December 21, 2011

Practical Race Condition Vulnerabilities in Web Applications

What are Race Conditions?

314

Race conditions in software are when two concurrent threads of execution access a shared resource in a way that unintentionally produces different results depending on the time at which the code is executed. For example, a multi-threaded program may spawn 2 threads that have access to the same location in memory. Thread #1 might store the value 300 in this location and expect it to still be 300 a few instructions later. Since thread #2 is executing at the same time as thread #1, thread #2 may overwrite the memory location with another value while thread #1 still expects it to be 300. Sometimes it will happen, sometimes it will not. It depends if thread #2 is "lucky enough" to execute just after thread #1 wrote the value 300.

Another example: Suppose Alice, Bob, and Carol have accounts at an online banking service. Alice, Bob, and Carol each have $100 in their accounts. Alice and Bob both transfer $10 to Carol at the exact same time. The web server starts processing Alice's transaction first. The server checks that Alice has sufficient funds in her account then gets Carol's balance into a local variable and adds $10. The next step would be to deduct $10 from Alice's account and then store Carol's updated balance in the database, but the web server decides to pause the execution of Alice's transaction and work on Bob's. Bob's transaction runs to completion, decrementing his balance $10 and incrementing Carol's balance by $10 (remember that Carol's balance in the database was still $100, Alice's request hasn't changed it yet). Now Carol has $110 in her account. The web server goes back to processing Alice's transaction. The variable that holds Carol's updated balance still has the value $110, so the web server sets Carol's balance to $110 unaware that Bob's transaction had already added $10 to Carol's account, and Carol ends up with $110 but Alice and Bob each lost $10.

You wouldn't normally think that a web application written in PHP (which doesn't even have multi-threading) would be vulnerable to race condition attacks, but they can be.

CPU Time Sharing

Computer programs seem to execute simultaneously, but unless you have multiple processors or a multi-core CPU, they actually don't. What really happens is that the operating system shares the CPU time between processes and threads. The OS will start some process executing, then a few microseconds later will let some other process use the CPU. After that process has been executing for a few microseconds, yet another process gets its turn to use the CPU. Doing this shares the CPU with all running processes, and makes it look like everything is running at the same time. Wikipedia has an excellent article on computer multitasking.

314

Online Bank Account Example

Consider the following PHP code for withdrawing money or credits from some kind of online account. Suppose getBalance() and setBalance() are functions that load/store an account balance from a MySQL database.

```php
function withdraw($amount)
{
   $balance = getBalance();
   if($amount <= $balance)
   {
       $balance = $balance - $amount;
       echo "You have withdrawn: $amount";
       setBalance($balance);
   }
   else
   {
       echo "Insufficient funds.";
   }
}
```

Just get the balance, see if there is enough money in the account, and if so, proceed with the transaction and deduct the amount from the balance. Easy, right? Wrong. This function, even though it executes in less than half of a millisecond on my PC, is vulnerable to a race condition attack that allows an attacker to withdraw funds without decreasing the balance. The attack is practical and can even be executed over the internet.

Proof of Concept

314

To demonstrate the attack, I implemented the withdraw function using a 1-entry MySQL table to hold the balance, and setup a PHP script that accepts a URL of the form `poc.php?wd=10` to withdraw the value specified in the URL (in this case, 10). I then wrote a python script to make 128 simultaneous requests to the URL:

```
import os
os.fork() #2
os.fork() #4
os.fork() #8
os.fork() #16
os.fork() #32
os.fork() #64
os.fork() #128
print os.popen('php -r ' + \
                '"echo file_get_contents(\'http://defuse.ca/poc.php?wd=10\');"').read()
```

Executing this script will make 128 requests to withdraw $10. If all 128 requests work properly, and the balance starts out at $10,000, we expect the balance to be $10,000 - 128 * $10 = $8720 after the script finishes executing. But due to the race condition, there's actually more money left in the account than there should be.

Table 1. Actual result of running the Python script.

