hexens × FUEL

APR.24

SECURITY REVIEW REPORT FOR FUEL

CONTENTS

- About Hexens
- Executive summary
    - Overview
    - Scope
- Auditing details
- Severity structure
    - Severity characteristics
    - Issue symbolic codes
- Findings summary
- Weaknesses
    - Decimal precision oversight in cross-layer token transactions
    - FuelERC20GatewayV4.sendMetadata() produces a corrupted message to the Portal because of the missed type conversion
    - Discrepancies between FuelERC20Gateway and DepositMessage
    - Failure to transfer the base asset to the recipient
    - Use safeTransferFrom instead of transferFrom for ERC721 transfers
    - Fee-on-transfer tokens are not supported by FuelERC20GatewayV4 and can be stolen from it
    - Funds can get stuck, because of block header recommitting

- Sent funds may get stuck inside of the bridge
- Redundant payable modifiers
- Inconsistent role assignments in FuelMessagePortalV3
- Wrong comment
- Add error message if withdrawal limit exceeded
- Use custom error
- Gas optimization on signature check
- _generate_sub_id_from_metadata is not used consistently

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

The audit was centered on Fuel's token bridge, encompassing the Fuel Message Portal and Gateway solidity smart contracts, Fuel Bridge Message Predicate and Transaction Script, and Fuel Bridge fungible token Sway smart contract.

Our security assessment involved a comprehensive review of the Solidity and Sway smart contracts' code, spanning a total of 3 weeks. Throughout this audit, we identified one high-severity issue, six medium-severity issues, and eight low-to-informational-severity issues.

All of our reported issues were either fixed or acknowledged by the development team and subsequently validated by us.

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

SCOPE

The analyzed resources are located on:

https://github.com/FuelLabs/fuel-bridge/tree/main/packages/solidity-contracts

https://github.com/FuelLabs/fuel-bridge/tree/main/packages/message-predicates

https://github.com/FuelLabs/fuel-bridge/tree/main/packages/fungible-token

The issues described in this report were fixed. Corresponding commits are mentioned in the description.

AUDITING DETAILS

STARTED
01.04.2024

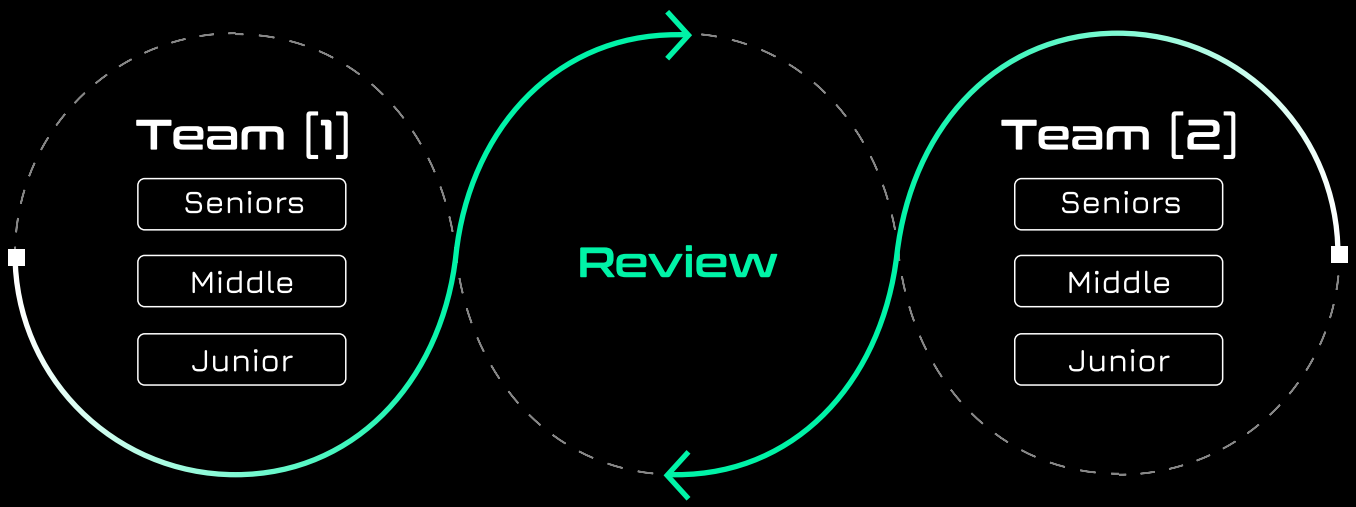
DELIVERED
15.04.2024

Review Led by

MIKHAIL EGOROV
Senior 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.

