4 SecOps Playbook: Umfassende Runtime Defense Strategie für Supabase und zKalkulator

SecOps Playbook: Umfassende Runtime Defense Strategie für Supabase und zKalkulator

1. Strategische Einordnung und Architektur der Active Defense

1.1 Paradigmenwechsel: Von passiver Sicherheit zu Active Runtime Defense

In der modernen Webarchitektur, insbesondere bei Serverless- und Edge-Computing-Plattformen wie Supabase, reicht der traditionelle Perimeterschutz nicht mehr aus. Das Projekt **zKalkulator** operiert in einer Umgebung, in der die Grenzen zwischen Client, Edge und Datenbank fließend sind. Statische Sicherheitsmaßnahmen wie Firewalls und statische Code-Analysen (SAST) sind notwendig, aber gegen dynamische Laufzeitangriffe – wie Fuzzing, Credential Stuffing oder Business Logic Abuse – oft machtlos. Dieses Playbook definiert daher den Übergang zu einer **Active Runtime Defense**.

Active Defense bedeutet in diesem Kontext, dass die Systemkomponenten (Deno Edge Functions und die PostgreSQL-Datenbank) nicht nur passiv Anfragen verarbeiten, sondern als aktive Sensoren und Aktoren im Sicherheitsgefüge agieren. Das System muss in der Lage sein, feindseliges Verhalten in Echtzeit zu erkennen, den Angreifer durch Täuschung (Deception) zu binden oder gezielt zu blockieren und den Vorfall manipulationssicher zu protokollieren.[1] Die Architektur stützt sich dabei auf drei verteidigte Zonen, die tiefgestaffelt (Defense in Depth) agieren.

Die erste Verteidigungslinie liegt am **Network Edge**. Hier werden hochvolumige, automatisierte Angriffe durch Deno Edge Functions abgefangen, bevor sie teure Datenbankressourcen binden oder Geschäftslogik manipulieren können. Die zweite Linie ist die **Database Active Layer**. Da Supabase PostgreSQL als zentrales Nervensystem nutzt, implementieren wir hier Logik zur Erkennung von Anomalien, die den Edge passiert haben – beispielsweise langsame, "Low-and-Slow" Enumerationsangriffe. Die dritte Linie ist der **Audit & Compliance Core**, der durch kryptografische Verkettung sicherstellt, dass Beweise für Angriffe nicht von kompromittierten Administratorenkonten gelöscht werden können.[2]

1.2 Bedrohungslandschaft und Risikoanalyse für zKalkulator

Die spezifische Architektur von zKalkulator auf Supabase exponiert mehrere Angriffsvektoren, die durch Standardmaßnahmen nicht abgedeckt sind. Die Analyse der Bedrohungslandschaft zeigt eine hohe Wahrscheinlichkeit für automatisierte Angriffe auf die Geschäftslogik.

Ein primäres Risiko stellt das **Sequential Scanning (Fuzzing)** dar. Angreifer nutzen die Vorhersehbarkeit von Identifikatoren (IDs) oder API-Endpunkten, um systematisch Datensätze zu extrahieren. Wenn zKalkulator beispielsweise sequenzielle Integer-IDs für Berechnungen oder Nutzerprofile verwendet (z.B. /api/calc/1001, /api/calc/1002), ermöglicht dies eine triviale Enumeration des gesamten Datenbestands.[3] Selbst bei Verwendung von UUIDs können Angreifer durch Timing-Analysen oder Fehlermeldungen ("User not found" vs. "Permission denied") Informationen über die Existenz von Ressourcen gewinnen (User Enumeration).[4]

Ein weiteres signifikantes Risiko sind **Bot-Netzwerke und Scraping**. Da zKalkulator potenziell wertvolle Berechnungsdaten oder Preisalgorithmen hostet, werden Bots versuchen, diese Logik massenhaft abzufragen. Dies führt nicht nur zum Abfluss von geistigem Eigentum, sondern auch zu "Denial of Wallet" (DoW) Angriffen. Da Serverless-Funktionen und Datenbankzugriffe bei Supabase nutzungsbasiert abgerechnet werden, kann ein Angreifer durch Millionen von legitimen Anfragen immense Kosten verursachen, ohne eine klassische Schwachstelle auszunutzen.[1]

