#hardening #codereview #vulnerabilities #coding

- **Name**: Iskander Nafikov
- **E-mail**: i.nafikov@innopolis.university
- **GitHub**: https://github.com/iskanred

Secure Coding

Objectives

In this lab I will:

1. Practice reading program code and finding potential security breaches in it.
2. Explain how exactly do your findings threaten security of the analyzed code.
3. Categorize each finding as threatening either integrity, confidentiality, or availability.
4. Fix each finding in the code; explain how your modification fixes the problem.
5. Introduce hardenings to avoid the risks created by similar problems.

Description

> **Task description**
>
> You are given an archive with two files, hash.c and hash.h, that contain a naive hash table implementation. The code contains programming mistakes that make it vulnerable in some way. You will need to:
>
> 1. Copy `hash.c` and `hash.h` to `hash_fixed.c` and `hash_fixed.h`, respectively.
> 2. Find as many security related programming mistakes as you can in `hash.c` and `hash.h`.
> 3. For every mistake you find (comment directly in the code for each step):
>   a. Clearly locate the mistake in the code.
>   b. Explain why it is a mistake and how it affects security of the code.
>   c. Categorize the mistake as affecting integrity, confidentiality, or availability.
>   d. Fix the mistake in `hash_fixed.c` and `hash_fixed.h`.
>   e. Explain why do you expect your fix to remove the problem.
>   f. Introduce a hardening to avoid the risks created by the mistake.

4. Summarize the hardening instructions from 3.f, for every problem, in file `hardenings.txt`.

- Below is the `hash.c` file

```
/**
*
* @Name : hash.c
*
**/
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hash.h"

// The format of mistake description is the following:
// "[#][X] ...", where:
// # - is an ID of mistake
// X is either A, C, or I meaning the security breach related to
Availability, Confidentiality, or Integrity correspondingly
// ... is a description of the mistake

/* [8][A] Weak hashing algorithm implementation resulting in a high
probability of collision
           which can lead to problem with performance, and hence
availability */unsigned HashIndex(const char* key) {
    unsigned sum = 0;
/* [9][A] Infinite iteration because pointer `c` is never NULL. This
leads to segmentation fault */
/* [10][I] Initializing 'char *' with an expression of type 'const char
*' discards qualifiers.
          This can lead to accidental unintended change of values during
iteration */    for (char* c = key; c; c++) {
/* [11][I] char `c` can contain negative values (depending on a
compiler), while int `sum` is unsigned.
          This leads to potential overflow when adding negative values.
*/        sum += *c;
    }

/* [12][I] Modulation by MAP_MAX is absent which leads to access array
memory out of bounds */
    return sum;
}
```

```
HashMap* HashInit() {
/* [13][A] Not checking for successful allocation can lead to
dereferencing
           null pointers which will lead to segmentation fault */
return malloc(sizeof(HashMap));
}

/* [14][I] No key size validation may lead to data loss  */
void HashAdd(HashMap *map, PairValue *value) {
    unsigned idx = HashIndex(value->KeyName);

/* [15][I] Checking if this key already exists is absent
         which leads to overriding or storing the same key twice or more
*/    if (map->data[idx])
/* [16][I] Replacing element instead of putting it to the head of the
list */
        value->Next = map->data[idx]->Next;

    map->data[idx] = value;
}

PairValue* HashFind(HashMap *map, const char* key) {
    unsigned idx = HashIndex(key);

/* [17][I] Iterating with non-const loop variable may lead to accidental
changing the element during iteration */
    for ( PairValue* val = map->data[idx]; val != NULL; val = val->Next )
{
/* [18][I] Incorrect string comparison leading to overriding, storing the
same key or losing an element  */
        if (strcpy(val->KeyName, key))
            return val;
    }

    return NULL;
}

/* [19][I] No key size validation may lead to data loss  */
void HashDelete(HashMap *map, const char* key) {
    unsigned idx = HashIndex(key);

    for( PairValue* val = map->data[idx], *prev = NULL; val != NULL; prev
= val, val = val->Next ) {
/* [20][I] Incorrect string comparison leading to overriding, storing the
same key or losing an element  */
```

```
        if (strcpy(val->KeyName, key)) {
            if (prev)
                prev->Next = val->Next;
            else
                map->data[idx] = val->Next;
        }
    }
/* [21][A] Missing memory deallocation for `val` leading to memory leaks
*/
}

void HashDump(HashMap *map) {
    for( unsigned i = 0; i < MAP_MAX; i++ ) {
/* [23][I] Iterating with non-const pointer loop variable may lead to
accidental changing the element during iteration */
        for(PairValue* val = map->data[i]; val != NULL; val = val->Next )
{
/* [24][C] Format string is missing, could lead to format string
vulnerabilities
           https://ctf101.org/binary-exploitation/what-is-a-format-
string-vulnerability/ */            printf(val->KeyName);
        }
    }
}
```

