hexens × FUEL

JUNE.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.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered the Predeposit smart contract as developed by Fuel Labs.

Our security assessment was a full review of the smart contract, spanning a total of 3 days.

During our audit, we have identified 2 medium severity vulnerabilities, various minor vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/FuelLabs/predeposit-contracts/tree/6c71bdc76c1b291e2e2565648a333c1603375e81

The issues described in this report were fixed in the following commit:

https://github.com/FuelLabs/predeposit-contracts/tree/6d2b80a4579065d240d920af8f0006d54d620c03

AUDITING DETAILS

STARTED
10.06.2024

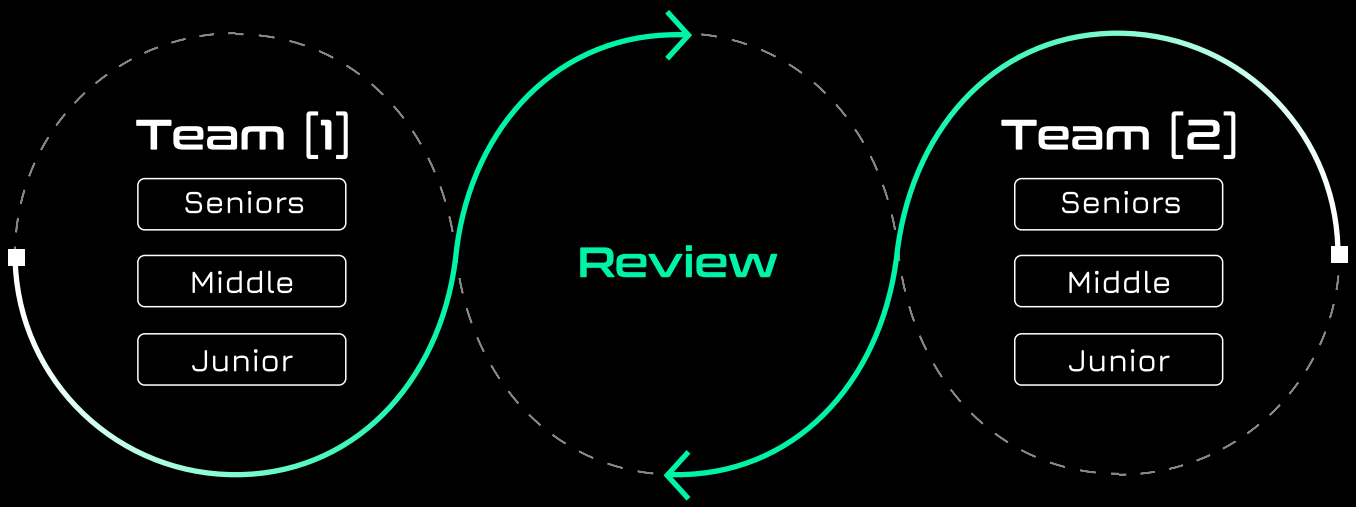
DELIVERED
12.06.2024

Review Led by

KASPER ZWIJSEN
Head of Audits | 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.

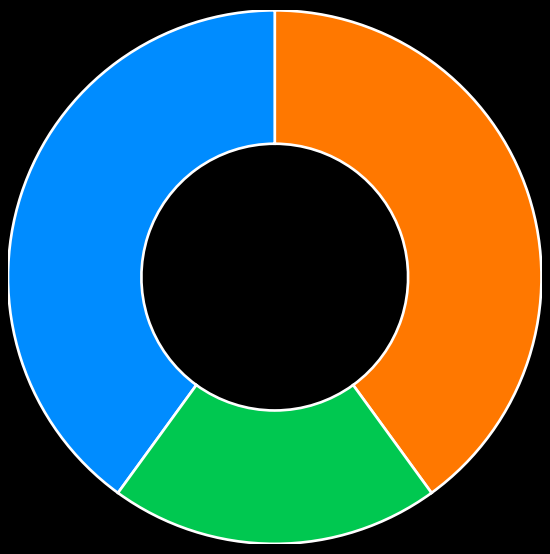
ISSUE SYMBOLIC CODES

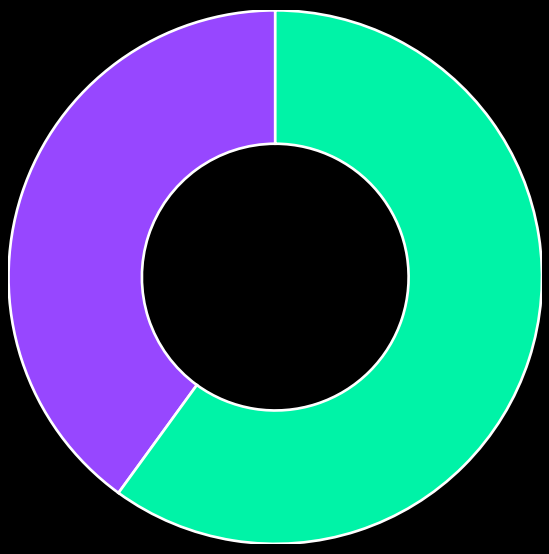
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	0
Medium	2
Low	1
Informational	2

Total: 5





WEAKNESSES

This section contains the list of discovered weaknesses.

