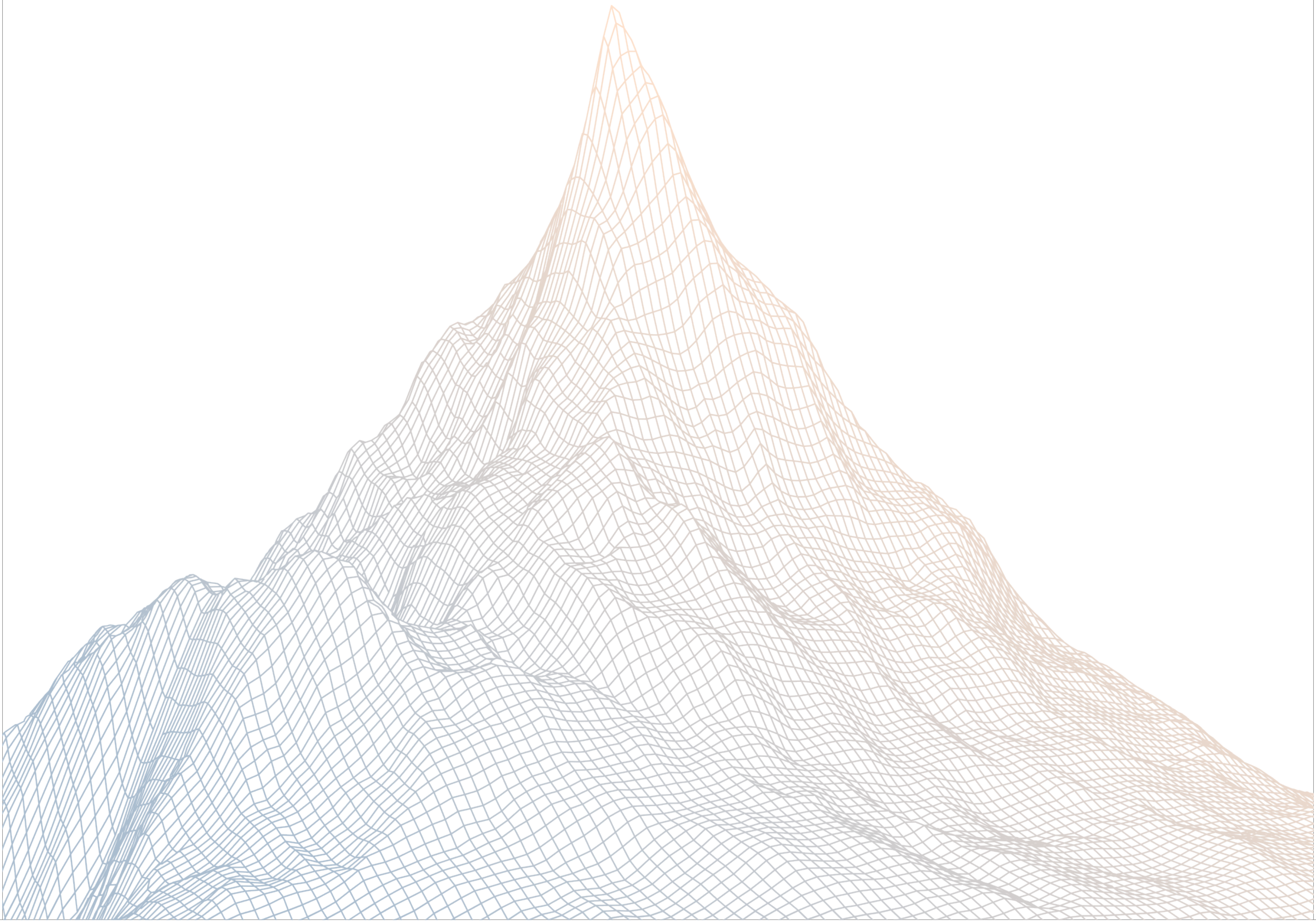
Zenith

Initia Labs

SDK
Security Assessment

VERSION 1.1



AUDIT DATES: January 30th to February 4th, 2026

AUDITED BY: DreyAnd
Jakub Heba

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About Initia

InterwovenKit is a React library that provides components and hooks to connect dApps to Initia and Interwoven Rollups.

