



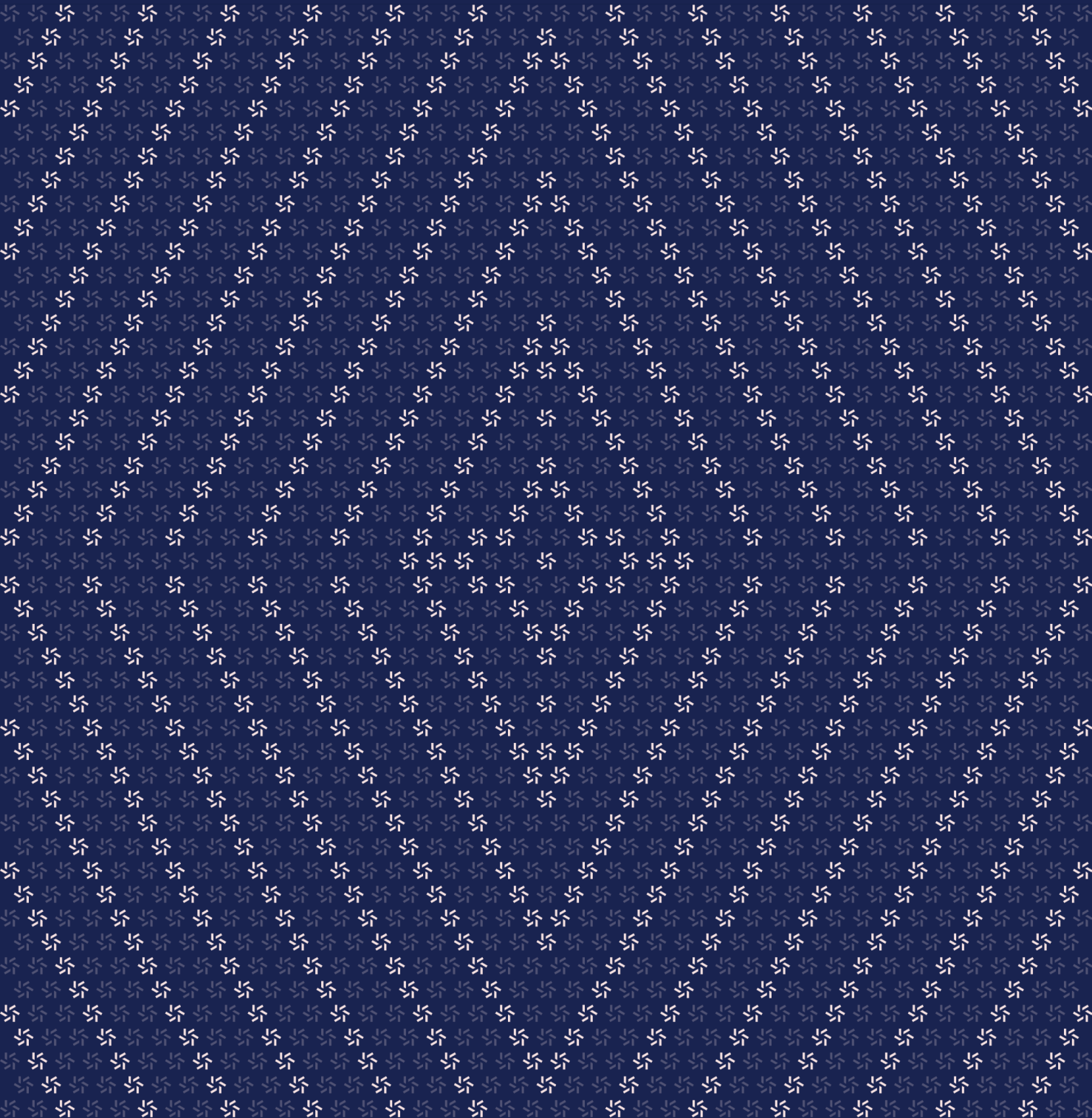
Prepared for
Stirling Poon
Nicu Badescu
James Flamant
Kraken

