TryHackMe Advent of Cyber 2025

Day 20 Challenge Report

Race Condition Exploitation & Web Application Concurrency

1. Executive Summary

This report documents the completion of Day 20 of the TryHackMe Advent of Cyber 2025 event. The challenge focused on race condition vulnerabilities in web applications-specifically how concurrent HTTP requests can exploit timing gaps to bypass inventory controls and process multiple transactions simultaneously. Successfully exploited TBFC shopping cart application using Burp Suite's parallel request capabilities, purchased more items than available stock, pushed inventory to negative values, and retrieved both challenge flags. Demonstrated practical understanding of Time-of-Check to Time-of-Use (TOCTOU) vulnerabilities, shared resource conflicts, atomicity violations, and mitigation strategies including database transaction locks, idempotency keys, and rate limiting.

2. Understanding Race Conditions

2.1 What is a Race Condition?

A race condition occurs when two or more actions happen simultaneously, and the system's outcome depends on the unpredictable order in which they complete. In web applications, this vulnerability manifests when multiple users or automated requests simultaneously access or modify shared resources-such as inventory counts, account balances, or transaction records-without proper synchronization mechanisms. If concurrent access isn't properly controlled, the result can be duplicate transactions, oversold inventory items, unauthorized data modifications, or inconsistent database states.

Think of a race condition like two cashiers trying to sell the last item in a store simultaneously. Both check the inventory system at the same moment, both see one item available, and both process their sales. The result? Two customers believe they purchased the last item, but only one actually exists. The race condition created an impossible state-negative inventory. This exact scenario plays out thousands of times daily in e-commerce platforms that don't implement proper concurrency controls.

2.2 Real-World Impact

Race condition vulnerabilities have caused significant real-world incidents:

- **Financial Loss:** E-commerce platforms allowing orders beyond available stock, forcing companies to fulfill orders at a loss or face customer lawsuits.
- **Double Spending:** Banking applications processing duplicate withdrawals from the same account balance, resulting in fraudulent overdrafts.
- **Privilege Escalation:** Authentication systems granting admin access to regular users due to race conditions in permission checks.
- **Data Corruption:** Concurrent updates overwriting each other, leading to inconsistent or invalid database states that corrupt critical business data.

3. Types of Race Conditions

3.1 Time-of-Check to Time-of-Use (TOCTOU)

TOCTOU race conditions occur when a program checks a condition first, then uses the result of that check later-but the underlying data changes between the check and the use. The check becomes stale before the action executes, causing the system to operate on invalid assumptions.

Example Scenario:

User A checks inventory: 1 SleighToy available. User B checks inventory: 1 SleighToy available (same moment). User A proceeds to checkout-stock becomes 0. User B proceeds to checkout-stock becomes -1 (impossible state). Both users received confirmation emails, but only one toy exists physically.

Code Example (Vulnerable):

```
// Check stock if (product.stock > 0) {     // Time gap here - race condition window!
// Another request might modify stock     product.stock -= 1;     createOrder(); }
```

The vulnerability exists in the time gap between checking stock (product.stock > 0) and using that information (product.stock -= 1). This gap might be milliseconds, but that's sufficient for concurrent requests to exploit the condition.

3.2 Shared Resource Conflicts

Shared resource race conditions happen when multiple users or processes simultaneously modify the same data without proper locking mechanisms. Since both modifications happen concurrently, the final result depends on which operation completes last-creating unpredictable outcomes known as 'last-write-wins' scenarios.

Example Scenario:

Admin updates product price to $50. Marketing simultaneously updates product description. Both modifications hit the database at the same moment. Depending on timing, either: (1) Price updates to $50 but description doesn't change, or (2) Description updates but price reverts to old value. One update silently overwrites the other-data loss without error messages.

Database Perspective:

```
// Request 1 UPDATE products SET price = 50 WHERE id = 123;  // Request 2 (simultaneously)
UPDATE products SET description = 'New' WHERE id = 123;  // Without transaction isolation,
results unpredictable
```

3.3 Atomicity Violations

Atomic operations should complete entirely as a single unit-either fully executed or not at all (all-or-nothing principle). Atomicity violations occur when multi-step operations run separately, allowing other requests to interfere between steps, causing inconsistent states.

