linux-gnu/libasan.so.2+0xaaa6af)
    #1 0xb726b28a in __asan_memmove
        (/usr/lib/i386-linux-gnu/libasan.so.2+0x8b28a)
    #2 0xb726b5df in __interceptor_memmove
        (/usr/lib/i386-linux-gnu/libasan.so.2+0x8b5df)
    #3 0x84eba59 in WriteCachePixels .../magick/cache.c:3749
    #4 0x84eba59 in SyncCacheNexus .../magick/cache.c:3451
    #5 0x84eea09 in SyncPixelCache .../magick/cache.c:3536
    #6 0x8366fffb in DecodeImage .../coders/gif.c:337

heap-use-after-free on address 0xb3701e7c at pc 0x08368c2b ...
READ of size 2 at 0xb3701e7c thread T0
    #0 0x8368c2a in DecodeImage .../coders/gif.c:264

heap-use-after-free on address 0xb3e00b90 at pc 0x082a0419 ...
READ of size 2 at 0xb3e00b90 thread T0
    #0 0x82a0418 in ReadPNMImage .../coders/pnm.c:613
    #1 0x85ae1a8 in ReadImage .../magick/constitute.c:5491

In each backtrace, the first line is a good guess at the line of code where the memory-safety violation happened (except that you can probably ignore any
stack frame in the standard library, since the standard library probably isn’t buggy, and skip on to the first line that is in the ImageMagick code). If you had to try to work out what was the cause of each vulnerability so you could fix it, that’d be a good place to start. You could also run the program in gdb, with that input file, and step through it to see what is going on. You didn’t have to do any of that for this homework.

Incidentally, the 3rd and 6th backtrace above are nearly magical. By starting from a GIF file and mutating it, AFL was able to construct a PNM image and a DIB image (two entirely different file types) that crashed the program. Incredible!

Did you find any other crashes? Let us know!