Trial #	Balance After Python Script Execution	Profit (actual - expected)
1	8900	+180
2	8830	+110
3	8810	+90

4	8880	+160

As you can see, if this were an actual online financial service (e.g. Paypal), I'd be able to make hundreds of dollars in only a few seconds. Just to be thorough, I can verify that $1280 was actually withdrawn by looking at the output of the Python script:

```
$ grep "You have withdrawn: 10" test2  | wc -l
128
```

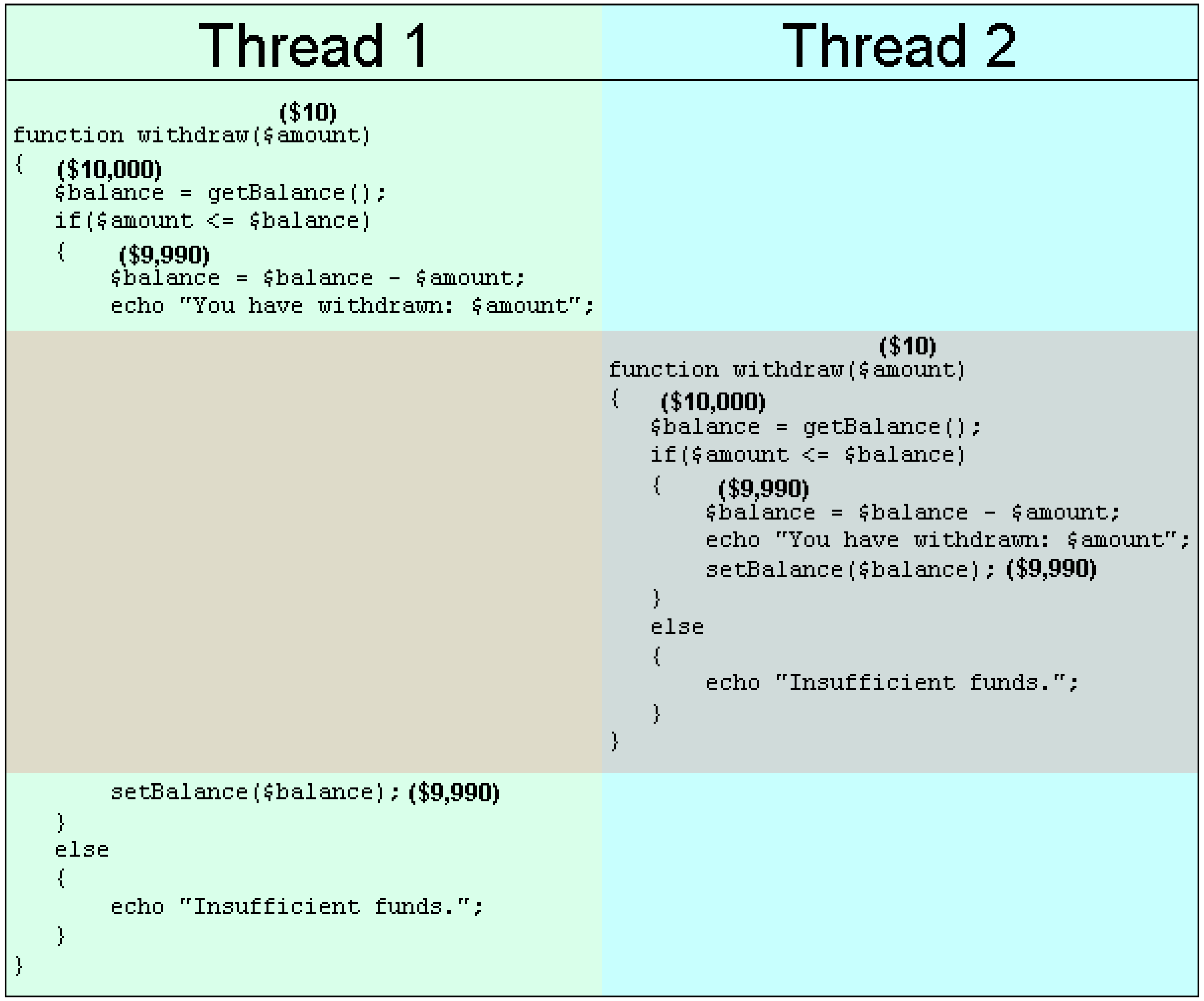
314

These tests were done using the VPS that hosts this website (Debian 6, running Apache 2.2.16) to host the withdraw script and my Debian 6 PC at home to run the Python script. I've also confirmed that the attack works just as well when the withdraw script is hosted on the same PC that executes the Python script. It works when they're separated by a 100mbps Ethernet LAN, too. You can download all of the proof of concept files here.

What's going on here?

