Hybrid EIP-712 Verification Testing Methodology

October 29, 2025
Shaun Miller (mourning_dove)

This testing methodology and all associated code snippets are provided ***for educational, research, and security audit purposes only****. They are designed exclusively for use on* ***private, local, or isolated test environments*** *(e.g., local `forked` instances, `Foundry` chains, or `Ganache`) and* ***must not*** *be deployed on or used to interact with any public, mainnet, or production blockchain network. Any unauthorized or illegal activity is strictly prohibited and not endorsed by this documentation.*

This document details a hybrid testing methodology for securing `EIP-712` typed data signing and on-chain verification. The approach leverages the speed of `Foundry` for smart contract integrity combined with `JavaScript/Ethers.js` for realistic off-chain signature generation. Our generalized test suite is **open-source** and is designed for use across a multiple protocols. This methodology provides a foundation that security researchers are encouraged to utilize and expand upon for developing new proofs of concept (`PoCs`).

Contents

1 Off-Chain Signatures **2**
1.1 EIP712 Signature 2
1.2 Signing in Ethers 2

2 On-Chain Verification **2**

3 Hybrid Testing **2**
3.1 Simple Smart Contract that Checks signature 2
3.2 Calldata Generation 2
3.3 Hybrid Test Integration 2

4 Fuzz Testing **2**

5 Conclusion **3**

1 Off-Chain Signatures

1.1 EIP712 Signature

1.2 Signing in Ethers

2 On-Chain Verification

Unit tests verify the core cryptographic and access control properties directly on-chain. Nothing should touch the javascript yet. We will do something a little more complicated with call data for the hybrid tests. Here we will just run through how to verify in foundry. Key tests include:

1. **Digest Fidelity:** Asserting that the contract's computed `EIP-712` digest matches the expected digest generated by a trusted off-chain utility.

2. **Expired Deadline:** Ensuring a valid signature is rejected if the `deadline` is in the past.

3. **Replay Protection:** Verifying the `nonce` mechanism correctly prevents reuse of a previously consumed signature.

4. **Domain Separation:** Verifying that modifying the `chainId` or `verifyingContract` (address of `SignedVault`) invalidates the signature.

3 Hybrid Testing

3.1 Simple Smart Contract that Checks signature

Here we create and deploy a smart contract that verifies the user signature.

3.2 Calldata Generation

A critical weakness in `EIP-712` systems is the mismatch between off-chain signing libraries and on-chain verification logic. This phase uses a `JavaScript` test harness to simulate a real user signing a message. Here we go through a test that generates data, signs the data, and produces the call data all in JavaScript. Foundry will call this script and use the calldata to input into our contract that checks the signature.

3.3 Hybrid Test Integration

The `Foundry` test suite incorporates a custom external script call to the `JavaScript` environment.

- **Setup:** The `Foundry` test creates the necessary `Transfer` struct and passes the raw data to the `JavaScript` harness via a specific command-line call.

- **Execution:** The `JavaScript` environment signs the data and outputs the resulting `R`, `S`, `V` signature as a hex string to `stdout`.

- **Verification:** The `Foundry` test captures the signature from `stdout` and uses it in the `SignedVault.verify` function, ensuring the full end-to-end flow is validated.

4 Fuzz Testing

Maybe we will comeback to fuzz testing.

5 Conclusion