2.2 Scope

The engagement involved a review of the following targets:

Target	interwovenkit
Repository	https://github.com/initia-labs/interwovenkit
Commit Hash	`3bdbd95 (feature/autosign-signature-derived-wallet)`

Files	`examples/vite/src/Providers.tsx`
	`src/data/config.ts`
	`src/data/signer.ts`
	`src/data/tx.ts`
	`src/data/ui.ts`
	`src/pages/tx/TxRequest.tsx`
	`public/app/InterwovenKitProvider.tsx`
	`public/data/hooks.ts`

2.3 Audit Timeline

January 30, 2026	Audit start
February 4, 2026	Audit end
February 4, 2026	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	6
Low Risk	3
Informational	2
Total Issues	**11**

3 Findings Summary

ID	Description	Status
M-1	Improper Base Domain Validation inside of the Auto-signing flow	Resolved
M-2	Missing spend limit on BasicAllowance feegrant	Acknowledged
M-3	Missing AllowedMsgAllowance wrapper on feegrant	Resolved
M-4	Unvalidated fee parameter in auto-sign transaction submission	Resolved
M-5	Derived wallet private key exposed via Jotai atom state	Resolved
M-6	Auto-sign validates message types but not message contents	Acknowledged
L-1	Website verification matches any registry chain instead of selected chain	Resolved
L-2	Missing opener protection on target="_blank" links	Resolved
L-3	Hardcoded Chain ID in SIWE Flow may lead to User confusion	Resolved
I-1	Outdated NPM Dependencies may introduce vulnerabilities	Resolved
I-2	Deployment hardening pointers	Resolved

4 Findings

4.1 Medium Risk

A total of 6 medium risk findings were identified.

[M-1] Improper Base Domain Validation inside of the Auto-signing flow

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

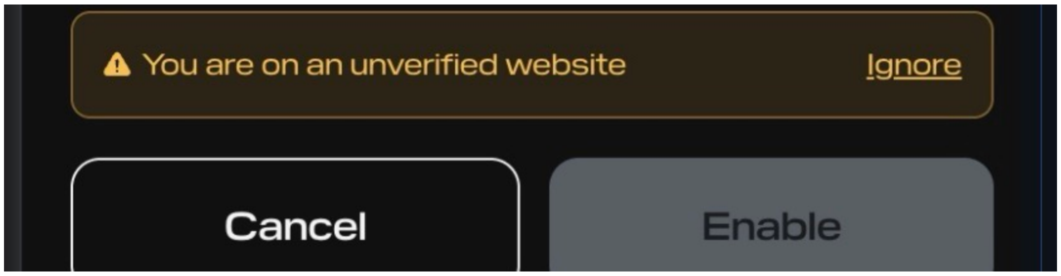
Target:

- EnableAutoSign.tsx

Description:

When enabling the **auto-sign functionality**, we're granting two blockchain-level permissions to a Privy-managed "embedded wallet", which can **execute EVM calls on our behalf** potentially including withdrawals/deposits without any additional approvals from MetaMask and other intermediaries.

If the user is about to enable auto-signing from a (malicious) domain that isn't a part of the Initia registry, they will be warned about it like the following:



Relevant code:

```
const isVerified = chains.some((chain) ⇒ {
 if (!chain.website) return false;
```

```
  const registryDomain = getBaseDomain(new URL(chain.website).hostname);
  const websiteDomain = getBaseDomain(window.location.hostname);
  return registryDomain === websiteDomain;
});
```