This attack is possible because web servers like Apache process queries asynchronously. That means, if two requests come in at nearly the same time, they'll get executed at the same time (on a multi-core CPU) or their execution will be interleaved by the operating system's CPU time sharing system. The result is that a few of the requests end up being processed like this:

314



```
                                       ($10)
function withdraw($amount)
{   ($10,000)
    $balance = getBalance();
    if($amount <= $balance)
    {     ($9,990)
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
```

```
                                       ($10)
function withdraw($amount)
{   ($10,000)
    $balance = getBalance();
    if($amount <= $balance)
    {     ($9,990)
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
        setBalance($balance); ($9,990)
    }
    else
    {
        echo "Insufficient funds.";
    }
}
```

```
        setBalance($balance); ($9,990)
    }
    else
    {
        echo "Insufficient funds.";
    }
}
```

After the two requests are processed, the balance should be $9980, but since the second request is processed while the first is still being processed, we end up with a balance of $9990. Both withdraws work (imagine that the echo statement was replaced by code to increment another account's balance), so $20 is withdrawn but only $10 is deducted from the balance.

Solutions

Random Delay

Adding a random delay to the withdraw script reduces the effectiveness of the attack, but doesn't prevent it. The following code can be used to add a random delay of up to 10 seconds before processing the withdraw:

```
$seconds = rand(1, 10);
$nanoseconds = rand(100, 1000000000);
time_nanosleep($seconds, $nanoseconds);
```

314

This makes the attack harder, but not impossible:

Table 2. Results with random delay.

Trial #	Balance After Python Script Execution	Profit (actual - expected)
1	8730	+10
2	8720	0
3	8720	0
4	8730	+10

Don't use this method. Unless you use a CSPRNG to generate the random delay time, an attacker can probably extract the weak PRNG's state, compute the future delay times, and delay his own queries so the withdraw function executes at the same time. The attacker can also increase the number of simultaneous queries to improve the chances of the withdraws occurring simultaneously. The long delay will annoy users, too.

System V Semaphore

If PHP is compiled with `--enable-sysvsem` then it will have System V-like semaphore abilities. We can make the withdraw function secure using the sem_get, sem_acquire, and sem_release functions:

314

```
function withdraw($amount)
{
  $sem = sem_get(1234, 1);
  if (sem_acquire($sem))
  {
      $balance = getBalance();
      if($amount <= $balance)
      {
          $balance = $balance - $amount;
          echo "You have withdrawn: $amount <br />";
          setBalance($balance);
      }
      else
      {
          echo "Insufficient funds.";
      }
       sem_release($sem);
   }
}
```

In the call to sem_get, "1234" is the semaphore key (identifier) and "1" is the maximum number of threads/processes that can "acquire" the semaphore at once. If a thread has acquired semaphore 1234 but hasn't yet released it, and another thread calls sem_acquire on the same semaphore, the call will block until the first thread releases it. This prevents the race condition attack.

You'll probably not want to use '1234' as the semaphore key. It would be better to, for example, use the ID of the user whose balance is being modified. Also, keep in mind that the semaphore functions are not universally supported. The following code can be used to emulate the semaphore functions on systems that do not support them (source):

```
<?php
if ( !function_exists('sem_get') ) {
   function sem_get($key) { return fopen(__FILE__.'.sem.'.$key, 'w+'); }
   function sem_acquire($sem_id) { return flock($sem_id, LOCK_EX); }
```

```
  function sem_release($sem_id) { return flock($sem_id, LOCK_UN); }
}
?>
```

314

I've combined the System V semaphore functions and the file lock method into a single cross-platform Mutex class that you can download here.

Table 3. Results with System V semaphore.

Trial #	Balance After Python Script Execution	Profit (actual - expected)
1	8720	0
2	8720	0
3	8720	0
4	8720	0

Atomic Database Operations

If you are using an ACID-compliant database, which guarantees atomicity, then the critical sections can be implemented as atomic database queries. Beware, however, of TOCTTOU bugs: any code not inside a database query will still be subject to race conditions. You really do need to be careful of what guarantees your database provides! If implemented correctly, it is almost certainly better than using a lock.

Thanks to Tiemo Kieft for pointing me to this solution.

Multiple Servers

If queries are being processed by more than one server, the System V semaphore method won't prevent the attack. Different semaphores will exist on each server so they won't help if an attacker floods each server with queries. To implement a multi-server mutex you'll need to give each server a common file system and use the file lock method, or use the locking mechanism provided by your database software.

Useful Links

- "On Race Condition Vulnerabilities in Web Applications" - A 2008 paper on nearly the same subject.
- Stack Overflow - PHP Mutual Exclusion
- The proof of concept code used in this article
- Wikipedia - Time Of Check To Time Of Use (TOCTTOU)
- Wikipedia - Computer Multitasking
- https://www.owasp.org/index.php/Testing_for_Race_Conditions_%28OWASP-AT-010%29

314