Prepared by
Katerina Belotskaia
Jinseo Kim
Zellic

August 18, 2026

Pull Oracle

Smart Contract Security Assessment



Contents

About Zellic	4
1. Overview	5
1.1. Executive Summary	5
1.2. Goals of the Assessment	5
1.3. Non-goals and Limitations	5
1.4. Results	5
2. Introduction	7
2.1. About Pull Oracle	7
2.2. Methodology	7
2.3. Scope	9
2.4. Project Overview	9
2.5. Project Timeline	10
3. Detailed Findings	11
3.1. Function names do not reflect validation behavior	11
3.2. Dead code in the hashing library	13
3.3. Constant derivations and edge-case branches are not covered by tests	16
4. System Design	19
4.1. Component: Signer registry and governance	19
4.2. Component: Payload authentication	21

5.	Assessment Results	26
5.1.	Disclaimer	26

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the [#1 CTF \(competitive hacking\) team](#) worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io and follow [@zellic_io](#) on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io.



1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Kraken from August 3rd to August 6th, 2026. During this engagement, Zellic reviewed Pull Oracle's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is there a way to cause positive attestation of any payload not created by CF Benchmarks?
 - Is it possible to mislead usage of the payload/attested data to attest to data that CF Benchmarks did not provide?
-

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

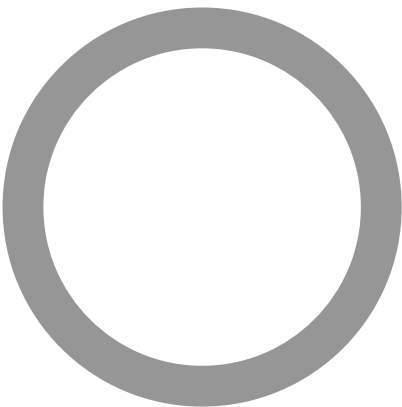
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Pull Oracle contracts, we discovered three findings, all of which were informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	3



2. Introduction

2.1. About Pull Oracle

Kraken contributed the following description of Pull Oracle:

CF Benchmarks Oracle V1 is a price oracle that attests that prices and other financial data are authentically originated from CF Benchmarks.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3. Scope

The engagement involved a review of the following targets:

Pull Oracle Contracts

Type	Solidity
Platform	EVM-compatible
Target	Zip file named delphi-e17a34a7b2b71e26c444a66510f797ca7cc374ab-with-diff.zip sha256: 5d1d56b35c52829fde5480951ee428b6b8b5e6214456e53274703e3fae63410
Version	e17a34a7b2b71e26c444a66510f797ca7cc374ab
Programs	src/CfbChainEncoding.sol src/CfbHashing.sol src/CfbPayloadAuthenticator.sol src/CfbPayloadDecoder.sol src/CfbProtocolBounds.sol src/CfbSignature.sol src/CfbSignerRegistry.sol src/CfbTypedDataValidation.sol src/CfbVerifier.sol src/CfbVerifierErrors.sol src/CfbVerifierUpgradeable.sol src/ICfbVerifier.sol proxydeployer/src/ProxyDeployer.sol

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.2 person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
✈ Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
✈ Engineer
kate@zellic.io ↗

Jinseo Kim
✈ Engineer
jinseo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

August 3, 2026 Start of primary review period

August 4, 2026 Kick-off call

August 6, 2026 End of primary review period

3. Detailed Findings

3.1. Function names do not reflect validation behavior

Target	CfbPayloadDecoder.sol		
Category	Code Maturity	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

CfbPayloadDecoder contains several functions whose names describe something other than what they do. Every call site in the codebase at the time of writing is correct, so this is a matter of legibility rather than behavior.

1. An assertion named as an accessor, with a discarded return value

The `_readValidFeedIdHash` function is declared returns `(bytes32 feedIdHash)`, but its only call site invokes it as a bare expression statement and discards the returned value:

```
// validate()
_readValidFeedIdHash(payload);
```

Its behavior is purely assertive; read the field, and revert `InvalidFeedIdHash` if it is zero. The `_read` prefix places it in the same family as `readFeedIdHash`, which reads the same field at the same offset, performs no check, and genuinely returns a value.

The discarded return also costs a duplicate read: `feedIdHash` is read inside `_readValidFeedIdHash` and again at `CfbPayloadAuthenticator.sol:_verifyPayload`.