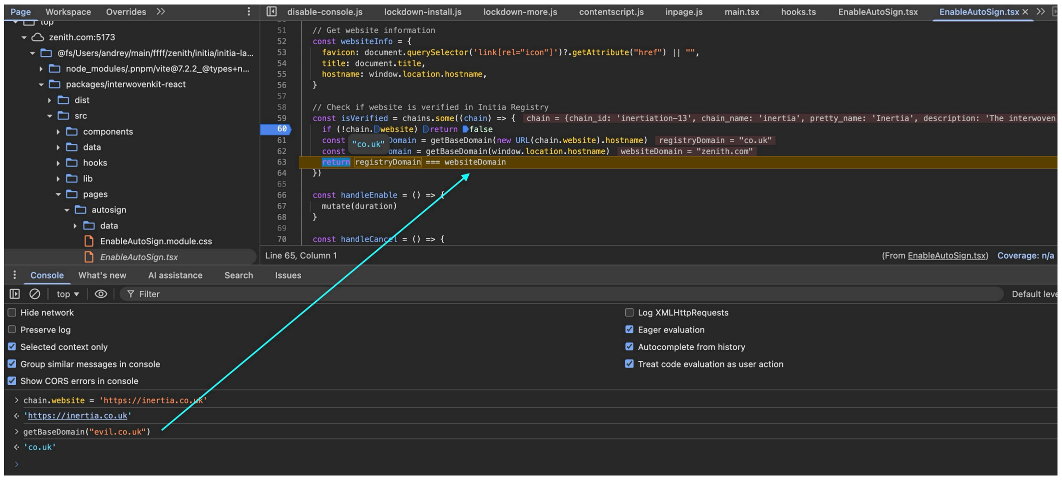
However, the `getBaseDomain()` function in `EnableAutoSign.tsx` has a problematic implementation as it's checking domains using the dots as separators. This isn't ideal as some eTLDs can have multiple parts, i.e `.co.uk`:

```
function getBaseDomain(hostname: string): string {
  const parts = hostname.split(".")
  if (parts.length < 2) return hostname
  return parts.slice(-2).join(".")
}
```

This means that in the following conditions:

- The developers implement a chain inside https://registry.testnet.initia.xyz/chains.json which has a multi-level eTLD (`.co.uk`, `.co.jp`, `com.au`, etc.)
- Someone finds a way to add their own website into the registry.

An attacker can register a domain like `evil.co.uk` and have it **completely bypass ANY warnings** during the auto-signing process and trick the victim into doing it.



```
  // Get website information
  const websiteInfo = {
    favicon: document.querySelector('link[rel="icon"]')?.getAttribute("href") || "",
    title: document.title,
    hostname: window.location.hostname,
  }

  // Check if website is verified in Initia Registry
  const isVerified = chains.some((chain) => {  chain = {chain_id: 'inertiation-13', chain_name: 'inertia', pretty_name: 'Inertia', description: 'The interwoven
    if (!chain.website) return false
    const "co.uk" Domain = getBaseDomain(new URL(chain.website).hostname)  registryDomain = "co.uk"
    const      omain = getBaseDomain(window.location.hostname)  websiteDomain = "zenith.com"
    return registryDomain === websiteDomain
  })

  const handleEnable = () => {
    mutate(duration)
  }

  const handleCancel = () => {
```

- [ ] Hide network
- [ ] Preserve log
- [x] Selected context only
- [x] Group similar messages in console
- [x] Show CORS errors in console
- [ ] Log XMLHttpRequests
- [x] Eager evaluation
- [x] Autocomplete from history
- [x] Treat code evaluation as user action

```
> chain.website = 'https://inertia.co.uk'
< 'https://inertia.co.uk'
> getBaseDomain("evil.co.uk")
< 'co.uk'
>
```

This may allow **financial loss**, such as transferring **victim NFTs**, **draining funds**, **execution of malicious contract interactions**, **spending user's funds on gas** without any additional verification on behalf of the victim.

Recommendations:

- Replace `getBaseDomain()` with a Public Suffix List (PSL)-aware implementation (e.g., `tldts` / `psl`) to correctly handle multi-level eTLDs (e.g., `co.uk`, `com.au`, etc.).
- Avoid "base domain" matching for trust decisions where possible. Prefer verifying against an explicit whitelist of **exact origins** (or exact hostnames) rather than suffix-based equivalence.
- Add unit tests covering multi-level eTLDs and edge cases (e.g., `app.foo.co.uk` vs `evil.co.uk`) to prevent regressions.

Initia: Resolved with @500b2b26f8...

Zenith: Verified.

[M-2] Missing spend limit on `BasicAllowance` feegrant

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Acknowledged	`LIKELIHOOD:` Medium

Target:

- `packages/interwovenkit-react/src/pages/autosign/data/actions.ts`

Description:

In `actions.ts`, within the `useEnableAutoSign` mutation function, the `feegrantMessage` is constructed using `BasicAllowance.fromPartial({ expiration })` which only sets the expiration timestamp. The Cosmos SDK feegrant module interprets an absent `spend_limit` field as unlimited spending authorization.