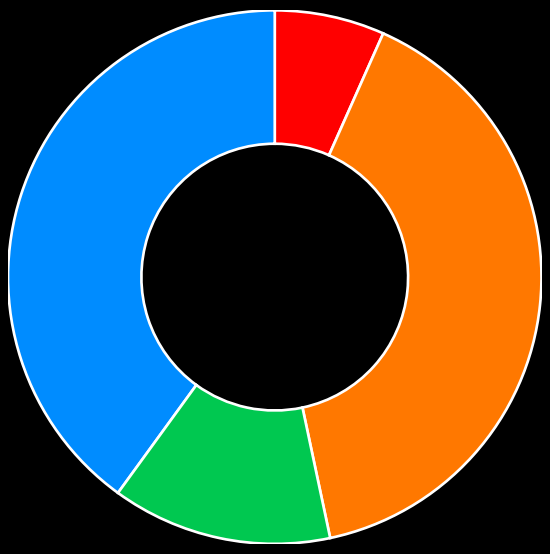
ISSUE SYMBOLIC CODES

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.

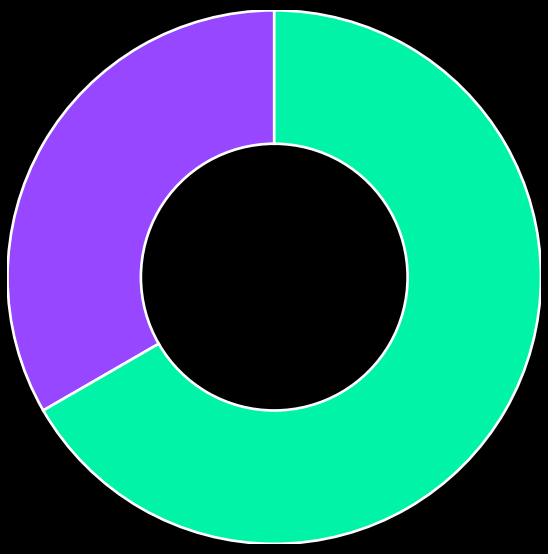
FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	1
Medium	6
Low	2
Informational	6

Total: 15



- High
- Medium
- Low
- Informational



- Fixed
- Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

FUEL1-13

DECIMAL PRECISION OVERSIGHT IN CROSS-LAYER TOKEN TRANSACTIONS

SEVERITY: High

PATH:

fuel-bridge/packages/solidity-contracts/contracts/messaging/gateway/FuelERC20Gateway/FuelERC20GatewayV4.sol
fuel-bridge/packages/fungible-token/bridge-fungible-token/src/main.sw

REMEDIATION:

Modify the sendMetadata() function in the FuelERC20GatewayV4 gateway to include the token's decimals when passing metadata to the sway bridge on the Fuel chain. The sway bridge should adjust minted amount based on token's decimals from metadata.

STATUS: Fixed

DESCRIPTION:

The sway bridge contract uses the same number of decimals for all bridged tokens, which is determined by the **DECIMALS** configurable constant or default value of **9** decimals.

```
const DEFAULT_DECIMALS: u8 = 9u8;

configurable {
    DECIMALS: u64 = 9u64,
    BRIDGED_TOKEN_GATEWAY: b256 =
0x00000000000000000000000096c53cd98B7297564716a8f2E1de2C83928Af2fe,
}
```

```
impl SRC20 for Contract {
    ...

    #[storage(read)]
    fn decimals(asset: AssetId) -> Option<u8> {
        match storage.tokens_minted.get(asset).try_read() {
            Some(_) => Some(DECIMALS.try_as_u8().unwrap_or(DEFAULT_DECIMALS)),
            None => None,
        }
    }
}
```

In contrast, the Fuel gateway on Ethereum (**FuelERC20GatewayV4.sol**) allows bridging any tokens by default, as the **whitelistRequired** state variable is set to **false**. Tokens bridged through this gateway may have different decimal amounts. For example, WETH has 18 decimals, whereas USDC has 6 decimals.

When a token is bridged from Ethereum to Fuel, a new **sub_id** is generated by the sway bridge contract based on the token's address on Ethereum, and the corresponding bridged amount is minted.

```
  let sub_id = _generate_sub_id_from_metadata(message_data.token_address,
  message_data.token_id);
```

```
  mint(sub_id, amount);
```

However, both bridged **WETH** and **USDC** tokens will have the same decimal amount specified by **DECIMALS**.

On the Fuel Gateway's (**FuelERC20GatewayV4.sol**) side, during the bridging process, token decimals are adjusted to match the specified **FUEL_ASSET_DECIMALS = 9**. This means that for the WETH token, the amount is divided by **10^9**, while for the USDC token, the amount remains unchanged.

```
function _adjustDepositDecimals(uint8 tokenDecimals, uint256 amount) internal
pure virtual returns (uint256) {
    if (tokenDecimals > FUEL_ASSET_DECIMALS) {
        unchecked {
            uint256 precision = 10 ** (tokenDecimals - FUEL_ASSET_DECIMALS);
            if (amount % precision != 0) {
                revert InvalidAmount();
            }
            return _divByNonZero(amount, precision);
        }
    }
    return amount;
}
```

