Homework #1 Race Condition Vulnerability and Dirty CoW

Problem 1:

Race conditions are a class of software bugs that occur when the behavior of a program depends on the relative timing of events, such as the order in which threads or processes execute. These issues can lead to unpredictable and unintended behavior, as multiple threads or processes attempt to access shared resources concurrently.

Race conditions can arise when security-critical process occurs in stages. Attacker makes changes between stages often, between stages that gives authorization, but before stages that transfers ownership. Race conditions may be more prevalent than buffer overflows but race conditions are harder to exploit. If a program has race condition, the attacker may affect the program output by influencing the related events.

Race conditions happen when multiple processes/threads access and manipulate the **same** data concurrently. The outcome of the execution depends on a particular order.

Two common types of race conditions are general race conditions and TOCTTOU (Time-Of-Check To Time-Of-Use) race conditions.

1. **General Race Conditions:**

Problem: In a general race condition, multiple threads or processes access shared resources concurrently without proper synchronization. This can lead to unpredictable outcomes as the order of execution determines the final result.

Exploitation: An attacker might exploit a general race condition by manipulating the timing of their actions to interfere with the normal flow of the program. For example, they might try to access a resource in between the time it is checked and the time it is used, leading to unexpected behavior.

2. **TOCTTOU Race Conditions:**

Problem: TOCTTOU race conditions occur when checking for a condition before using a resource. Time of Check to Time of Use (TOCTTOU) race conditions occur in computer systems when a program's behavior depends on the state of a resource at two different times. This

vulnerability arises when the state of a resource is checked (Time of Check, TOC) and then used or operated on (Time of Use, TOU) at a later time, but the resource's state has changed between the check and the use.

Exploitation: The race conditions can be exploited in the following ways:

1. Choose a target file: Add the following line to /etc/passwd to add a new user – test:U6aMy0wojraho:0:0:test:/root:/bin/bash
2. Launch attack (Run the vulnerable program): Two processes that race against each other: vulnerable process and attack process
   – Run the vulnerable program in an infinite loop
   – password_input is the string to be inserted in /etc/passwd
3. Monitor the result: Check if the password is modified
4. Run the exploit: Start the race, win the race and verify if we have the root.

Race Condition Vulnerabilities:

1. Data Corruption: Concurrent access to shared data without proper synchronization can result in data corruption. If multiple threads write to the same location simultaneously, the final state of the data may be unpredictable and incorrect.

2. Deadlocks: Improperly synchronized access to resources can lead to deadlocks, where multiple threads or processes are stuck waiting for each other to release resources. This can result in a system-wide halt.

3. Security Vulnerabilities: Race conditions can introduce security vulnerabilities, such as TOCTTOU issues, where an attacker can manipulate the timing of their actions to gain unauthorized access to resources or escalate privileges.

Exploitation Techniques:

Timing Attacks: Exploiting the timing differences between checks and uses to manipulate the program's behavior.

Resource Exhaustion: Creating a large number of threads or processes to increase the likelihood of a race condition occurring.

Data Manipulation: Modifying shared data between the check and use phases to achieve unintended results.

To mitigate race conditions, proper synchronization mechanisms, such as locks or semaphores, should be employed to ensure that only one thread or process can access critical sections of code at a time. Additionally, using atomic operations and transactions can help address these vulnerabilities.

Problem 2:

For problem 2, we have been asked if the given Set-UID program has a race condition problem or not. The answer for this part is yes, it does have a race condition problem. The attack window is between lines 2 and 3.

In order for this to succeed, the real user should have the access. The line code 3 is going to fail.

```
1   filename = "/tmp/XYZ";
2   fd = open (filename, O_RDWR);
3   status = access (filename, W_OK);
    ...
    ... (code omitted) ...
    ...
10  if (status == ACCESS_ALLOWED) {
11     write_to_file(fd);
12  } else {
13     fprintf(stderr, "Permission denied\n");
14  }
```

The race condition arises because there is a time gap between checking the existence of the file using access() and opening it using open(). During this time gap, the file may be created or deleted by another process or thread, leading to a potential inconsistency.

Exploiting the race condition typically involves the following steps:

1. Check if the file exists: Before the access() call, the file may not exist. An attacker could create a symbolic link with the same name ("/tmp/xyz") pointing to a sensitive file that the attacker wants to read or modify.

2. Symbolic link attack: If an attacker can create a symbolic link pointing to a critical file during the time window between access() and open(), they could potentially manipulate the target file. For example, they could point the symbolic link to a file with sensitive information, and the subsequent open() call would operate on that file instead of the intended one.

Let's look at an example of how an attacker might exploit this:

```
# In one terminal (attacker)
ln -s /etc/passwd /tmp/xyz

# In another terminal (victim's code)
fd = open(filename, O_RDWR);  # The open call follows the symbolic link
status = access(filename, W_OK);  # The access call checks the symlink, not
```

In this example, the attacker creates a symbolic link /tmp/xyz pointing to /etc/passwd. The victim's code, during the race condition window, opens the symbolic link, leading to potential unauthorized access to sensitive information in /etc/passwd.

