

V12 reports three critical flaws in x402 SDKs for AI agent payments

Source copy kept by wheresthe.ai for the story "V12 reports three critical flaws in x402 SDKs for AI agent payments" (<https://wheresthe.ai/d/intl-v12-reports-three-critical-flaws-in-x402-sdks-for-ai-ag-2026-10/>).

Source: <https://v12.sh/blog/x402>

Text saved 23 Oct 2026 (the page as published).

SHA-256 of the saved text: ec3ba9e1b8abbbeb56045e7954468531f78dd09fc42799db9fae7adef1200707

x402 lets software pay for an API request as part of an ordinary HTTP exchange. A server responds with 402 Payment Required , the client signs a payment payload and retries the request with the payload attached, which then proceeds. This is useful for automated clients and AI agents that need to buy data or services without a person approving every request.

V12 reviewed the x402 codebase and found 25 vulnerabilities .

This post focuses on the three critical findings. An attacker could make a Solana payment appear successful without transferring tokens, collect a login signature for an unrelated website, or get an automated client to authorize spending its entire balance of a supported token.

Each issue crossed a different trust boundary: what counted as a payment, which website a signature was for, and who decided how much the wallet could spend.

The findings

A typical x402 exchange involves a client, a server selling a resource, and a facilitator that checks and submits payments. The server supplies payment details, including the amount, token, network, and recipient. The client signs, and the facilitator checks that the payment satisfies those requirements.

That flow needs protection in both directions. A merchant must be able to trust that an accepted payment actually pays them. A client must be able to contact an unfamiliar server without giving it unrestricted control over its wallet.

An approval was accepted as a payment, without transferring funds

The TypeScript Solana facilitators for the v1 and v2 exact schemes checked that a transaction called an allowed token program. They then parsed its third instruction as TransferChecked , the instruction used to transfer tokens.

The missing step was checking which instruction the transaction actually contained. Simplified, the parsing step looked like this:

```
1
const transferIx = instructions [ 2 ] ;
2
const programAddress = transferIx . programAddress . toString () ;
3
4
// After checking that the token program is allowed:
5
const parsedTransfer = programAddress === TOKEN_PROGRAM_ADDRESS . toString ()
6
? parseTransferCheckedInstructionToken ( transferIx as never )
7
: parseTransferCheckedInstruction2022 ( transferIx as never ) ;
```

Despite its name, the parser did not enforce the instruction type. It decoded the bytes into fields without checking that the first byte identified a transfer.

Another instruction, `ApproveChecked`, had the same data layout. The two differed only in the first byte:

`TransferChecked: 0c 40420f0000000000 06`

`ApproveChecked: 0d 40420f0000000000 06`

An approval grants spending permission; it does not move tokens. Its account layout also lined up with the transfer parser: the account naming the delegate was read as the payment destination. An attacker could put the merchant's expected token-account address there and satisfy the destination check, while the amount and token checks passed too.

Our local test used the real token encoders and parser and confirmed that the verification path returned `isValid: true` for this approval. The settlement code also skipped the post-settlement payment check for this path. The expected on-chain outcome followed from the token program's approval logic; the test did not submit a live transaction.

This was a complete payment bypass: a merchant relying on the affected facilitator could deliver a paid resource while receiving no tokens. An attacker needed a valid token account but no token balance for the approval, and the facilitator would pay the transaction fee. Repeating the attack could consume both the merchant's paid resources and the facilitator's fee budget.

A server could request a login signature for another website

The `Sign-In-With-X` extension lets clients authenticate using their wallets. On Ethereum, it builds a `Sign-In` with Ethereum message containing the website, wallet address, and a one-time login challenge, called a `nonce`.

The automatic wrapper signed messages supplied in a 402 response without checking that their domain and uri matched the website the client had contacted. The client copied those fields directly from the server:

1

```
const completeInfo : CompleteSIWxInfo = {
```

2

```
  domain : serverExtension . domain ,
```

3

```
  address : signer . address ,
```

4

```
  statement : serverExtension . statement ,
```

5

```
  uri : serverExtension . uri ,
```

6

```
  version : serverExtension . version ,
```

7

```
  chainId : serverExtension . chainId ,
```

8

```
  nonce : serverExtension . nonce ,
```

9

```
  issuedAt : serverExtension . issuedAt ,
```

10

```
};
```

For example, an agent could request data from data.example and receive a login message naming app.example . With an automatic signer, the wrapper would sign it and send the signature back to data.example .

An attacker controlling the data server could first obtain a fresh login challenge from the target website in a session they controlled. They could place that challenge in the extension, collect the client's signature, and submit it to the target site to log in as the client's wallet.

The target site's normal domain and signature checks would still pass: the message named that site and carried a genuine signature. The missing check belonged on the client, before signing.

Our test confirmed that the real SIWE library produced a validly formatted message for a domain unrelated to the requested host. The critical impact was session takeover wherever the target accepted that relayed login challenge. The attacker could perform whatever actions the authenticated session allowed, independently of the price of the original API request.

Default client setup had no spending limit

The documented default client setup did not enforce a per-payment limit or a total spending budget. It selected the first supported payment option without applying a spending policy. The server supplied the amount, token, recipient, and network that reached the signing code.

The relevant flow can be reduced to:

```
1
// Simplified: server-provided requirements become payment fields.
2
const selected = accepts [ 0 ] ;
3
const payment = {
4
  to : selected . payTo ,
5
  value : selected . amount ,
6
  asset : selected . asset ,
7
  network : selected . network ,
8
};
```

A hostile API could therefore turn a routine data request into a request for the wallet's full balance of a compatible token. If necessary, an earlier small payment could reveal the wallet address, allowing the server to look up its balance before choosing the larger amount.

With an automatic signer and no application-supplied policy, the client could sign that authorization. For a token supporting the signed transfer mechanism, the attacker could submit it and collect the funds. The practical limits were the wallet's balance and any external signer restrictions.

The HTTP decoding path also parsed payment requirements without calling the existing schema validator. Our local test showed malformed requirements reaching the selector even though that validator would have rejected them. But schema validation alone would not fix the spending issue: a correctly formatted request can still ask for

far more money than the client intended to pay.

This made the issue critical for funded automated clients. A single hostile server response could obtain authorization to drain a supported token balance without a separate approval prompt.

Disclosure and fixes

We reported the 3 critical and 13 high severity findings to the x402 team on August 12, 2026. The 3 critical vulnerabilities were fixed by August 20th. A further 9 medium and low severity findings were submitted on September 9th.

The security requirements are straightforward. The Solana verifier should check the instruction type before parsing it as a transfer. The sign-in client should compare the requested domain and URI with the contacted website before signing. Payment clients should validate incoming requirements and enforce explicit per-payment and total budgets, alongside permitted tokens, networks, and recipients.

These checks must run before the client signs or the facilitator accepts a payment. Optional policies only protect applications that actually enable them.

Conclusion

Automatic payments depend on clear limits. A successful transaction must transfer the required funds, a login signature must belong to the website being contacted, and a price supplied by a server must fit the client's spending policy.

These vulnerabilities were found by V12. V12 is your security agent for mission-critical software.

Find bugs like this in your code: <https://v12.sh/> .