Grenoble INP **UGA**
Master CyberSecurity **Year 2021-22**

Software Security & Secure Programming

Written Assignment - Tuesday November the 30th, 2021

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

This exam contains two exercises which are independent each others.

Exercise 1 (∼ 12 pts, about 40 minutes)

Appendix A is the code of function `grub_username_get()` found in a previous version of the Linux Grub2 bootloader, which happened to contain several vulnerabilities, some of them being exploitable. These vulnerabilities are (classical !) memory errors due to buffer overflows. In the following we assume that:

- Buffer `buf` has been properly allocated before the call, and its length (`buf_size`) is 1024 bytes.
- The attacker controls – as a regular user – the value of the local variable `key` through function `grub_getkey()`. In particular she/he is expected to fill buffer `buf` with its user name, some editing facilities being provided (i.e., backspacing and erasing the whole input).

Q1. Within function `grub_username_get()` buffer `buf` is written at line 29. Obviously (!), this write access may lead to a so-called *off-by-two* error.

1. explain what is meant by *off-by-two* error;
2. how such an error could occur (what should do the attacker to trigger this vulnerability) ?
3. what could be the possible consequences of this error from a security point view (i.e, what could be the attacker *gains*) ?
4. how to rewrite the code in oder to prevent this error ?

Q2. Function `grub_memset`, called at line 34, is similar to the standard function `memset`: it is used here to "clear" (with 0's) the suffix of `buf` which have not been filled by the user in the while loop. The motivation is the following:

> Typing "root" as username, cur_len is 5, and the grub_memset() function will clear (set to zero) bytes form 5 to 1024-5. This way of programming is quite robust. For example, if the typed username is stored in a clean 1024-byte array, then we can compare the whole 1024-bytes with the valid username, rather than comparing both strings. This protects against some short of side-channels attacks, like timing attacks.

Explain what is meant here, and how an attacker could get secret information (like valid usernames) if (classical) string comparison was used later on, without having called `grub_memset()` in this way.

Q3. Unfortunately, this call to function `grub_memset` at line 34 may trigger itself a memory error, writing outside `buf` bounds ... In particular the value of `cur_len` is *attacker controlled.*

1. what happens for instance if the user **immediately** enters **one** backspace followed by a return when running `grub_username_get()` (noticing that `cur_len` is an *unsigned int*, and remembering that in C arithmetic operations on unsigned int are performed modulo 2^{32}, with wrap-around) ?
2. Assuming `buf` address is `0xabcd` what would be the value of `buf + cur_len` ? And the value of `buf_size - cur_len` ?
3. As a consequence, explain how the attacker may overwrite (a part of) the execution stack in a controlled way. Why is this (in general) dangerous and potentially exploitable ? Draw a **picture** of the stack when `grub_memset` is under execution, illustrating how having entered several backspaces may allow to overwrite **critical data** in the stack . . .

Q4. According to the previous questions the attacker may "only" overwrite a part of the execution stack with 0's, which is (hardly) exploitable under regular execution conditions. However, at this early stage of the boot sequence, the IVT (Interrupt Vector Table) resides at address `0x0`, and it contains pointer to security sensitive functions. Moreover:

- There is no memory protection. The whole memory is readable/writable/executable.
- There is no Stack Smashing Protector (SSP).
- There is no Address Space Layout Randomization (ASLR).

Explain if and how these protections would allowed to mitigate this attack.

Q5. How to (slightly !) rewrite the code of `grub_username_get` in order to prevent this vulnerability ?

You can have a look (after the exam !) to the following web page if you want more information about this vulnerability:

`http://hmarco.org/bugs/CVE-2015-8370-Grub2-authentication-bypass.html`

Exercise 2 (∼ 8 pts), about 20 minutes

We consider the following C code, where function `checkKey()` is supposed to check if the public input string `inputKey` is equal to the secret value `secretKey`. This function uses the auxiliary function `equalString` to perform a string comparison. We assume that all buffers are properly allocated, of size `KLENGTH` (no buffer overflows !).

```c
#define KLENGTH 16

char secretKey[KLENGTH] ; // secret value

int equalString(char *s1, char *s2) {
   int i ;
   for (i=0 ; i<KLENGTH ; i++)
        if (s1[i] != s2[i])
              return 0 ;     // s1 and s2 are not equal ...
   return 1 ; // s1 and s2 are equal
}

int checkKey(char *inputKey) {
  if (! equalString(inputKey, secretKey)
     printf("wrong key !\n");
}
```

Q1. When running several times function `checkKey()` with an incorrect `inputKey` (i.e., not equal to `secretKey`) an attacker may get some information about the secret key by measuring the execution times. What is this leaking information ? More precisely, if the secret key is `"xxxxxxxxxxxxxxxx"` and the input key is `"xxxxyyyyyyyyyyyy"` what the attacker may learn ?

Q2. What is the cost (i.e., the maximal number of tries) to guess the correct key using a brute-force attack (without measuring at all the execution times) ? And what would be the cost of a timing attack (exploiting the vulnerability discussed in question Q1) ?

Q3. A *golden rule* to avoid some timing attack is to use the so-called **constant-time** programming paradigm:

> Secret information may only be used as an instruction input if that input has no impact on what resources will be used and for how long.

Explain why functions `equalString` and `checkKey` are not constant-time.

Q4. Rewrite function `equalString` using the constant-time paradigm. Is function `checkKey()` now constant-time ? Is the timing attack still possible ?

Q5. Knowing that this critical code (buffer `secretKey` and function `checkKey()`) is written in C, what would be the possible solution(s) to protect an external **process** to access and/or execute it ? And what about an external **thread** ?

Appendix A - function grub_username_get (Exercise 1)

```c
static int grub_username_get (char buf[], unsigned buf_size) {
  unsigned cur_len = 0;
  int key;

  while (1)
    {
      key = grub_getkey ();
      if (key == '\n' || key == '\r')
        break;

      if (key == '\e')
        {
          cur_len = 0;
          break;
        }

      if (key == '\b')  // backspace key
        {
          cur_len--;
          grub_printf ("\b");
          continue;
        }

      if (!grub_isprint (key))
        continue;

      if (cur_len + 2 < buf_size)
        {
          buf[cur_len++] = key; // Off-by-two !!
          grub_printf ("%c", key);
        }
    }

  grub_memset( buf + cur_len, 0, buf_size - cur_len);

  grub_xputs ("\n");
  grub_refresh ();
  return (key != '\e');
}
```