Schließlich besteht das Risiko der **Insider Bedrohung und Vertuschung**. In vielen Organisationen haben Entwickler oder Administratoren weitreichende Rechte auf der Datenbank. Ein Angreifer, der solche Credentials erbeutet (oder ein böswilliger Insider), könnte versuchen, Spuren seines Zugriffs in

den audit_logs zu löschen. Ohne eine kryptografische Verankerung der Logs ist die Integrität der Forensik nicht gewährleistet.[6]

2. Erste Verteidigungslinie: Edge Defense & Deno Runtime

Die Verteidigung am Edge ist die effizienteste Methode, um Angriffe abzuwehren, da sie die Latenz für den Angreifer erhöht und die Last von der zentralen Infrastruktur fernhält. Supabase Edge Functions, basierend auf der Deno Runtime, bieten hierfür die ideale Plattform.[8]

2.1 Implementierung von Stateful Rate Limiting mit Deno KV

Herkömmliche Rate-Limiting-Ansätze auf Basis von IP-Adressen in einer einzelnen Server-Instanz sind in einer Serverless-Umgebung wirkungslos, da der Zustand (State) zwischen den Funktionsaufrufen nicht persistent ist. Externe Speicher wie Redis sind oft performant, fügen aber Latenz und Komplexität hinzu. Für zKalkulator nutzen wir **Deno KV**, einen in die Runtime integrierten Key-Value-Store, der globale Konsistenz und atomare Transaktionen bietet.[9]

Algorithmische Wahl: Sliding Window vs. Fixed Window

Die Wahl des Algorithmus ist entscheidend für die Effektivität. Ein einfaches **Fixed Window** (z.B. "max. 100 Requests pro Minute") setzt den Zähler zu festen Zeitpunkten zurück (z.B. 12:00:00, 12:01:00). Intelligente Angreifer nutzen dies aus, indem sie ihre Anfragen an die Fenstergrenzen legen (z.B. 100 Requests um 12:00:59 und 100 um 12:01:01). Dies führt zu einer Lastspitze von 200 Requests innerhalb von zwei Sekunden, die vom System durchgelassen wird.[11]

Für zKalkulator implementieren wir daher ein **Sliding Window Log** oder einen approximierten Sliding Window Counter. Bei diesem Verfahren bewegt sich das Zeitfenster kontinuierlich mit dem aktuellen Zeitpunkt. Stellen Sie sich einen Zeitstrahl vor, auf dem jeder Request als Punkt markiert wird. Das "Fenster" ist ein Rahmen fester Breite (z.B. 60 Sekunden), der sich über diesen Zeitstrahl schiebt. Wenn ein neuer Request eintrifft, prüft das System nicht, wie viele Requests "in der aktuellen Kalenderminute" passiert sind, sondern wie viele Requests im Zeitraum [Jetzt - 60s, Jetzt] liegen. Requests, die zeitlich aus dem Fenster herausfallen, werden nicht mehr gezählt. Dies glättet Lastspitzen effektiv und verhindert "Burst"-Attacken an den Zeitgrenzen.[12]

Technische Implementierung mit Deno KV

Deno KV ermöglicht es uns, diese Logik performant umzusetzen, da es atomare Operationen unterstützt. Das bedeutet, wir können den Zähler lesen, prüfen und inkrementieren, ohne dass Race Conditions bei parallelen Requests auftreten.[10] Im Vergleich zu Redis (z.B. Upstash) zeigt Deno KV zwar in einigen Benchmarks höhere Latenzen bei "Cold Starts" in entfernten Regionen, bietet aber durch die Integration in die Plattform eine deutlich einfachere Verwaltung und keine zusätzlichen externen HTTP-Roundtrips, wenn die Funktion warmgelaufen ist.[14]

Das folgende TypeScript-Modul demonstriert eine robuste Implementierung für zKalkulator. Es nutzt eine hierarchische Schlüsselstruktur (["rate_limit", identifier, timestamp]), um Einträge effizient zu gruppieren und abzufragen.