The bridged amount of USDC is not adjusted to 9 decimals by the sway bridge contract. This means that a user would receive a 1000x smaller amount of bridged USDC.

```
    function _deposit(
        address tokenAddress,
        uint256 amount,
        uint256 l2MintedAmount,
        bytes memory messageData
    ) internal virtual {
        ////////////
        // Checks //
        ////////////
        if (l2MintedAmount == 0) revert CannotDepositZero();
        if (l2MintedAmount > uint256(type(uint64).max)) revert InvalidAmount();

        ////////////
        // Effects //
        ////////////
        uint256 updatedDeposits = _deposits[tokenAddress] + l2MintedAmount;
        if (updatedDeposits > type(uint64).max) revert BridgeFull();

         if (whitelistRequired && updatedDeposits > _depositLimits[tokenAddress])
{
            revert GlobalDepositLimit();
        }

        _deposits[tokenAddress] = updatedDeposits;

        ////////////
        // Actions //
        ////////////
        //send message to gateway on Fuel to finalize the deposit
        sendMessage(CommonPredicates.CONTRACT_MESSAGE_PREDICATE, messageData);

        //transfer tokens to this contract and update deposit balance
              IERC20MetadataUpgradeable(tokenAddress).safeTransferFrom(msg.sender,
address(this), amount);

        //emit event for successful token deposit
               emit  Deposit(bytes32(uint256(uint160(msg.sender))),  tokenAddress,
amount);
    }
```

```
impl SRC20 for Contract {
    #[storage(read)]
    fn total_assets() -> u64 {
        storage.total_assets.try_read().unwrap_or(0)
    }

    #[storage(read)]
    fn total_supply(asset: AssetId) -> Option<u64> {
        storage.tokens_minted.get(asset).try_read()
    }

    #[storage(read)]
    fn name(asset: AssetId) -> Option<String> {
        let l1_address = _asset_to_l1_address(asset);
        storage.l1_names.get(l1_address).read_slice()
    }

    #[storage(read)]
    fn symbol(asset: AssetId) -> Option<String> {
        let l1_address = _asset_to_l1_address(asset);
        storage.l1_symbols.get(l1_address).read_slice()
    }

    #[storage(read)]
    fn decimals(asset: AssetId) -> Option<u8> {
        match storage.tokens_minted.get(asset).try_read() {
            Some(_) => Some(DECIMALS.try_as_u8().unwrap_or(DEFAULT_DECIMALS)),
            None => None,
        }
    }
}
```

FUELERC20GATEWAYV4.SENDMETADATA() PRODUCES A CORRUPTED MESSAGE TO THE PORTAL BECAUSE OF THE MISSED TYPE CONVERSION

SEVERITY: Medium

PATH:

fuel-bridge/packages/solidity-contracts/contracts/messaging/gateway/FuelERC20Gateway/FuelERC20GatewayV4.sol:L190-L199

fuel-bridge/packages/fungible-token/bridge-fungible-token/src/data_structures/message_data.sw:L21-L30

REMEDIATION:

Convert the type to uint256:

uint256(MessageType.METADATA)

STATUS: Fixed

DESCRIPTION:

The **FuelERC20GatewayV4.sendMetadata()** function encodes the **MessageType** enum using uint8 instead of uint256, as you can see in the code snippet below.

As a result, it will produce messages similar to the following (in hex):

```
13acd573529992578e7f0fdbe54c747a8a9d91f757947b56d343741d40a0c791030000000000
00000000000000e7f1725e7734ce288f8367e1bb143e90bb3f0512000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000800000000000000000000000000000000000000000000000000
000000000000000c0000000000000000000000000000000000000000000000000000000000000000
000003544b4e0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000005546f6b656e00000000
0000000000000000000000000000000000000000
```

Where the type of the message (`03`) is in the 33th byte instead of the 64th.

The contract on the other end will try to read the 64th position and revert or process a wrong execution path.

```
        bytes memory metadataMessage = abi.encodePacked(
            assetIssuerId,
            MessageType.METADATA,
            abi.encode(
                tokenAddress,
                uint256(0), // token_id = 0 for all erc20 deposits
                IERC20MetadataUpgradeable(tokenAddress).symbol(),
                IERC20MetadataUpgradeable(tokenAddress).name()
            )
        );
```

```
    pub fn parse(msg_idx: u64) -> Self {
        let message_type: u8 = input_message_data(msg_idx,
OFFSET_MESSAGE_TYPE).get(31).unwrap(); // Get the last byte

        match message_type {
            DEPOSIT =>
MessageData::Deposit(DepositMessage::parse_deposit_to_address(msg_idx)),
            CONTRACT_DEPOSIT =>
MessageData::Deposit(DepositMessage::parse_deposit_to_contract(msg_idx)),
            CONTRACT_DEPOSIT_WITH_DATA =>
MessageData::Deposit(DepositMessage::parse_deposit_to_contract_with_data(msg
_idx)),
            METADATA =>
MessageData::Metadata(MetadataMessage::parse(msg_idx)),
            _ => revert(0),
        }
    }
```

DISCREPANCIES BETWEEN FUELERC20GATEWAY AND DEPOSITMESSAGE

SEVERITY: Medium

PATH:

fuel-bridge/packages/fungible-token/bridge-fungible-token/src/data_structures/deposit_message.sw

REMEDIATION:

The FuelERC20Gateway solidity contract and the bridge sway contract should adhere to the same message data layout.

