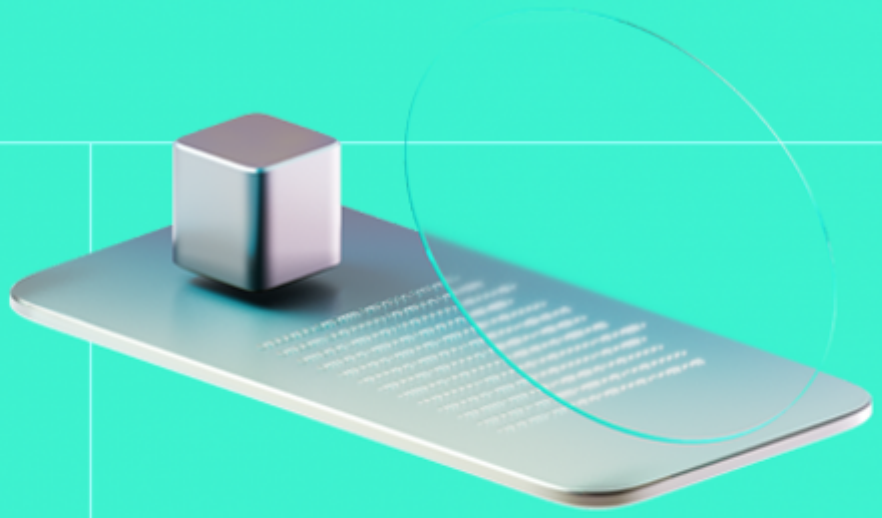
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Archethic

Date: 12/07/2024

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

Archethic (UCO) is a groundbreaking Layer 1 biometric blockchain with 11 revolutionary patents, integrating blockchain and biometrics for unparalleled security and privacy. It features a unique consensus algorithm and a patented biometric cold wallet, ensuring unmatched security, scalability, and a seamless user experience. AeBridge is a pioneering bridge solution that enables users to handle fund transfers between EVM and the Archethic chain, enhancing interoperability and expanding the capabilities of the Archethic ecosystem.

Document

Name	Smart Contract Code Review and Security Analysis Report for Archethic
Audited By	Turgay Arda Usman, Grzegorz Trawiński
Approved By	Ataberk Yavuzer
Website	https://www.archethic.net
Changelog	04/07/2024 - Preliminary Report
	12/07/2024 - Final Report
Platform	Ethereum, Archetic, BSC, Polygon
Language	Solidity
Tags	Bridge, ERC20, Atomic Swap
Methodology	https://hackenio.cc/sc_methodology

Review

Scope

Repository	https://github.com/archethic-foundation/bridge-contracts/tree/11cf88221d00c9ea029ae5a4cf08f14705199ce1
Commit	11cf882

Audit Summary

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

9	7	2	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	1
High	0
Medium	2
Low	5

Vulnerability	Status
F-2024-4134 - Missing Storage Gaps	Accepted
F-2024-4145 - Solution is a subject to chain re-org	Accepted
F-2024-4132 - Checks Effects Interactions Pattern Violation	Fixed
F-2024-4137 - Reentrancy Leading to Signature Replay in Withdrawals	Fixed
F-2024-4138 - Fee-on-Transfer Accounting-Related Issues	Fixed
F-2024-4139 - Missing Funds Transfer In Contract Creation	Fixed
F-2024-4140 - The provisionHTLC Function Can Be Front-Run	Fixed
F-2024-4141 - The provisionHTLC Function Accepts Arbitrary Amount	Fixed
F-2024-4142 - The mintHTLC Function Lacks Lockout Mechanism	Fixed

Documentation quality

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

Code quality

- The code mostly follows style guides and best practices.
  - See informational issues and observations for more details.
- The development environment is configured.

Test coverage

Code coverage of the project is around **73.08%** (branch coverage).

Table of Contents

System Overview

aeBridge is a p2p Bridge solution that aims its users to handle funds transfers between EVM and Archethic chain. It has the following contracts:

HTLC_ERC — HTLC contract customized for ERC20 transfers.

