Initia Security Review

Pashov Audit Group

Conducted by: defsec, Bloqarl, 5m477, Shurikenzer

June 17th 2025 - June 23th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **initia-labs/widget** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Widget and Router API

Initia Widget is a React SDK that seamlessly integrates Initia blockchain wallet connections, bridging, and transaction signing into your web applications with ready-to-use hooks and components. The scope also included Router API which is a proxy for Skip API with additional supports for Initia Ecosystem.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - c7c7fc23a5d65cafbd8d748711a607abf695052a

fixes review commit hash - de9d3602dc1afcb40312cadf12c6dbf1fb1a1169

Scope

The following smart contracts were in scope of the audit:

- `src/components/`
- `src/data/`
- `src/lib/router/`
- `src/pages/`
- `src/public/`
- `src/styles/`
- `console`
- `index`
- `src/app/`
- `src/shared/`
- `src/types/`
- `src/utils/`
- `constants`
- `env`
- `main`
- `sentry.ts`

7. Executive Summary

Over the course of the security review, defsec, Bloqarl, 5m477, Shurikenzer engaged with Initia to review Widget and Router API. In this period of time a total of **28** issues were uncovered.

Protocol Summary

Protocol Name	Widget and Router API
Repository	https://github.com/initia-labs/widget
Date	June 17th 2025 - June 23th 2025
Protocol Type	Widget

Findings Count

Severity	**Amount**
Medium	5
Low	23
Total Findings	**28**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Raw request chain to external service	Medium	Resolved
[M-02]	Unvalidated External Image URLs from NFT Metadata	Medium	Resolved
[M-03]	Stale SigningStargateClient when changing accounts	Medium	Resolved
[M-04]	Inadequate URL protocol whitelist in link sanitization	Medium	Resolved
[M-05]	XSS possible via unfiltered src in img.initia.xyz proxy endpoint	Medium	Resolved
[L-01]	ERC20 approval failure handling	Low	Acknowledged
[L-02]	Fee validation calculated but not enforced allows transaction failure	Low	Resolved
[L-03]	Docker image runs as root user	Low	Resolved
[L-04]	Missing Kubernetes security controls risk container escape	Low	Acknowledged
[L-05]	Fixed gas limits in LayerZero cause transaction and fund locks	Low	Acknowledged
[L-06]	Information disclosure through console logging	Low	Resolved
[L-07]	Tabnabbing vulnerability in external links	Low	Acknowledged
[L-08]	Missing security headers	Low	Acknowledged
[L-09]	Overly permissive CORS configuration	Low	Acknowledged

[L-10]	X-Powered-By: express header leaks technology stack	Low	Resolved
[L-11]	Oversized packet in ORDERED IBC channel can cause channel closure	Low	Resolved
[L-12]	Missing await prevents proper error handling in Erc20Service	Low	Resolved
[L-13]	Improper exception type and lack of input validation in OP bridge check	Low	Resolved
[L-14]	Lack of validation on NFT metadata in WithNormalizedNft	Low	Acknowledged
[L-15]	No response validation for external skip API	Low	Acknowledged
[L-16]	Insecure TLS (1.0 & 1.1) active on public endpoints	Low	Resolved
[L-17]	Missing HTTP request timeouts in axios clients on SKIP APIs	Low	Resolved
[L-18]	Missing HTTP request timeouts across modules in widgets	Low	Acknowledged
[L-19]	Unhandled exception in quantitySuperRefine()	Low	Acknowledged
[L-20]	Missing curve validation in encodeEthSecp256k1Pubkey	Low	Acknowledged
[L-21]	Insufficient confirmation depth for reorg protection	Low	Acknowledged
[L-22]	Unescaped NFT metadata in CollectionDetails.tsx	Low	Acknowledged
[L-23]	Unvalidated Wallet Image Source	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Raw request chain to external service

Severity

Impact: Medium

Likelihood: Medium

Description

Vulnerable code:

- router-api/src/app/transfer/transfer.controller.ts.
- router-api/src/app/transfer/transfer.service.ts.
- router-api/src/app/transfer/external/skip.service.ts.

The code defines two open proxy-style endpoints (/api/rest/* and /api/rpc/*) that forward the entire raw HTTP Request object, unfiltered, through the application layers and finally into an outbound axios.request(...) call targeting the Skip API (SKIP_GO_API_URL).

There is no:

- Input validation.
- Allowed path filtering.
- Header or method restrictions.
- Body sanitation.
- Authentication or access control.

The result is a generic unauthenticated passthrough interface to an external API, over which clients can execute arbitrary calls with custom payloads.

Note: Note: This vulnerability is documented in detail within the individual findings "No Response Validation for External Skip API" and "Unvalidated Raw Request Injection via msgsDirect and txTrack." This section emphasizes how these vulnerabilities interact to create a compound security risk that amplifies the potential impact beyond what each issue presents in isolation.

```
// transfer.controller.ts:
@All("/api/rest/*path")
apiRest(@Req() req: Request) {
  return this.transferService.api(req);
}
```

```
// transfer.service.ts:
api(req: Request) {
  return this.skipService.api(req);
}
```

```
// skip.service.ts:
api(req: Request) {
  const url = new URL(req.url, SKIP_GO_API_URL);
  return this._requestSkip(url, req.body, req.method);

}
```

Impact:

- If SKIP_GO_API_URL is misconfigured, attackers can target internal resources.
- Ability to spoof headers such as Host, Authorization, Origin.
- Attackers can forward arbitrary requests through your backend.