STATUS: Fixed

DESCRIPTION:

In the **depositWithData()** function of the **FuelERC20GatewayV4** solidity contract, the message **data** is constructed with the arbitrary data as the last 8th element as shown below:

```
bytes memory depositMessage = abi.encodePacked(
    assetIssuerId,
        uint256(data.length == 0 ? MessageType.DEPOSIT_TO_CONTRACT :
MessageType.DEPOSIT_WITH_DATA),
    bytes32(uint256(uint160(tokenAddress))),
    uint256(0), // token_id = 0 for all erc20 deposits
    bytes32(uint256(uint160(msg.sender))),
    to,
    l2MintedAmount,
    data
);
```

However, the **parse_deposit_to_contract_with_data()** function in the **DepositMessage** sway library expects 8 elements of size 32 bytes in the message data and interprets the 8th element as the **decimals** value:

```
pub fn parse_deposit_to_contract_with_data(msg_idx: u64) -> Self {
    Self {
        amount: input_message_data(msg_idx, OFFSET_AMOUNT).into(),
        from: input_message_data(msg_idx, OFFSET_FROM).into(),
        token_address: input_message_data(msg_idx,
OFFSET_TOKEN_ADDRESS).into(),
        to:
Identity::ContractId(ContractId::from(b256::from(input_message_data(msg_idx,
OFFSET_TO)))),
        token_id: input_message_data(msg_idx, OFFSET_TOKEN_ID).into(),
        decimals: input_message_data(msg_idx,
OFFSET_DECIMALS).get(31).unwrap(),
        deposit_type: DepositType::ContractWithData,
    }
}
```

This discrepancy may lead to a subversion of the logic in the sway bridge contract. However, currently **DepositMessage.decimals** isn't used in the code.

FAILURE TO TRANSFER THE BASE ASSET TO THE RECIPIENT

SEVERITY: Medium

PATH:

fuel-bridge/packages/message-predicates/contract-message-predicate/script_asm.rs:L54

fuel-bridge/packages/fungible-token/bridge-fungible-token/src/main.sw:L336-348

REMEDIATION:

Transfer the base asset amount to the recipient.

STATUS: Fixed

DESCRIPTION:

The transaction script **script_asm.rs** facilitates the transfer of the base asset (**0x000...** or **ZERO_B256**) to the bridge contract.

```
op::call(REG_DATA_PTR, REG_MSG_AMOUNT, REG_ASSET_PTR, RegId::CGAS),
```

Where **REG_ASSET_PTR** equals **0x000...** and **REG_MSG_AMOUNT** contains the amount of the base asset from the input message.

```
op::gtf(
    REG_MSG_AMOUNT,
    RegId::ZERO,
    GTFArgs::InputMessageAmount.into(),
)
```

However, the bridge contract fails to transfer the base asset to the recipient, only transferring bridged tokens (e.g. USDC).

```
let asset_id = AssetId::new(contract_id(), sub_id);
```

```
match message_data.deposit_type {
    DepositType::Address | DepositType::Contract => {
        transfer(message_data.to, asset_id, amount)
    },
    DepositType::ContractWithData => {
                            let    dest_contract    =    abi(MessageReceiver,
message_data.to.as_contract_id().unwrap().into());
        dest_contract
            .process_message {
                coins: amount,
                asset_id: asset_id.into(),
            }(msg_idx);
    }
};
```

Consequently, the base asset amount is trapped in the bridge contract.

USE SAFETRANSFERFROM INSTEAD OF TRANSFERFROM FOR ERC721 TRANSFERS

SEVERITY: Medium

PATH:
fuel-bridge/packages/solidity-contracts/contracts/messaging/gateway/FuelERC721Gateway.sol::finalizeWithdrawal():L168-L182

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

As per OpenZeppelin's documentation:
"Note that the caller is responsible to confirm that the recipient is capable of receiving ERC721 or else they may be permanently lost. Usage of safeTransferFrom prevents loss, though the caller must understand this adds an external call which potentially creates a reentrancy vulnerability."
Hence, in the function **FuelERC721Gateway.sol::finalizeWithdrawal()#L168-L182**, **ERC721** token is sent to **to** with the **transferFrom** method:

```
        IERC721Upgradeable(tokenAddress).transferFrom(address(this), to,
tokenId);
```

