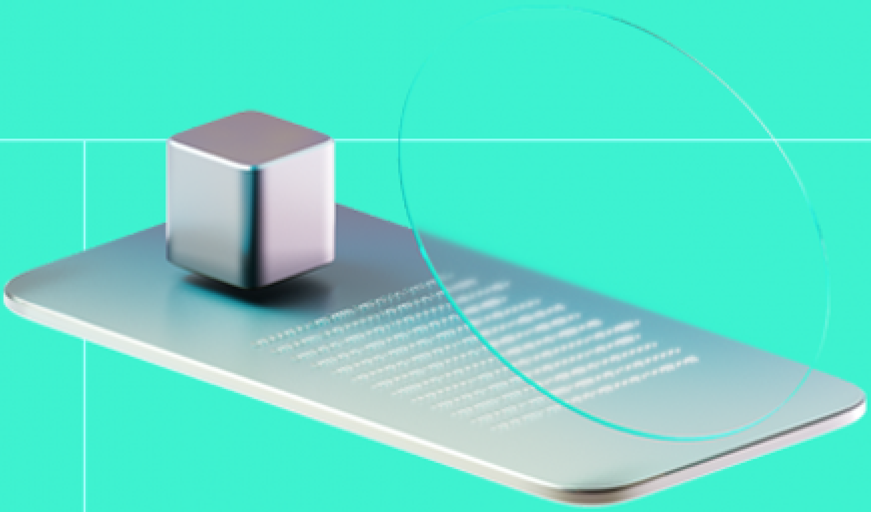
Smart Contract Code Review And Security Analysis Report



Customer: HOT Omni Token

Date: 09/10/2024

We express our gratitude to the HOT Omni Token team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

The HOT Omni token wallet consists of a multi-chain system that allows transfers of tokens among different chains (bridge).

Document

Name	Smart Contract Code Review and Security Analysis Report for HOT Omni Token
Audited By	David Camps Novi, Nataliia Balashova
Approved By	Przemyslaw Swiatowiec
Website	-
Changelog	24/09/2024 - Preliminary Report; 09/10/2024 - Final Report;
Platform	Ethereum, Base, Arbitrum and Polygon. In the future, integration with more EVM chains will be introduced.
Language	Solidity
Tags	Bridge, Signatures, Wallet.
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/hot-dao/omni-wallet-solidity
Commit	40ff89f

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

4	3	1	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	1
Low	3

Vulnerability	Severity
F-2024-6207 - Precompile ecrecover May Lead to Signature Malleability	Medium
F-2024-6222 - Use of transfer() to Send Native Assets may Revert	Low
F-2024-6225 - Lack of EIP712 Compliance May Result in Signature Replay	Low
F-2024-6228 - Unchecked Return Value in Token Transfer May Lead to Unexpected Behavior	Low

Documentation quality

- Functional requirements are mostly provided.
- Technical description is partially provided.

Code quality

- NatSpec is provided for `MetaWallet` but missing for `RlpEncode`..
- The development environment is configured.
- Best Practices are not followed: F-2024-6235, F-2024-6223, F-2024-6221.

Test coverage

Code coverage of the project is **~60%**.

- Deployment and basic interactions are covered with tests.
- Negative cases coverage are partially covered.
- Interactions by several users are not tested thoroughly.

Table of Contents

System Overview **6**

Privileged Roles 6

Potential Risks **7**

Findings **8**

Vulnerability Details 8

Observation Details 16

Disclaimers 23

Appendix 1. Definitions **24**

Severities 24

Potential Risks 24

Appendix 2. Scope **25**

System Overview

The HOT Omni token wallet consists of a multi-chain system allowing token transfers among chains (bridge).

- MetaWallet - Contract that handles deposits and withdrawals of native and ERC20 tokens via signature.
- RlpEncode - Library to manage the encoding required for signature validation.

Privileged roles

- Owner:
    - Updates the addresses of the owner and the verifyAddress.
    - Closes the wallet contract deposit and withdraw operations.
    - Can withdraw native and ERC20 tokens from the contract.