PoolBase — Pool to manage assets for Archethic's bridge on EVM's side.

ETHPool — Pool to manage ETH asset for Archethic's bridge on EVM's side.

ERCPool — Pool to manage ERC assets for Archethic's bridge on EVM's side.

ChargeableHTLC_ERC — HTLC contract with chargeable fee towards pool's safety module.

ChargeableHTLC_ETH — HTLC contract with chargeable fee towards pool's safety module.

HTLCBase — base logic for HashTime-Lock Contract.

SignedHTLC_ERC — HTLC contract with signature verification before withdraw for ERC20 swap.

SignedHTLC_ETH — HTLC contract with signature verification before withdraw for ether swap

Privileged roles

- The owner of the BasePool can upgrade the contract, lock and unlock the contract, update lock time period, and update Archetic Pool signer address.
- Other contracts are permissionless.

Risks

- The use of `tx.origin` to set the `from` address in the `ChargeableHTLC_ERC` constructor poses a significant security risk, as it can expose the contract to phishing and reentrancy attacks. Specifically, `tx.origin` refers to the original external account that initiated the transaction, which can be manipulated in scenarios where multiple contracts interact. This can lead to unauthorized actions if a malicious contract tricks a user into initiating a transaction, causing `tx.origin` to be the user's address instead of the intended contract.
- The audit does not cover all code in the repository. Contracts outside the audit scope may introduce vulnerabilities, potentially impacting the overall security due to the interconnected nature of smart contracts.
- The functioning of the system significantly relies on specific external contracts. Any flaws or vulnerabilities in these contracts adversely affect the audited project, potentially leading to security breaches or loss of funds.

Findings

Vulnerability Details

F-2024-4140 - The provisionHTLC Function Can Be Front-Run - Critical

Description:

The solution is an atomic swap bridge between EVM and Archethic blockchain. Whenever the swap is initiated on the Archethic side, the peer on EVM side can proceed with swap by means of `provisionHTLC` function. This function accepts the secret and signature generated by the Archethic's smart contract. As a result, the function deploys new instance of `SignedHTLC_ETH` contract with the purpose of one time usage. This contract saves the information that `msg.sender` is the final `receipient` of funds. The `SignedHTLC_ETH` instance receives the native token from the `ETHPool` contract. The `ETHPool` is funded in advance by the protocol owner.

In the next step the user can call the `withdraw` from the `SignedHTLC_ETE` to receive the native token. To accomplish this, the another secret and signature generated by the Archethic's smart contract must be provided.

However, it was identified that the `provisionHTLC` function can be front-runned by the attacker. The signature verified within this function does not include e.g. `msg.sender` value. So, any user can call `provisionHTLC` function, while having obtained the valid secret and signature, e.g. from the mempool. Additionally, the Client's team confirmed that on the Archethic's side the EVM's initiator address is not being tracked and verified. Thus, the solution could generate final artefacts required for `withdraw` function.

Apart from that, the `provisionHTLC` function can be front-runned by the attacker to consume the secret and signature, preventing legitimate user from finalising the swap.

```
function provisionHTLC(bytes32 _hash, uint256 _amount, uint _lockTime, bytes mem
        checkUnlocked();

        if (_hash == bytes32(0)) {
            revert InvalidHash();
        }

        if (_amount == 0) {
            revert InvalidAmount();
```

```
        }

[...]
        if (_lockTime == 0 || _lockTime < block.timestamp || (_lockTime - block.t
            revert InvalidLockTime();
        }

        if(address(_refProvisionedSwaps[_hash]) != address(0)) {
            revert AlreadyProvisioned();
        }

        bytes32 _archethicHTLCAddressHash = sha256(_archethicHTLCAddress);
        bytes32 messagePayloadHash = keccak256(abi.encode(_archethicHTLCAddressHa
        bytes32 signedMessageHash = ECDSA.toEthSignedMessageHash(messagePayloadHa

        address signer = ECDSA.recover(signedMessageHash, _v, _r, _s);
        if (signer != archethicPoolSigner) {
            revert InvalidSignature();
        }

        delete signer;
        delete messagePayloadHash;
        delete signedMessageHash;

        IHTLC htlcContract = _createSignedHTLC(_hash, _amount, _lockTime);
        _refProvisionedSwaps[_hash] = htlcContract;

        _provisionedSwaps.push(address(htlcContract));
        setSwapByOwner(msg.sender, address(htlcContract), _archethicHTLCAddress,
        emit ContractProvisioned(htlcContract, _amount);
    }
```