If this **to** is a contract and is not aware of incoming **ERC721** tokens, the sent token could be locked up in the contract forever.

```
function finalizeWithdrawal(
    address to,
    address tokenAddress,
    uint256 /*amount*/,
    uint256 tokenId
) external payable override whenNotPaused onlyFromPortal {
    bytes32 fuelContractId = messageSender();
    require(_deposits[tokenAddress][tokenId] == fuelContractId, "Fuel bridge
does not own this token");

    delete _deposits[tokenAddress][tokenId];

    IERC721Upgradeable(tokenAddress).transferFrom(address(this), to,
tokenId);
    //emit event for successful token withdraw
    emit Withdrawal(bytes32(uint256(uint160(to))), tokenAddress,
fuelContractId, 1);
}
```

Call the safeTransferFrom() method instead of transferFrom() for ERC721 transfers:

openzeppelin-contracts-upgradeable/contracts/token/ERC721/ERC721Upgradeable.sol at e3c57ac7590cc021c0de8c3cef18600b23123abb · OpenZeppelin/openzeppelin-contracts-upgradeable

```
function finalizeWithdrawal(
    address to,
    address tokenAddress,
    uint256 /*amount*/,
    uint256 tokenId
) external payable override whenNotPaused onlyFromPortal {
    bytes32 fuelContractId = messageSender();
    require(_deposits[tokenAddress][tokenId] == fuelContractId, "Fuel bridge does not own this token");

    delete _deposits[tokenAddress][tokenId];

-   IERC721Upgradeable(tokenAddress).transferFrom(address(this), to, tokenId);
+   IERC721Upgradeable(tokenAddress).safeTransferFrom(address(this), to, tokenId);
    //emit event for successful token withdraw
    emit Withdrawal(bytes32(uint256(uint160(to))), tokenAddress, fuelContractId, 1);
}
```

FEE-ON-TRANSFER TOKENS ARE NOT SUPPORTED BY FUELERC20GATEWAYV4 AND CAN BE STOLEN FROM IT

SEVERITY: Medium

PATH:
/fuel-bridge/packages/solidity-contracts/contracts/messaging/gateway/FuelERC20Gateway/FuelERC20GatewayV4.sol:L268-L269

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

FuelERC20GatewayV4 does not support fee-on-transfer tokens because accounting doesn't check the difference between the requested transfer amount and the real balance after the transfer.

Deposit of a fee-on-transfer token will result in a deficit on the balance of **FuelERC20GatewayV4** because the amount minted to the caller will remain the same, while the amount received will decrease because of the fee.

If the same user requests a withdrawal from the L2, they will either not receive tokens because of the deficit or consume tokens from another user's deposit.

```
        //transfer tokens to this contract and update deposit balance
        IERC20MetadataUpgradeable(tokenAddress).safeTransferFrom(msg.sender,
address(this), amount);
```

Rewrite the deposit functions to collect payment at the beginning of execution and utilize the following scheme:

```
uint256 balanceBefore = IERC20(tokenAddress).balanceOf(address(this));
IERC20MetadataUpgradeable(tokenAddress).safeTransferFrom(msg.sender,
address(this), amount);
uint256 balanceAfter = IERC20(tokenAddress).balanceOf(address(this));

uint256 toMint = balanceAfter - balanceBefore;
```

Use toMint afterward to determine the funds to mint on the L2.

FUEL1-6

FUNDS CAN GET STUCK, BECAUSE OF BLOCK HEADER RECOMMITTING

SEVERITY: Medium

PATH:

FuelChainState:commit()

REMEDIATION:

Consider adding a check, which disallows slot rewriting.

STATUS: Fixed

DESCRIPTION:

When committing a block header, the **COMMITTER_ROLE** invokes the **commit()** function, which writes the **blockHash** and **timestamp** to storage. To access messages generated within the L2 (Fuel) blockchain, users are required to call **relayMessage()**. This function includes a check for **finalized()**, ensuring that a specified period of **TIME_TO_FINALIZE** has passed after the block header commitment. However, the **commit()** function lacks a check, leading to an issue where the **blockHash** and **timestamp** could potentially be overwritten. Consequently, this oversight may result in user funds becoming inaccessible until another **TIME_TO_FINALIZE** period has elapsed.

```
    function relayMessage(
        Message calldata message,
        FuelBlockHeaderLite calldata rootBlockHeader,
        FuelBlockHeader calldata blockHeader,
        MerkleProof calldata blockInHistoryProof,
        MerkleProof calldata messageInBlockProof
    ) external payable virtual override whenNotPaused {
        if (withdrawalsPaused) {
            revert WithdrawalsPaused();
        }

        //verify root block header
        if (!
_fuelChainState.finalized(rootBlockHeader.computeConsensusHeaderHash(),
rootBlockHeader.height)) {
            revert UnfinalizedBlock();
        }

        //verify block in history
        if (
            !verifyBinaryTree(
                rootBlockHeader.prevRoot,
                abi.encodePacked(blockHeader.computeConsensusHeaderHash()),
                blockInHistoryProof.proof,
                blockInHistoryProof.key,
                rootBlockHeader.height
            )
        ) revert InvalidBlockInHistoryProof();

        //verify message in block
        bytes32 messageId = CryptographyLib.hash(
            abi.encodePacked(message.sender, message.recipient, message.nonce,
message.amount, message.data)
        );

        if (messageIsBlacklisted[messageId]) {
            revert MessageBlacklisted();
        }
```

```
        if (
            !verifyBinaryTree(
                blockHeader.outputMessagesRoot,
                abi.encodePacked(messageId),
                messageInBlockProof.proof,
                messageInBlockProof.key,
                blockHeader.outputMessagesCount
            )
        ) revert InvalidMessageInBlockProof();

        //execute message
        _executeMessage(messageId, message);
    }
```

```
    function commit(bytes32 blockHash, uint256 commitHeight) external
whenNotPaused onlyRole(COMMITTER_ROLE) {//@audit should add require check for
rewriting cases
        uint256 slot = commitHeight % NUM_COMMIT_SLOTS;
        Commit storage commitSlot = _commitSlots[slot];
        commitSlot.blockHash = blockHash;
        commitSlot.timestamp = uint32(block.timestamp);

        emit CommitSubmitted(commitHeight, blockHash);
    }
```