2. A partially validating function named as the complete one

The `hashPayload` reads as the safe counterpart to `hashValidatedPayload`; however, it does not perform the full validation. It performs the version and length checks while skipping `_readValidFeedIdHash` and `_validateDelegationCertificateFields`. It therefore accepts a payload carrying a zero `feedIdHash`, a zero operational key, a zero `feedsRoot`, a zero `schemeId`, a noncanonical delegation signature, or a `validToMillis` below the delegation window.

Impact

There is no impact on the current deployment. The finding is therefore informational: the names can lead a reader to expect checks that do not run, and can leave redundancy in place, as with the

discarded return value of `_readValidFeedIdHash`.

Recommendations

We recommend the following.

1. **Rename assertive helpers to state their role.** For example, rename `_readValidFeedIdHash` to `_requireValidFeedIdHash`, returning nothing.
2. **Document what `hashPayload` does not check.** Making `hashPayload` call `validate` would also make the name accurate, but test expectations should be checked first, since negative cases may rely on it accepting payloads that are structurally valid and semantically malformed.

Remediation

This issue has been acknowledged by Kraken, and fixes were implemented in the following commits:

- [434d0b5f](#) ↗
- [4851853a](#) ↗
- [f9739320](#) ↗

Kraken provided the following comment:

Functions renamed/removed. Overall, `CfbPayloadDecoder` aligned to match and clarify usage in `CfbPayloadAuthenticator`.

3.2. Dead code in the hashing library

Target	CfbHashing.sol		
Category	Code Maturity	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

Three functions in CfbHashing have no caller in the production sources, and one domain constant has no reader anywhere. Both hashChainEncodingId and emptyChainEncodingMerkleNode are unreferenced *throughout the repository*, while the emptyFeedIdMerkleNode is unused in production but does carry a test.

Function	Line	src/	test/	Consumed domain constant
hashChainEncodingId	103	0	0	DOMAIN_CHAIN_ENCODING_ID (:20)
emptyChainEncodingMerkleNode	116	0	0	DOMAIN_CHAIN_ENCODING_MERKLE_EMPTY (:24)
emptyFeedIdMerkleNode	95	0	1	DOMAIN_FEED_ID_MERKLE_EMPTY (:15)

Each of the three domain constants in the right-hand column is consumed only by its unreferenced function, so those are transitively unreachable from production code as well. Also, a DOMAIN_SCHEME_VALUE constant is orphaned outright; no function in the library reads it, because VALUE_SCHEME_ID is a hardcoded literal rather than a derivation from it.

The hashChainEncodingId is the more notable of the two unreferenced functions, because a hardcoded copy of its output sits 11 lines below it:

```
bytes32 internal constant EVM_ABI_V1_ENCODING_ID =
    0x605016be6c607dcde9404ed8bac43b6159e1bc60422bacbe3dc1e04768bc9475; //
    :28-29
```

We verified independently that this literal equals hashChainEncodingId(keccak256("evm/abi-v1")) and matches the specification.

Note the inconsistency between the two empty-node helpers: emptyFeedIdMerkleNode is verified by derivation in CfbHashing.t.sol:53-57, while its chain-encoding counterpart has no test at all, despite the canonical vectors containing the expected value at profile.chainEncodingMerkleEmptyNode.

Impact

There is no functional impact. Unused internal library functions are not included in the deployed bytecode, so there is no gas or code-size cost and no attack surface.

The concern is that unused code accumulates without the review pressure applied to code on a live path.

Recommendations

We recommend the following.

1. Use `hashChainEncodingId` to validate the literal.

```
_assertEqBytes32(
  CfbHashing.EVM_ABI_V1_ENCODING_ID,
  CfbHashing.hashChainEncodingId(keccak256("evm/abi-v1")),
  "evm profile id derivation"
);
```

2. Connect `DOMAIN_SCHEME_VALUE` to `VALUE_SCHEME_ID` by asserting the derivation in a test.

```
_assertEqBytes32(
  CfbHashing.VALUE_SCHEME_ID,
  keccak256(abi.encodePacked(PAYLOAD_VERSION,
    CfbHashing.DOMAIN_SCHEME_VALUE)),
  "value scheme id derivation"
);
```

3. Test the empty-node helpers. Both expected values are already present in the canonical vectors, at `profile.feedIdMerkleEmptyNode` and `profile.chainEncodingMerkleEmptyNode` in `docs/vectors/cfb-da-v1-merkle.json`, and `CfbDaV1MerkleInterop.t.sol` already loads that file, so covering both costs two assertions. If the helpers are instead being kept as a reference implementation for SDK authors, state that in a comment.