Example Scenario:

User purchases item: Step 1 executes-charge credit card ($100 deducted). Gap occurs-another request changes product price to $200. Step 2 executes-create order record with new $200 price. Result: Customer charged $100 but order shows $200, creating accounting nightmare.

The Problem:

Multi-step processes (charge card, update inventory, create order, send confirmation) must be atomic. If any step can be interrupted by concurrent requests, the system enters inconsistent states. Database transactions solve this by wrapping all steps in BEGIN TRANSACTION...COMMIT blocks that either succeed completely or roll back entirely.

4. Challenge Environment Setup

4.1 Configuring Firefox with FoxyProxy

1. Opened Firefox browser
2. Clicked FoxyProxy icon in toolbar
3. Selected 'Burp' profile to route all traffic through proxy

4.2 Launching Burp Suite

4. Clicked Burp Suite icon on Desktop
5. Selected 'Temporary project in memory' (no file saving)
6. Clicked 'Start Burp' to launch application
7. Navigated to Proxy > Intercept and disabled interception (button shows 'Intercept off')

Critical Step: Disabling intercept allows requests to pass through Burp Suite without manual forwarding while still logging all HTTP traffic for analysis. This is essential for capturing the checkout request we'll exploit later.

5. Making Legitimate Request

5.1 Accessing TBFC Shopping Cart

Navigated to http://10.66.151.203 in Firefox. Login page appeared with authentication form.

Credentials Used:

- Username: attacker
- Password: attacker@123

5.2 Product Selection

After authentication, main dashboard displayed limited-edition products:

- **SleighToy Limited Edition:** Only 10 units available
- **Bunny Plush (Blue):** Limited stock

5.3 Normal Checkout Process

1. Clicked 'Add to Cart' for SleighToy Limited Edition
2. Clicked 'Checkout' button to proceed to checkout page
3. Clicked 'Confirm & Pay' to finalize purchase
4. Received success message confirming order

Important: This legitimate request was critical because Burp Suite captured the exact HTTP request (headers, cookies, POST parameters) sent to /process_checkout endpoint. We'll replay this request multiple times simultaneously to exploit the race condition.

6. Exploiting the Race Condition

6.1 Capturing the Checkout Request

1. Switched to Burp Suite application
2. Navigated to Proxy > HTTP history tab
3. Located POST request to /process_checkout endpoint
4. Right-clicked request → 'Send to Repeater'

This action copied the complete HTTP request-including session cookies, CSRF tokens, product IDs, and authentication headers-into Burp's Repeater tool for manipulation and replay.

6.2 Creating Request Group

1. Switched to Repeater tab (request now visible)
2. Right-clicked first tab → 'Add tab to group'
3. Clicked 'Create tab group'
4. Named group 'cart' for organization

6.3 Duplicating Requests

1. Right-clicked request tab inside 'cart' group
2. Selected 'Duplicate tab'
3. Entered '15' when prompted for number of copies

Result: 15 identical tabs, each containing the exact same POST request to /process_checkout. Each tab represents a separate order attempt for the same product.

6.4 Parallel Request Execution

1. Used Repeater toolbar 'Send' dropdown menu
2. Selected 'Send group in parallel (last-byte sync)'
3. Clicked 'Send group (parallel)'

Technical Explanation: 'Last-byte sync' is critical. Burp holds all 15 connections open, sends 14.9999 of each request, then releases all final bytes simultaneously. This maximizes timing overlap, forcing all 15 requests to hit the server's stock-checking code at the exact same microsecond-the optimal condition for exploiting TOCTOU vulnerabilities.