Potential Risks

- The nonces used for deposits and withdraws have different origin: whilst the deposit nonce is calculated on-chain in the MetaWallet contract, the withdraw nonce is generated in another part of the system that cannot be validated. Since the input nonce is divided by the constant `NONCE_TS_SHIFT`, it should be created taking solidity division truncation into account so that it does not provide a nonce of 0, or avoiding undesired values.
- During withdraw, a require statement checks whether the nonce timestamp is expired. In case it is, the user can make a refund. However, this refund is managed outside of the scope.
- The MetaWallet contract includes the functions `withdrawToken()` and `withdrawEth()`, which allow the contract `owner` to retrieve any funds.
- Assembly code is present in the audited contracts, decreasing the readability and reliance of the system.
- The audited project works as a bridge and, as such, it has a high dependency on off-chain and centralized processes to manage the communication and funds across different chains. This is a big part of the functionality that cannot be reviewed in this audit. A user relies on the protocol managers to correctly and fairly manage their funds once they perform a deposit and in case such a user requires a withdraw.
- The project utilizes Solidity version 0.8.20 or highe, which includes the introduction of the PUSH0 (0x5f) opcode. This opcode is currently supported on the Ethereum mainnet but may not be universally supported across other blockchain networks. Consequently, deploying the contract on chains other than the Ethereum mainnet, such as certain Layer 2 (L2) chains or alternative networks, might lead to compatibility issues or execution errors due to the lack of support for the PUSH0 opcode. In scenarios where deployment on various chains is anticipated, selecting an appropriate Ethereum Virtual Machine (EVM) version that is widely supported across these networks is crucial to avoid potential operational disruptions or deployment failures.
- The methods `deposit()` for tokens, and `withdraw()` are payable, allowing callers to send some `msg.value` as a fee. However, there is no mechanism to control this fee or set any limits (i.e. there is no percentage fee or value recorded in the contract).

Findings

Vulnerability Details

F-2024-6207 - Precompile ecrecover May Lead to Signature Malleability - Medium

Description:

The method `recoverSigner()` uses Solidity's global function `ecrecover()` to obtain the signer of a signed message, given the signature parameters r, s and v. However, such function is susceptible to signature malleability.

This vulnerability stems from the `ecrecover()` function's inability to discern between legitimately unique signatures and those that have been manipulated but are still considered valid by the Ethereum blockchain's signature verification standards. By exploiting this flaw, an external actor can create signatures that will be accepted by the system, enabling non-authorized transactions.

```
function recoverSigner(bytes32 ethSignedMessageHash, bytes memory signature)
    internal
    pure
    returns (address)
{
    (bytes32 r, bytes32 s, uint8 v) = splitSignature(signature);

    return ecrecover(ethSignedMessageHash, v, r, s);
}
```

Assets:

- MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]

Status: Fixed

Classification

Impact: 4/5

Likelihood: 3/5

Exploitability: Independent

Complexity: Medium

Severity: Medium

Recommendations

Remediation: It is advisable to use OpenZeppelin's ECDSA library's `recover()` instead of the built-in `ecrecover()` function. The ECDSA library provides robust and reliable signature verification, reducing the vulnerability to replay attacks and ensuring the integrity of the contract interactions.

Resolution: Fixed in commit ID `422795e`: ECDSA library's `recover()` was implemented instead of the built-in `ecrecover()` function.

F-2024-6222 - Use of transfer() to Send Native Assets may Revert - Low

Description:

The contract uses built-in `transfer()` function for transferring native tokens.

The `transfer()` function was commonly used in earlier versions of Solidity for its simplicity and automatic reentrancy protection. However, it was identified as potentially problematic due to its fixed gas limit of `2300`.

The usage of `transfer()` function can lead to unintended function call revert when the receiving contract's `receive()` or `fallback()` functions require more than `2300` Gas for processing.

```
function withdrawEth(uint256 amount) public onlyOwner {
    payable(owner).transfer(amount);
}

function withdraw(
    uint128 nonce,
    address contract_id,
    address receiver_id,
    uint128 amount,
    bytes memory signature
) public {
    uint128 nonce_ts = nonce / NONCE_TS_SHIFT;
    require(nonce_ts > minTimestamp);
    require(nonce_ts < maxTimestamp);
    require(
        nonce_ts > uint128(block.timestamp) - WITHDRAW_DELAY_SECONDS,
        "Nonce time is expired, you can make a refund"
    ); // this transfer can only be refunded

    require(!usedNonces[nonce], "Nonce already used");
    require(
        verify(
            nonce,
            abi.encodePacked(contract_id),
            abi.encodePacked(receiver_id),
            amount,
            signature
        ),
        "Invalid signature"
    );
    usedNonces[nonce] = true;
    if (contract_id == NATIVE_TOKEN) {
        payable(receiver_id).transfer(amount);
    } else {
        IERC20(contract_id).transfer(receiver_id, amount);
    }
}
```

Assets:

- MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]

Status: Fixed

Classification

Impact: 4/5

Likelihood: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to use built-in `call()` function instead of `transfer()` to transfer native assets. This method does not impose a gas limit, it provides greater flexibility and compatibility with contracts having more complex business logic upon receiving the native tokens. When working with then `call()` function ensure that its execution is successful by checking the returned boolean value.