Recommendations

Recommendations are already in place for independent findings related to this one. Additionally:

- Strip and Sanitize Headers.
- Only allow safe methods like GET or explicitly defined POST operations.
- Apply DTO validation if the body is used.

[M-02] Unvalidated External Image URLs from NFT Metadata

Severity

Impact: Medium

Likelihood: Medium

Description

In multiple components (`CollectionDetails.tsx`, `NftDetails.tsx`, etc.), NFT image URLs sourced from on-chain metadata or third-party APIs are passed directly into rendering components like `<NftThumbnail />` without validation or sanitization.

Example vulnerable code:

```
<NftThumbnail src={nft.image} />
```

This creates a **trust boundary violation**, since NFT images may originate from:

- Untrusted domains or IPFS gateways.
- Inline `data:` URIs (e.g. `data:image/svg+xml`).
- Malformed or malicious payloads (e.g. embedded `<script>` in SVG).

These images ultimately reach shared components such as `Image.tsx`, which may not enforce strict URL validation.

Impact

An attacker can:

- Execute **XSS** via malicious SVG or `data:` URIs.
- **Spoof branding** or UI with attacker-controlled images.
- Load **offensive or phishing** content.
- Exploit **cache poisoning** or unexpected proxy behavior.

Code Location

link

Recommendations

Enforce strict validation before rendering any external image:

- Allow only `https:` URLs or trusted `ipfs://` schemes.
- Block `data:`, `javascript:`, and unknown protocols.
- Use a shared utility (e.g. `isSafeImageUrl`) for consistent validation.

Harden rendering components (`NftThumbnail.tsx`, `Image.tsx`):

- Add domain allowlists or trusted gateway filters.
- Reject or sanitize inline SVGs and `image/svg+xml` MIME types.
- Apply content-type checks if fetched via proxy.

Strengthen metadata sourcing:

- Use trusted NFT registries (e.g. `nft.storage`, `mintscan`).
- Consider signature validation or allowlisting known creators.

[M-03] Stale `SigningStargateClient` when changing accounts

Severity

Impact: Medium

Likelihood: Medium

Description

`clientCache` keys only by chainId. If the user switches to a different wallet address, the cached client, tied to the old `OfflineSigner` instance, will be reused, causing all subsequent transactions to be signed by the previous account. a) Client Cache Implementation:

```
// packages/widget-react/src/data/signer.ts
const clientCache = new Map<string, SigningStargateClient>
//(); // Problem: Only chains are cached

export function useCreateSigningStargateClient() {
  const findChain = useFindChain();
  const registry = useRegistry();
  const aminoTypes = useAminoTypes();
  const offlineSigner = useOfflineSigner();

  return async (chainId: string) => {
    if (clientCache.has(chainId)) { // @audit Only checks chainId
      return clientCache.get(chainId)!;
    }

    const { rpcUrl } = findChain(chainId);
    const client = await SigningStargateClient.createWithSigner(
      /* ... */
    );

    clientCache.set(chainId, client); // @audit Stores without address context
    return client;
  };
}
```

b) `OfflineSigner` Initialization:

```
// Same file, signer.ts
export function useOfflineSigner() {
  const address = useInitiaAddress(); // Changes when wallets switch
  const { signMessageAsync } = useSignMessage();

  return new OfflineSigner(address, (message) => signMessageAsync({ message }));
}
```

Root Cause Analysis

- Cache Key Design Flaw: The cache uses only chainId as the key (Map<string, SigningStargateClient>) Ignores the currently connected wallet address (useInitiaAddress())

- State Contamination sequence: User -> App: Connects Wallet A App -> Cache: Stores Client (chainId:123 + Wallet A signer) User -> App: Switches to Wallet B App -> Cache: Retrieves same Client (chainId:123) App -> Blockchain: Signs TX with Wallet A (wrong account!)

- Silent Failure Mode: No errors are thrown UI shows current wallet (B) while actually signing with old wallet (A)

Code Location

https://github.com/initia-labs/widget/blob/c7c7fc23a5d65cafbd8d748711a607abf695052a/packages/widget-react/src/data/signer.ts#L181

Recommendations

Key the cache by both chainId and address, or clear the cache on signer change, For example:

```
const clientCache = new Map<string, SigningStargateClient>();

function cacheKey(chainId: string, address: string) {
  return `${chainId}:${address}`;
}
…
const key = cacheKey(chainId, signerAddress);
if (clientCache.has(key)) return clientCache.get(key)!;
…
clientCache.set(key, client);
```

[M-04] Inadequate URL protocol whitelist in link sanitization

Severity

Impact: Medium

Likelihood: Medium

Description

In `ExplorerLink.tsx` and `ManageChainsItem.tsx`, user-controlled URLs (e.g. the txPage template or the chain's website field) are passed directly through xss() and rendered into an `<a href="…">` without validating the URL scheme. While the xss library escapes dangerous characters, it does not enforce a protocol whitelist by default.