TypeScript

```
// rate_limiter.ts
// Implementierung eines Sliding Window Rate Limiters mit Deno KV
// Fokus auf atomare Konsistenz und Performance
const kv = await Deno.openKv();
interface RateLimitConfig {
  limit: number;      // Maximale Anzahl erlaubter Anfragen
  windowMs: number;   // Größe des Zeitfensters in Millisekunden
}
/**
 * Prüft, ob ein Request für einen gegebenen Identifier (z.B. IP oder User-ID)
 * das Rate Limit überschreitet.
```

```
 */
export async function checkRateLimit(
  identifier: string,
  config: RateLimitConfig
): Promise<{ allowed: boolean; remaining: number; reset: number }> {

  const now = Date.now();
  const windowStart = now - config.windowMs;

  // Key-Design: Wir nutzen eine hierarchische Struktur für schnelles Scannen.
  // Der Prefix ermöglicht es uns, alle Einträge für einen User zu isolieren.
  const keyPrefix = ["rate_limit", identifier];
  // 1. Bereinigung und Zählung (Atomic Check)
  // Wir listen alle Einträge auf, die innerhalb des aktuellen Fensters liegen.
  // Deno KV 'list' ist hier effizient, da wir nur Keys im relevanten Zeitbereich betrachten.
  // Ältere Einträge werden implizit ignoriert (und können asynchron bereinigt werden).

  const entries = kv.list({ prefix: keyPrefix, start: });

  let count = 0;
  // Iteration über die gefundenen Einträge im Fenster
  for await (const entry of entries) {
    count++;
  }
  // Entscheidung: Blockieren, wenn Limit erreicht
  if (count >= config.limit) {
    return {
      allowed: false,
      remaining: 0,
      reset: now + config.windowMs // Konservative Schätzung des Reset-Zeitpunkts
    };
  }
  // 2. Registrierung des neuen Requests (Atomic Write)
  // Wir nutzen Atomic Operations, um sicherzustellen, dass der Eintrag auch unter Last konsistent
geschrieben wird.
  // Ein zufälliger Suffix verhindert Kollisionen bei millisekundengenau gleichen Requests.
  const timestampKey =;

  // Setze den Eintrag mit einer automatischen Ablaufzeit (TTL).
  // Dies übernimmt die Garbage Collection für uns, sodass die DB nicht volläuft.
  const res = await kv.atomic()
   .set(timestampKey, 1, { expireIn: config.windowMs * 2 })
   .commit();
  if (!res.ok) {
    // Bei einem Commit-Fehler (z.B. Konflikt) sollten wir "Fail Open" oder "Fail Closed" entscheiden.
    // Für Sicherheitssysteme ist Logging essenziell.
    console.error("Rate Limit Commit failed - Concurrency conflict potentially high");
  }
  return {
```

```
    allowed: true,
    remaining: config.limit - count - 1,
    reset: now + config.windowMs
  };
}
```

Die Verwendung von expireIn ist hier ein entscheidendes Detail für die "Runtime Defense". Es automatisiert die Bereinigung alter Daten ("Garbage Collection") direkt auf Datenbankebene, was die Performance der Edge Function langfristig stabil hält, da keine manuellen DELETE-Operationen notwendig sind.[9]

2.2 Bot-Erkennung und Validierung mit Cloudflare Turnstile

Während Rate Limiting volumetrische Angriffe abwehrt, benötigen wir eine semantische Unterscheidung zwischen Mensch und Maschine, um intelligente Bots abzuwehren. Cloudflare Turnstile bietet hierfür eine datenschutzfreundliche Lösung ("Smart Challenge"), die oft ohne interaktive Rätsel auskommt. Die Integration erfolgt direkt in der Edge-Schicht, um ungültigen Traffic gar nicht erst an die Supabase-Datenbank durchzulassen.[16]

Der kritische Sicherheitsaspekt ist hier die **serverseitige Validierung**. Ein Client-seitiges Widget allein ist wertlos, da ein Angreifer den API-Aufruf simulieren und das Widget umgehen kann. Das Token, das der Client erhält, muss zwingend in der Deno Edge Function gegen die Cloudflare-API (siteverify) geprüft werden.