Remediation

This issue has been acknowledged by Kraken, and a fix was implemented in commit [89ce44f1](#).

Kraken provided the following comment:

Unused functions and variables in CfbHashing removed. All important literals converted to derived form in the production code, and literal form in the test code. Documentation pins all such literals.

3.3. Constant derivations and edge-case branches are not covered by tests

Target	Project-wide		
Category	Protocol Risks	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The test suite is strong, and the items below should be read as exceptions rather than as a characterization of it. Running the project's own coverage workflow reports 92.54% of lines, 93.21% of statements, 94.55% of branches, and 97.59% of functions across `src/`, with 106 tests and none failing.

We identified three areas where coverage could be strengthened.

1. Hardcoded constants and hash constructions are never checked against their derivations

Several values in `src/` are written as literals whose relationship to the normative construction is asserted nowhere:

- `CfbHashing.VALUE_SCHEME_ID` and `CfbHashing.EVM_ABI_V1_ENCODING_ID` are hex literals. The specification defines each as a derivation — `H(1 || D_SCHEME_VALUE)` and `H(1 || D_CHAIN_ENCODING_ID || H("evm/abi-v1"))` respectively — but no test computes the derivation and compares.
- `CfbHashing.hashCertificate` has no direct check of its construction. It produces the certificate digest on which the entire delegation chain rests and is used in production (`CfbPayloadAuthenticator.sol`), yet its 125-byte preimage from specification §7 is compared against nothing beyond end-to-end scenarios.
- `CfbHashing.emptyChainEncodingMerkleNode` has no test at all.
- The 16 domain constants — `DOMAIN_PAYLOAD`, `DOMAIN_CERTIFICATE`, `DOMAIN_FEED_ID`, the four `DOMAIN_FEED_ID_MERKLE_{LEAF, NODE, EMPTY, ROOT}`, `DOMAIN_SCHEME_VALUE`, `DOMAIN_SCHEME_GENERIC`, `DOMAIN_COMMITMENT`, `DOMAIN_CHAIN_ENCODING_ID`, `DOMAIN_CHAIN_ENCODING_ENTRY`, and the four `DOMAIN_CHAIN_ENCODING_MERKLE_{LEAF, NODE, EMPTY, ROOT}` — are each the keccak256 of an ASCII label written in-line in the source. A typo in any label yields a silently different domain, and nothing compares either the labels or their resulting digests against the specification's table.

2. No property test for chain-encoding envelope parsing

The `CfbChainEncoding.verifyEnvelope` has tests for the size and depth bounds, but every envelope in the suite is built by a helper that keeps the declared length and the actual buffer in agreement. However, there is no test that breaks that agreement, so these cases are never run:

- A `CHAIN_ENCODING_ENTRY_PREFIX_BYTES + uint256(nativeDataLength)` exceeds the length. The 101-byte test case stops at the minimum-size check before reaching this line.
- An envelope whose length does not equal `entryLength + 1 + proofCount * 32` while the length stays within `[102, 8192]` and `proofCount <= 8`. The depth-nine test case stops at the proof-depth check, and the 8,193-byte case at the maximum-size check.

3. Three branches are never executed by the suite

Three branches in `src/` are never executed.

The first is a two-part condition whose halves differ in reachability, so it is listed twice:

Location	Uncovered code	Reachable from outside?
<code>CfbPayloadAuthenticator</code> .sol:131	<code>error != ECDSA.RecoverError.NoError</code>	Yes — an <code>r</code> that is not the x-coordinate of any curve point makes <code>ecrecover</code> return zero, which OpenZeppelin reports as <code>InvalidSignature</code> . The smallest such value is <code>r = 5</code> . The suite's only in-range value is <code>r = 1</code> , which <i>is</i> a valid x-coordinate, so recovery succeeds there and the revert happens later as a key mismatch.
<code>CfbPayloadAuthenticator</code> .sol:131	<code>signer == address(0)</code>	No — in the pinned OpenZeppelin version, every path that returns <code>NoError</code> also returns a nonzero signer.
<code>CfbPayloadDecoder</code> .sol:146	<code>length < VERSION_BYTES</code>	Yes — a zero-length payload: both <code>decode (: 77)</code> and <code>hashPayload (: 70)</code> read the version byte before any other length check.
<code>CfbSignerRegistry</code> .sol:119	<code>!record.active in narrowSignerValidity</code>	Yes — calling it for a signer that was never registered or has been deactivated.