This means the grantee, so the derived auto-sign wallet, can consume an unbounded amount of the granter's tokens for transaction fees. The feegrant is created via `MsgGrantAllowance` with `typeUrl: "/cosmos.feegrant.v1beta1.BasicAllowance"` and the encoded allowance contains no spending cap whatsoever. Once enabled, this grant persists until the specified expiration or explicit revocation.

```
const feegrantMessage = {
 typeUrl: "/cosmos.feegrant.v1beta1.MsgGrantAllowance",
 value: MsgGrantAllowance.fromPartial({
 granter: initiaAddress,
 grantee: derivedWallet.address,
 allowance: {
 typeUrl: "/cosmos.feegrant.v1beta1.BasicAllowance",
 value: BasicAllowance.encode(BasicAllowance.fromPartial({
    expiration })).finish(),
 },
 }),
};
```

Any entity capable of triggering transaction signing through the derived wallet can drain the user's entire balance through repeated high-fee transactions. Even intentionally failing transactions consume fees on most Cosmos SDK chains, enabling rapid fund extraction without successful transaction execution.

Recommendations:

Always configure a `spend_limit` field on the `BasicAllowance` with a reasonable cap, such as equivalent to 100-1000 typical transaction fees. Consider implementing `PeriodicAllowance` with periodic reset caps to limit sustained attack windows.

Initia: We agree with the finding. But we would want to better understand the transaction/usage intensity for each of the appchain better to be able to set a practical yet effective allowances values. Thus, we will implement this as a future improvement separate from the scope of this audit.

Zenith: Issue was acknowledged by the Client.

[M-3] Missing `AllowedMsgAllowance` wrapper on feegrant

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target:

- `packages/interwovenkit-react/src/pages/autosign/data/actions.ts`

Description:

The feegrant implementation in `actions.ts` uses `BasicAllowance` directly without wrapping it in `AllowedMsgAllowance`. The Cosmos SDK provides `AllowedMsgAllowance` specifically to restrict which message types a feegrant can pay fees for. Without this wrapper, the derived wallet grantee can use the feegrant to pay fees for any transaction type it signs, not just the intended `MsgExec` authz wrapper messages.

The current implementation at the `feegrantMessage` construction site creates an unrestricted fee payment capability that operates independently of the authz message type restrictions configured via `GenericAuthorization`.

```
const feegrantMessage = {
 typeUrl: "/cosmos.feegrant.v1beta1.MsgGrantAllowance",
 value: MsgGrantAllowance.fromPartial({
 granter: initiaAddress,
 grantee: derivedWallet.address,
 allowance: {
 typeUrl: "/cosmos.feegrant.v1beta1.BasicAllowance",
 value: BasicAllowance.encode(BasicAllowance.fromPartial({
    expiration })).finish(),
 },
 }),
};
```

Even if authz grants are properly scoped, the unrestricted feegrant allows the derived wallet to submit arbitrary transaction types with fees charged to the user's account. This creates an independent attack vector for fee burning that bypasses the intended message type controls.

Recommendations:

Wrap the `BasicAllowance` inside an `AllowedMsgAllowance` that explicitly permits only `/cosmos.authz.v1beta1.MsgExec` as the allowed message type. This ensures the feegrant can only pay fees for the intended authz execution flow.

Initia: Resolved with PR-146.

Zenith: Verified.

[M-4] Unvalidated fee parameter in auto-sign transaction submission

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target:

- `packages/interwovenkit-react/src/data/tx.ts`

Description:

The `submitTxSync` and `submitTxBlock` functions in `tx.ts` accept a caller-supplied `fee: StdFee` parameter through the `TxParams` interface. When `validateAutoSign` returns `true`, these functions proceed to call `signWithDerivedWallet` which sets `fee.granter` to the user's address and signs the transaction without any validation of the fee amount.