Das folgende Diagramm und der Code verdeutlichen den Workflow der "Active Defense" bei der Bot-Erkennung. Anstatt eine Anfrage mit ungültigem Token einfach nur abzulehnen, nutzen wir die Information, um das Bedrohungsniveau (Suspicion Score) der IP-Adresse zu erhöhen.

Code-Implementierung der Validierung (Deno):

TypeScript

```
// turnstile_validator.ts
// Validiert das Token und integriert Active Defense Maßnahmen (Reputation Tracking)
export async function validateTurnstileToken(token: string, ip: string): Promise<boolean> {
  // Das Secret Key sollte sicher in den Environment Variables gespeichert sein
  const SECRET_KEY = Deno.env.get("TURNSTILE_SECRET_KEY")?? "";

  const formData = new FormData();
  formData.append('secret', SECRET_KEY);
  formData.append('response', token);
  formData.append('remoteip', ip);
  try {
    const result = await fetch('https://challenges.cloudflare.com/turnstile/v0/siteverify', {
      body: formData,
      method: 'POST',
    });
    const outcome = await result.json();

    if (!outcome.success) {
      // LOGGING: Detaillierte Fehlercodes für Forensik (z.B. 'timeout-or-duplicate')
      console.warn(`Turnstile validation failed for IP ${ip}:`, outcome['error-codes']);

      // ACTIVE DEFENSE: Erhöhe den Suspicion Score in Deno KV
      // Wenn eine IP mehrfach ungültige Tokens sendet, wird sie temporär komplett gesperrt.
      await incrementSuspicionScore(ip);
```

```
      return false;
    }
    return true; // Token valide
  } catch (e) {
    console.error("Turnstile API communication error", e);
    // Fail Secure: Im Zweifel den Zugriff verweigern, um das System zu schützen
    return false;
  }
}
async function incrementSuspicionScore(ip: string) {
    const kv = await Deno.openKv();
    const key = ["security", "suspicion", ip];
    // Inkrementiere Score atomar. Ablaufzeit 24h.
    await kv.atomic().sum(key, 1n).set(key, 1n, { expireIn: 86400 }).commit();
}
```

Diese Logik stellt sicher, dass zKalkulator nicht nur passiv schützt, sondern "lernt", welche IPs feindselig agieren. Ein Angreifer, der versucht, alte Tokens wiederzuverwenden ("Replay Attack"), wird durch die siteverify-API erkannt, und unsere Active Defense Logik registriert diesen Versuch als Indikator für böswilliges Verhalten.[18]

3. Zweite Verteidigungslinie: Database Active Defense (PostgreSQL)

Sollte ein Angreifer die Edge-Barrieren überwinden – beispielsweise durch Nutzung valider Credentials oder durch Angriffe, die unterhalb der Rate-Limit-Schwellen bleiben ("Low and Slow") – muss die Datenbank selbst als Verteidigungssystem fungieren. Supabase bietet durch die PostgreSQL-Basis mächtige Werkzeuge wie pgcrypto für Kryptografie, plpgsql für prozedurale Logik und pg_net für asynchrone Netzwerkkommunikation.

3.1 Anomalieerkennung: Sequenzielle Scans und Fuzzing

Ein häufiges Angriffsmuster gegen Anwendungen wie zKalkulator ist das systematische Scannen von Ressourcen-IDs, bekannt als "Enumeration" oder "Fuzzing". Angreifer versuchen, durch Inkrementieren von IDs (z.B. User ID 100, 101, 102...) auf nicht autorisierte Daten zuzugreifen.[3]

Der Algorithmus zur Erkennung sequenzieller Muster

Statische WAFs (Web Application Firewalls) betrachten oft nur den einzelnen Request. Die Datenbank hingegen besitzt den historischen Kontext. Wir implementieren einen "Stateful Anomaly Detection"-Mechanismus direkt in PostgreSQL. Die Kernidee basiert auf der Analyse der mathematischen Beziehung zwischen den angefragten IDs bei fehlgeschlagenen Zugriffen (404 Not Found oder 403 Forbidden).