Impact

There is no impact on the deployment at the time of writing. Every value and construction we checked is correct, with the domain constants and both derived identifiers independently verified

during review. The uncovered branches are correct as written.

The risk is one of maintenance. A change to a hardcoded constant, or to the field order or width within a preimage, could diverge from the normative specification without failing any test.

Recommendations

We recommend the following.

1. Assert the derivations in CfbHashing.t.sol instead of pinning literals:

```
_assertEqBytes32(
    CfbHashing.VALUE_SCHEME_ID,
    keccak256(abi.encodePacked(PAYLOAD_VERSION,
        CfbHashing.DOMAIN_SCHEME_VALUE)),
    "value scheme id derivation"
);
_assertEqBytes32(
    CfbHashing.EVM_ABI_V1_ENCODING_ID,
    CfbHashing.hashChainEncodingId(keccak256("evm/abi-v1")),
    "evm profile id derivation"
);
```

2. Add a fuzz suite for `verifyEnvelope`, targeting the items listed under the second heading of the Description section above.
3. Cover the three branches listed under the third heading of the Description section above — one test case each: `r = 5` with a canonical `s`, a zero-length payload, and `narrowSignerValidity` against a revoked signer.

Remediation

This issue has been acknowledged by Kraken, and fixes were implemented in the following commits:

- [333dc23e](#) ↗
- [b0a25ea7](#) ↗
- [06cd5ed7](#) ↗
- [4e77ec00](#) ↗

Kraken provided the following comment:

Literals covered in 3.2 remediation. Additional tests added per subpoint. Additional coverage test.

4. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

4.1. Component: Signer registry and governance

Description

Sources: *src/CfbSignerRegistry.sol* and *src/CfbVerifierUpgradeable.sol* contracts

What it is responsible for. It is responsible for all mutable states in the system and the authority to change it. It records which primary signers this deployment trusts and over which interval, along with the pause flag and the gate on replacing the implementation. Four roles may alter that state: `SIGNER_ROLE`, `PAUSER_ROLE`, `UPGRADER_ROLE`, and the default admin. The payload-authentication component queries it on every verification, and no other part of the system writes storage.

What it does. It registers primary signers against a proof that they control the key, narrows or revokes their validity, gates pause and unpauses, authorizes UUPS upgrades, and answers the authorization query that `_requireValidSignerForDelegation` asks during each verification.

How it does it. Two mappings hold the state — `_signers` keyed by address, and `_signerRegistrationNonces` — layered over OpenZeppelin's `AccessControlDefaultAdminRules`, `Pausable`, `EIP712`, and `UUPSUpgradeable`. Registration requires an EIP-712 signature from the prospective signer itself, bound through the domain separator to this chain and this contract, so an authority cannot enroll an address whose key it does not hold.

The design turns on one decision; *registration is permanent*. Once registered is set it is never cleared; only active flips. That is what makes the payload-authentication component's key-separation invariant durable — an expired, narrowed, or revoked primary signer remains permanently barred from acting as an operational key.

Invariants

The following properties are held.

- **Registration is once-only and monotonic.** An address can be registered at most once, and `registered` never returns to false. It is enforced by the `record.registered` check in `_addSigner` and by the absence of any code path that clears the flag.
- **Every registration proves key possession, scoped to this deployment.** The proof is an EIP-712 signature by the signer over `(signer, validFromMillis, validToMillis, nonce)`, with the domain separator binding it to the chain ID and the verifier address. It is enforced by `_requireSignerProof`; replay across chains and across deployments is rejected.