- Below is the `hash.h` file

```
/**
*
* @Name : hash.h
*
**/
#ifndef __HASH__
#define __HASH__

// The format of mistake description is the following:
// "[#][X] ...", where:
// # - is an ID of mistake
// X is either A, C, or I meaning the security breach related to
Availability, Confidentiality, or Integrity correspondingly
// ... is a description of the mistake

/* [1][I] Opening internal entry `PairValue as an interface may result in
data corruption
          if someone will assign / remove `Next` by their own */
```

```
typedef struct {
        #define KEY_STRING_MAX 255
       char KeyName[KEY_STRING_MAX];
       int  ValueCount;
        struct PairValue* Next;
    } PairValue;

    typedef struct {
        #define MAP_MAX 128
/* [2][I] Non-const pointer of `data` may lead to data corruption */
/* [3][C] Array is initialized with some left data in memory which can
lead to data leak */
       PairValue* data[MAP_MAX];
    } HashMap;

/* [4][I] Non-const pointer of `map` may lead to data corruption */
    HashMap* HashInit();
    void HashAdd(HashMap *map, PairValue *value);
    void HashDelete(HashMap *map, const char* key);
/* [5][I] Non-const pointer of `map` may lead to data corruption */
    PairValue* HashFind(HashMap *map, const char* key);
/* [6][I] Non-const pointer of `map` may lead to data corruption */
    void HashDump(HashMap *map);
#endif
```

- Here are the fixes: `hash_fixed.c`

```
/**
*
* @Name : hash.c
*
**/
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hash_fixed.h"

/* Internal entry now is not exposed */
typedef struct Entry {
    #define KEY_STRING_MAX 255U
    const char key[KEY_STRING_MAX + 1]; // +1 for '\0' at the end
    struct Entry* prev;
    struct Entry* next;
} Entry;
```

```c
void NullifyMapDataArray(HashMap* map) {
    for (unsigned int i = 0U; i < MAP_MAX; ++i) {
        map->data[i] = (Entry*) NULL;
    }
}

HashMap* HashInit() {
    HashMap* map = malloc(sizeof(HashMap));

/* Check if allocation succeeded */
    if (map == NULL) {
        fprintf(stderr, "Memory allocation for HashMap failed\n");
        exit(EXIT_FAILURE);
    }

/* We will not see any memory "garbage" */
    NullifyMapDataArray(map);
    return map;
}

/* Validate user's input key length */
void __ValidateKeySize(const char* key) {
    if (strlen(key) > KEY_STRING_MAX) {
        fprintf(stderr, "Length of key > maximum of %u characters\n",
KEY_STRING_MAX);
        exit(EXIT_FAILURE);
    }
}

// I didn't change hashing algorithm because
// it will be more complex task then
unsigned int __HashIndex(const char* key) {
    __ValidateKeySize(key);

    unsigned int sum = 0U;

    for (const char* currentCharPtr = key; *currentCharPtr != '\0';
++currentCharPtr) {
/* Correct and explicit unsigned type conversion for arithmetic operation
*/
        sum += (unsigned int)(unsigned char) *currentCharPtr;
    }

/* Modulating to avoid overflow */
    return sum % MAP_MAX;
```

```c
}

Entry* __FirstEntryByHashIndex(const HashMap *map, unsigned int
hashIndex) {
    return (Entry*) map->data[hashIndex];
}

Entry* __HashFindByHashIndexAndKey(const HashMap *map, unsigned int
hashIndex, const char* key) {
    Entry* firstEntry = __FirstEntryByHashIndex(map, hashIndex);

    for (Entry* currentEntry = firstEntry; currentEntry != NULL;
currentEntry = currentEntry->next) {
/* Using of strcmp instead of strcpy */
        if (strcmp(currentEntry->key, key) == 0) {
            return currentEntry;
        }
    }

    return NULL;
}

const char* HashFind(const HashMap *map, const char* key) {
    __ValidateKeySize(key);
    return __HashFindByHashIndexAndKey(map, __HashIndex(key), key)->key;
}

void HashAdd(HashMap *map, const char* key) {
    __ValidateKeySize(key);

    unsigned int hashIndex = __HashIndex(key);
    Entry* existedEntry = __HashFindByHashIndexAndKey(map, hashIndex,
key);
    Entry* firstEntry = __FirstEntryByHashIndex(map, hashIndex);

    if (existedEntry == NULL) {
        Entry* newEntry = malloc(sizeof(Entry));
        //          dst         src
        strcpy(newEntry->key, key);

        // Insert new entry to the head of the list
        map->data[hashIndex] = newEntry;

        // Key with the same hash already exist
        if (firstEntry != NULL) {
            firstEntry->prev = newEntry;
```

```
            newEntry->next = firstEntry;
        }
    }
}

void HashDelete(HashMap *map, const char* key) {
    __ValidateKeySize(key);

    unsigned int hashIndex = __HashIndex(key);
    Entry* existedEntry = __HashFindByHashIndexAndKey(map, hashIndex,
key);

    if (existedEntry != NULL) {
        Entry* existedPrev = existedEntry->prev;
        Entry* existedNext = existedEntry->next;

        // If existed entry is head of list
        if (existedPrev == NULL) {
            map->data[hashIndex] = existedNext;
        } else {
            existedPrev->next = existedNext;
        }

        if (existedNext != NULL) {
            existedNext->prev = existedPrev;
        }

/* Freeing unused object to avoid memory leak */
        free(existedEntry);
    }
}

void HashDump(const HashMap *map) {
    printf("{ ");
    for (unsigned int i = 0; i < MAP_MAX; ++i) {
        for (Entry* currentEntry = map->data[i]; currentEntry != NULL;
currentEntry = currentEntry->next) {
/* Using format string */
            printf("%s ", currentEntry->key);
        }
    }
    printf("}\n");
}
```