Assets:

- Pool/PoolBase.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]

Status: Fixed

Classification

Impact: 5/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Critical

Recommendations

Remediation: It is recommended to prevent the possibility of front-running and sweeping the funds by the not legitimate user. It is recommended to add recipient address to the signature verification. Additionally, it is recommended to track and verify the EVM's initiator address on the Archethic side.

Resolution: The `msg.sender` is now added to the signature and its verification [commit ID: 074d3b14c8bd42f35b8d57e225bdd12b1d510a1c].

Evidences

Proof of Concept

Reproduce:

1. As a protocol owner, prepare and deploy the `ETHPool` contract. Transfer native tokens to the `ETHPool` instance.
2. Simulate the swap transaction on the Archethic side and prepare the secret and the signature.
3. As a user, attempt to execute a call to the `provisionHTLC` function with correct input parameters.
4. As an attacker, observe the mempool. Note the transaction triggered in step 3. Front-run the transaction as attacker.
5. Observe that front-runned transaction finished successfully. Note that `SignedHTLC_ETH` instance has the `receipient` set to the frontrunner address.
6. Observe the transaction triggered by the legitimated user reverted with the `AlreadyProvisioned` error.

```
it("Hacken: provisionHTLC can be front-runned by the malicious user", async
    const { pool, accounts, archPoolSigner } = await loadFixture(deployPool)

    await accounts[1].sendTransaction({
        to: pool.getAddress(),
        value: ethers.parseEther("2.0"),
    });

    const buffer = new ArrayBuffer(32);
    const view = new DataView(buffer);
    view.setUint32(0x0, networkConfig.chainId, true);
    const networkIdUint8Array = new Uint8Array(buffer).reverse();

    const archethicHtlcAddress = "00004970e9862b17e9b9441cdbe7bc13aeb4c906a7
```

```
const archethicHtlcAddressHash = ethers.sha256(`0x${archethicHtlcAddress

const sigPayload = concatUint8Arrays([
    hexToUintArray(archethicHtlcAddressHash.slice(2)), // Archethic HTLC
    hexToUintArray("bd1eb30a0e6934af68c49d5dd5ad3e3c3d950ff977a730af56b5
    networkIdUint8Array
])

const hashedSigPayload2 = hexToUintArray(ethers.keccak256(`0x${uintArray
const signatur
```

See more

F-2024-4138 - Fee-on-Transfer Accounting-Related Issues - Medium

Description:

The functions below transfer funds from the caller to the receiver via `safeTransfer()`, but do not ensure that the actual number of tokens received is the same as the input amount to the transfer. If the token is a fee-on-transfer token, the balance after the transfer will be lower than expected, leading to accounting issues. One way to address this problem is to measure the balance before and after the transfer, and use the difference as the amount, rather than the stated amount.

```
function _transferAsWithdraw() internal override {
    ...
}

function _transferAsRefund() internal override {
    SafeERC20.safeTransfer(token, from, amount + fee);
}

function _transferAsWithdraw() internal override virtual {
        SafeERC20.safeTransfer(token, recipient, amount);
}
function _transferAsRefund() internal override virtual {
    SafeERC20.safeTransfer(token, from, amount);
}
```

Assets:

- HTLC/ChargeableHTLC_ERC.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]
- HTLC/HTLC_ERC.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]

Status: Fixed

Classification

Impact: 3/5

Likelihood: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: To mitigate potential vulnerabilities and ensure accurate accounting with fee-on-transfer tokens, modify your contract's token transfer logic to measure the recipient's balance before and after the transfer. Use this observed difference as the actual transferred amount for any further logic or calculations.