Wenn ein normaler Benutzer zufällig auf nicht existierende Ressourcen stößt, sind die IDs meist unkorreliert. Ein Skript hingegen erzeugt eine lineare Sequenz. Die Differenz (Δ) zwischen aufeinanderfolgenden angefragten IDs ist bei einem Angriff konstant (oft $\Delta = 1$).

Implementierungsstrategie:

1. **Logging:** Wir erstellen eine ungeloggte Tabelle security_logs (für maximale Schreibperformance ohne WAL-Overhead), die fehlgeschlagene Zugriffe speichert.[19]
2. **Analyse:** Ein Trigger analysiert bei jedem neuen Eintrag die letzten N Fehlversuche desselben Nutzers oder derselben IP.
3. **Mustererkennung:** Wir berechnen die Differenzen zwischen den IDs. Ist die Varianz dieser Differenzen nahe Null, handelt es sich um einen maschinellen Scan.[20]

Der folgende SQL-Code demonstriert die Logik zur Erkennung dieser Sequenzen. Er nutzt Window Functions (LAG), um die Abstände zwischen den Zugriffen zu berechnen.
SQL

```
-- SQL-Logik zur Erkennung sequenzieller ID-Zugriffe innerhalb einer Session
WITH last_accesses AS (
   SELECT resource_id
   FROM security_access_logs
   WHERE user_id = current_user_id
   AND event_type = 'NOT_FOUND'
   ORDER BY created_at DESC
   LIMIT 5 -- Betrachte die letzten 5 Fehlversuche
),
diffs AS (
   -- Berechne die Differenz zur vorherigen ID in der sortierten Liste
   SELECT resource_id - LAG(resource_id) OVER (ORDER BY resource_id) as diff
   FROM last_accesses
)
SELECT
   CASE
      -- Wenn alle Differenzen identisch sind (z.B. immer +1), ist es ein Scan
      WHEN COUNT(DISTINCT diff) = 1 THEN 'SEQUENTIAL_ATTACK_DETECTED'
      ELSE 'RANDOM_NOISE'
   END as pattern
FROM diffs
WHERE diff IS NOT NULL;
```

Diese Abfrage ist extrem leistungsfähig, da sie rein mathematische Muster erkennt und nicht auf Signaturen angewiesen ist. Sie erkennt auch "Reverse Scans" (100, 99, 98...) oder Scans mit Schrittweite 2 (100, 102, 104...).

3.2 Active Deception: Honeytokens

Active Defense beinhaltet das Element der Täuschung. Wir implementieren **Honeytokens** (Köder-Daten) in die users oder calculations Tabellen von zKalkulator. Diese Datensätze sehen für einen Angreifer attraktiv aus, gehören aber keinem echten Nutzer und sollten im normalen Betrieb niemals abgerufen werden.[21]

Der Mechanismus ist simpel, aber effektiv:

1. Wir fügen einen Datensatz mit einer bekannten ID (z.B. 999999 oder eine spezielle UUID) ein.
2. Wir erstellen einen Datenbank-Trigger, der auf SELECT, UPDATE oder DELETE Operationen auf dieser spezifischen Zeile reagiert.
3. Sobald der Trigger feuert, wissen wir mit an Sicherheit grenzender Wahrscheinlichkeit, dass ein unautorisierter Scan oder Datenabfluss stattfindet, da kein legitimer Prozess diese ID anfragen würde.[23]

Implementation mit pg_net für asynchrone Alarme:

Kritisch ist hierbei, dass der Alarm asynchron erfolgen muss. Würden wir eine blockierende HTTP-Anfrage im Trigger nutzen, würde die Datenbanktransaktion warten, bis der HTTP-Call fertig ist. Der Angreifer würde eine Verzögerung bemerken (Timing Attack). Mit der Extension pg_net können wir den Alarm in einen Hintergrundprozess auslagern, sodass der Angreifer sofort eine Antwort erhält (um keinen Verdacht zu schöpfen), während im Hintergrund die Sicherheitsmaßnahmen anlaufen.24