- And `hash_fixed.h`

```
/**
*
* @Name : hash.h
*
**/
#ifndef __HASH__
#define __HASH__
/**
HashMap with string-type keys
*/

    typedef struct {
        #define MAP_MAX 128U
       void* data[MAP_MAX];
    } HashMap;

    HashMap* HashInit();

    void HashAdd(HashMap *map, const char* key);

    void HashDelete(HashMap *map, const char* key);

    const char* HashFind(const HashMap *map, const char* key);

    void HashDump(const HashMap *map);

#endif
```

- Below is the file `hardenings.txt`:

```
It's hard to avoid all the risks that are presented by the code because
the code contains many bugs, so
the risks may be avoided by fixing this bugs primarily.

However, I tried to collect some advice that can help to mitigate the
risks.

1. User Account Security:
    Create a user account with limited privileges specifically for
running application:

    ```
    sudo adduser application
    sudo chown application:application hash
    su - application -c ./hash
```
```

~~~~
    ```

2. Make sure your C files and any executables are not globally writable:
    ```
    chmod 755 hash
    chmod 644 hash.c hash.h
    ```

3. Use Access Control Lists:
    Utilize ACLs to fine-tune access permissions for users and groups on
your files and directories.

    ```
    sudo setfacl -m u:applicatoin:rw hash
    ```

4. Enable Memory Protection Features:
    ```
    gcc -o hash hash.c -fstack-protector-strong -D_FORTIFY_SOURCE=2 -O2 -
Wformat -Wformat-security
    ```

    * `-fstack-protector-strong` option enables stack protection
mechanisms to help prevent stack buffer overflow attacks.
    * `-D_FORTIFY_SOURCE=2` option enables additional compile-time and
runtime checks for certain built-in functions to help mitigate the risks
associated with buffer overflow vulnerabilities. Setting _FORTIFY_SOURCE
to 2 activates more aggressive optimizations and checks.
    * `-O2` is an optimization level flag that tells the compiler to
optimize the code.
    * `-Wformat -Wformat-security` are warning flags that enable warnings
related to `printf`-style format specifiers in functions that are called
with format strings.

5. Utilize Address Space Layout Randomization:
    ASLR is enabled by default in modern Linux systems, providing an
additional layer of protection against buffer overflow attacks.

    ```
    cat /proc/sys/kernel/randomize_va_space
    ```

    Ensure it returns `2` (full randomization).

6. Compile-time Hardening:
    Compile with warnings enabled to catch issues early:
~~~~

````
    ```bash
    gcc -Wall -Wextra -Wpedantic -o hash hash.c
    ```

7. Static Code Analysis:
    Integrate a static analysis tool into a development workflow.

    ```bash
    cppcheck hash.c hash.h
    ```

8. Use Isolated Environments:
    Consider using `Docker` to isolate the execution environment for your applications.
    This reduces the risk of your application affecting or being affected by the host OS.
````