Resolution: The support for fee-on-transfer tokens is now disabled by enforcing revert whenever fee was collected by the ERC20 token [commit ID: 8d5d72f546f34dbdde4762168fbdf3ce6f5465ed].

F-2024-4139 - Missing Funds Transfer In Contract Creation - Medium

Description:

The `mintHTLC` function in the `ERCPool` contract and its internal `_createChargeableHTLC` method do not include a call to `token.safeTransferFrom`. This omission means that when creating a new `ChargeableHTLC_ERC` contract, the required ERC20 tokens are not automatically transferred from the user's balance to the HTLC contract. Consequently, the `ChargeableHTLC_ERC` contract must be manually funded separately after its creation, leading to a potential failure in automated workflows and an increased risk of user error.

```
function _createChargeableHTLC(bytes32 _hash, uint256 _amount, uint _lockTime) o
    uint256 _fee = swapFee(_amount, token.decimals());
    uint256 _recipientAmount = _amount - _fee;
    ChargeableHTLC_ERC htlcContract = new ChargeableHTLC_ERC(token, _recipientAmou
    return htlcContract;
}
```

In the current implementation, the `mintHTLC` function, or more specifically, the `_createChargeableHTLC` function, creates a new `ChargeableHTLC_ERC` contract without transferring the specified amount of ERC20 tokens from the user's address. This is contrary to the `ETHPool` implementation, where the required ether is sent within the `_createChargeableHTLC` method. The lack of an automated transfer mechanism for ERC20 tokens means that the new `ChargeableHTLC_ERC` contracts must be funded manually. This discrepancy between the handling of native coins and ERC20 tokens can cause the `mintHTLC` process to fail unless the user manually transfers the tokens to the HTLC contract.

Assets:

- Pool/ERCPool.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]

Status: Fixed

Classification

Impact: 3/5

Likelihood: 4/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: To resolve this issue, incorporate a `token.safeTransferFrom` call in the `mintHTLC` function or within the `_createChargeableHTLC` method. This will ensure that the specified amount of ERC20 tokens is transferred from the user's balance to the new HTLC contract automatically, mirroring the behavior for native ether transfers.

Resolution: The `_createChargeableHTLC` function now implements the ERC20 `transferFrom` operation [commit ID: b2c055020ab36a40ffebb7da2c8e4fffd1c704c4].

Evidences

Proof of Concept

Reproduce:

1. As a protocol owner, prepare and deploy the `ERCPool` contract.
2. As a user, execute call to the `mintHTLC` function with correct input parameters.
3. Observe the transaction finished successfully and the `ChargeableHTLC_ERC` contract is created. Note that no prior call for token approval was required.
4. As a user, attempt to call the `withdraw` function with correct input parameters. Observe the transaction reverts with the `InsufficientFunds` error.

F-2024-4132 - Checks Effects Interactions Pattern Violation - Low

Description:

It was identified that `HTLCBase.sol` contract has an instance of Checks-Effects-Interactions (CEI) pattern violation, where state variables are updated after the external calls to the token contract. As explained in Solidity Security Considerations, it is best practice to follow the CEI pattern while interacting with external contracts to avoid reentrancy-related issues. One reentrancy vulnerability was identified during the security assessment, however it is impractical, as contracts are meant to be one-time use. The finding is reported as a deviation from leading security practices.

```
function _withdraw(bytes32 _secret) internal {
    if (status != HTLCStatus.PENDING) {
    revert AlreadyWithdrawn();
    }
    if (sha256(abi.encodePacked(_secret)) != hash) {
    revert InvalidSecret();
    }
    if (!_enoughFunds()) {
    revert InsufficientFunds();
    }



    secret = _secret;
    _transferAsWithdraw();
    status = HTLCStatus.WITHDRAWN;
    emit Withdrawn();
}
...
function _refund() internal {
    if (status != HTLCStatus.PENDING) {
        revert AlreadyRefunded();
    }
    if (!_enoughFunds()) {
        revert InsufficientFunds();
    }
       _transferAsRefund();
    status = HTLCStatus.REFUNDED;
    emit Refunded();
 }
```

Assets:

- HTLC/HTLCBase.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]

Status: Fixed

Classification

Impact: 2/5

Likelihood: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation:

- It is recommended to follow the CEI pattern when interacting with external contracts.
- It is recommended to use reentrancy locks.

