Grenoble INP **UGA**
Master CyberSecurity **Year 2024-25**

Software Security & Secure Programming

Written Assignment - Tuesday November the 19th, 2024

Duration: 1h15 – Answers can be written either in English or in French – All documents allowed.

Exercise 1 (∼ 6 pts)

We consider the code given below where:

- global variables `T` and `y` contain **high** (i.e., confidential) values stored in secure memory locations;
- function `foo` is public, its code is visible and its parameter `x` is **low** (i.e., user controlled).

```
1  char T[] ; // confidential
2  int y ;    // confidential
3
4  int foo (int x) {
5    int r=0 ;
6    while (r < x) {
7          if (T[r] > 0)
8             r=x ;
9          else
10            r=r+1 ;
11   }
12   T[y] = r+x ;
13   return r ;
14 }
```

Q1. Tell why function `foo` is **not constant-time**[1], briefly explaining your answer.

Q2. Indicate **precisely** what information an attacker may gain on `T` and `y` when executing `foo` (as many time he/she wants). You will distinguish between two situations:

1. `foo` has not been compiled with Address Sanitizer
2. `foo` has been compiled with Address Sanitizer

Q3. Re-write `foo` as a **constant-time function**, preserving its nominal behavior (i.e., only reducing potential side-channels).

[1] from a confidentiality point of view

Exercise 2 (~ 6 pts)

Two recent hardware-level enhancements have been proposed to mitigate software vulnerabilities targeting the control-flow integrity (CFI) of the program under execution.

Intel CET consists in duplicating the normal execution stack with a **shadow stack** in order to store a copy of the current sequence of function **return-addresses**. This shadow stack being non accessible from the applications it allows to check the integrity of the return address when calling a `ret` instruction.

ARM PAC consists in **signing the pointers** used to access code or data memory locations (including the return addresses). Before dereferencing such pointers their signature is checked to make sure that it has not been altered by an attacker.

Q1. Give a (short!) vulnerable code which **cannot** be exploited thanks to these protections.

Q2. Briefly compare these two solutions in terms of memory and execution time overhead.

Q3. Give a of short example of vulnerable code those exploitation would be prevented by ARM PAC only.

Exercise 3 (~ 8 pts)

The file `vuln.c` (in Appendix A) contains a lightweight C library allowing to create and manage a set of user accounts. Each user account consists in:

- a user id (a strictly positive integer), which **uniquely** identifies a user;
- a boolean value telling if this user is "administrator" or not;
- a user name (possibly non unique, we don't care)
- a sequence of dummy user information called "setting", stored as pairs (index,value), where both index and values are integers.

Account creation can be performed by privileged users only, whereas updating settings can be performed by any users. File `test.c` (in Appendix B) gives an example of code using this library.

Q1. File `vuln.c` contains a **buffer overflow** vulnerability allowing an **unprivileged attacker** to successfully **promote as admin**[2] a user id initially created **as non-admin**. The attacker can **only** use the API functions provided for a non-privileged user.

Give a new version of `test.c` allowing to trigger and exploit this vulnerability, explaining how it works.

Q2. Tell whether each following solutions might help to detect/prevent the exploitation of this vulnerablity, briefly justifying your answer:

1. compiling the code with the *stack protector* option (i.e., adding stack canaries)
2. compiling the code with *Address Sanitizer*
3. ensuring that the stack is *non executable*

Q3. How would you patch `vunl.c` to correct this vulnerability at the source level?

[2] such that function `is_admin()` would return true

Appendix A: `vuln.c`

```
1
2
3
4  #define MAX_USERNAME_LEN 39
5  #define SETTINGS_COUNT 10
6  #define MAX_USERS 100
7
8  // The following type and variables are not accessible to non-privileged users
9
10 typedef struct {
11     long userid;
12     char username[MAX_USERNAME_LEN + 1];
13     long setting[SETTINGS_COUNT];
14     int isAdmin;
15 } user_account;
16
17 // Simulates an internal store of active user accounts
18 user_account *accounts[MAX_USERS];
19
20 // Internal counter of user accounts
21 static int userid_next = 0;
22
23 // The following function can be called by privileged users only
24
25 // Creates a new user account and returns it's unique identifier
26 int create_user_account(int isAdmin, const char *username) {
27     printf("Creating user account for %s ...", username) ;
28     if (userid_next >= MAX_USERS) {
29         fprintf(stderr, "maximum user number exceeded"); return -1;
30     }
31
32     user_account *ua;
33     if (strlen(username) > MAX_USERNAME_LEN) {
34         fprintf(stderr, "username is too long"); return -1;
35     }
36     ua = malloc(sizeof (user_account));
37     if (ua == NULL) {
38         fprintf(stderr, "malloc failed to allocate memory"); return -1;
39     }
40     ua->isAdmin = isAdmin;
41     ua->userid = userid_next++;
42     strcpy(ua->username, username);
43     memset(&ua->setting, 0, sizeof ua->setting); // empty setting
44     accounts[userid_next] = ua;
45     return userid_next;
46 }
47
48
49
50
51
52
```

```
53 // The following function can be called by any user
54
55 // Updates the matching setting for the specified user and returns the status
       of the operation
56 // A setting is some arbitrary string associated with an index as a key
57 int update_setting(int user_id, const char *index, const char *value) {
58     int i, v;
59
60     printf("Updating setting for user id %d ... ", user_id) ;
61     if (user_id < 0 || user_id >= MAX_USERS) {
62         fprintf(stderr, "invalid user id"); return 0;
63     };
64
65     if (value==NULL) {
66         fprintf(stderr, "invalid value setting"); return 0;
67     };
68
69     // convert strings value and index to int
70     v = atoi(value) ; i = atoi(index) ;
71
72     if (index==NULL || i >= SETTINGS_COUNT) {
73         fprintf(stderr, "invalid index setting"); return 0;
74     };
75
76     accounts[user_id]->setting[i] = v;
77     return 1;
78 }
79
80 // Returns whether the specified user is an admin
81 int is_admin(int user_id) {
82     if (user_id < 0 || user_id >= MAX_USERS) {
83         fprintf(stderr, "invalid user id"); return 0;
84     }
85     return accounts[user_id]->isAdmin;
86 }
```

Appendix B: `test.c`

```
1
2
3 int main() {
4     // Creates an admin username called "alice"
5     int user1 = create_user_account(true, "alice");
6     // Creates a non-admin username called "bob"
7     int user2 = create_user_account(false, "bob");
8
9     // Updates the setting '2' of user1 to the number '10'
10    update_setting(user1, "2", "10");
11
12    return 0;
13 }
```