NUS CS-3235: Computer Security October 23, 2023

Quiz 2

Lecturer: Reza Shokri

***** Indicate **true** or **false** for the following statements and **explain why**. Assume the attacker provides input to a memory unsafe function.

(i) (4 Marks) If the system enables the non-executable stack defense, it's impossible for an attacker to exploit a memory vulnerability to execute the system call `execve("/bin/sh")`.

(ii) (4 Marks)If the system enables the address space layout randomization (ASLR) mechanism, the memory safety is guaranteed.

Answer

(i) False ; The attacker can employ a return-to-libc attack strategy.

(ii) False ; the attacker can still

- Read or write in memory using format string attacks.
- Overwrite the return address, causing system crashes.
- Execute blink ROP attacks.
- If the randomization lacks sufficient entropy, the attacker might guess the correct memory address.

You will only get for the explanation if you mentioned that the attacker can brute force without saying that the entropy of the randomness is low.

* Recall that, in qmail, most modules run as seperate processes under separate user ids (UIDs).

(i) (4 Marks) What is the "security principle" for assigning separate UIDs to each process instead of using the same UID for all?

(ii) (3 Marks) What are the vulnerabilities of using the same UID for all modules?

Answer

(i) The principle of least privilege or privilege separation and isolation (compartmentalization) is the best fit. By assigning unique UIDs, each module operates with only the permissions it needs, reducing the potential for broader system exploitation.

(ii) When every module shares the same UID, compromising one module might jeopardize the entire system. An attacker who gains unauthorized access to one module could potentially manipulate or extract information from any other module.

*

(i) (4 Marks) Consider the permissions for file "`/usr/bin/passwd`":
`-rwsr-xr-x 1 root root 68208 May 28 2020 /usr/bin/passwd`
Please explain why is a common user able to change his password using `passwd`?

(ii) (4 Marks) You want to create a directory where only the file owner and the root can rename or remove the files. How can you achieve this?

Answer

(i) The string "rwsr-xr-x" is read as three groups of three:

- The first group, "rws" says that the owner of this file has full access to the file, and that it is a setuid program which will execute with the owner's identity.
- The second group, "r-x", means that users who belong to this 'group' have read and execute permission, but not write.
- The third group, also "r-x", means that others (those who are neither the owner nor group-members) also can read and execute, but not write.

The permissions showcase the setuid bit (s in -rwsr-xr-x). This means that when a regular user runs the passwd command, it executes with the permissions of its owner, which is root in this case. Thus, even if a user isn't the root, they can change their own password since the command runs with root permissions.

(ii) You should employ the sticky bit. When activated (usually seen as a t in the permissions string), the sticky bit ensures that only the file's owner, the directory's owner, or the root can delete or rename files within that directory. A classic example of this in practice is the /tmp directory.

* Control Flow Integrity (CFI) plays an important role in memory safety, please explain how CFI works by answering the following questions:

(i) (4 Marks) Which of the memory attacks (that we covered in class) can be prevented by CFI? Explain how.

(ii) (4 Marks) Which of the memory attacks (that we covered in class) cannot be prevented by CFI? Explain why.

(iii) Explain the two types of function calls.

- Direct Call (2 Marks)
- Indirect Call (2 Marks)

(iv) (2 Marks) Which type does need protection using CFI? Provide an explanation.

(v) (2 Marks) Where are CFI labels placed, and what is the issue with using one single label for all blocks? Explain why.

Answer

(i) Attacks that modify the flow of the program: code injection attacks, return-oriented programming, buffer overflow.
Explaination: The CFI ensures that these illegal jumps and malicious code insertions are detected and halted.

(ii) Data leak, String Format Attack (read-only), Integer overflow, and DoS. .
Explaination: Attacks like Data Leak, String Format Attack, and Integer Overflow might not directly tamper with the control flow. Thus, CFI might not detect or prevent these attacks. For DoS, even if CFI halts the program, the purpose still achieved

(iii)

- direct calls: A call where the address of the function is known at compile-time.
- indirect calls: A call where the address is determined at runtime, often using function pointers or virtual functions.

(iv) Focus on indirect calls (call, ret, jmp), because direct calls have constant targets and the target address cannot be changed. Thus, we need to check the integrity of indirect calls.

(v) Insert a label before the target address of an indirect call. No, using the same label for all blocks would allow mimicry attacks .

* Consider the following C program. The attacker can supply arbitrary input on stdin. The 64-bit system does not have any memory defense (Note, the CPU is little-endian. Little-endian is an order in which the "little end" (least significant value in the sequence) is stored first. For example, the hexadecimal number `4F52` will be stored as `524F` in memory with little-endian.)

```
1
2  void secret(){
3    int secret = 3235;
4    printf("My secret is %d\n", secret);
5  }
6
7  void vuln(){
8    char name[30]; //  30 bytes
9    int grades;    //  8 bytes
10   int ranking;
11   puts("What's your name?");
12   gets(name);
13 }
14
15 int main() {
16   vuln();
17   puts("Bye Bye");
18   return 0;
19 }
```

The address of secret function is `0xfffff58b`.

(i) (6 Marks) To get the value of the `secret`, what should be the attacker's input? Please write your payload.

(ii) (6 Marks) What technique can be used to prevent the leakage of secret? Describe how it works.

Answer

(i) A*46+0x8b0xf50xff0xff.

(ii) Can use canary to prevent buffer overflow. Insert a canary before the return address. If the attacker overwrites the return address, it will change the value of canary, which can be detected by the program.