Resolution: The `_withdraw` and `_refund` functions now implement CEI pattern [commit ID: c350ca9d70ab4b68e5e2dec5e9178aa652fcac9a].

F-2024-4134 - Missing Storage Gaps - Low

Description:

When working with upgradeable contracts, it is necessary to introduce storage gaps to allow for storage extension during upgrades.

Storage gaps are a convention for reserving storage slots in a base contract, allowing future versions of that contract to use up those slots without affecting the storage layout of child contracts.

The affected contract(s): `PoolBase.sol`

Note: OpenZeppelin Upgrades checks the correct usage of storage gaps.

Assets:

- Pool/PoolBase.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]

Status: Accepted

Classification

Impact: 3/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation:

Introduce Storage Gaps in the affected contracts.

To create a storage gap, declare a fixed-size array in the base contract with an initial number of slots. This can be an array of `uint256` so that each element reserves a 32 byte slot. Use the name `__gap` or a name starting with `__gap_` for the array so that OpenZeppelin Upgrades will recognize the gap.

To help determine the proper storage gap size in the new version of your contract, you can simply attempt an upgrade using `upgradeProxy` or just run the validations with `validateUpgrade` (see docs for Hardhat or Truffle). If a storage gap is not being reduced properly, you will

see an error message indicating the expected size of the storage gap.

Resolution: The Client's team confirmed issue and plan to implement a fix in a next version

F-2024-4141 - The provisionHTLC Function Accepts Arbitrary Amount - Low

Description:

The solution is an atomic swap bridge between EVM and Archethic blockchain. Whenever the swap is initiated on the Archethic side, the peer on EVM side can proceed with swap by means of `provisionHTLC` function. This function accepts the secret and signature generated by the Archethic's smart contract. Additionally, this function accepts the `_amount` input parameter to define an amount of native tokens to be given to the EVM peer. However, this input parameter is not included in the signature verification, thus, any amount can be provided, e.g. the whole balance of `ETHPool`.

The Client's team confirmed that on the Archethic's side the `amount` parameter is verified, thus manipulation of the native tokens to be transferred is mitigated. Nevertheless, this approach appears to be single point of failure and it is considered a deviation from the leading security practices.

```
function provisionHTLC(bytes32 _hash, uint256 _amount, uint _lockTime, bytes mem
        checkUnlocked();

        if (_hash == bytes32(0)) {
            revert InvalidHash();
        }

        if (_amount == 0) {
            revert InvalidAmount();
        }

[...]

        bytes32 _archethicHTLCAddressHash = sha256(_archethicHTLCAddress);
        bytes32 messagePayloadHash = keccak256(abi.encode(_archethicHTLCAddressHa
        bytes32 signedMessageHash = ECDSA.toEthSignedMessageHash(messagePayloadHa

        address signer = ECDSA.recover(signedMessageHash, _v, _r, _s);
        if (signer != archethicPoolSigner) {
            revert InvalidSignature();
        }
[...]
    }
```

Assets:

- Pool/PoolBase.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]

Status: Fixed

Classification

Impact: 5/5

Likelihood: 1/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to include the `_amount` input parameter in the signature verification check to prevent any possibility of manipulation and undesired transfer of excessive amount of funds.

Resolution: The `_amount` is now included in the signature and its verification [commit ID: ebf2d4714a11eaffc28abadf0d751e15feca7be1].

F-2024-4142 - The mintHTLC Function Lacks Lockout Mechanism - Low

Description:

The `PoolBase` contract allows user to either begin swap procedure by means of the `mintedSwaps` or continue swap initiated on the Archethic side by means of the `provisionHTLC`. The `provisionHTLC` function is protected with the `checkUnlocked` function, whereas `mintHTLC` is not. The `checkUnlocked` acts as pausability pattern implementation.