SQL

```
-- Trigger Funktion für Honeytoken-Zugriff
CREATE OR REPLACE FUNCTION trigger_honeytoken_alert()
RETURNS TRIGGER AS $$
BEGIN
  -- Prüfung, ob die Honeytoken-ID betroffen ist
  IF (NEW.id = 999999 OR OLD.id = 999999) THEN

    -- Sende asynchronen Alarm via pg_net an die Edge Function
    -- 'PERFORM' führt die Funktion aus und verwirft das Ergebnis
    PERFORM net.http_post(
      'https://api.zkalkulator.internal/admin/ban-user',
      jsonb_build_object(
        'reason', 'HONEYTOKEN_TOUCHED',
        'attacker_user_id', auth.uid(), -- Supabase Auth User ID
        'query_context', current_query(),
        'timestamp', NOW()
      ),
      '{"Content-Type": "application/json"}'::jsonb
    );

    -- Optional: Um Datenleck zu verhindern, werfen wir sofort einen Fehler.
    -- Alternativ könnte man Fake-Daten zurückgeben ("Deception").
    RAISE EXCEPTION 'Access Denied: Security Incident Logged';
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
-- Binden des Triggers an die sensible Tabelle
CREATE TRIGGER check_honeytoken
BEFORE UPDATE OR DELETE ON public.sensitive_calculations
FOR EACH ROW EXECUTE FUNCTION trigger_honeytoken_alert();
```

Hinweis zur Performance: Die Nutzung von pg_net innerhalb von Triggern hat einen sehr geringen Overhead (ca. 2.7% Latenzerhöhung bei Inserts), da die Requests lediglich in eine interne Queue-Tabelle (net.http_request_queue) geschrieben werden und nicht synchron ausgeführt werden. Dies macht die Methode skalierbar auch für hochfrequentierte Systeme.[26]

4. Dritte Verteidigungslinie: Audit & Compliance (Tamper-Proof Ledger)

Die letzte Verteidigungslinie sichert die Integrität der Geschichte. Wenn ein Angreifer administrativen Zugriff erlangt, ist sein erstes Ziel oft das Audit-Log, um Spuren zu verwischen. Ein Standard-PostgreSQL-Log ist mutierbar (UPDATE audit_logs SET...). Um dies zu verhindern, implementieren wir einen **kryptografisch verketteten Ledger (Hash Chain)**.

4.1 Konzept der Hash Chain

Ähnlich wie bei einer Blockchain enthält jeder Eintrag im Audit-Log nicht nur die Daten des Ereignisses, sondern auch den kryptografischen Hash des *vorherigen* Eintrags. Dies erzeugt eine untrennbare Kette.

$Hash_n = SHA256(Hash_{n-1} + Data_n + Timestamp_n)$

Wenn ein Angreifer einen historischen Eintrag ($n-5$) ändert oder löscht, ändert sich dessen Hash. Da dieser Hash aber Teil der Berechnung für den nachfolgenden Eintrag ($n-4$) ist, stimmt die Kette ab diesem Punkt nicht mehr überein. Die Manipulation wird bei der Verifikation sofort evident.[2]

4.2 Technische Implementierung in PostgreSQL

Wir nutzen die pgcrypto Extension für die SHA-256 Hashfunktionen. Das Schema erfordert eine Tabelle, die auf sich selbst referenziert (logisch, über den Hash), und einen Trigger, der die Integrität bei jedem Insert sicherstellt.

