Race conditions

They occur when websites process requests concurrently without adequate safeguards. This can lead to multiple distinct threads interacting with the same data at the same time, resulting in a "collision" that causes unintended behavior in the application

The period of time during which a collision is possible is known as the "race window". This could be the fraction of a second between two interactions with the database

Limit overrun race conditions

The most well-known type of race condition enables you to exceed some kind of limit imposed by the business logic of the application the application transitions through a temporary sub-state; that is, a state that it enters and then exits again before request processing is complete.This introduces a small race window during which you can repeatedly claim the discount as many times as you like.

- Bypassing an anti-brute-force rate limit
- Redeeming a gift card multiple times
- Rating a product multiple times
- Withdrawing or transferring cash in excess of your account balance
- Reusing a single CAPTCHA solution

Limit overruns are a subtype of so-called "time-of-check to time-of-use" (TOCTOU) flaws

Detecting and exploiting limit overrun race conditions with Burp Repeater

- Identify a single-use or rate-limited endpoint that has some kind of security impact or other useful purpose
- Issue multiple requests to this endpoint in quick succession to see if you can overrun this limit

The primary challenge is timing the requests so that at least two race windows line up, causing a collision. This window is often just milliseconds and can be even shorter.

The single-packet attack enables you to completely neutralize interference from network jitter by using a single TCP packet to complete 20-30 requests simultaneously.

Detecting and exploiting limit overrun race conditions with Turbo Intruder

To use the single-packet attack in Turbo Intruder:

1. Ensure that the target supports HTTP/2. The single-packet attack is incompatible with HTTP/1.
2. Set the `engine=Engine.BURP2` and `concurrentConnections=1` configuration options for the request engine.
3. When queueing your requests, group them by assigning them to a named gate using the `gate` argument for the `engine.queue()` method.
4. To send all of the requests in a given group, open the respective gate with the `engine.openGate()` method.

```python
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=1,
                           engine=Engine.BURP2
                           )

    # queue 20 requests in gate '1'
    for i in range(20):
        engine.queue(target.req, gate='1')

    # send all requests in gate '1' in parallel
    engine.openGate('1')
```

the `race-single-packet-attack.py` template provided in Turbo Intruder's default examples directory.

Hidden multi-step sequences

In practice, a single request may initiate an entire multi-step sequence behind the scenes, transitioning the application through multiple hidden states that it enters and then exits again before request processing is complete

If you can identify one or more HTTP requests that cause an interaction with the same data, you can potentially abuse these sub-states to expose time-sensitive variations of the kinds of logic flaws that are common in multi-step workflows.

Methodology for Hidden multi-step Sequence:

Predict potential collisions

After mapping out the target site as normal, you can reduce the number of endpoints that you need to test by asking yourself the following questions:

- Is this endpoint security critical?

- Is there any collision potential?

Probe for clues

To recognize clues, you first need to benchmark how the endpoint behaves under normal conditions. You can do this in Burp Repeater by grouping all of your requests and using the **Send group in sequence (separate connections)** option.

Next, send the same group of requests at once using the single-packet attack (or last-byte sync if HTTP/2 isn't supported) to minimize network jitter

Prove the concept

Try to understand what's happening, remove superfluous requests, and make sure you can still replicate the effects.

Advanced race conditions can cause unusual and unique primitives, so the path to maximum impact isn't always immediately obvious. It may help to think of each race condition as a structural weakness rather than an isolated vulnerability.

Multi-endpoint race conditions

Perhaps the most intuitive form of these race conditions are those that involve sending requests to multiple endpoints at the same time.

LAB

try to place item after checkout window in parallel to cause the system to believe that the item is also needed to be checkout but due to the speed in parallel connection is goes on without a check the item is bought at a negative price.

Aligning multi-endpoint race windows

This common problem is primarily caused by the following two factors:

- Delays introduced by network architecture
- Delays introduced by endpoint-specific processing

Connection warming

Back-end connection delays don't usually interfere with race condition attacks because they typically delay parallel requests equally, so the requests stay in sync.

One way to do this is by "warming" the connection with one or more inconsequential requests to see if this smoothes out the remaining processing times.

If the first request still has a longer processing time, but the rest of the requests are now processed within a short window, you can ignore the apparent delay and continue testing as normal.

Abusing rate or resource limits

Web servers often delay the processing of requests if too many are sent too quickly. By sending a large number of dummy requests to intentionally trigger the rate or resource limit, you may be able to cause a suitable server-side delay. This makes the single-packet attack viable even when delayed execution is required.

Single-endpoint race conditions

Sending parallel requests with different values to a single endpoint can sometimes trigger powerful race conditions.

Consider a password reset mechanism that stores the user ID and reset token in the user's session.sending two parallel password reset requests from the same session, but with two different usernames, could potentially cause the following collision

Email address confirmations, or any email-based operations, are generally a good target for single-endpoint race conditions.

Session-based locking mechanisms

Some frameworks attempt to prevent accidental data corruption by using some form of request locking. For example, PHP's native session handler module only processes one request per session at a time

It's extremely important to spot this kind of behavior as it can otherwise mask trivially exploitable vulnerabilities. If you notice that all of your requests are being processed sequentially, try sending each of them using a different session token.

Partial construction race conditions

Many applications create objects in multiple steps, which may introduce a temporary middle state in which the object is exploitable.

the way for exploits whereby you inject an input value that returns something matching the uninitialized database value, such as an empty string, or `null` in JSON, and this is compared as part of a security control.

Time-sensitive attacks

Sometimes you may not find race conditions, but the techniques for delivering requests with precise timing can still reveal the presence of other vulnerabilities.

Consider a password reset token that is only randomized using a timestamp. In this case, it might be possible to trigger two password resets for two different users, which both use the same token. All you need to do is time the requests so that they generate the same timestamp.

How to prevent race condition vulnerabilities

- Avoid mixing data from different storage places.
- Ensure sensitive endpoints make state changes atomic by using the datastore's concurrency features. For example, use a single database transaction to check the payment matches the cart value and confirm the order.
- As a defense-in-depth measure, take advantage of datastore integrity and consistency features like column uniqueness constraints.
- Don't attempt to use one data storage layer to secure another. For example, sessions aren't suitable for preventing limit overrun attacks on databases.
- Ensure your session handling framework keeps sessions internally consistent. Updating session variables individually instead of in a batch might be a tempting optimization, but it's extremely dangerous. This goes for ORMs too; by hiding away concepts like transactions, they're taking on full responsibility for them.
- In some architectures, it may be appropriate to avoid server-side state entirely. Instead, you could use encryption to push the state client-side, for example, using JWTs. Note that this has its own risks, as we've covered extensively in our topic on JWT attacks.