An attacker who can manipulate the link template can supply a javascript: URI (for example, `"javascript:alert(document.cookie)"`), resulting in DOM-based XSS when a user clicks the link.

```
// ExplorerLink.tsx
<a
  href={xss(txPage.replace(/\$\{txHash\}/g, txHash))}
  target="_blank"
  rel="noopener noreferrer"
>
  {text}
</a>

// ManageChainsItem.tsx
{website && (
  <a
    href={xss(website)}
    target="_blank"
    rel="noopener noreferrer"
  >
    {website}
  </a>
)}
```

Code Location

ExplorerLink.tsx#L30-L31

Recommendations

- Enforce a strict URL protocol whitelist before rendering. For example, wrap each URL in a helper that:

1. Parses via `new URL(url, window.location.origin)`.
2. Checks protocol `=== 'https:' || protocol === 'http:'`.

Only then passes it to xss (or better, omit xss and rely on React's safe attribute encoding). For example:

```
// utils/safeLink.ts
export function sanitizeLink(href: string): string {
  try {
    const url = new URL(href, window.location.href)
    if (!['http:', 'https:'].includes(url.protocol)) {
      throw new Error('Invalid protocol')
    }
    return url.toString()
  } catch {
    return '#'
  }
}

// In ExplorerLink.tsx & ManageChainsItem.tsx
import { sanitizeLink } from '@/utils/safeLink'

…
<a
  href={sanitizeLink(txPage.replace(/\$\{txHash\}/g, txHash))}
  target="_blank"
>
  …
</a>
```

3. Alternatively, configure xss with custom safeAttrValue logic to reject any non-HTTP(S) protocol. To enforce a strict HTTP/HTTPS protocol whitelist before rendering any user-controlled URL. Two possible approaches:

Helper-based sanitization:

```
// utils/sanitizeLink.ts
export function sanitizeLink(href: string): string {
  try {
    const url = new URL(href, window.location.href);
    if (!['http:', 'https:'].includes(url.protocol)) {
      throw new Error('Invalid protocol');
    }
    return url.toString();
  } catch {
    return '#';
  }
}

// In ExplorerLink.tsx & ManageChainsItem.tsx
<a
  href={sanitizeLink(txPage.replace(/\$\{txHash\}/g, txHash))}
  target="_blank"
  rel="noopener noreferrer"
>
  {text}
</a>
```

Custom xss configuration:

```
import xss from 'xss';

const urlXssOptions = {
  whiteList: {},
  stripIgnoreTag: true,
  stripIgnoreTagBody: ['script'],
  safeAttrValue: (tag, name, value) => {
    try {
      const url = new URL(value, window.location.href);
      if (!['http:', 'https:'].includes(url.protocol)) {
        return '#';
      }
      return xss.safeAttrValue(tag, name, value);
    } catch {
      return '#';
    }
  }
};

<a href={ xss(txPageUrl, urlXssOptions) } …>
  {text}
</a>
```

[M-05] XSS possible via unfiltered `src` in `img.initia.xyz` proxy endpoint

Severity

Impact: High

Likelihood: Low

Description

An XSS vulnerability exists due to the use of unvalidated and unfiltered URLs passed into an image proxy service (`img.initia.xyz`). This affects the image rendering logic within the `Image.tsx` component, which directly binds the `src` attribute from potentially untrusted input.

The function `getProxyImage` prepends `https://img.initia.xyz/?url=` to the provided URL unless it starts with a `data:image/` prefix. However, this allows attackers to inject malicious payloads via base64-encoded HTML/JavaScript content through a `data:text/html;base64,...` scheme.

A working XSS payload example:

```
https://img.initia.xyz/?url=data:text/html;
base64,PHNjcmlwdD5hbGVydCgiWFNTIGhlcmUiKTwvc2NyaXB0Pg==
```

Since the widget directly renders the image using the `src` from `getProxyImage`, the browser treats the URL as valid and executes the embedded script.

Additionally, there is no validation or sanitization of the incoming `src` value in the component. The code path:

```
<Img
  ...
  src={getProxyImage(src)}
  ...
/>
```

shows that even malformed or malicious input is proxied and rendered.

Code Location : link

Recommendations

- Add **whitelisting or domain validation** on the `src` parameter before binding it to the image tag.
- Sanitize or reject any `data:` URIs that do not begin with `data:image/`.
- Consider rendering images via a safe CDN or a stricter proxy layer that prevents HTML/JS payloads.
- Update the `Image.tsx` component to include input validation and reject unsafe schemes like `data:text/html`.