The fee value flows directly from the calling application code through to the signed transaction. There is no upper bound check, no comparison against simulated gas estimates, and no sanity validation before the granter-paid fee is committed to the signed transaction bytes.

```
const submitTxBlock = async (
 txParams: TxParams,
 timeoutMs = 30 * 1000,
 intervalMs = 0.5 * 1000,
 ): Promise<DeliverTxResponse> ⇒ {
 const chainId = txParams.chainId ?? defaultChainId
 try {
 const { messages, memo = "", fee } = txParams
 const client = await createSigningStargateClient(chainId)

 const isAutoSignValid = await validateAutoSign(chainId, messages)
 const signedTx = isAutoSignValid
 ? await signWithDerivedWallet(chainId, address, messages, fee, memo)
 : await signWithEthSecp256k1(chainId, address, messages, fee, memo)
```

A malicious or compromised integrating application can specify arbitrarily large fee values (such as 1000 INIT per transaction) and auto-sign will execute them without any user confirmation or wallet popup. Combined with the unlimited feegrant from other issue reported, this enables immediate and complete balance drainage through a single transaction or rapid succession of transactions.

Recommendations:

Implement mandatory fee validation in auto-sign mode that enforces maximum gas price, maximum total fee, and comparison against simulation-based estimates. Consider computing fees internally from gas simulation rather than accepting caller-provided values, or require explicit user confirmation when fees exceed a configured threshold.

Initia: We fixed this by removing trust in caller-provided fee values from the auto-sign path. For auto-sign transactions, the SDK now computes the delegated fee internally using gas simulation and chain gas prices, rather than using caller-supplied fee.amount or fee.gas.

Applications can still hint the fee token with preferredFeeDenom, but the final fee amount and gas are always derived by the SDK. We also added per-chain relative bounds (gasMultiplier, maxGasMultiplierFromSim) and optional fee-denom allowlists for additional control. If internal fee computation fails for any reason, the transaction falls back to the normal manual-signing flow.

Resolved with PR-145

Zenith: Verified.

[M-5] Derived wallet private key exposed via Jotai atom state

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target:

- `packages/interwovenkit-react/src/pages/autosign/data/store.ts`

Description:

In `store.ts`, the `derivedWalletsAtom` stores the complete `DerivedWallet` interface including the `privateKey: Uint8Array` field in a Jotai atom. Jotai atoms are stored in React context provider state, which resides in standard JavaScript heap memory. This memory is accessible to any JavaScript code executing in the same browser context, including third-party scripts loaded on the page such as analytics, advertising, social widgets, browser extensions with content script permissions, and any successful XSS payload.

The code comment noting "memory-only storage" as a security feature against persistence attacks does not address runtime memory access threats.

```
/* Memory-only storage: private keys exist only in browser memory and are
    cleared on page refresh.
 * The wallet can always be re-derived from the same signature, so
    persistence is unnecessary
 * and would increase attack surface. */
export const derivedWalletsAtom = atom<Record<string, DerivedWallet>>({});
```

Any malicious script achieving code execution on the page can extract the `privateKey` bytes from the Jotai store. With the private key, an attacker can sign transactions offline and indefinitely, using the still-active authz and feegrant permissions to drain user funds from any location without further interaction with the victim's browser.

Recommendations:

Explore using Web Crypto API's `CryptoKey` with `extractable: false` for key material where possible. Implement strict Content Security Policy headers to limit script sources.

Initia: We will explore the Web Crypto API but as far as we can see now, it will require quite a significant change in our signing architecture. For now, we've implemented a scoped remediation to reduce private key exposure in the autosign flow. Resolved with PR-144

Zenith: Verified.

[M-6] Auto-sign validates message types but not message contents

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target:

- `packages/interwovenkit-react/src/pages/autosign/data/validation.ts`

Description:

The `useValidateAutoSign` function in `validation.ts` implements auto-sign eligibility checking by verifying that all messages in a transaction have their `typeUrl` field present in the configured `messageTypes[chainId]` array. The validation logic at `messages.every((msg) ⇒ chainMessageTypes.includes(msg.typeUrl))` performs only type-level checking without any inspection of message parameters, recipient addresses, amounts, or contract calls.