SQL

```
-- 1. Extension aktivieren
CREATE EXTENSION IF NOT EXISTS pgcrypto;
-- 2. Ledger Tabelle erstellen
CREATE TABLE public.audit_ledger (
    id BIGSERIAL PRIMARY KEY,
    previous_hash TEXT NOT NULL, -- Kryptografischer Link zum Vorgänger
    user_id UUID,                -- Wer hat gehandelt?
    action_type TEXT,            -- Was wurde getan?
    payload JSONB,               -- Detaildaten
    created_at TIMESTAMPTZ DEFAULT NOW(),
    row_hash TEXT NOT NULL,      -- Hash dieser Zeile (Der "Siegel")

    -- Constraint: previous_hash muss existieren (außer für den allerersten Eintrag, den Genesis Block)
    CONSTRAINT fk_previous_hash CHECK (id = 1 OR previous_hash IS NOT NULL)
);
-- Index für schnelle Validierung der Kette
CREATE INDEX idx_ledger_hash ON public.audit_ledger(row_hash);
```

Der entscheidende Teil ist der Trigger, der *vor* dem Einfügen (BEFORE INSERT) den Hash berechnet. Er muss den aktuellsten Hash aus der Tabelle lesen und in die Berechnung des neuen Hashes einbeziehen.

SQL

```
CREATE OR REPLACE FUNCTION chain_log_entry()
RETURNS TRIGGER AS $$
DECLARE
    last_hash TEXT;
    payload_string TEXT;
BEGIN
    -- 1. Hole den Hash der letzten Zeile (höchste ID)
    -- Hier ist ein potentieller Race-Condition-Punkt bei extrem hoher Last.
    -- Für Audit-Logs ist dies meist akzeptabel oder durch Locking lösbar.
    SELECT row_hash INTO last_hash
    FROM public.audit_ledger
    ORDER BY id DESC
    LIMIT 1;
    -- Genesis Block Handling: Wenn Tabelle leer, setze Start-Hash
    IF last_hash IS NULL THEN
        last_hash := 'GENESIS_HASH_0000000000000000000000000000000000000000000000000';
    END IF;
    -- 2. Setze previous_hash im neuen Datensatz
    NEW.previous_hash := last_hash;
    -- 3. Berechne den Hash der aktuellen Zeile
    -- Wir konkatenieren alle relevanten Spalten und den previous_hash
    payload_string := last_hash |
|
                    COALESCE(NEW.user_id::text, '') |
```

```
|
                NEW.action_type |
|
                NEW.payload::text |
|
                NEW.created_at::text;

   -- Erzeuge SHA-256 Hash und speichere als Hex-String
   NEW.row_hash := encode(digest(payload_string, 'sha256'), 'hex');
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER trg_audit_chain
BEFORE INSERT ON public.audit_ledger
FOR EACH ROW EXECUTE FUNCTION chain_log_entry();
```

4.3 Durchsetzung der "Append-Only" Eigenschaft

Die kryptografische Kette nützt nichts, wenn man Einträge einfach überschreiben kann. PostgreSQL erlaubt es uns, Schreibzugriffe granular zu steuern. Wir installieren einen Trigger, der jegliche UPDATE oder DELETE Operationen auf der audit_ledger Tabelle bedingungslos blockiert. Selbst ein User mit Schreibrechten auf die Tabelle kann diese Sperre nicht umgehen, solange der Trigger aktiv ist.[27]

SQL

```
CREATE OR REPLACE FUNCTION block_modification()
RETURNS TRIGGER AS $$
BEGIN
   -- Wirf eine harte Exception bei jedem Änderungsversuch
   RAISE EXCEPTION 'TAMPERING ATTEMPT: Audit Ledger is append-only! Action blocked.';
   RETURN NULL;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER trg_block_mod
BEFORE UPDATE OR DELETE ON public.audit_ledger
FOR EACH ROW EXECUTE FUNCTION block_modification();
```

Damit wird die Tabelle effektiv unveränderlich ("Immutable"). Ein Angreifer müsste den Trigger löschen (DROP TRIGGER), was wiederum in den Datenbank-Systemlogs (die idealerweise extern gespiegelt werden) Spuren hinterlassen würde.

5. Operative Integration: Monitoring und Response

Active Defense ist ein Kreislauf aus Erkennung, Reaktion und Analyse. Die technischen Komponenten müssen in einen operativen Workflow (SecOps) eingebunden werden.