SENT FUNDS MAY GET STUCK INSIDE OF THE BRIDGE

SEVERITY: Low

REMEDIATION:

Either add another way to retrieve the stuck funds or store the refund amount before calling the process_message in case of a failure and then `0` it out after a successful call.

STATUS: Acknowledged

DESCRIPTION:

In the **bridge-fungible-token/src/main.sw** contract while processing the message there are a couple of points in the contract where if the check isn't passing and the call will fail for one or another reason, it calls the **register_refund()** function which will allow the user to later come and claim the stuck funds.

bridge-fungible-token/src/main.sw:L343-361

```
fn register_refund(
    from: b256,
    token_address: b256,
    token_id: b256,
    amount: b256,
) {
    let asset = sha256((token_address, token_id));

    let previous_amount =
storage.refund_amounts.get(from).get(asset).try_read().unwrap_or(ZERO_U256);
    let new_amount = amount.as_u256() + previous_amount;

    storage.refund_amounts.get(from).insert(asset, new_amount);
    log(RefundRegisteredEvent {
        from,
        token_address,
        token_id,
        amount,
    });
}
```

When bridging the tokens, the user has the ability to specify a contract which would be called as a callback so they can do further processing with the funds. However due to missing checks, there may happen a case where the called contract reverts and no refund was registered and thus the tokens will get stuck in the contract.

bridge-fungible-token/src/main.sw:L170-187

```
match message_data.len {
    ADDRESS_DEPOSIT_DATA_LEN => {
        transfer(message_data.to, asset_id, amount);
    },
    CONTRACT_DEPOSIT_WITHOUT_DATA_LEN => {
        transfer(message_data.to, asset_id, amount);
    },
    _ => {
        if let Identity::ContractId(id) = message_data.to {
            let dest_contract = abi(MessageReceiver, id.into());
            dest_contract
                .process_message {
                    coins: amount,
                    asset_id: asset_id.into(),
                }(msg_idx);
        };
    },
}
```

FUEL1-18

REDUNDANT PAYABLE MODIFIERS

SEVERITY: Low

PATH:

fuel-bridge/packages/solidity-contracts/contracts/messaging/gateway/FuelERC20Gateway/FuelERC20GatewayV4.sol

REMEDIATION:

Remove payable modifiers from deposit(), depositWithData(), and sendMetadata().

STATUS: Acknowledged

DESCRIPTION:

The functions **deposit()**, **depositWithData()**, and **sendMetadata()** within the **FuelERC20GatewayV4** contract are marked as payable.

However, the ETH amount sent in **msg.value** is not utilized in the call to the **FuelMessagePortal.sendMessage()** function and the subsequent message sent to the Fuel chain.

Consequently, ETH becomes trapped in the contract until an admin rescues it using **rescueETH()**.

```
function depositWithData(
    bytes32 to,
    address tokenAddress,
    uint256 amount,
    bytes calldata data
) external payable virtual whenNotPaused {
    uint8 decimals = _getTokenDecimals(tokenAddress);
    uint256 l2MintedAmount = _adjustDepositDecimals(decimals, amount);

    bytes memory depositMessage = abi.encodePacked(
        assetIssuerId,
                uint256(data.length  ==  0  ?  MessageType.DEPOSIT_TO_CONTRACT  :
MessageType.DEPOSIT_WITH_DATA),
        bytes32(uint256(uint160(tokenAddress))),
        uint256(0), // token_id = 0 for all erc20 deposits
        bytes32(uint256(uint160(msg.sender))),
        to,
        l2MintedAmount,
        data
    );
    _deposit(tokenAddress, amount, l2MintedAmount, depositMessage);
}
```

INCONSISTENT ROLE ASSIGNMENTS IN FUELMESSAGEPORTALV3

SEVERITY: Informational

PATH:

fuel-bridge/packages/solidity-contracts/contracts/fuelchain/FuelMessagePortal/v3/FuelMessagePortalV3.sol

REMEDIATION:

To ensure clarity and maintain consistency in role assignments, restrict the pauseWithdrawals function to the PAUSER_ROLE, while delegating the unpauseWithdrawals and setMessageBlacklist functions to the DEFAULT_ADMIN_ROLE.