Resolution: Fixed in commit ID `422795e`: the reported mthods now use the built-in `call()` function instead of `transfer()` to transfer native assets.

F-2024-6225 - Lack of EIP712 Compliance May Result in Signature Replay - Low

Description:

The function `getMessageRaw()` is used to recreate the message hash containing the data to be signed and validated within the system.

```
function getMessageRaw(
    uint128 nonce,
    bytes memory contract_id,
    bytes memory receiver_id,
    uint128 amount
) internal view returns (bytes memory) {
    bytes[] memory rlpList = new bytes[](5);
    rlpList[0] = RLPEncode.encodeUint128(nonce, 16);
    rlpList[1] = RLPEncode.encodeUint64(chainId, 8);
    rlpList[2] = RLPEncode.encodeBytes(contract_id);
    rlpList[3] = RLPEncode.encodeBytes(receiver_id);
    rlpList[4] = RLPEncode.encodeUint128(amount, 16);

    return RLPEncode.encodeList(rlpList);
}
```

As defined by the standard for signature validation in solidity, the EIP712, each message hash is recommended to include the `verifyingContract` address (i.e. `address(this)` in this case). This will prevent the replay of a signature in a future contract that may use the same purpose.

However, the current implementation does not include the `verifyingContract` address and, thus, it is vulnerable to an eventual signature replay.

Assets:

- MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]

Status: Accepted

Classification

Impact: 4/5

Likelihood: 1/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Consider implementing a fully-compliant message hash with the `EIP712`. In order to do so, it is recommended to include the verifying contract address into the hash.

Resolution: The development team accepted the finding and the risks arising from it.

F-2024-6228 - Unchecked Return Value in Token Transfer May Lead to Unexpected Behavior - Low

Description:

The methods `withdraw()` and `withdrawToken()` use the `ERC20` function `transfer()` in order to withdraw tokens from the contract.

```
function withdraw(
    uint128 nonce,
    address contract_id,
    address receiver_id,
    uint128 amount,
    bytes memory signature
) public {
    uint128 nonce_ts = nonce / NONCE_TS_SHIFT;
    require(nonce_ts > minTimestamp);
    require(nonce_ts < maxTimestamp);
    require(
        nonce_ts > uint128(block.timestamp) - WITHDRAW_DELAY_SECONDS,
        "Nonce time is expired, you can make a refund"
    ); // this transfer can only be refunded

    require(!usedNonces[nonce], "Nonce already used");
    require(
        verify(
            nonce,
            abi.encodePacked(contract_id),
            abi.encodePacked(receiver_id),
            amount,
            signature
        ),
        "Invalid signature"
    );
    usedNonces[nonce] = true;
    if (contract_id == NATIVE_TOKEN) {
        payable(receiver_id).transfer(amount);
    } else {
        IERC20(contract_id).transfer(receiver_id, amount);
    }
}

function withdrawToken(address tokenAddress, uint256 amount)
    public
    onlyOwner
{
    IERC20(tokenAddress).transfer(owner, amount);
}
```

However, the returned value (a bool) from the call `IERC20(address).transfer()` is not checked. As a consequence, the contract is blind to the success of that execution: if the transfer was not successful, it would not be caught by the contract, so the signature and nonce would be marked as used whilst the transaction failed.

Additionally, some tokens do not return any bool from the `transfer()` function, they don't return anything at all. One known example is the Binance Coin. Thus, checking the return value is not enough or safe.

Assets:

- MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]

Status: Fixed

Classification

Impact: 4/5

Likelihood: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to use the SafeERC20 library to handle token transfers. The `MetaWallet` contract also uses the imported Uniswap contract `TransferHelper` for a similar use. However, it is encouraged to use the SafeERC20 library for all token transfers instead of the `TransferHelper`.

Resolution: Fixed in commit ID `422795e`: the SafeERC20 library was implemented to handle ERC20 token transfers.

Observation Details

F-2024-6221 - Old Solidity Version May Result in Unsafe Code - Info

Description:

The project uses the floating `pragma ^0.8.0`.

This may result in the contracts being deployed using the wrong `pragma` version, which is different from the one they were tested with. For example, they might be deployed using an outdated `pragma` version which may include bugs that affect the system negatively.

Additionally, using such an outdated version of Solidity can lead to several issues, including missing out on important bug fixes, security enhancements, and improved language features introduced in later versions. Staying up-to-date with recent Solidity versions is crucial for maintaining the security, efficiency, and overall quality of smart contracts.