Problem 3:

```
int main()
{
   struct stat stat1, stat2;
   int fd1, fd2;

   if (access("/tmp/XYZ", O_RDWR)) {
      fprintf(stderr, "Permission denied\n");
      return -1;
   }
   else fd1 = open("/tmp/XYZ", O_RDWR);

   if (access("/tmp/XYZ", O_RDWR)) {
      fprintf(stderr, "Permission denied\n");
      return -1;
   }
   else fd2 = open("/tmp/XYZ", O_RDWR);

   The program then checks whether fd1 and fd2 refer to
   the same file, if so, the program will write to
   fd1 (or fd2). Otherwise, the program will do nothing
   and exit.
}
```

For this problem, there are 3 race conditions. Although, there is an assumption that we are starting with a regular file that the user has permission. The Access part uses real permission whereas open part uses effective permission. First, we start with access and then we encounter open, the open might now have the permission to access the same file hence; this was the first race condition. When we move along, we see another access, just like before this access has real permission but then upcoming open might not have the same file access and open has effective permission so the send race condition happens. Similarly, when we encounter a third access and open block, the third race condition will happen just like the same way it did last two times.

Problem 4:

On analyzing the code:

*char *addr = (char *)map;*

printf("%s\n", map + 8);

I can see that, the content of the memory-mapped file is "Florida_State_University," and the memory address map points to the beginning of this content, adding 8 to map means we are skipping the first 8 characters. Therefore, the printf statement will start printing characters from the 9th position of the original string. Since, the given statement is:

"Florida_State_University"

it will be printed from the 9th character which in this case is State_University. Hence, the printf statement will print:

State_University

Problem 5:

The Dirty COW (Copy-On-Write) vulnerability is a race condition that existed in the Linux Kernel. The vulnerability, officially identified as CVE-2016-5195, was discovered in 2007 and gained significant attention due to its potential impact on the security of Linux systems. It affects all Linux based operating systems, including Android. The Dirty COW vulnerability allows an attacker to exploit a race condition in the memory management subsystem of the Linux Kernel to gain write access to read-only memory mappings. This, in turn, enables the attacker to modify read-only data, potentially leading to privilege escalation and unauthorized access.

Here's a step-by-step explanation of how the Dirty COW vulnerability is exploited:

1. Copy-On-Write Mechanism:

- In a typical Copy-On-Write scenario, when a process requests a copy of a piece of memory, the operating system allocates a new block of memory and marks it as copy-on-write. This means that as long as the memory remains unchanged, the processes sharing this memory region can refer to the same physical memory pages.

2. Race Condition Exploitation:

- The vulnerability arises due to a race condition in the way the Linux Kernel handles the Copy-On-Write mechanism. Specifically, it involves a race between the time a page is marked as read-only and the time it is actually copied.

3. Triggering the Race Condition:

- An attacker starts by creating multiple threads or processes that simultaneously access the same read-only memory mapping.

4. Memory Page Flag Manipulation:

- The attacker, through one of the threads, manipulates the page tables to mark the targeted memory page as read-only.

5. Race Window:

- There exists a brief window of time between marking the page as read-only and the actual copying of the page. During this window, other threads or processes may still have read access to the page.

6. Write Access to Read-Only Memory:

- While the page is still being shared and before the copy process completes, the attacker exploits this window to write malicious data into the supposedly read-only memory.

7. Privilege Escalation:

- By modifying critical data structures or executing malicious code in the context of a privileged process (such as a setuid root program), the attacker can gain elevated privileges.

8. Root Privilege:

- Once the attacker has successfully modified critical data structures or executed malicious code, they can potentially gain root privileges, allowing them to control the system and access sensitive information.

It's important to note that the Dirty COW vulnerability has been patched in subsequent Linux Kernel updates. Therefore, keeping the operating system and software up-to-date is crucial for maintaining a secure system. Additionally, this case underscores the importance of addressing race conditions in software development to prevent security vulnerabilities.

Problem 6:

While the textbook might specifically highlight the /etc/passwd file as an example, in a Unix-based system, there are other critical files that, if manipulated, could lead to gaining root privileges. Here are two additional files that attackers might target:

1. **/etc/shadow:**
   Explanation: The /etc/shadow file stores encrypted user passwords. In a typical Unix-based system, user passwords are not stored directly in /etc/passwd for security reasons. Instead, the hashed passwords and other security-related information are stored in /etc/shadow. If an attacker gains write access to /etc/shadow and can manipulate the stored password hashes, they may be able to replace the root password hash with one of their choosing, effectively granting them root privileges.

2. **/etc/sudoers:**
   Explanation: The /etc/sudoers file determines which users are allowed to run specific commands with elevated privileges using the sudo command. If an attacker can modify the sudoers file, they may grant themselves or another user unrestricted sudo access, allowing them to execute arbitrary commands with root privileges without providing a password. This can be a powerful avenue for privilege escalation.

In both cases, the key is to target files that are crucial for user authentication and authorization or files that control elevated privileges. Successful manipulation of these files during a Dirty COW race condition exploit could lead to unauthorized access and potential root privilege escalation. As always, it's essential to apply security best practices, keep software and systems updated, and implement proper access controls to mitigate such vulnerabilities.