STATUS: Fixed

DESCRIPTION:

The **FuelMessagePortalV3** contract introduces new functions for pausing and unpausing withdrawals, as well as setting message blacklists. However, the role assignments for these functions deviate from the established pattern seen in the previous version. In the V1 contract, the **PAUSER_ROLE** is only responsible for pausing functionality, while other administrative tasks are handled by the **DEFAULT_ADMIN_ROLE**:

```
    function pause() external virtual onlyRole(PAUSER_ROLE) {
        _pause();
    }

    function unpause() external virtual onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
    }
```

In contrast, the V3 contract assigns both pausing and unpausing of withdrawals to the **PAUSER_ROLE**, potentially leading to confusion and inconsistent governance.

```
    function pauseWithdrawals() external payable onlyRole(PAUSER_ROLE) {
        withdrawalsPaused = true;
    }

    function unpauseWithdrawals() external payable onlyRole(PAUSER_ROLE) {
        withdrawalsPaused = false;
    }

     function setMessageBlacklist(bytes32 messageId, bool value) external payable
onlyRole(PAUSER_ROLE) {
        messageIsBlacklisted[messageId] = value;
    }
```

FUEL1-8

WRONG COMMENT

SEVERITY: Informational

PATH:

fuel-bridge/packages/fungible-token/bridge-fungible-token/src/utils.sw#::encode_data():L85-L103

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **FuelBridgeBase.sol::finalizeWithdrawal()** function is intended to accept parameters of type **address**, **address**, **uint256** and **uint256**, representing the recipient's address, the token contract address, the withdrawal amount, and the token ID, respectively:

```
    function finalizeWithdrawal(
        address to,
        address tokenAddress,
        uint256 amount,
        uint256 tokenId
    ) external payable virtual;
}
```

In the Fuel chain, there is a function which intended to encode data for this call:
utils.sw#::encode_data():L85-L103

```
pub fn encode_data(to: b256, amount: b256, bridged_token: b256, token_id: b256) -
> Bytes {
    // capacity is 4 + 32 + 32 + 32 + 32 = 132
    let mut data = Bytes::with_capacity(132);

    // first, we push the selector 1 byte at a time
     // the function selector for finalizeWithdrawal on the base layer gateway
contract:
    // finalizeWithdrawal(address,address,uint256,bytes32) = 0x64a7fad9 // @audit
(in case bytes32-0xaa5057d4)
    data.push(0x64u8);
    data.push(0xa7u8);
    data.push(0xfau8);
    data.push(0xd9u8);

    data.append(Bytes::from(to));
    data.append(Bytes::from(bridged_token));
    data.append(Bytes::from(amount));
    data.append(Bytes::from(token_id));

    data
}
```

However, in the comments provided in the code, the last parameter is incorrectly specified as **bytes32** instead of **uint256**.

Despite the incorrect comment, the function correctly generates the function selector based on the correct parameter types (**address**, **address**, **uint256**, **uint256**). The function selector (**0x64a7fad9**) aligns with the correct parameter types.

```
// finalizeWithdrawal(address,address,uint256,bytes32) = 0x64a7fad9
```

The comment should specify that the function accepts parameters of type address, address, uint256, and uint256 instead of bytes32 for the last parameter.

```
--   // finalizeWithdrawal(address,address,uint256,bytes32) = 0x64a7fad9
++   // finalizeWithdrawal(address,address,uint256,uint256) = 0x64a7fad9
```

ADD ERROR MESSAGE IF WITHDRAWAL LIMIT EXCEEDED

SEVERITY: Informational

PATH:

packages/fungible-token/bridge-fungible-token/src/main.sw::withdraw():L237-L276

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

The **withdraw** function facilitates token withdrawals from the contract. The function adjusts the **amount** to match the base layer's decimal format and then, it subtracts the adjusted **amount** from the current token balance stored in the contract.

If the withdrawal **amount** exceeds the available token balance an underflow condition may occur, leading to a contract revert.

```
        storage
            .tokens_minted
            .insert(
                asset_id,
                storage
                    .tokens_minted
                    .get(asset_id)
                    .read() - amount,
            );
```

