hexens × FUEL

SEPT.24

SECURITY REVIEW REPORT FOR FUEL

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

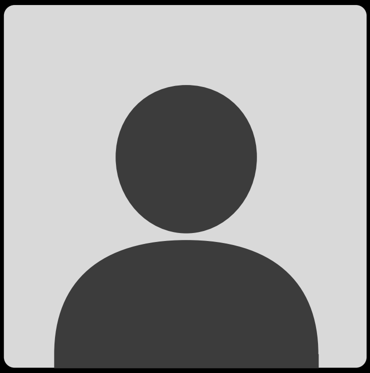
SCOPE

The analyzed resources are located on:

https://github.com/FuelLabs/fuel-connectors/tree/main/packages/evm-predicates/predicate
https://github.com/FuelLabs/fuel-connectors/tree/main/packages/solana-connector/predicate

The issues described in this report were acknowledged.

AUDITING DETAILS



STARTED

17.09.2024

DELIVERED

20.09.2024

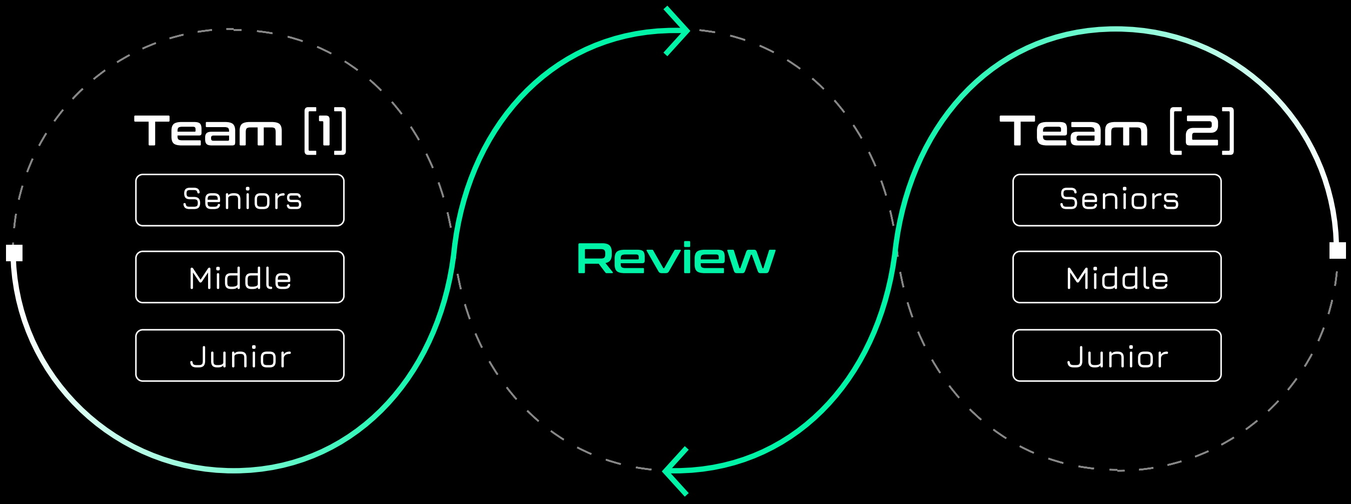
Review Led by

NOUREDDINE BENOMARI

Security Researcher | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

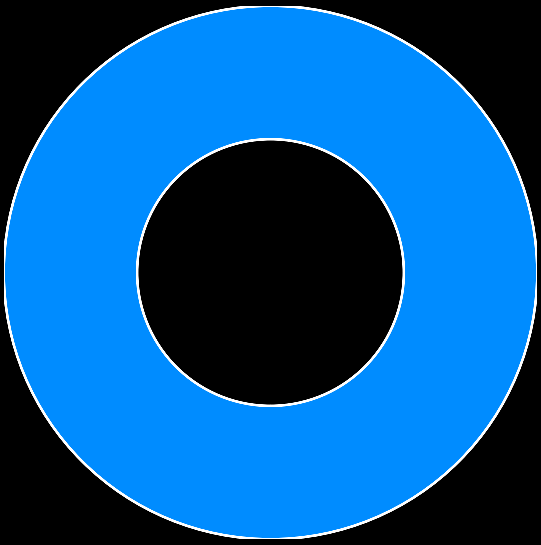
ISSUE SYMBOLIC CODES

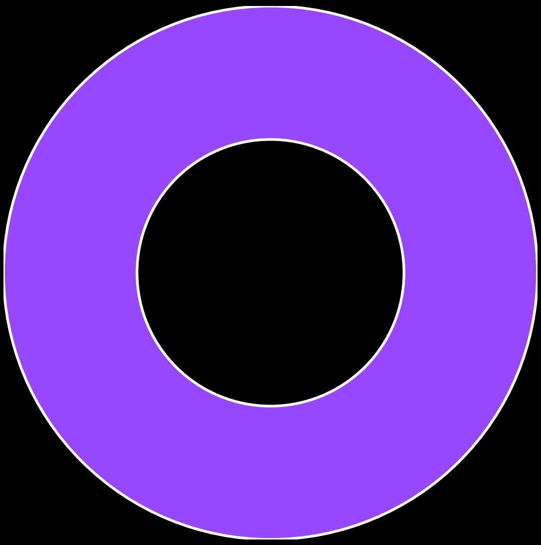
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	0
Medium	0
Low	0
Informational	2

Total: 2





WEAKNESSES

This section contains the list of discovered weaknesses.