When a user enables auto-sign for broad execution messages like `/initia.move.v1.MsgExecute`, `/cosmwasm.wasm.v1.MsgExecuteContract`, or `/minievm.evm.v1.MsgCall`, the validation permits any invocation of these message types regardless of what contract is called or what parameters are passed.

```
/* Validate whether a transaction can be auto-signed by checking enabled
    status and message types */
export function useValidateAutoSign() {
 const { data } = useAutoSignStatus();
 const messageTypes = useAutoSignMessageTypes();

 return async (chainId: string, messages: EncodeObject[]) ⇒ {
 // Check condition 1: All messages must be in allowed types
 const allMessagesAllowed = messages.every((msg) ⇒ {
 const chainMessageTypes = messageTypes[chainId];
 if (!chainMessageTypes) return false;
 return chainMessageTypes.includes(msg.typeUrl);
 });

 // Check condition 2: AutoSign must be enabled for the chain
 const isAutoSignEnabled = data?.isEnabledByChain[chainId] ?? false;
```

```
    return allMessagesAllowed && isAutoSignEnabled;
  };
}
```

A malicious dApp can construct valid message types containing harmful parameters - calling attacker-controlled smart contracts, executing token transfer functions to drain balances, setting unlimited token approvals, or transferring NFTs. The user has no opportunity to review transaction details before auto-signing executes the malicious payload.

Recommendations:

Implement parameter-level validation for high-risk message types including contract address allowlists, recipient address blocklists, and spending amount limits. Consider requiring explicit per-contract or per-function approvals rather than one message type authorization.

Initia: We agree with the finding and plan to implement the suggested parameter-level control and validation. But since this affects developer workflow across multiple appchains, we plan to fix this as a separate iteration of the autosign feature outside of this audit after discussing the best DX and form factor with partner teams.

Zenith: Client acknowledged the issue and will resolve it in the future.

4.2 Low Risk

A total of 3 low risk findings were identified.

[L-1] Website verification matches any registry chain instead of selected chain

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target:

- `packages/interwovenkit-react/src/pages/autosign/EnableAutoSign.tsx`

Description:

In `EnableAutoSign.tsx`, the `isVerified` boolean is computed using `chains.some((chain) ⇒` which iterates over the entire `useInitiaRegistry()` chain list to find any matching domain. The comparison uses `getBaseDomain(new URL(chain.website).hostname)` against `getBaseDomain(window.location.hostname)` without filtering to the specific chain identified by `pendingRequest.chainId`.

This means verification passes if the current site's domain matches the website of any chain in the registry, regardless of which chain the user is actually enabling auto-sign for.

```
// Check if website is verified in Initia Registry
const isVerified = chains.some((chain) ⇒ {
 if (!chain.website) return false;
 const registryDomain = getBaseDomain(new URL(chain.website).hostname);
 const websiteDomain = getBaseDomain(window.location.hostname);
 return registryDomain ≡≡ websiteDomain;
});
```

An attacker operating a legitimate chain registered in the Initia registry can use their verified domain status to appear trustworthy when requesting auto-sign enablement for a completely different chain. Users see a "verified" indicator and may trust the request, not realizing the verification applies to an unrelated chain entry.

Recommendations:

Modify the verification logic to filter the chain lookup to only `chains.find((chain) ⇒ chain.chainId ≡ pendingRequest.chainId)` and verify against that specific chain's registered website. Display clear indication of which chain's verification is being shown.

Initia: Resolved with PR-141

Zenith: Verified.

[L-2] Missing opener protection on `target="_blank"` links

`SEVERITY`: Low	`IMPACT`: Low
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target:

- `packages/interwovenkit-react/src/pages/autosign/EnableAutoSign.tsx`

Description:

In `EnableAutoSign.tsx`, the documentation link element uses `target="_blank"` to open the Initia docs URL in a new tab but does not include the `rel="noopener noreferrer"` attribute. Without these attributes, the newly opened page receives a reference to `window.opener` which points back to the original InterwovenKit page.

This is a well-documented browser security issue where the opened page (or any page it redirects to, or any scripts it loads) can execute `window.opener.location = "https://phishing-site.com"` to navigate the original tab without user interaction.

```
<a
 href="https://docs.initia.xyz/user-guides/wallet/auto-signing/introduction"
 target="_blank"
 className={styles.learnMoreLink}
 >
```

If the linked documentation site is compromised, experiences a supply-chain attack on its dependencies, or redirects through a vulnerable URL, the attacker can silently redirect the user's original wallet interface tab to a phishing page. The user returns to find what appears to be their wallet but is actually an attacker-controlled site ready to capture credentials or request malicious transactions.