To enhance user experience, incorporating a custom error message in such scenarios can provide users with informative feedback about why their transaction failed.

```
    fn withdraw(to: b256) {
        let amount = msg_amount();
        let asset_id = msg_asset_id();
        let sub_id = _asset_to_sub_id(asset_id);
        require(amount != 0, BridgeFungibleTokenError::NoCoinsSent);

         // attempt to adjust amount into base layer decimals and burn the sent
tokens
        let adjusted_amount = adjust_withdrawal_decimals(
            amount,
            DECIMALS
                .try_as_u8()
                .unwrap_or(DEFAULT_DECIMALS),
            BRIDGED_TOKEN_DECIMALS
                .try_as_u8()
                .unwrap_or(DEFAULT_BRIDGED_TOKEN_DECIMALS),
        ).unwrap();
        storage
            .tokens_minted
            .insert(
                asset_id,
                storage
                    .tokens_minted
                    .get(asset_id)
                    .read() - amount,
            );
        burn(sub_id, amount);

           // send a message to unlock this amount on the base layer gateway
contract
        let sender = msg_sender().unwrap();
        send_message(
            BRIDGED_TOKEN_GATEWAY,
            encode_data(to, adjusted_amount, BRIDGED_TOKEN, sub_id),
            0,
```

```
        );
        log(WithdrawalEvent {
            to: to,
            from: sender,
            amount: amount,
        });
    }
```

Add an error message if the withdrawal amount exceeds the token balance:

```
    // Retrieve the current balance of the asset
    let current_balance = storage.tokens_minted.get(asset_id).read();

    // Ensure that the withdrawal amount does not exceed the current balance
                                require(amount        <=        current_balance,
BridgeFungibleTokenError::InsufficientBalance);

       // Update the storage with the adjusted balance after deducting the
withdrawal amount
    storage.tokens_minted.insert(asset_id, current_balance - amount);
```

FUEL1-10

USE CUSTOM ERROR

SEVERITY: Informational

PATH:

fuel-bridge/packages/solidity-contracts/contracts/fuelchain/FuelMessagePortal.sol:L308

fuel-bridge/packages/solidity-contracts/contracts/fuelchain/FuelMessagePortal/v2/FuelMessagePortalV2.sol:L88

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In **FuelMessagePortal.sol** and **FuelMessagePortalV2.sol** contracts various errors are currently handled using native Solidity **revert** statements and custom errors, with one exception:

```
revert("Message relay failed");
```

Use custom error in this case as well, it will improve gas optimization and enhance code quality.

FUEL1-14

GAS OPTIMIZATION ON SIGNATURE CHECK

SEVERITY: Informational

PATH:

solidity-contracts/contracts/lib/Cryptography.sol:addressFromSignature

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The implementation provided in the remediation section will save approximately 600 gas.

```
function addressFromSignature(bytes memory signature, bytes32 message) internal pure returns (address) {
      // ECDSA signatures must be 64 bytes (<https://eips.ethereum.org/EIPS/eip-2098)>
    require(signature.length == 64, "signature-invalid-length");

    // Signature is concatenation of r and v-s, both 32 bytes
                    //     <https://github.com/celestiaorg/celestia-specs/blob/ec98170398dfc6394423ee79b00b71038879e211/src/specs/data_structures.md#signature>
    bytes32 vs;
    bytes32 r;
    bytes32 s;
    uint8 v;

    (r, vs) = abi.decode(signature, (bytes32, bytes32));

    // v is first bit of vs as uint8
     // yParity parameter is always either 0 or 1 (canonically the values used have been 27 and 28), so adjust accordingly
    v = 27 + uint8(uint256(vs) & (1 << 255) > 0 ? 1 : 0);

    // s is vs with first bit replaced by a 0
    s = bytes32((uint256(vs) << 1) >> 1);

    return addressFromSignatureComponents(v, r, s, message);
}
```

Change the addressFromSignature() function implementation to this:

```
        function addressFromSignature(bytes memory signature, bytes32 message)
internal pure returns (address) {
          // ECDSA signatures must be 64 bytes (https://eips.ethereum.org/EIPS/
eip-2098)
        require(signature.length == 64, "signature-invalid-length");
        bytes32 vs;
        bytes32 r;
        bytes32 s;
        uint8 v;
        assembly {
            r := mload(add(signature, 0x20))
            vs := mload(add(signature, 0x40))
        }
        unchecked {
              // We do not check for an overflow here since the shift operation
results in 0 or 1.
            v = uint8((uint256(vs) >> 255) + 27);
                                                              s     =     vs     &
bytes32(0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff);
        }
        return addressFromSignatureComponents(v, r, s, message);
    }
```

FUEL1-17

_GENERATE_SUB_ID_FROM_METADATA IS NOT USED CONSISTENTLY

SEVERITY: Informational

PATH:

fuel-bridge/packages/fungible-token/bridge-fungible-token/src/main.sw

REMEDIATION:

Use _generate_sub_id_from_metadata() instead of directly computing sub id for maintaining consistency and code clarity.

STATUS: Fixed

DESCRIPTION:

The **register_refund()** function computes sub id directly from the token address and token id. However, the **_generate_sub_id_from_metadata()** function is designed to fulfill that purpose.

```
let asset = sha256((token_address, token_id));
```

The same happens for the **claim_refund()** function.

```
let asset = sha256((token_address, token_id));
```

```
fn _generate_sub_id_from_metadata(token_address: b256, token_id: b256) ->
b256 {
    sha256((token_address, token_id))
}
```


hexens
×
FUEL