Race Condition Vulnerability

S7.1. Does the following `Set-UID` program have a race condition vulnerability?

```
if (!access("/etc/passwd", W_OK)) {
    /* the real user has the write permission*/
    f = open("/tmp/X", O_WRITE);
    write_to_file(f);
}
else {
   /* the real user does not have the write permission */
   fprintf(stderr, "Permission denied\n");
}
```

S7.2. Assume we develop a new system call, called `faccess(int fd, int mode)`, which is identical to `access()`, except that the file to be checked is specified by the file descriptor `fd`. Does the following program have a race condition problem?

```
int f = open("/tmp/x", O_WRITE);
if (!faccess(f, W_OK)) {
    write_to_file(f)
} else {
    close(f);
}
```

S7.3. How many race conditions do attackers have to win in the following program?

```
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.
}
```

S7.4. In the `open()` system call, it first checks whether the user has the required permission to access the target file, then it actually opens the file. There seems to be a check-and-then-use pattern. Is there a race condition problem caused by this pattern?

S7.5. The least-privilege principle can be used to effectively defend against the race condition attacks discussed in this chapter. Can we use the same principle to defeat buffer-overflow attacks? Why or why not? Namely, before executing the vulnerable function, we disable the root privilege; after the vulnerable function returns, we enable the privilege back.

S7.6. The following root-owned `Set-UID` program needs to write to a file, but it wants to ensure that the file is owned by the user. It uses `stat()` to get the file owner's ID, and compares it with the real user ID of the process. If they do not match, the program will exit. Please describe whether there is a race condition in the program? If so, please explain how you can exploit the race condition. The manual of `stat()` can be found online.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main()
{
  struct stat statbuf;
  uid_t real_uid;
  FILE* fp;

  fp = fopen("/tmp/XYZ", "a+");
  stat("/tmp/XYZ", &statbuf);

  printf("The file owner's user ID: %d\n", statbuf.st_uid);
  printf("The process's real user ID: %d\n", getuid());

  // Check whether the file belongs to the user
  if (statbuf.st_uid == getuid()) {
    printf("IDs match, continue to write to the file.\n");
    // write to the file ...
    if (fp) fclose(fp);
  } else {
    printf("IDs do not match, exit.\n");
    if (fp) fclose(fp);
    return -1;
  }

  return 0;
}
```

S7.7. The following root-owned `Set-UID` program needs to write to a file, but it wants to

ensure that the file is owned by the user. It uses `fstat()` to get the file owner's ID, and compares it with the real user ID of the process. If they do not match, the program will exit. Please describe whether there is a race condition in the program? If so, please explain how you can exploit the race condition. The manual of `fstat()` and `fileno()` can be found online.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main()
{
  struct stat statbuf;
  uid_t real_uid;
  FILE* fp;

  fp = fopen("/tmp/XYZ", "a+");
  fstat(fileno(fp), &statbuf);

  printf("The file owner's user ID: %d\n", statbuf.st_uid);
  printf("The process's real user ID: %d\n", getuid());

  // Check whether the file belongs to the user
  if (statbuf.st_uid == getuid()) {
    printf("IDs match, continue to write to the file.\n");
    // write to the file ...
    if (fp) fclose(fp);
  } else {
    printf("IDs do not match, exit.\n");
    if (fp) fclose(fp);
    return -1;
  }

  return 0;
}
```

S7.8. If we can lock a file, we can solve the race condition problem by locking a file during the check-and-use window, because no other process can use the file during the time window. Why don't we use this approach to solve the race condition problems discussed in this chapter?

S7.9. Does the following privileged `Set-UID` program have a race condition problem? If so, where is the attack window? Please also describe how you would exploit this race condition window.

```
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  }
```

S7.10. Please use the least-privilege principle to fix the race condition problem in the following program.

```
if (access("/tmp/XYZ", W_OK) == ACCESS_ALLOWED) {
    f = open("/tmp/XYZ", O_WRITE);
    write_to_file(f);
}
else {
   fprintf(stderr, "Permission denied\n");
}
```