8.2. Low Findings

[L-01] ERC20 approval failure handling

Sequential approval processing with fail-fast behavior creates inconsistent on-chain state when partial failures occur.

link

```
for (const approval of tx.evm_tx.required_erc20_approvals) {
  const response = await tokenContract.approve(spender, amount)
  await response.wait() // Throws on failure, exits loop
}
```

ERC20 approvals are individual on-chain transactions that cannot be rolled back. The loop lacks transaction-level error isolation, causing:

1. **State Inconsistency:** Earlier approvals succeed while later ones fail.
2. **Gas Waste:** Failed approvals consume gas without completing the bridge operation.
3. **User Confusion:** Partial approval state requires manual intervention.

Recommendation Replace the sequential approval loop with a parallel processing approach that handles individual failures gracefully while maintaining transaction independence.

[L-02] Fee validation calculated but not enforced allows transaction failure

The code calculates whether users have sufficient balance for transaction fees but doesn't actually prevent submission when fees are insufficient. The validation logic exists but is commented out.

```
// BridgeFields.tsx lines 140-145
const feeErrorMessage = useMemo(() => {
  for (const fee of route?.estimated_fees ?? []) {
    const balance = balances?.[fee.origin_asset.denom]?.amount
    if (!balance || BigNumber(balance).lt(fee.amount ?? 0)) {
      return `Insufficient ${fee.origin_asset.symbol} for fees`
    }
  }
}, [balances, route])

const disabledMessage = useMemo(() => {
  // ... other validations
  if (feeErrorMessage) return // feeErrorMessage  <-- ISSUE: Commented out!
},
```

Recommendation

Consider enabling fee validation on the code.

[L-03] Docker image runs as root user

The Dockerfile for this project uses the default root user to run the application.

Running as root within a container may:

- Allow **privilege escalation** if the container is compromised.
- Violate **best practices** and some Kubernetes security policies (e.g., PodSecurityAdmission).
- Lead to **inconsistent file permissions** when using shared volumes.
- Introduce **additional risk** if kernel vulnerabilities are present.

Recommendation

Update the Dockerfile to drop root privileges by creating and switching to a non-root user. Here’s a suggested change:

```
# Add to the base image section
RUN useradd --create-home --shell /bin/bash appuser

# In the app image section
RUN chown -R appuser:appuser /app

# Drop root privileges before running the app
USER appuser
```

[L-04] Missing Kubernetes security controls risk container escape

The Kubernetes deployment configuration in `charts/templates/deployment.yaml` lacks essential security controls, creating multiple attack vectors that enable container escape, lateral movement within the cluster, and resource abuse. Despite handling sensitive financial operations, the deployment uses minimal security hardening.

Specific Issues Identified:

1. **No Network Policies**: Pods can communicate unrestricted with any other pods/services in the cluster
2. **Insufficient Resource Limits**: Fixed 1GB memory limit vulnerable to exhaustion attacks
3. **Missing Security Context Controls**: Lacks privilege escalation prevention and capability dropping
4. **No Pod Security Standards**: Missing admission controls for container security validation
5. **Unencrypted Internal Traffic**: No service mesh or mTLS for inter-service communication
6. **Inadequate Monitoring**: Basic health checks only, no security event monitoring

Code Location: link

Recommendation

Consider implementing Kubernetes security policies.

[L-05] Fixed gas limits in `LayerZero` cause transaction and fund locks

The LayerZero integration uses hardcoded gas limits (500,000 for lzReceive and 800,000 for compose operations) that don't adapt to payload complexity or network conditions. This creates a critical vulnerability where complex operations fail due to insufficient gas, potentially locking funds in bridge contracts.

link

```
export function createLzQuoteSendEvm(
  amountIn: string,
  targetChainLzId: number,
) {
  return [
    targetChainLzId,
    toDestinationAddressDataEvm(DUMMY_HEX_ADDRESS),
    BigInt(amountIn),
    BigInt(amountIn),
    Options.newOptions()
      .addExecutorLzReceiveOption(500000)
      .addExecutorComposeOption(0, 800_000)
      .toHex(),
    "0x", // composeData doesn't affect the fees
    "0x",
  ];
}
```

Recommendation Consider moving gas limits to environment configuration.

[L-06] Information disclosure through console logging

The application logs information including transaction details, error messages, and potentially user data to the browser console through various console.log, console.warn, and console.error statements throughout the codebase.

Recommendation

Implement conditional logging and sanitize sensitive data:

```
const isDevelopment = process.env.NODE_ENV === 'development'

function secureLog(message: string, data?: any) {
  if (isDevelopment) {
    console.log(message, sanitizeData(data))
  }
}
```

[L-07] Tabnabbing vulnerability in external links

External links using `target="_blank"` don't include the `rel="noopener noreferrer"` attribute, which can lead to reverse tabnabbing attacks where malicious sites can manipulate the parent window through the window.opener reference.

Example Issue : https://hackerone.com/reports/1145563

Code Location:

https://github.com/initia-labs/widget/blob/c7c7fc23a5d65cafbd8d748711a607abf695052a/packages/widget-react/src/components/ExplorerLink.tsx#L33.

https://github.com/initia-labs/widget/blob/c7c7fc23a5d65cafbd8d748711a607abf695052a/packages/widget-react/src/components/ExplorerLink.tsx#L33.

https://github.com/initia-labs/widget/blob/c7c7fc23a5d65cafbd8d748711a607abf695052a/packages/widget-react/src/pages/wallet/tabs/ManageChainsItem.tsx#L43.