FUEL7-11

CONNECTOR PREDICATE MAY REVERT INSTEAD OF RETURNING FALSE IF THE WITNESS INDEX IS GREATER OR EQUAL TO THE WITNESS COUNT

SEVERITY: Informational

REMEDIATION:

Consider catching None before unwrapping and return false in this case. This approach may be better because it will provide better information for this case.

STATUS: Acknowledged

DESCRIPTION:

The fn **main()** from evm-predicates and solana-predicates may revert if the **witness_index** is greater or equal to the witness count.

```
fn main(witness_index: u64) -> bool {
    // Retrieve the Ethereum signature from the witness data in the Tx at
the specified index.
    let signature: B512 = tx_witness_data(witness_index).unwrap();
```

```
fn main(witness_index: u64) -> bool {
    let signature: B512 = tx_witness_data(witness_index).unwrap();
```

The function **tx.sw::tx_witness_data()** may return **None** if the witness index is >= **tx_witnesses_count()**.

```
pub fn tx_witness_data<T>(index: u64) -> Option<T> {
    if index >= tx_witnesses_count() {
        return None
    }
```

In that case where **tx.sw::tx_witness_data()** returns **None**, **option.sw::unwrap()** is called on **None**, and as a result it will revert with panic.

```
    pub fn unwrap(self) -> T {
        match self {
            Self::Some(inner_value) => inner_value,
            _ => revert(0),
        }
```

FUEL7-12

THE SIGNATURES ARE NOT USER-FRIENDLY

SEVERITY: Informational

REMEDIATION:

We recommend using a more explicit signature scheme containing the fields of a fuel transaction in plain text.

STATUS: Acknowledged

DESCRIPTION:

Both predicates accept signatures in the form of:

- Ethereum: the prefix + tx id (32 bytes hash)
- Solana: tx id (32 bytes hash) translated to ASCII

This means that users will not be able to visually verify the validity of the data they are about to sign if they don't manually compute and match the tx id.

Modern signature schemes usually contain transaction details in plain text (e.g., EIP-712) to eliminate any risks of blind signatures.

packages/evm-predicates/predicate/src/main.sw#L62-L98

```
fn personal_sign_hash(transaction_id: b256) -> b256 {
    // Hack, allocate memory to reduce manual `asm` code.
    let data = SignedData {
        transaction_id,
        ethereum_prefix: ETHEREUM_PREFIX,
        empty: ZERO_B256,
    };


    // Pointer to the data we have signed external to Sway.
    let data_ptr = asm(ptr: data.transaction_id) {
        ptr
    };


    // The Ethereum prefix is 28 bytes (plus padding we exclude).
    // The Tx ID is 32 bytes at the end of the prefix.
    let len_to_hash = 28 + 32;


    // Create a buffer in memory to overwrite with the result being the
hash.
    let mut buffer = b256::min();


    // Copy the Tx ID to the end of the prefix and hash the exact len of the
prefix and id (without
    // the padding at the end because that would alter the hash).
    asm(
        hash: buffer,
        tx_id: data_ptr,
        end_of_prefix: data_ptr + len_to_hash,
        prefix: data.ethereum_prefix,
        id_len: 32,
        hash_len: len_to_hash,
    ) {
        mcp end_of_prefix tx_id id_len;
        k256 hash prefix hash_len;
    }


    // The buffer contains the hash.
    buffer
}
```

packages/solana-connector/predicate/src/main.sw#L25-L55

```
pub fn b256_to_ascii_bytes(val: b256) -> Bytes {
    let bytes = Bytes::from(val);
    let mut ascii_bytes = Bytes::with_capacity(64);
    let mut idx = 0;


    while idx < 32 {
        let b = bytes.get(idx).unwrap();
        ascii_bytes.push(ASCII_MAP[(b >> 4).as_u64()]);
        ascii_bytes.push(ASCII_MAP[(b & 15).as_u64()]);
     idx = idx + 1;
    }


    ascii_bytes
}


configurable {
    SIGNER: b256 = ZERO_B256,
}


fn main(witness_index: u64) -> bool {
    let signature: B512 = tx_witness_data(witness_index).unwrap();
    let encoded = b256_to_ascii_bytes(tx_id());
    let result = ed_verify(SIGNER, signature, encoded);


    if result.is_ok() {
        return true;
    }


    // Otherwise, an invalid signature has been passed and we invalidate the Tx.
    false
}
```

hexens × FUEL