Recommendations:

Add `rel="noopener noreferrer"` to all anchor tags using `target="_blank"`. This severs the `window.opener` reference and prevents the opened page from manipulating the original tab's navigation.

Initia: Resolved with PR-143.

Zenith: Verified.

[L-3] Hardcoded Chain ID in SIWE Flow may lead to User confusion

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target:

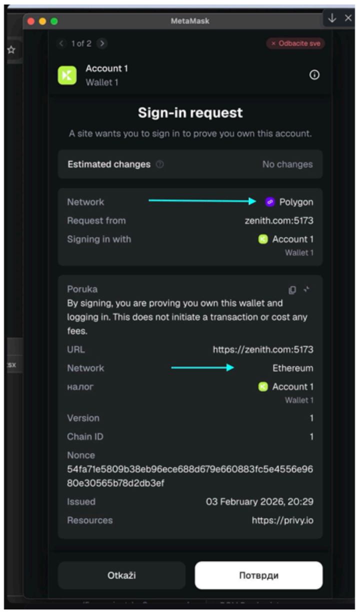
- privy.ts

Description:

The **SIWE** (Sign-In With Ethereum) authentication flow uses a hardcoded ‘chainId: "eip155:1"‘ (Ethereum mainnet) regardless of which network the user's wallet is actually connected to:

```
const message = await privyContextRef.current.siwe.generateSiweMessage({
 chainId: "eip155:1", // Always Ethereum, regardless of actual network
 address,
});
```

When a user connects their wallet while on a **non-Ethereum network** (e.g., Polygon, Arbitrum, Base), the SIWE message presented to them claims they are authenticating on **Ethereum (Chain ID: 1)**, even though their wallet is connected to a **different chain.** The user signs this message, and Privy accepts the authentication.



Account 1
Wallet 1

Sign-in request

A site wants you to sign in to prove you own this account.

Estimated changes	No changes
Network	Polygon
Request from	zenith.com:5173
Signing in with	Account 1 Wallet 1

Poruka
By signing, you are proving you own this wallet and logging in. This does not initiate a transaction or cost any fees.

URL	https://zenith.com:5173
Network	Ethereum
налог	Account 1 Wallet 1
Version	1
Chain ID	1

Nonce
54fa71e5809b38eb96ece688d679e660883fc5e4556e96
80e30565b78d2db3ef

Issued	03 February 2026, 20:29
Resources	https://privy.io

As it stands, aside from it possibly leading to simple user confusion / loss of trust on the UI side, it is just a **SIWE specification violation**, as the Chain ID in a SIWE message should match the chain the user is actually on, per EIP-4361.

However, it may lead to:

- **Future Authentication failures**: if Privy enforces stricter chain validation in new updates,
- **Session Management Issues**: Chain-specific session management or permissions would be incorrectly scoped to Ethereum rather than the user's actual network.

Recommendations:

- Dynamically determine the chain ID from the connected wallet instead of hardcoding it.
- Add validation to ensure the SIWE chain ID matches the wallet's connected chain before authentication.

Initia: Resolved with @6d44b625b4e1083f8a2eb62eb8cbdf5e226e13b1

Zenith: Verified.

4.3 Informational

A total of 2 informational findings were identified.

[I-1] Outdated NPM Dependencies may introduce vulnerabilities

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target:

- package.json

Description:

`pnpm audit` reports **6 vulnerabilities** (3 high / 1 moderate / 2 low) affecting **transitive dependencies** (dependencies of dependencies), rather than packages that are directly imported by this repository's application/library source code.

Observed vulnerable packages and dependency chains (as reported by `pnpm audit`):

- *`qs`* **(HIGH, arrayLimit bypass DoS)**

Path: `examples/vite` → `vite-plugin-node-polyfills` → `node-stdlib-browser` → `url` → `qs` Notes: This chain is associated with browser polyfills for Node.js stdlib behavior. It is not a direct application-level dependency.

- *`preact`* **(HIGH, JSON VNode injection)**

Path: `examples/vite` → `@privy-io/react-auth` → `@coinbase/wallet-sdk` → `preact` Notes: This is the most runtime-relevant chain among the findings, but usage is internal to third-party wallet SDK dependencies.