Recommendation

Add `rel="noopener noreferrer"` to all external links.

[L-08] Missing security headers

The application lacks essential security headers that protect against common web vulnerabilities.

1. Content Security Policy (CSP) - No CSP header implemented.
2. HTTP Strict Transport Security (HSTS) - No HSTS enforcement.
3. X-Frame-Options - No clickjacking protection.
4. X-Content-Type-Options - No MIME type sniffing protection.
5. X-XSS-Protection - No XSS filtering directive.
6. Referrer-Policy - No referrer information control.
7. Permissions-Policy - No feature policy restrictions.

8. Cross-Origin-Embedder-Policy (COEP) - No cross-origin isolation.

9. Cross-Origin-Opener-Policy (COOP) - No cross-origin window isolation.

10. Cross-Origin-Resource-Policy (CORP) - No cross-origin resource protection.

Recommendation

Consider implementing the headers.

[L-09] Overly permissive CORS configuration

The Router API is configured with an overly permissive Cross-Origin Resource Sharing (CORS) policy that allows requests from any origin.

```
const app = await NestFactory.create<NestExpressApplication>(AppModule, {
  cors: true,  // Allows all origins, methods, and headers
  // ... other configuration
});
```

Example HTTP Response :

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Content-Type: application/json; charset=utf-8
Content-Length: 196
ETag: W/"c4-kJDgytQyX2ET4r6NyTo7CdMjllo"
Date: Sat, 21 Jun 2025 19:44:51 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{"environment":"dev",
"l1_nft_transfer_hook_address":
"0x42cd8467b1c86e59bf319e5664a09b6b5840bb3fac64f5ce690b5041c530565a",
"network_type":"mainnet",
"version":{"commit_sha":"commit_sha","tag":"tag"}}
```

Recommendation Consider implementing origin allowlisting.

[L-10] X-Powered-By: express header leaks technology stack

The application responds with the X-Powered-By: Express HTTP header, revealing that it is built with the Express.js framework. This information leakage may assist an attacker in crafting targeted exploits based on known vulnerabilities or misconfigurations in specific versions of Express or related middleware.

Example Response

```
HTTP/1.1 404 Not Found
X-Powered-By: Express
Access-Control-Allow-Origin: *
Content-Type: application/json; charset=utf-8
Content-Length: 63
ETag: W/"3f-BunLb98SCK6azHy0RO08GDnFBek"
Date: Wed, 18 Jun 2025 17:52:36 GMT
Connection: keep-alive
Keep-Alive: timeout=5
```

Recommendation

Suppress the X-Powered-By header in Express by adding the following middleware :

```
const express = require('express');
const app = express();

app.disable('x-powered-by');
```

[L-11] Oversized packet in ORDERED IBC channel can cause channel closure

In both the ICS-20 and ICS-721 specifications, the following statement is made:

> "Chains should ensure that there is some length limit on the entire packet data to ensure that the packet does not become a DOS vector. However, these do not need to be protocol-defined limits. If the receiver cannot accept a packet because of length limitations, this will lead to a timeout on the sender side."

This means that if a packet exceeds the receiver chain's accepted size and is therefore rejected, the sender side will experience a timeout. In the case of an ORDERED channel, such a timeout can lead to the channel being closed, as stated in the IBC specification:

> "In ORDERED channels, a timeout of a single packet in the channel closes the channel."

Therefore, sending an oversized packet to an ORDERED channel can cause the entire channel to close, disrupting all future communication between the two chains through that channel. For this reason, the `/nft` API should enforce a size limit on the returned `AminoMsg` messages to prevent sending oversized packets that could lead to timeouts.

[L-12] Missing `await` prevents proper error handling in `Erc20Service`

The functions `Erc20Service::_getRemoteTokenContract()` and `Erc20Service::_getTokenDenom()` incorrectly attempt to catch exceptions from asynchronous calls without using `await`. This results in the `try-catch` block not catching the error thrown from the asynchronous function, causing unhandled promise rejections.

```
private async _getRemoteTokenContract(...) {
  try {
=>   return fetchRemoteTokenContract(wrapperContract, localContract, rest);
  } catch (error) {
    ...
  }
}

private async _getTokenDenom(tokenContract: string, rest: string) {
  try {
=>   return fetchTokenDenom(tokenContract, rest);
  } catch (error) {
    ...
  }
}
```

Here, the `fetchRemoteTokenContract()` and `fetchTokenDenom()` functions are asynchronous. Since they are not awaited, the `try-catch` block will not catch any exceptions thrown during their execution.

Recommendation

Add `await` when calling the asynchronous functions so that any exceptions are correctly caught:

```
private async _getRemoteTokenContract(...) {
  try {
    const res = await fetchRemoteTokenContract
      (wrapperContract, localContract, rest);
    return res;
  } catch (error) {
    ...
  }
}
```

[L-13] Improper exception type and lack of input validation in OP bridge check

Vulnerable code: router-api/src/app/transfer/external/op.service.ts.

The `_isOpDenomValid` method performs a validation check to determine whether a provided l2Denom matches the expected hashed format derived from a bridgeId and l1Denom. If the bridgeId is not a decimal string, the method throws an `InternalServerErrorException`, which is a server-side error intended for application bugs, not for invalid client input.

Relevant code:

```
if (!/^\d+$/.test(bridgeId)) {
  throw new InternalServerErrorException(
    `Invalid bridgeId format: expected a decimal string, got "${bridgeId}"`,
  );
}
```

Impact: The logic is internal and the impact is limited, but it reflects weak input validation boundaries and incorrect use of HTTP semantics:

- Misuse of exception type may result in unintended 500 Internal Server Errors, even for simple client mistakes.
- May result in unhandled errors or unclear API responses if bridgeId is user-provided and invalid.

Recommendation Replace with appropriate exception:

```
throw new BadRequestException
  (`Invalid bridgeId format: expected decimal string`);
```

[L-14] Lack of validation on NFT metadata in `WithNormalizedNft`

Vulnerable code: packages/widget-react/src/pages/wallet/tabs/nft/WithNormalizedNft.tsx.

This component is used to fetch and normalize NFT metadata for rendering across the UI. It does so by:

- Pulling metadata from a URI provided by the on-chain nftResponse.
- Passing the resulting NormalizedNft object to child components (which often render fields like name, image, description, etc.).

Key concern: the fetched metadata is entirely unvalidated and unsanitized, and may originate from untrusted or unsanitized external sources.

```
const { data: metadata = {} } = useNftMetataQuery(nftResponse.nft.uri)
const nft = normalizeNft(nftResponse, metadata)
return children(nft)
```

Impact: This exposes the application to:

- Security: `data:image/svg+xml,...` payloads could embed malicious or deceptive content.
- Layout Abuse: Unbounded name or description fields may break UI alignment.

Recommendation React does escape raw HTML in JSX, but it does not block malicious image sources, overly long strings, or spoofed names. Harden `normalizeNft()`

Ensure that fields like image and name are explicitly:

- Trimmed.
- Length-limited.
- Source-verified.

[L-15] No response validation for external skip API

Vulnerable code: router-api/src/app/transfer/external/skip.service.ts.

I acknowledge this is a known issue, the Router API proxies responses from Skip without verification, and currently no integrity checks or contract allowlists are enforced. While syncing all contract addresses may be complex, I still recommend implementing runtime schema validation (e.g., with zod) to detect malformed or tampered responses early. This lightweight measure significantly reduces risk, even without full contract verification.

Recommendation

Example using zod:

```
const SkipRouteSchema = z.object({
  routes: z.array(
    z.object({
      source_chain_id: z.string(),
      dest_chain_id: z.string(),
      tx: z.any(), // Placeholder for the actual tx structure
      //(to be defined strictly later)
    })
  )
});
```

What this does: Declares that a valid Skip route response must be an object with a field routes. Routes must be an array of objects where each object has:

- source_chain_id: a string.
- dest_chain_id: a string.
- tx: any data (this is a placeholder, and should ideally be more strictly typed).

```
const parsed = SkipRouteSchema.safeParse(data);
```

Takes the raw data from the Skip API and attempts to parse it using the schema.

- safeParse is non-throwing: it returns { success: true, data } or { success: false, error }

```
if (!parsed.success) {
  throw new BadRequestException("Invalid response from Skip");
}
return parsed.data;
```

If the response does not match the expected structure, an exception is thrown, preventing unsafe processing. If valid, the typed and safe parsed.data is returned for further use.

Define one schema per method (Chains, Assets, Route, Msgs, etc.). Reject any unexpected or malformed fields.

[L-16] Insecure TLS (1.0 & 1.1) active on public endpoints

Several public-facing services in the Initia stack still permit legacy TLS versions 1.0 and 1.1, which are known to be vulnerable to multiple cryptographic attacks (BEAST, POODLE, etc.) and lack modern cipher suites. This affects the confidentiality and integrity of all data in transit for the following endpoints:

- Router API URL (e.g. https://api.initia.xyz).
- Registry API (https://registry.initia.xyz).
- Scan Endpoint etc.

Allowing TLS 1.0/1.1 undermines secure communications, exposing users' wallet interactions, transaction data, and cross-chain messages.

Recommendations

- Disable TLS 1.0 and 1.1 on all web servers, load balancers, and API gateways serving these domains.
- Enforce TLS 1.2+ with strong cipher suites (e.g. ECDHE_A* ciphers, AES-GCM, or ChaCha20-Poly1305).
- Implement HTTP Strict Transport Security (HSTS) with a long max-age and includeSubDomains.

[L-17] Missing HTTP request timeouts in axios clients on SKIP APIs

Every call to Skip's HTTP API in SkipService._requestSkip() uses axios.request(...) without a timeout. A slow or unresponsive Skip backend can block the Router API's event loop threads, causing request handlers to hang indefinitely and ultimately leading to resource exhaustion (Denial-of-Service).