- **Registration nonces are globally single-use.** It is enforced by `_signerRegistrationNonces`. Its value is that a proof collected and stored off chain becomes provably unusable once spent.
- **Windows only ever shrink.** A new window must be contained in the existing one. It is enforced by the containment test in `narrowSignerValidity`.
- **Revocation is one-way and only from active.** An unregistered or already revoked signer cannot be revoked. It is enforced by the `!record.active` guard in `revokeSigner`.
- **The default-admin delay is always in (0, 30 days].** Enforced identically at initialization and in the `changeDefaultAdminDelay` override, so a zero delay that would defeat the two-step admin transfer cannot be set.
- **Pause is asymmetric.** The `PAUSER_ROLE` stops verification; only `DEFAULT_ADMIN_ROLE` resumes it.

These are properties deliberately *not* held:

- **The four authorities need not be distinct.** The `_requireAuthority` function checks only that each is nonzero, so one address may simultaneously hold the admin, signer, pauser, and upgrader.
- **Primary signers are not checked against role holders.** The registry has no knowledge of the role system, so a primary signer may also hold `DEFAULT_ADMIN_ROLE` or any other.
- **Nothing prevents registering an address already serving as an operational key.** Operational keys are never recorded on chain, so the check is impossible here. The consequence is described under the Attack Surface section below.
- **There is no reinitializer.** A future implementation needing to initialize new state must add its own.

Test coverage

Cases covered

- EIP-712 binding: digest changes across chain IDs and across verifier deployments and proof replay rejected in both directions
- Proof from the wrong signer, proof issued for a different validity window, and malformed proof scalars
- Nonce zero, nonce reused globally, and nonce *not* consumed when the proof fails
- Registration window inclusive at whole-second boundaries and registration outside the window
- Maximum validity accepted at exactly the 1,095-day limit — zero, duplicate, and inverted windows rejected
- Narrowing accepted, widening rejected; revoked signer cannot be re-added; repeated revocation of inactive or unregistered signers rejected
- Access control: non-holders cannot add, revoke, or narrow

Cases not covered

- `narrowSignerValidity` against an inactive or unregistered signer

Attack surface

What an unprivileged caller can reach. They can reach three view getters — `signerRecord`, `signerRegistrationDigest`, and `signerRegistrationNonceUsed` — plus `renounceRole` for their own address, which writes nothing when the caller holds no role because `_revokeRole` tests membership first. The `acceptDefaultAdminTransfer` function carries no role modifier but is restricted to the address the sitting admin already nominated. *No unprivileged path writes storage.*

The one exception is `initialize`, which has no access control at all and is guarded only by the `initializer` modifier. If a proxy is ever deployed without `initializer` calldata in the same transaction, the first caller takes default admin, all three authorities, and both signer slots.

This is the *privileged ladder*, in descending order of what a compromised key achieves:

- `UPGRADER_ROLE` is total. A valid UUPS implementation of the holder's choosing replaces all logic.
- `DEFAULT_ADMIN_ROLE` can grant itself `SIGNER_ROLE`, `UPGRADER_ROLE`, or `PAUSER_ROLE`.
- `SIGNER_ROLE` can register a new primary signer, but only one whose key it controls, since registration demands proof of possession. That is sufficient for complete forgery of attestations; a signer it controls can certify any operational key over any feed manifest.
- `PAUSER_ROLE` can only halt and cannot resume.

Griefing through legitimate privileged actions. Two operations take feeds offline immediately, for as long as the 24-hour delegation window (`CfbPayloadDecoder.DELEGATION_WINDOW_MILLIS`) leaves certificates outstanding:

1. Registering an address that is currently an operational key in outstanding certificates — every payload signed by that key then fails `SignerRoleCollision`.
2. Narrowing a signer's window — because authorization tests *containment* rather than overlap, any live certificate extending past the new boundary is rejected at once — even for observations inside the remaining overlap.

4.2. Component: Payload authentication

Description

Sources: `src/CfbPayloadAuthenticator.sol`, `src/CfbPayloadDecoder.sol`, `src/CfbHashing.sol`, and `src/CfbSignature.sol`

What it is responsible for. It is responsible for deciding whether a byte string handed over by an untrusted caller is an authentic observation issued by a signer this deployment trusts and whether it is recent enough for the caller's own tolerance. It is the trust core; every other path in the system either delegates to it or interprets its output.