- *`h3`* **(HIGH, request smuggling TE.TE)**

Path: `examples/vite` → `@privy-io/react-auth` → `@walletconnect/ethereum-provider` → `@walletconnect/keyvaluestorage` → `unstorage` → `h3` Notes: `h3` is an HTTP framework; in this client-side context it is not expected to be instantiated as an HTTP server, reducing practical exploitability.

- *`lodash`* **(MODERATE, prototype pollution in** *`_.unset`* **/** *`_.omit`***)**

Path: `examples/vite` → `wagmi` → `@wagmi/connectors` → `@gemini-wallet/core` → `@metamask/rpc-errors` → `@metamask/utils` → `lodash` Notes: Risk depends on upstream usage of the affected lodash APIs on attacker-controlled objects.

- *`elliptic`* **(LOW, risky cryptographic primitive implementation)**

Path: `examples/vite` → `vite-plugin-node-polyfills` → `node-stdlib-browser` → `crypto-browserify` → `browserify-sign` → `elliptic` Notes: This appears in a polyfill crypto dependency chain rather than in the primary signing stack (e.g., ethers/CosmJS).

- *`diff`* **(LOW, DoS in** *`parsePatch`* **/** *`applyPatch`***)**

Path: `packages/interwovenkit-react` → `vite-plugin-dts` → `@microsoft/api-extractor` → `diff` Notes: This is build-time tooling rather than runtime code shipped to end users.

Overall, this represents a **dependency hygiene / supply chain risk**. Given the transitive nature of the findings and the presence of multiple build-time or polyfill-related chains, the expected real-world impact on the deployed product is low; however, remediation is still recommended to reduce exposure and future risk.

Recommendations:

- Update the **direct parent dependencies** that introduce these vulnerable transitives (e.g., wallet SDKs, connector packages, and build tooling), then re-run `pnpm audit` to confirm remediation.
- Where parent updates are not immediately available, enforce patched versions via *`pnpm.overrides`* (when a patched version exists), reinstall, and re-audit to ensure the override is applied.
- Reduce the dependency attack surface by removing **unused polyfills/tooling** (notably Node polyfills) if they are not required for runtime behavior.
- Add CI checks to continuously monitor dependency risk (e.g., `pnpm audit --audit-level=high`) and store reports as artifacts for review and traceability.
- Reassess severity based on **runtime reachability**: prioritize fixes for vulnerabilities that are plausibly reachable in the shipped client bundle over dev/build-time-only findings.

Initia: Resolved with @a3ca9d808034...

Zenith: Verified.

[I-2] Deployment hardening pointers

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target:

- README.md

Description:

The repository README / docs do not provide guidance for integrators on deploying the widget UI with standard web security response headers (e.g., **Content-Security-Policy (CSP)**, **X-Frame-Options /** *`frame-ancestors`*, and related hardening headers). While these headers are typically enforced by the host application/CDN (not by the widget itself), the absence of documented requirements and recommendations increases the chance that integrators deploy with insecure defaults.

This is relevant because the UI includes high-impact actions (deposit/withdraw/approvals). Without clickjacking protections and a restrictive CSP, the impact of UI redressing and any future injection primitive (in the host app or a dependency) is materially increased.

Additionally, the README should note that HTTP responses should set an explicit character set (e.g., `Content-Type: text/html; charset=utf-8`) to avoid ambiguity and reduce encoding-related risk.

Recommendations:

- Add a short "Deployment Security Headers" section to the README that:
- explains why CSP and clickjacking protections matter for wallet/transaction UIs,
- recommends enforcing clickjacking protections via `Content-Security-Policy: frame-ancestors ...` (preferred) and optionally `X-Frame-Options` for legacy coverage,
- provides a strong CSP example and notes that integrators must allowlist only the exact third-party origins they use (e.g., auth/wallet providers) in `connect-src` / `frame-src`,
- includes a reminder to set `Content-Type` with an explicit charset (e.g., `text/html; charset=utf-8`) for HTML responses (and to use appropriate `Content-Type` values for JS/CSS with charset where applicable),
- suggests simple verification steps (`curl -v`, browser DevTools) to confirm the headers are present in production responses.

Initia: Resolved with @cba24141ba...

Zenith: Verified.