6.5 Exploitation Results

The server attempted to process all 15 checkout requests simultaneously. Each request executed the vulnerable code path:

```
Request 1: Check stock (10 available) → Proceed Request 2: Check stock (10 available) → Proceed Request 3: Check stock (10 available) → Proceed ... Request 15: Check stock (10 available) → Proceed  All requests passed stock check!  Request 1: Deduct stock → 9 remaining Request 2: Deduct stock → 8 remaining ... Request 15: Deduct stock → -5 remaining
```

Outcome: All 15 orders confirmed successfully despite only 10 items existing. Stock pushed to negative value (-5). System accepted impossible state without error.

7. Verification & Flags

7.1 SleighToy Limited Edition

Refreshed TBFC shopping cart web application. Dashboard now displayed:

- Multiple confirmed orders for SleighToy
- Stock count: Negative value
- **Flag displayed: THM{WINNER_OF_R@CE007}**

7.2 Bunny Plush (Blue)

Repeated identical exploitation process for Bunny Plush:

1. Added Bunny Plush to cart
2. Proceeded to checkout
3. Captured /process_checkout request in Burp
4. Duplicated request 15 times
5. Sent group in parallel with last-byte sync
6. Refreshed page to verify negative stock and retrieve flag

Flag: THM{WINNER_OF_Bunny_R@ce}

8. Mitigation Strategies

8.1 Atomic Database Transactions

Implementation:

```
BEGIN TRANSACTION;   SELECT stock FROM products WHERE id = 123 FOR UPDATE;   -- Lock prevents other transactions   UPDATE products SET stock = stock - 1;   INSERT INTO orders ...; COMMIT;
```

The 'FOR UPDATE' clause locks the row, preventing concurrent transactions from reading the same stock value. All operations execute atomically-either all succeed or all roll back.

8.2 Final Stock Validation

Implementation:

```
BEGIN TRANSACTION;   current_stock = SELECT stock FROM products WHERE id = 123;   IF current_stock < 1 THEN     ROLLBACK;     RETURN 'Out of stock';   END IF;   UPDATE products SET stock = stock - 1; COMMIT;
```

Double-check stock immediately before committing transaction. Even if initial check passed, validate again at commit time to catch race conditions.

8.3 Idempotency Keys

Implementation:

```
// Client generates unique key per checkout POST /checkout Idempotency-Key: uuid-12345  // Server tracks processed keys IF idempotency_key EXISTS IN processed_requests THEN   RETURN cached_response; ELSE   process_checkout();   STORE idempotency_key; END IF;
```

Idempotency keys ensure duplicate requests (intentional retries or attacker replay) return the same result without re-processing. Critical for payment systems

.

8.4 Rate Limiting & Concurrency Controls

Implementation options:

- **Per-User Rate Limiting:** Maximum 3 checkout attempts per minute per user
- **Per-Session Locks:** Only one active checkout per session (queue subsequent requests)
- **Request Throttling:** Artificial delays between requests from same IP/session

9. Key Skills Developed

- Race condition theory and categorization
- TOCTOU vulnerability identification
- Burp Suite Repeater advanced features
- Parallel request execution techniques
- Last-byte synchronization understanding
- Web application concurrency testing
- Database transaction design principles
- Mitigation strategy implementation

10. Conclusion

Day 20 of the TryHackMe Advent of Cyber 2025 provided comprehensive hands-on training in race condition exploitation and web application concurrency vulnerabilities. Successfully identified and exploited Time-of-Check to Time-of-Use (TOCTOU) vulnerabilities in TBFC's shopping cart system, demonstrating how parallel HTTP requests can bypass inventory controls through timing manipulation.

The challenge highlighted critical security lessons: race conditions aren't theoretical-they're actively exploited in production systems causing financial loss and data inconsistency. Proper synchronization mechanisms (database transactions, row-level locking, idempotency keys) are essential, not optional. Testing with single requests isn't sufficient-applications must be stress-tested with concurrent load to identify timing vulnerabilities. Burp Suite's parallel request capabilities provide powerful testing tools for security researchers. Understanding the gap between check and use in code is critical for identifying TOCTOU vulnerabilities during code review.

Challenge Status: **COMPLETED ✓ - Both Flags Retrieved!**