What it does. The `_verifyPayload(payload, signature, maxAgeMillis)` function either reverts with a typed error or returns an authenticated header — feed hash, timestamp, scheme,

data commitment, and the two identities involved. It never interprets the data itself.

```
/// @notice Authenticated signed header fields returned without interpreting
    the data commitment.
struct VerifiedPayload {
    bytes32 feedIdHash;
    uint64 timeMillis;
    bytes32 schemeId;
    bytes32 dataCommitment;
    address primarySigner;
    address operationalKey;
    uint8 version;
}
```

How it does it. The protocol uses two-level key delegation with nested signatures. A long-lived primary signer, registered on chain, issues a certificate authorizing a short-lived operational key for exactly 24 hours, one exact data scheme, and a Merkle-committed set of feed identifiers. That certificate is embedded in the payload; the operational key then signs the whole payload, certificate signature included.

The payload is a fixed-width frame of 230 + 32d bytes:

offset	size	field	signed by
0	1	version = 1	
1	20	operationalKeyId	
21	8	validToMillis	certificate fields – primary signer
29	32	feedsRoot	
61	32	schemeId	
93	64	certificateSignature	(EIP-2098, over the four fields above)
157	8	timeMillis	
165	32	dataCommitment	observation
197	32	feedIdHash	
229	1	feedProofCount = d	
230	32*d	feedProof	

The detached 64-byte operational signature is a separate argument and is not part of the frame. The nesting is what prevents splicing: the payload digest covers certificateSignature, so a different certificate cannot be attached to the same observation without invalidating the operational signature.

Verification proceeds in the order below:

1. Validate every structural invariant of the frame
2. Read feedIdHash, timeMillis
3. Check freshness against the destination chain's clock (_requireFresh)

4. Derive `validFromMillis = validToMillis - 24h`
5. Require the observation to fall inside the certificate window
(`_requirePayloadTimeInsideDelegation`)
6. Recover the certificate signer, first `ecrecover`, over the certificate digest
(`_recoverDelegationSigner`)
7. Require registry authorisation for that signer (`_requireValidSignerForDelegation`)
8. Require the feed to be inside the committed manifest
(`_requireFeedIdMerkleMembership`)
9. Verify the operational signature, second `ecrecover`, over the whole payload
(`_requireKeySignature`)
10. Reject if the operational key was ever a registered primary signer

Invariants

These are properties that hold for every payload the component accepts.

- **Framing is exact.** The accepted length is one of exactly nine values, $230 + 32d$ for $0 \leq d \leq 8$, and d equals the declared proof count. It is enforced by `_requireCfbDaV1Length`.
- **Time is triply constrained.** The observation is not in the future, is no older than the caller's own limit (itself capped at 31 days), and lies inside the certificate's exact 24-hour window, and that window lies *wholly inside* the primary signer's registered window. It is enforced by `_requireFresh`, `_requirePayloadTimeInsideDelegation`, and `_requireValidSignerForDelegation`.
- **Delegation is authorized.** The certificate's four fields carry a signature recovering to a signer that is registered *and* currently active. It is enforced by `_recoverDelegationSigner` followed by `_requireValidSignerForDelegation`.
- **The feed is within the delegated scope.** The claimed `feedIdHash` is a member of the Merkle manifest the primary signer committed to in `feedsRoot`. It is enforced by `_requireFeedIdMerkleMembership`.
- **The observation is signed by the delegated key.** The detached signature recovers to the operational key named in the certificate, over a digest covering the entire payload. It is enforced by `_requireKeySignature`.
- **Key roles never merge.** An operational key is never an address that has *ever* been registered as a primary signer. It is enforced by `_isRegisteredSigner`.
- **Signatures are canonical.** Both signatures are EIP-2098 compact with $0 < r < N$ and $0 < s \leq N/2$, so exactly one valid encoding exists per (key, digest). It is enforced by `CfbSignature.isCanonicalCompact`.
- **Verification writes no storage.** It is enforced by the type system — all four entry points are `view`, and the self-call inside `isValidPayload` compiles to `STATICCALL`.
- **Digest encodings are injective.** No two distinct field tuples produce the same digest at any of the 17 hash constructions.

