Race Conditions

⋮ 22/07/2022

Jul 21, 2022

6 min read
Race-Co Web-Notes

Race Condition

References

RACE CONDITION BUGS!

PayloadsAllTheThings/Race Condition at master · swisskyrepo/PayloadsAllTheThings

GitHub - ngalongc/bug-bounty-reference: Inspired by https://github.com/djadmin/awesome-bug-bounty, a list of bug bounty write-up that is categorized by the bug nature

Race Condition

GitHub - alexbieber/Bug_Bounty_writeups: BUG BOUNTY WRITEUPS - OWASP TOP 10 🔴🔴🔴🔴✔

Race-Condition Vulnerability Lab

pentest-guide/Race-Condition at master · Voorivex/pentest-guide

Writeups Bug Bounty hackerone

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.

Mechanism

A race condition happens when two sections of code that are designed to be executed in a sequence get executed out of sequence.

In computer science, concurrency is the ability to execute different parts of a program simultaneously without affecting the outcome of the program.

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.

	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

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		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

Incorrect Calculation Due to a Race Condition

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

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 This picture.

	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)

Normal Execution of Two Threads Operating on the Same Bank Account

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 this:

	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)

Faulty Transfer Results Due to a Race Condition

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

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/).

`Hunting for Race Conditions`

Hunting for race conditions is simple. But often it involves an element of luck.

Step 1: Find Features Prone to Race Condition

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

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)
```

Just don't test for 5000 transfer request and you don't have this balance 😃

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.

Step 4: Create a Proof of Concept

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.

Escalating Race Conditions

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!

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!

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`.

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 applies synchronization measures accordingly.

`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.