12

RACE CONDITIONS

Race conditions are one of the most interesting vulnerabilities in modern web applications. They stem from simple programming mistakes developers often make, and these mistakes have proved costly: attackers have used race conditions to steal money from online banks, e-commerce sites, stock brokerages, and cryptocurrency exchanges.

Let's dive into how and why these vulnerabilities happen, and how you can find them and exploit them.

Mechanisms

A *race condition* happens when two sections of code that are designed to be executed in a sequence get executed out of sequence. To understand how this works, you need to first understand the concept of concurrency. In computer science, *concurrency* is the ability to execute different parts of a program simultaneously without affecting the outcome of the program. Concurrency can drastically improve the performance of programs because different parts of the program's operation can be run at once.

Concurrency has two types: multiprocessing and multithreading. *Multiprocessing* refers to using multiple *central processing units (CPUs)*, the hardware in a computer that executes instructions, to perform simultaneous computations. On the other hand, *multithreading* is the ability of a single CPU to provide multiple *threads*, or concurrent executions. These threads don't actually execute at the same time; instead, they take turns using the CPU's computational power. When one thread is idle, other threads can continue taking advantage of the unused computing resources. For example, when one thread is suspended while waiting for user input, another can take over the CPU to execute its computations.

Arranging the sequence of execution of multiple threads is called *scheduling*. Different systems use different scheduling algorithms, depending on their performance priorities. For example, some systems might schedule their tasks by executing the highest-priority tasks first, while another system might execute its tasks by giving out computational time in turns, regardless of priority.

This flexible scheduling is precisely what causes race conditions. Race conditions happen when developers don't adhere to certain safe concurrency principles, as we'll discuss later in this chapter. Since the scheduling algorithm can swap between the execution of two threads at any time, you can't predict the sequence in which the threads execute each action.

To see why the sequence of execution matters, let's consider an example (courtesy of Wikipedia: *https://en.wikipedia.org/wiki/Race_condition*). Say two concurrent threads of execution are each trying to increase the value of a global variable by 1. If the variable starts out with a value of 0, it should end up with a value of 2. Ideally, the threads would be executed in the stages shown in Table 12-1.

Table 12-1: Normal Execution of Two Threads Operating on the Same Variable

	Thread 1	Thread 2	Value of variable A
Stage 1			0
Stage 2	Read value of A		0
Stage 3	Increase A by 1		0
Stage 4	Write the value of A		1
Stage 5		Read value of A	1
Stage 6		Increase A by 1	1
Stage 7		Write the value of A	2

But if the two threads are run simultaneously, without any consideration of conflicts that may occur when accessing the same resources, the execution could be scheduled as in Table 12-2 instead.

Table 12-2: Incorrect Calculation Due to a Race Condition

	Thread 1	Thread 2	Value of variable A
Stage 1			0
Stage 2	Read value of A		0
Stage 3		Read value of A	0
Stage 4	Increase A by 1		0
Stage 5		Increase A by 1	0
Stage 6	Write the value of A		1
Stage 7		Write the value of A	1

In this case, the final value of the global variable becomes 1, which is incorrect. The resulting value should be 2.

In summary, race conditions happen when the outcome of the execution of one thread depends on the outcome of another thread, and when two threads operate on the same resources without considering that other threads are also using those resources. When these two threads are executed simultaneously, unexpected outcomes can occur. Certain programming languages, such as C/C++, are more prone to race conditions because of the way they manage memory.

When a Race Condition Becomes a Vulnerability

A race condition becomes a vulnerability when it affects a security control mechanism. In those cases, attackers can induce a situation in which a sensitive action executes before a security check is complete. For this reason, race condition vulnerabilities are also referred to as *time-of-check* or *time-of-use* vulnerabilities.

Imagine that the two threads of the previous example are executing something a little more sensitive: the transfer of money between bank accounts. The application would have to perform three subtasks to transfer the money correctly. First, it has to check if the originating account has a high enough balance. Then, it must add money to the destination account. Finally, it must deduct the same amount from the originating account.

Let's say that you own two bank accounts, account A and account B. You have $500 in account A and $0 in account B. You initiate two money transfers of $500 from account A to account B at the same time. Ideally, when two money transfer requests are initiated, the program should behave as shown in Table 12-3.