These are properties deliberately *not* held, which callers must account for:

- **No binding to chain or contract.** Neither digest carries a chain ID or verifier address, so an attestation is valid at every deployment trusting the same primary signer. Governance is bound (registration proofs use EIP-712); the data it authorizes is not.
- **No scheme binding in `verifyPayload`.** The `schemeId` is returned but compared to nothing; the typed paths do the comparison, and a direct `verifyPayload` consumer must do it themselves.

Test coverage

Cases covered

- Every wire offset, by fuzz round trip of all 11 decoded fields against an independent builder (`CfbPayloadDecoderFuzz.t.sol`)
- Proof framing: declared depth against actual depth, over the full 0–8 range (`CfbPayloadDecoderFuzz.t.sol`)
- Length boundaries: $\text{MIN} - 1, \text{MAX} + 1$, every incomplete proof length, depths 0, 8, and 9
- Feed-ID hashing: preimage independently reconstructed, length fuzzed across 0–256, character set checked against a predicate oracle (`CfbStringValidationFuzz.t.sol`)
- Signature scalars, exhaustively at the boundaries: r in $\{0, 1, N-1, N, N+1, \text{max}\}$ and s in $\{0, \text{HALF_N}, \text{HALF_N}+1, \text{max}\}$, parity bit alone and combined (`CfbSignature.t.sol:21-46`)
- Signature length $\{0, 1, 32, 63, 65\}$ — both exact boundaries around 64
- Freshness: zero and maximum age accepted inclusively, future payload, stale payload, age limit above the 31-day cap
- Delegation window: accepted at both endpoints, rejected at ± 1 ms
- Merkle membership under an attacker holding the signing key: a raw leaf and a raw intermediate node written into `feedsRoot` **with the certificate re-signed**, both rejected (`CfbVerifierPayload.t.sol`) — adversarial zero-valued and patterned siblings required to reconstruct the signed root
- Key-role collision across all four signer states — active, expired, narrowed, revoked — plus a never-registered key permitted

Cases not covered

- Zero-length payload — the `length < VERSION_BYTES` branch (`CfbPayloadDecoder.sol:146`) is unexecuted — this is the path on which `_hashValidatedPayload` would underflow.
- The `hashCertificate` has no direct preimage check.

Attack surface

An attacker controls *the entire input*: all $230 + 32d$ payload bytes, all 64 signature bytes, and `maxAgeMillis`. Nothing about the call is privileged and nothing is rate-limited. Everything else —

the signer registry, the pause flag, the roles — is deployment state the attacker cannot touch through this component.

Controlled input	What constrains it
version	Strict equality with 1
frame length	One of nine discrete values — proof count must agree
operationalKeyId	Must equal the recovery of the detached signature and must never have been a registered primary signer
validToMillis	At least 86,400,000 (guards the window subtraction) — derived window must contain the observation and sit inside the signer's window
feedsRoot	Covered by the certificate signature, which must recover to an active registered signer — pinned <i>before</i> the proof is folded
schemeId	Covered by the certificate signature — compared against caller expectation in the typed paths
certificateSignature	Canonical EIP-2098 scalars; must recover to an authorized signer over the certificate digest
timeMillis	Not future, within caller's limit, inside the delegation window
dataCommitment	Opaque here — reconstructed and compared in the typed paths
feedIdHash	Nonzero — must be a Merkle member of the pinned root
feedProofCount, siblings	Count bounded to 8 and consistent with the frame length — the fold must reach the pinned root
detached signature	Exactly 64 bytes, canonical scalars, must recover to the operational key
maxAgeMillis	Capped at 31 days — this bounds the caller's <i>laxity</i> , so a careless integrator cannot accept a year-old attestation

Each degree of freedom is individually wide, but they are not independent; every one of them must simultaneously satisfy a signature the attacker cannot produce. Changing `feedsRoot` or `schemeId` breaks the certificate signature. Changing any byte at all breaks the operational signature, since the payload digest covers the frame in full, certificate signature included.

5. Assessment Results

During our assessment on the scoped Pull Oracle contracts, we discovered three findings, all of which were informational in nature.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Where Zellic states that a finding has been addressed or fixed in one or more specific commits, this indicates only that those commits introduce changes that, in our assessment, address the finding relative to the codebase version originally reviewed. This does not imply that Zellic reviewed other changes contained in those commits, the overall state of the codebase at those commits, or the interplay between remediation measures for different findings.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.