FUEL4-1

PREDEPOSITS DOESN'T SUPPORT TOKENS WITH AMOUNT CHANGES IN TRANSFERS

SEVERITY: Medium

REMEDIATION:

Use the difference between balances before and after the transfers.

STATUS: Acknowledged

DESCRIPTION:

The **deposit()** and **depositWithPermit()** functions will put an incorrect amount of tokens into the **balances** mapping if the transfer amount changes during the token calls (like in fee-on-transfer tokens).

This will lead to other users losing funds on the **withdraw()** call.

contracts/PreDeposits/PreDeposits.sol:L65

```
IERC20(token).safeTransferFrom(_msgSender(), address(this), amount);
```

contracts/PreDeposits/PreDeposits.sol:L90

```
IERC20(token).safeTransferFrom(_msgSender(), address(this), amount);
```

FUEL4-3

PERMIT DEPOSIT DOESN'T SAVE THE DEPOSITPARAM ARGUMENT TO STORAGE

SEVERITY: Medium

PATH:

contracts/PreDeposits/PreDeposits.sol:L69-L92

REMEDIATION:

Add the update to the function.

STATUS: Fixed

DESCRIPTION:

The function accepts and emits the **depositParam** argument in the **Deposit** event but doesn't actually update it in the storage deposits mapping.

You can see the update in the twin **deposit()** function:

contracts/PreDeposits/PreDeposits.sol:L62-L63

```
            _tokenDeposit.depositParam = depositParam;
            deposits[_msgSender()][token] = _tokenDeposit;
```

```
function depositWithPermit(
 address token,
 uint240 amount,
 uint16 depositParam,
 uint256 deadline,
 uint8 v,
 bytes32 r,
 bytes32 s
) external whenNotPaused {
 deposits[_msgSender()][token].balance += amount;

 ERC20Permit(token).permit(
  _msgSender(),
  address(this),
  amount,
  deadline,
  v,
  r,
  s
 );

 IERC20(token).safeTransferFrom(_msgSender(), address(this), amount);
 emit Deposit(_msgSender(), token, amount, depositParam);
}
```

PERMIT DEPOSIT CAN BE DOS'ED

SEVERITY: Low

PATH:

contracts/PreDeposits/PreDeposits.sol::depositWithPermit():L69-L92

REMEDIATION:

Consider adding try/cath, or if block, so if the permit() is already called, just check the allowance of msg.sender and skip the call to permit().

STATUS: Fixed

DESCRIPTION:

The **PreDeposits.sol::depositWithPermit()** function currently utilizes an inner call to the **permit()** function of the **openzeppelin/contracts/token/ERC20/extensions/ERC20Permit.sol**. However, this flow exposes **depositWithPermit()** to a griefing attack, where an attacker can forcibly block the victim's transaction.
The attack proceeds as follows: the attacker front-runs the victim's transaction, extracts the parameters from the mempool, and places a transaction that directly calls **ERC20Permit(token).permit()** with the victim's params. Consequently, the victim's transaction reverts since the signature has already been used for **permit()** in the attacker's transaction.

```
    function depositWithPermit(
        address token,
        uint240 amount,
        uint16 depositParam,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external whenNotPaused {
        deposits[_msgSender()][token].balance += amount;

        ERC20Permit(token).permit(
            _msgSender(),
            address(this),
            amount,
            deadline,
            v,
            r,
            s
        );

        IERC20(token).safeTransferFrom(_msgSender(), address(this), amount);
        emit Deposit(_msgSender(), token, amount, depositParam);
    }
```

ENABLE WITHDRAWALS TO OTHER ADDRESSES

SEVERITY: Informational

PATH:

contracts/PreDeposits/PreDeposits.sol::withdraw()#L95-L111

REMEDIATION:

Add address recipient to the withdraw() function.

STATUS: Fixed

DESCRIPTION:

The current implementation of the **PreDeposits.sol** contract restricts withdrawals to the sender's address. This limitation can be restrictive for users who might want to withdraw their funds to a different address. Allowing withdrawals to other addresses can offer flexibility and convenience to users, facilitating smoother fund management and better user experience.

```
    function withdraw(address token, uint240 amount) external whenNotPaused
{
        // Underflow checks already in effect with new solidity versions
        deposits[_msgSender()][token].balance =
            deposits[_msgSender()][token].balance -
            amount;

        if (token == address(0)) {
            (bool success, ) = _msgSender().call{ value: amount }("");
            if (!success) {
                revert RecipientRevert();
            }
        } else {
            IERC20(token).safeTransfer(_msgSender(), amount);
        }

        emit Withdraw(_msgSender(), token, amount);
    }
```

THE MIGRATE FUNCTION IS NOT IMPLEMENTED

SEVERITY: Informational

PATH:

contracts/PreDeposits/PreDeposits.sol::migrate():L134-L140

REMEDIATION:

Consider implementing the migrate() function.

STATUS: Acknowledged

DESCRIPTION:

The migrate() function is typically used for migration management. However, in this case, it is not implemented and will always revert.

```
function migrate(
    address /*token*/,
    address /*migrationFacilitator*/,
    bytes calldata /*facilitatorData*/
) external view whenNotPaused {
    revert("UNIMPLEMENTED");
}
```


hexens
×
FUEL