```
// app/transfer/external/skip.service.ts
private async _requestSkip<T = unknown>(
  url: URL,
  body?: BodyInit,
  method?: string,
) {
  // …
  const { data } = await axios.request<T>({
    data: body,
    headers,
    method,
    url: url.toString(),    // ← no timeout configured
  });
  return data;
}
```

Other Affected Files

- `shared/account/account.service.ts` `AccountService._l2AccountExists()` `AccountService._createL2Account()`
- `shared/registry/registry.service.ts` `RegistryService.getChains()` (the `axios.get("/chains.json")` call)

Recommendations Configure a sensible per-request timeout (e.g. 5 000 ms) on the axios client. For example:

```
// at top of skip.service.ts
const skipHttp = axios.create({
  baseURL: SKIP_API_URL,
  timeout: 5_000,          // 5 seconds
});

// then inside _requestSkip:
const { data } = await skipHttp.request<T>({
  url: url.pathname + url.search,
  method,
  headers,
  data: body,
});
```

This ensures that if Skip does not respond within 5 s, the call errors can be retried or surfaced to the client, rather than hanging forever.

[L-18] Missing HTTP request timeouts across modules in widgets

Several hooks and helpers in `src/pages/bridge/data/` spin up ky clients without ever specifying a timeout. Any unresponsive or maliciously redirected endpoint can cause those requests to hang indefinitely (or until Ky's internal default, which may be very long), blocking the React Suspense boundaries and effectively denying service in the UI. This akso affects other modules listed below. Affected Files `account.ts` – `waitForAccountCreation()` `assets.ts` – `useSkipAssets()` `balance.ts` – `useSkipBalancesQuery()` `chains.ts` – `useSkipChains()` `simulate.ts` – `useRouteQuery()` `skip.ts` – global `useSkip()` client factory `tx.ts` – all `skip.get(…) / skip.post(…)` calls in `useBridgeTx`, `useTrackTxQuery`, `useTxStatusQuery`

`packages/widget-react/src/pages/bridge/op/data.ts`: `useLayer1RestClient()` → creates `ky.create({ prefixUrl: restUrl })` `useWithdrawals()` → `ky.create({ prefixUrl: executorUrl })` `useLatestOutput()` & `useOutput()` → both use `restClient` from `useLayer1RestClient()` `useOutputResponse()` → `ky.create({ prefixUrl: executorUrl })`

Transaction Module `src/pages/tx/TxRequest.tsx` (gas-prices fetch)

Wallet Activity `src/pages/wallet/tabs/activity/data.ts` (`useTxs`) `wallet/tabs/nft/queries.ts` (all `ky.create` calls)

Any other module that does `ky.create({ prefixUrl })` without timeout

Sample Snippet

```
// src/pages/bridge/data/skip.ts
export function useSkip() {
  const { routerApiUrl } = useConfig()
  // no timeout configured here
  return useMemo(() => ky.create({ prefixUrl: routerApiUrl }), [routerApiUrl])
}
```

```
// src/pages/bridge/op/data.ts

// 1) Global Layer-1 client without timeout
function useLayer1RestClient() {
  const { restUrl } = useLayer1()
  return useMemo(() => ky.create({ prefixUrl: restUrl }), [restUrl])
}

// 2) Withdrawals query without timeout
await ky
  .create({ prefixUrl: executorUrl })
  .get(`withdrawals/${address}`, { searchParams })
  .json<WithdrawalTxListResponse>()

// 3) Output-response query without timeout
ky.create({ prefixUrl: executorUrl })
  .get(`withdrawal/${sequence}`)
  .json<OutputResponse>()
```

Recommendation Configure all `Ky` clients with a sensible per-request timeout (e.g. 5 000 ms) and an optional retry policy. The simplest, unified fix is to update your global `useSkip()` and `waitForAccountCreation` clients.

[L-19] Unhandled exception in `quantitySuperRefine()`

Within the Zod refinement, `BigNumber(quantity)` is invoked without guarding against invalid strings. If the quantity is not a valid numeric string (NaN), `BigNumber` will throw, bubbling an uncaught exception and possibly crashing the form.

```
// src/data/form.ts
if (BigNumber(quantity).isZero()) {
  …
}
```

Recommendation Pre-validate `quantity` as a numeric string before creating a `BigNumber`, or wrap in try/catch:

```
if (!/^\d+(\.\d+)?$/.test(quantity)) { // You can introduce a better regex
  ctx.addIssue({ /* invalid format message */ })
  return
}
```

[L-20] Missing curve validation in

`encodeEthSecp256k1Pubkey`

In packages/widget-react/src/data/patches/encoding.ts, the function `encodeEthSecp256k1Pubkey` function performs basic structural validation (33-byte length and `0x02/0x03` prefix) but does not cryptographically verify that the public key represents a valid point on the secp256k1 curve. While this doesn't pose an immediate security risk in the current implementation, it could lead to:

- Downstream Processing Issues: Some libraries may behave unpredictably when given invalid curve points.
- UI Confusion: Wallet interfaces might display incorrect derived addresses for malformed keys.

