Software Security Assignment 2 - Race Conditions

Mats Authen

September 16, 2014

1 Introduction

1.1 About

In this assignment, we were tasked with imagining an electronic ID solution, and thinking of at least two possible race conditions, and their effects, as well as countermeasures for them. My scenario is an online store, which operates with its own form of "currency", or points, which users first buy, then spend on merchandise.

1.2 Technical description

Race conditions occur when several threads or parallel processes attempt to modify a shared value at the same time.
This can become a serious problem in the case that one process or thread preforms a check to find out if the variable has the right value, and then does a specific action with the variable if the result of the check satisfies a certain condition. If another process or thread modifies the variable after the check, but before the action, the first process/thread will assume that the required condition is still met, and perform the action on the variable with the wrong value, possibly creating problems in the system.
Example:

```
1 if (i == 1) {
2         i += 9; // i should have the value of 10
3 }
4 printf("%d = 10!\n", i);
```

In the example above, another thread may have modified the variable i after the if, but before the arithmetic operation. In this case, the output may not be "10 = 10!", but could be something else, like "1337 = 10!".

2 Case one

2.1 The problem

Imagine that the following C++ code is running in the store's server, handling purchases. A user cannot buy anything that he/she cannot afford, because the store isn't allowed to set the users balance lower than 0:

```
1  void purchase(User user, Product product) {
2          int user_balance = user.balance();
3          int cost = product.get_product_cost();
4
5          if (user_balance >= cost) {
6                  user.new_balance(user_balance - cost);
7                  give_store_lots_of_money(cost);
8                  user.add_product(product);
9          }
10         else {
11                 too_low_user_balance(user);
12         }
13 }
```

Now imagine a user with evil intentions (Mallory). Mallory will devise a way to request multiple purchases simultaneously. Imagine that all the parallel threads run the if-condition and then allows the next one take over. All the processes will believe that Mallory is only making one purchase, and that he can afford it. In reality, he can afford one, but only one. What happens next is that Mallory's balance is set to an illegal value (he doesn't care since he has already paid the initial balance only), the store is awarded with more credits than there are, and Mallory gets all the products.

2.2 The fix

The simplest way to fix this would be to make sure that no users can request more than one purchase at a time. Use mutex or another implementation of a binary semaphore to lock the critical sector.
It would also be a good idea to make the new_balance() function safer, by having it preform its own check and returning an error, instead of blindly changing the balance.

3 Case two

3.1 The problem

Another scenario is a comment section on the product sites of the store. All comments are numbered internally to keep track of them. The number that

each of the comment is assigned acts like that comment's unique ID. Each time a user makes a comment, a new thread handles it, and uses the shared variable for next comment number.

```
1 void comment(User user) {
2         Comment* comment = new Comment(nr, user);
3         nr++;
4         add_comment(comment);
5 }
```

Imagine that the variable nr is 42. If a user attempts to make a comment, this function will be called in a new thread. It creates the new comment as shown on line 2. But before it can progress to line 3, where nr is updated, another user attempts to make a comment. The function is called again in a new thread, the new thread runs line 2 *before* the previous one can run line 3. The result is two comments, both with 42 as ID. The nr variable will now be 44, because it has been updated twice, but there will be no comments with an ID of 43, which may cause problems if one where to for instance attempt to iterate through all the comments. In addition, all actions that are supposed to be preformed on one of these comments will be preformed on both. If the author of one of them attempts to delete his comment, the other one may be deleted as well, because they have the same ID.

3.2 The fix

Use a mutex or another implementation of a binary semaphore to lock the critical region here, thus preventing two or more users to use this function to make a comment at the same time.
In addition, using a global variable like this to keep track of the IDs is generally a bad idea. But in reality one would simply use a database for these comments, and in MySQL there are security measures in place to prevent race conditions when assigning IDs with auto_increment.