Assets:

- MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]
- RlpEncode.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]

Status: Accepted

Recommendations

Remediation:

It is recommended to lock the pragma version of Solidity. Additionally, consider using one of the latest solidity versions that is also compatible with the rest of the contracts. Note that updating the solidity version requires a revision of the dependencies.

Resolution:

Partially fixed in commit ID `422795e`: the solidity version was updated to `^0.8.20`. However the pragma is still floating instead of locked to `0.8.20`.

F-2024-6223 - Visibility of State Variable is Not Defined - Info

Description: `NATIVE_TOKEN` state variable visibility is not set explicitly.

The default variable visibility is `internal`. Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.

Assets:

- MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]

Status: Accepted

Recommendations

Remediation: It is recommended to define the visibility of the state variable `NATIVE_TOKEN`.

Resolution: The development team accepted the finding and the risks arising from it.

F-2024-6235 - Unused Function Parameters - Info

Description:

The `hot_verify` function includes parameters `walletId`, `userPayload` and `metadata`, but neither of them is used within the function's logic. These parameters are passed in but are not referenced or processed, suggesting they may be unnecessary or their intended use has not been implemented, moreover redundant parameters consume extra Gas and decrease code readability.

Assets:

- MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]

Status:

Accepted

Recommendations

Remediation:

Consider removing parameters `walletId` and `metadata` from `hot_verify` function.

Resolution:

The development team accepted the finding and the risks arising from it.

F-2024-6236 - Missing Event Emission - Info

Description:

Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes with timelocks that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services. The alternative of directly querying the on-chain contract state for such changes is not considered practical for most users/usages.

Currently the `changeOwner` function does not emit an event so it is challenging for external observers to monitor this critical change.

```
    function changeOwner(address newOwner) public onlyOwner {
          owner = newOwner;
      }
```

`changeOwner` should emit an event upon execution to enhance transparency and enable tracking of ownership changes.

The following functions should also emit events: `close()`, `changeVerifyAddress()`, `withdraw() → native token transfer branch`.

Assets:

- MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]

Status: Accepted

Recommendations

Remediation: It is advisable to implement an appropriate event to enhance the contract's observability and maintainability.

Resolution: The development team accepted the finding and the risks arising from it.

F-2024-6260 - Single-Step Owner Transfer May Cause DOS - Info

Description: The `MetaWallet` contracts uses the function `changeOwner()` to update the contract `owner`:

```
function changeOwner(address newOwner) public onlyOwner {
    owner = newOwner;
}
```

However, the `owner` role is critical for the contract and introducing a wrong address will result in a Denial of Service of the methods `withdrawToken()`, `withdrawEth()`, `changeOwner()`, `close()` and `changeVerifyAddress()`. In case of need for withdrawing tokens at any given time by the owner, it will not be possible.

Assets:

- MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]

Status: Accepted

Recommendations

Remediation: It is recommended to implement a two-step ownership transfer, where the new owner must accept the role. An example can be found here.

Resolution: The development team accepted the finding and the risks arising from it.

F-2024-6261 - Missing Zero Address Checks - Info

Description:

The following methods lack zero address checks: `deposit()` for native tokens, `withdraw()` and `changeOwner()`.

By failing to do so, the address `0x0` can be used, resulting in the following consequences:

- `deposit()` and `withdraw()`: native tokens can be sent to `0x0`, resulting in a loss of funds.
- `changeOwner()`: the contract ownership will be lost, resulting in a Denial of Service of all functions that are only callable by the contract owner.

Assets:

- MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]

Status: Accepted

Recommendations

Remediation: Introduce zero address checks in the functions `deposit()`, `withdraw()` and `changeOwner()`.

Resolution: The development team accepted the finding and the risks arising from it.

F-2024-6262 - Unused Code Should be Removed - Info

Description: The following interface is defined in `MetaWallet.sol`. However, this piece of code is not used and should be removed.

```
interface IWETH is IERC20 {
    function deposit() external payable;

    function withdraw(uint256 amount) external;
}
```

Assets:

- MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]

Status: Fixed

Recommendations

Remediation: It is recommended to remove unused code from the contracts.

Resolution: Fixed in commit ID `40ff89f`: the unused code was removed from the contracts.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project’s performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/hot-dao/omni-wallet-solidity
Commit	40ff89f
Whitepaper	-
Requirements	./README.md
Technical Requirements	./README.md

Asset	Type
MetaWallet.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]	Smart Contract
RlpEncode.sol [https://github.com/hot-dao/meta-wallet/tree/main/solidity]	Smart Contract