5.1 Der Alarm-Loop mit pg_net

Wie in Abschnitt 3.2 angedeutet, spielt pg_net eine zentrale Rolle für die Echtzeit-Reaktion. Wenn ein Trigger (Honeytoken oder Sequenz-Scan) anschlägt, darf die Datenbank nicht isoliert bleiben.
Der Workflow sieht wie folgt aus:

1. **Erkennung:** Der SQL-Trigger identifiziert eine Anomalie.
2. **Benachrichtigung:** pg_net sendet ein JSON-Payload an einen internen Webhook (z.B. eine weitere Supabase Edge Function /notify-security).
3. **Reaktion (Edge Function):**
    - Die Funktion empfängt den Alarm.

- Sie reichert die Daten an (z.B. Geo-IP-Lookup der Angreifer-IP).
- Sie sendet eine formatierte Nachricht an den SecOps-Kanal (Slack/Discord/PagerDuty).
- **Automatisierter Ban:** Sie ruft die Supabase Admin API auf, um den User in der auth.users Tabelle auf banned zu setzen oder die Session zu invalidieren.[25]

Dieser "Closed Loop" ermöglicht Reaktionszeiten im Millisekundenbereich, ohne dass ein menschlicher Operator eingreifen muss.

5.2 Zentrales Logging mit Logflare

Für die langfristige Analyse und Forensik ist es wichtig, dass Logs nicht nur in der Datenbank liegen, sondern auch in einem spezialisierten Log-Management-System. Supabase integriert hierfür Logflare. Damit die Logs der Deno Edge Functions in Logflare effektiv durchsuchbar sind, müssen sie **strukturiert** (als JSON) ausgegeben werden, nicht als unstrukturierter Text.[29]

Best Practice für strukturiertes Logging in Deno:

```
TypeScript
// Anstatt console.log("User failed auth"); nutzen wir:
console.log(JSON.stringify({
  level: "warning",
  event_type: "security_incident",
  component: "edge_defense",
  ip_address: clientIp, // Aus Request Headern extrahiert
  user_agent: req.headers.get("user-agent"),
  details: {
    reason: "turnstile_invalid",
    attempt_count: 5,
    target_resource: "/api/calc/sensitive"
  }
}));
```

Logflare parst diese JSON-Objekte automatisch, sodass SecOps-Teams SQL-ähnliche Abfragen auf die Felder ausführen können (z.B. SELECT ip_address, count(*) FROM edge_logs WHERE event_type = 'security_incident' GROUP BY ip_address).

6. Zusammenfassung und Roadmap

Die Sicherheit von zKalkulator beruht nicht auf einer einzelnen "Firewall", sondern auf der synergetischen Verzahnung von Edge-Logik, Datenbank-Intelligenz und kryptografischer Beweisführung.

Implementierungs-Checkliste:

1. **Phase 1: Edge Hardening (Woche 1)**
   - Einrichtung von Cloudflare Turnstile und Integration des Validierungs-Codes in Deno.
   - Aktivierung von Deno KV und Deployment des Sliding Window Rate Limiters.
2. **Phase 2: Database Active Defense (Woche 2)**
   - Installation der Extensions pgcrypto und pg_net.
   - Deployment der Honeytoken-Trigger und der SQL-Logik zur Sequenz-Erkennung.
3. **Phase 3: Audit & Compliance (Woche 3)**
   - Migration der bestehenden Audit-Tabellen auf das Hash-Chain-Schema.
   - Penetration Test: Versuch, Logs via SQL-Injection oder Admin-Zugriff unbemerkt zu manipulieren.
4. **Phase 4: Integration (Woche 4)**
   - Verbindung der DB-Trigger mit dem Slack-Alerting via Edge Functions.
   - Dokumentation der "Banned User" Workflows für das Support-Team.

Durch die Umsetzung dieses Playbooks transformiert sich zKalkulator von einem passiven Ziel zu einem "Hard Target", das Angriffe nicht nur aushält, sondern sie aktiv erkennt, dokumentiert und neutralisiert.