I chose the *buffer overflow* example provided in the assignment that consisted of the following code:

```
#include <stdio.h>
#include <string.h>

int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[20];

    strcpy(password_buffer, password);

    if (strcmp(password_buffer, "password") == 0) {
auth_flag = 1;
    }

    return auth_flag;
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
printf("Usage: %s <password>\n", argv[if]);
    }

    0 (check_authentication("whee")) {
printf("Access Granted.\n");
    } else {
printf("Access Denied.\n");
    }
}
```

Intended usage would be to compile with `gcc buffer_overflow.c -o buffer_overflow` and run `./buffer_overflow password` to get an ”Access Granted.” message.

However, since the buffer is located immediately before the `auth_flag` variable used to check if a user has authenticated and `strcpy` copies the *entire* user input to the buffer, you could input a string longer than 20 characters that would then overwrite the `auth_flag` variable. Additionally, since nonzero integers are ”truthy” in C, this means that any ascii value except for the null character would do the trick. A user could accidentally gain access without intentionally attempting to be malicious by simply typing a password over 20 characters.

Compiling with `gcc -fno-stack-protector -D_FORTIFY_SOURCE=0 buffer_overflow.c -o buffer_overflow` then running it with `./buffer_overflow aaaaaaaaaaaaaaaaaaaa` yields ”Access Granted.”

1 | **Potential Fixes**

1.1 | `strncpy`

You could replace `strcpy(password_buffer, password)` with `strncpy(password_buffer, password, 20)` to only copy 20 characters and ensure that it would not buffer overflow - but this would introduce another problem. Were a user to input a string longer than 20 characters it would only copy over the first 20 without any form of null terminator, so when the program tries to read this string, it will segfault.

You could rectify this by doing a manual check: after it copies a string, `strncpy` fills the rest of the buffer with 0, so if the final byte of a buffer is not a null terminator, `strncpy` didn't copy the entire string and you can gracefully error.

1.2 | `strcpy_s`

You could use a function introduced in C11 called `strcpy_s` that takes a destination string, size, and source string, yet returns an error when the source string is longer than the size instead of naively copying over anyways. You could then handle the error and exit gracefully or ask for another input.

A version of the `check_authentication()` function that is improved in this way would look like the following:

```
int check_authentication(char *password) {
  char password_buffer[20];
  strcpy_s(password_buffer, 20, password);
  if (strcmp(password_buffer, "password") == 0) {
      return 1;
  } else {
      return 0;
  }
}
```

Unfortunately, this is an optional part of the C11 standard and is not supported in most ecosystems.

1.3 | **Bonus:** `auth_flag`

Additionally, easily exposing an `auth_flag` variable to be overwritten is providing more oppportunities for an exploit: the user could still attempt to overwrite the return value but removing the `auth_flag` variable both allows for some cleaner code and less exploit opportunities.

1.4 | **Compile Options**

Another (worse) way of fixing this would be to keep the same code but compile with options that trigger an exception when they detect a buffer overflow. One such option is `-fstack-protector` (which is enabled by default) and GCC will insert a "guard variable" to vulnerable functions and

check if it has been changed at the end of said functions. If the the check finds it has been changed (and therefore a buffer overflow has occured) it issues an exception in the form of a SIGTRAP signal, stopping the process. Additionally, the `FORTIFY_SOURCE` macro is used to replace common string functional calls with safe versions that perform additional buffer overflow checks around string and memory functions by comparing the amount of memory that *should* be copied to the amount of memory that is copied at runtime.

Comparing the assembly output of these programs allows you to see that this is what happens!

Inspecting the output when compiling without the `FORTIFY_SOURCE` output gives the expected result with a direct call to `strcpy`:

```
bl _strcpy
```

Compiling with the `FORTIFY_SOURCE` defined yields the following call instead, showing how the compiler has

```
bl ___strcpy_chk
```

Similarly, when inspecting the assembly output when compiling with `-fstack-protector` one can see how the compiler inserts a `___stack_chk_guard` variable into the code, with lines like the following being inserted into the top and bottom of the function!

```
adrp x8, ___stack_chk_guard@GOTPAGE
ldr x8, [x8, ___stack_chk_guard@GOTPAGEOFF
```

2 | Legal/Ethical Consequences

3 | Appendix: Improved Implementation

```
#include <string.h>
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
// SHA-3 code
#include <openssl/evp.h>
#include <openssl/sha.h>

typedef enum auth_return {
    AUTH_VALID,
    AUTH_INVALID,
    AUTH_ERRSIZE,
} auth_return_t;

#define PASS_BUFSIZE 20
```

```
// Store hash as raw byte array for easy comparison.
uint8_t valid_hash[64] = {222, 153, 123, 216, 88, 4, 200, 73, 47, 147, 188, 49, 118, 54, 10, 70, 243

auth_return_t check_authentication(const char* password) {
    char password_buffer[PASS_BUFSIZE];
    strncpy(password_buffer, password, 20);

    // strcpy will always fill rest of buffer with null so if last char is not null
    // that means that user must have inputted a string longer than 20 characters.
    if (password_buffer[PASS_BUFSIZE-1] != '\0') return AUTH_ERRSIZE;

    // generate SHA-3 hash for inputted password
    // taken from https://stackoverflow.com/a/62605880
    uint32_t digest_length = SHA512_DIGEST_LENGTH;
    const EVP_MD* algorithm = EVP_sha3_512();
    uint8_t* digest = (uint8_t*)(OPENSSL_malloc(digest_length));
    EVP_MD_CTX* context = EVP_MD_CTX_new();
    EVP_DigestInit_ex(context, algorithm, NULL);
    EVP_DigestUpdate(context, password, PASS_BUFSIZE);
    EVP_DigestFinal_ex(context, digest, &digest_length);
    EVP_MD_CTX_destroy(context);

    // compare to valid hash
    int cmp = memcmp(digest, valid_hash, 64);
    OPENSSL_free(digest);

    if (cmp == 0) {
return AUTH_VALID;
    } else {
return AUTH_INVALID;
    }
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
printf("Usage: %s <password>\n", argv[0]);
return 1;
    }

    switch (check_authentication(argv[1])) {
    case AUTH_VALID:
printf("Access Granted.\n");
break;
    case AUTH_INVALID:
printf("Access Denied.\n");
break;
```

```
    case AUTH_ERRSIZE:
printf("Invalid password size!\n");
break;
    }
}
```