```
// packages/widget-react/src/data/patches/encoding.ts
export function encodeEthSecp256k1Pubkey
  (pubkey: Uint8Array): EthSecp256k1Pubkey {
  if (pubkey.length !== 33 || (pubkey[0] !== 0x02 && pubkey[0] !== 0x03)) {
    throw new Error(

             "Public key must be compressed secp256k1, i.e. 33 bytes starting with 0x0
    )
  }
  return {
    type: pubkeyTypeInitia.ethsecp256k1,
    value: toBase64(pubkey),
  }
}
```

Risk Context This is primarily a defensive improvement rather than a critical fix. The current implementation is safe when: Keys come from trusted sources (wallets/key generators). Downstream systems properly validate keys.

The main benefits are: Earlier detection of malformed keys. More predictable behavior across different client implementations.

Recommendation For defense-in-depth, consider adding cryptographic validation using `@noble/secp256k1`:

```
import { Point } from "@noble/secp256k1";
import { toBase64 } from "@cosmjs/encoding";

export function encodeEthSecp256k1Pubkey
  (pubkey: Uint8Array): EthSecp256k1Pubkey {
  // 1. Structural checks
  if (pubkey.length !== 33 || (pubkey[0] !== 0x02 && pubkey[0] !== 0x03)) {
    throw new Error("Invalid compressed secp256k1 key format");
  }

  // 2. Cryptographic validation
  try {
    // Throws if:
    // - X-coordinate is >= curve field size
    // - Point is not on the curve
    // - Point is at infinity
    Point.fromHex(pubkey);
  } catch (err) {
    throw new Error(`Invalid secp256k1 public key: ${err.message}`);
  }

  return {
    type: pubkeyTypeInitia.ethsecp256k1,
    value: toBase64(pubkey),
  };
}
```

[L-21] Insufficient confirmation depth for reorg protection

He `waitForTxConfirmationWithClient` function in `194:225:packages/widget-react/src/data/tx.ts` only waits for a transaction to appear in a single block without requiring additional confirmations. This creates an issue where blockchain reorganizations can reverse "confirmed" transactions.

Code Location : tx.ts#L210-L211

The current implementation immediately returns upon finding a transaction in any block:

```
const tx = await client.getTx(txHash)
if (tx) {
  if (tx.code !== 0) throw new Error(tx.rawLog)
  return tx  // ❌ No confirmation depth check
}
```

- **Single Block Finality**: Treats any block inclusion as final confirmation.
- **No Reorganization Detection**: Missing fork detection mechanisms.

Recommendations

Consider implementing confirmation depth requirements.

[L-22] Unescaped NFT metadata in `CollectionDetails.tsx`

Vulnerable code: widget-react/src/pages/wallet/tabs/nft/CollectionDetails.tsx

This component renders collection.name and nft.name directly into JSX. If these fields contain malicious or malformed content (e.g., SVG payloads or homoglyph characters), they could result in:

- Visual spoofing of UI.
- Rendering glitches.

DOM injection if the string is later used in a context that bypasses React's escaping

```
<Page title={collection.name}>          // Unescaped collection name
<div className={styles.name}>{nft.name}</div>  // NFT name rendered directly
```

Impact: This exposes the application to:

- Cross-Site Scripting (XSS) if an SVG or data URI is rendered.
- Broken UI/UX: Unexpected characters or malformed strings may corrupt rendering.

Recommendations

React does escape certain HTML by default (e.g., <, >), but it does not:

- Validate content origin or structure.
- Prevent Unicode spoofing or complex rendering issues.
- Guarantee safety if developers later switch to dangerouslySetInnerHTML.

This makes it unsafe to render untrusted metadata directly, especially in apps working with NFTs or cross-chain metadata.

Escape or Sanitize Metadata For plain text:

```
const safeName = escapeHtml(nft.name);  // Custom util or use `he` lib
```

For rich text (if required):

```
import DOMPurify from "dompurify";
<div dangerouslySetInnerHTML={{ __html: DOMPurify.sanitize
  (nft.description) }} />
```

[L-23] Unvalidated Wallet Image Source

Vulnerable code: widget-react/src/pages/bridge/BridgeAccount.tsx

This image source (connected.image) is dynamically loaded from user-selected wallet metadata via useCosmosWallets(). There is no validation or sanitization before rendering it using the shared `<Image />` component.

```
{type
   === "src" && connected && <Image src={connected.image} width={18} height={18} />}
```

Impact:

- SVG-based XSS.
- Visual spoofing of the wallet icon (e.g. mimicking MetaMask).
- Proxy abuse if rendered through an unfiltered CDN.

Proof of Concept (PoC): Attacker provides wallet metadata (e.g. in a community wallet config) with:

```
```

Recommendations

Validate the image before rendering:

- Only allow https:// from trusted domains.
- Block data:, javascript:, or any unknown schemes.
- Consider server-side validation of wallet metadata.