* Consider the following C program. The attacker can supply arbitrary input on stdin. Assume the following code is executed on a 64-bit system that does not have memory defense. Recall that, on 64 bit systems, for performance reasons, the first 6 arguments of a function are placed on registers and the rest of the arguments are pushed on the stack in a reverse order.

```
1  void vuln() {
2      int secret = 3539; // 8 bytes
3      char name[32]; // 32 bytes
4      fgets(name, 64, stdin);
5      printf("Hello ");
6      printf(name);
7  }
8
9  int main() {
10     vuln();
11     puts("Bye Bye");
12     return 0;
13 }
```

(i) (5 Marks) What can the adversary provide as the input to learn the value of `secret`?

(ii) (4 Marks) Show how you can modify the code to fix this security bug. Please refer to the code line numbers that you need to modify.

Answer

(i) %10$d or repeat %d 10 times.

(ii) `printf("%s", name)`

* Consider the following C program. The attacker can supply arbitrary input on stdin. Assume the following code is executed on a 32-bit system that does not have memory defense. In this example, `fgets(buf, <size>, file_stream)` reads `<size>`-characters from the file_stream and appends a NULL character (‘\0’) at the last and writes it into `buf`.

```
void vuln(char *buf){
  char tmp[64]; // 64 bytes

  int i = 0;
  while (buf[i] != '\0'){
    tmp[i] = buf[i];
    i++;
  }
}

int main(void)
{
   char buf[256]; // 256 bytes
   fgets(buf,256,stdin);
   vuln(buf);
   printf("Hello world. \n");
   return 0;
}
```

(i) (5 Marks) Can an attacker trick the program to run the system call `execve("/bin/sh")`? Explain your answer.

(ii) (5 Marks) If the stack of the system grows towards higher addresses,(i.e. the opposite of what we discussed in the class), can the attacker trick the program to run the system call `execve("/bin/sh")`? Explain your answer.

Answer

(i) Yes, The adversary can overwrite vuln's return address by overflowing `tmp`.

(ii) No. ; If the stack grows up, then no other state is placed **above** tmp on the stack, so even if the adversary overflows tmp, it will not affect the execution of the program.

* Consider the following C program. The attacker can supply arbitrary input on stdin. The 32-bit system has disabled the non-executable stack defense, the address space layout randomization mechanism.

In this example, `fgets(buf, <size>, file_stream)` reads `<size>` characters from the file_stream and appends a NULL character ('\0') at the last and writes it into `buf`.

```
1  void vuln(char *buf){
2    char tmp[64]; // 64 bytes
3
4    int i = 0;
5    while (buf[i] != '\0'){
6      tmp[i] = buf[i];
7      i++;
8    }
9  }
10
11 int main(void)
12 {
13    char buf[256]; // 256 bytes
14    fgets(buf,256,stdin);
15    vuln(buf);
16    printf("Hello world. \n");
17    return 0;
18 }
```

(i) (5 Marks) If the system uses the terminator stack canary for return addresses, can the attacker trick the program to run the system call `execve("/bin/sh")`? Explain your answer.

(ii) (5 Marks) If the system uses the terminator stack canary for return addresses, can the attacker prevent the program from printing "Hello world. "? Explain your answer.

Answer

(i) No; A terminator stack canary includes a **NULL byte**, and if the adversary overwrites the return address on the stack, the canary value will necessarily not contain any NULL bytes (since otherwise the while loop would have exited).

(ii) Yes; The adversary could simply overwrite the canary value, which will terminate the program as vuln return

* (20 pt) In the following code there are a few independent buffer overflow bugs.

```
1  typedef struct {
2    char name[20];
3    int id;
4    int price;
5  } item_info_t;
6
7
8  int fun(int *ids, int *prices, char **names, int no_items) {
9    char str[100];
10   item_info_t *item_info;
11   int i;
12   for (i = 0; i < no_items; i++) {
13     item_info = malloc(sizeof(item_info_t));
14     snprintf(str, sizeof(str), names[i]);
15     str[sizeof(str) - 1] = 0;
16     printf("Buffer size is: (%d) \nData input: %s \n", strlen(str),
       str) ;
17
18     item_info->id = ids[i];
19     item_info->price = prices[i];
20     strcpy(item_info->name, str);
21
22     printf("Item (ID=%d), %s costs %d\n", item_info->id, item_info->
       name, item_info->price);
23
24     memset(item_info, 0, sizeof(&item_info));
25     free(item_info);
26   }
27 }
28
29
30 int main(int argc, char *argv[]) {
31   int i, idx, ids[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
32   int prices[10] = {1, 10, 2, 15, 20, 1, 4, 19, 23, 14};
33   char *names[10];
34
35   for (i = 0; i < sizeof(ids); i++) {
36     printf("%d ", ids[i]);
37   }
38   printf("\n");
39
40   /* Assume names are read from an arbitrary file */
41
42   fun(ids, prices, names, 10);
43
44   return 0;
45 }
```

(i) (10 Marks) Indicate those (and only those) line numbers which have a bug (at least four). Specify the type(s) of bug(s) and the attack that can exploit it.

(ii) (10 Marks) Propose 1 line fixes for each bug you spotted in (i), indicating which line numbers your fixes modify. Your fixes should not change the intended functionality of the program other than the buffer overflow errors.

Answer

(i) Line 14 (fmt strg), 20 (buffer overflow), 24 (Null deref due to memset) , 35-36 (buff overflow due to sizeof).

(ii) Use the correct function.
Line 14: `snprintf(str, sizeof(str), ``%s'', names[i]);`
line 35: `sizeof(ids) / sizeof(int)`
Line 20: use `strncpy`
Line 25: `free(item_info); item_info = NULL;`