Table 12-3: Normal Execution of Two Threads Operating on the Same Bank Account

	Thread 1	Thread 2	Balance of accounts A + B
Stage 1	Check account A balance ($500)		$500
Stage 2	Add $500 to account B		$1,000 ($500 in A, $500 in B)
Stage 3	Deduct $500 from account A		$500 ($0 in A, $500 in B)
Stage 4		Check account A balance ($0)	$500 ($0 in A, $500 in B)
Stage 5		Transfer fails (low balance)	$500 ($0 in A, $500 in B)

You end up with the correct amount of money in the end: a total of $500 in your two bank accounts. But if you can send the two requests simultaneously, you might be able to induce a situation in which the execution of the threads looks like Table 12-4.

Table 12-4: Faulty Transfer Results Due to a Race Condition

	Thread 1	Thread 2	Balance of accounts A + B
Stage 1	Check account A balance ($500)		$500
Stage 2		Check account A balance ($500)	$500
Stage 3	Add $500 to account B		$1,000 ($500 in A, $500 in B)
Stage 4		Add $500 to account B	$1,500 ($500 in A, $1,000 in B)
Stage 5	Deduct $500 from account A		$1,000 ($0 in A, $1,000 in B)
Stage 6		Deduct $500 from account A	$1,000 ($0 in A, $1,000 in B)

Note that, in this scenario, you end up with more money than you started with. Instead of having $500 in your accounts, you now own a total of $1,000. You made an additional $500 appear out of thin air by exploiting a race condition vulnerability!

Although race conditions are often associated with financial sites, attackers can use them in other situations too, such as to rig online voting systems. Let's say an online voting system performs three subtasks to process an online vote. First, it checks if the user has already voted. Then, it adds a vote to the vote count of the selected candidate. Finally, it records that that user has voted to prevent them from casting a vote again.

Say you try to cast a vote for candidate A twice, simultaneously. Ideally, the application should reject the second vote, following the procedure in Table 12-5.

Table 12-5: Normal Execution of Two Threads Operating on the Same User's Votes

	Thread 1	Thread 2	Votes for candidate A
Stage 1			100
Stage 2	Check whether the user has already voted (they haven't)		100
Stage 3	Increase candidate A's vote count		101
Stage 4	Mark the user as Already Voted		101
Stage 5		Check whether the user has already voted (they have)	101
Stage 6		Reject the user's vote	101

But if the voting application has a race condition vulnerability, execution might turn into the scenario shown in Table 12-6, which gives the users the power to cast potentially unlimited votes.

Table 12-6: User Able to Vote Twice by Abusing a Race Condition

	Thread 1	Thread 2	Votes for candidate A
Stage 1			100
Stage 2	Check whether the user has already voted (they haven't)		100
Stage 3		Check whether the user has already voted (they haven't)	100
Stage 4	Increase candidate A's vote count		101
Stage 5		Increase candidate A's vote count	102
Stage 6	Mark the user as Already Voted		102
Stage 7		Mark the user as Already Voted	102

An attacker can follow this procedure to fire two, ten, or even hundreds of requests at once, and then see which vote requests get processed before the user is marked as Already Voted.

Most race condition vulnerabilities are exploited to manipulate money, gift card credits, votes, social media likes, and so on. But race conditions can also be used to bypass access control or trigger other vulnerabilities. You can read about some real-life race condition vulnerabilities on the HackerOne Hacktivity feed (*https://hackerone.com/hacktivity?querystring=race%20condition/*).

Prevention

The key to preventing race conditions is to protect resources during execution by using a method of *synchronization*, or mechanisms that ensure threads using the same resources don't execute simultaneously.

Resource locks are one of these mechanisms. They block other threads from operating on the same resource by *locking* a resource. In the bank transfer example, thread 1 could lock the balance of accounts A and B before modifying them so that thread 2 would have to wait for it to finish before accessing the resources.

Most programming languages that have concurrency abilities also have some sort of synchronization functionality built in. You have to be aware of the concurrency issues in your applications and apply synchronization measures accordingly. Beyond synchronization, following secure coding practices, like the principle of least privilege, can prevent race conditions from turning into more severe security issues.

The *principle of least privilege* means that applications and processes should be granted only the privileges they need to complete their tasks. For example, when an application requires only read access to a file, it should not be granted any write or execute permissions. You should grant applications precisely the permissions that they need instead. This lowers the risks of complete system compromise during an attack.

Hunting for Race Conditions

Hunting for race conditions is simple. But often it involves an element of luck. By following these steps, you can make sure that you maximize your chances of success.

Step 1: Find Features Prone to Race Conditions

Attackers use race conditions to subvert access controls. In theory, any application whose sensitive actions rely on access-control mechanisms could be vulnerable.

Most of the time, race conditions occur in features that deal with numbers, such as online voting, online gaming scores, bank transfers, e-commerce payments, and gift card balances. Look for these features in an application and take note of the request involved in updating these numbers.

For example, let's say that, in your proxy, you've spotted the request used to transfer money from your banking site. You should copy this request to use for testing. In Burp Suite, you can copy a request by right-clicking it and selecting **Copy as curl command**.

Step 2: Send Simultaneous Requests

You can then test for and exploit race conditions in the target by sending multiple requests to the server simultaneously.

For example, if you have $3,000 in your bank account and want to see if you can transfer more money than you have, you can simultaneously send multiple requests for transfer to the server via the `curl` command. If you've copied the command from Burp, you can simply paste the command into your terminal multiple times and insert a `&` character between each one. In the Linux terminal, the `&` character is used to execute multiple commands simultaneously in the background:

```
curl (transfer $3000) & curl (transfer $3000) & curl (transfer $3000)
& curl (transfer $3000) & curl (transfer $3000) & curl (transfer $3000)
```

Be sure to test for operations that should be allowed once, but not multiple times! For example, if you have a bank account balance of $3,000, testing to transfer $5,000 is pointless, because no single request would be allowed. But testing a transfer of $10 multiple times is also pointless, since you should be able to do that even without a race condition. The key is to test the application's limits by executing operations that should not be repeatable.

Step 3: Check the Results

Check if your attack has succeeded. In our example, if your destination account ends up with more than a $3,000 addition after the simultaneous requests, your attack has succeeded, and you can determine that a race condition exists on the transfer balance endpoint.

Note that whether your attack succeeds depends on the server's process-scheduling algorithm, which is a matter of luck. However, the more requests you send within a short time frame, the more likely your attack will succeed. Also, many tests for race conditions won't succeed the first time, so it's a good idea to try a few more times before giving up.

Step 4: Create a Proof of Concept

Once you have found a race condition, you will need to provide proof of the vulnerability in your report. The best way to do this is to lay out the steps needed to exploit the vulnerability. For example, you can lay out the exploitation steps like so:

1. Create an account with a $3,000 balance and another one with zero balance. The account with $3,000 will be the source account for our transfers, and the one with zero balance will be the destination.
2. Execute this command:

   ```
   curl (transfer $3000) & curl (transfer $3000) & curl (transfer $3000)
   & curl (transfer $3000) & curl (transfer $3000) & curl (transfer $3000)
   ```

   This will attempt to transfer $3,000 to another account multiple times simultaneously.
3. You should see more than $3,000 in the destination account. Reverse the transfer and try the attack a few more times if you don't see more than $3,000 in the destination account.

Since the success of a race condition attack depends on luck, make sure you include instructions to try again if the first test fails. If the vulnerability exists, the attack should succeed eventually after a few tries.

Escalating Race Conditions

The severity of race conditions depends on the impacted functionality. When determining the impact of a specific race condition, pay attention to how much an attacker can potentially gain in terms of monetary reward or social influence.

For example, if a race condition is found on a critical functionality like cash withdrawal, fund transfer, or credit card payment, the vulnerability could lead to infinite financial gain for the attacker. Prove the impact of a race condition and articulate what attackers will be able to achieve in your report.

Finding Your First Race Condition!

Now you're ready to find your first race condition. Follow these steps to manipulate web applications using this neat technique:

1. Spot the features prone to race conditions in the target application and copy the corresponding requests.
2. Send multiple of these critical requests to the server simultaneously. You should craft requests that should be allowed once but not allowed multiple times.
3. Check the results to see if your attack has succeeded. And try to execute the attack multiple times to maximize the chance of success.
4. Consider the impact of the race condition you just found.
5. Draft up your first race condition report!