This disparity pose a risk that in the event of emergency the `provisionHTLC` is protected, but `mintHTLC` is not, thus, a swap can be triggered at any time on the EVM side.

```
    function provisionHTLC(bytes32 _hash, uint256 _amount, uint _lockTime, bytes
        checkUnlocked();

        if (_hash == bytes32(0)) {
            revert InvalidHash();
        }
    [...]
    function mintHTLC(bytes32 _hash, uint256 _amount) payable virtual external {
        if (_hash == bytes32(0)) {
            revert InvalidHash();
        }

        if (_amount == 0) {
            revert InvalidAmount();
        }

        _mintHTLC(_hash, _amount, _chargeableHTLCLockTime());
    }
    [...]
    function checkUnlocked() internal view {
        require(!locked, "Locked");
    }
    [...]
        function unlock() virtual external {
        _checkOwner();
        locked = false;
        emit Unlock();
    }
    [...]
    function lock() virtual external {
        _checkOwner();
        locked = true;
```

```
        emit Lock();
    }
```

Assets:

- Pool/PoolBase.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]

Status: Fixed

Classification

Impact: 2/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to add `checkUnlocked` check to the `mintHTLC` function.

Resolution: The `checkUnlocked` is now implemented for both `mintHTLC` and `provisionHTLC` functions [commit ID: 4dec7491587d14022c215c35005c34c0733b719e] .

F-2024-4145 - Solution is a subject to chain re-org - Low

Description: The solution is supposed to be an atomic-swap bridge. Thus, no latency of the tokens transfers is a main priority. However, this property makes the solution vulnerable to chain re-org. Within the code, no assertions were identified that checks whether a certain number of blocks passed to allow fulfil transfer.

This creates a risk when a swap is finalised on one chain, but reverted on the other due to chain re-org event.

Status: Accepted

Classification

Impact: 5/5

Likelihood: 1/5

Exploitability: Dependent

Complexity: Medium

Severity: Low

Recommendations

Remediation: It is recommended to consider chain re-org risk and apply security controls to prevent occurrence of one-side swap:

- Delay funds transfer on EVM side after certain number of blocks passed.
- Delay funds transfer on Archethic side after certain number of blocks passed.
- Whenever RPC API is in use, ensure it fetches data only fo finalised blocks.

Resolution: The Client's team is aware about the risk and accepted it.

F-2024-4137 - Reentrancy Leading to Signature Replay in Withdrawals - Info

Description:

A signature replay attack occurs when an attacker reuses a valid signature to perform unauthorized transactions multiple times. In the context of the `ChargeableHTLC_ERC` smart contract, the `withdraw` function could be vulnerable to such an attack if the contract's state is not properly updated before performing external calls.

The `withdraw` function in the `ChargeableHTLC_ERC` contract verifies a signature before allowing the withdrawal of tokens. This signature is derived from the `hash`, ensuring that only a valid signer can authorize withdrawals. However, a critical vulnerability exists in the `_withdraw` function's state update order. The function first performs the token transfer via the `_transferAsWithdraw` call and only afterward updates the contract's state by setting `status` to `WITHDRAWN` and `secret` to the provided `_secret`.

```
function withdraw(bytes32 _secret, bytes32 _r, bytes32 _s, uint8 _v) external {
   if (!_beforeLockTime(block.timestamp)) {
       revert TooLate();
   }
   bytes32 sigHash = ECDSA.toEthSignedMessageHash(hash);
   address signer = ECDSA.recover(sigHash, _v, _r, _s);
   if (signer != poolSigner) {
       revert InvalidSignature();
   }
   delete sigHash;
   delete signer;
   _withdraw(_secret);
}

function _withdraw(bytes32 _secret) internal {
    if (status != HTLCStatus.PENDING) {
        revert AlreadyWithdrawn();
    }
    if (sha256(abi.encodePacked(_secret)) != hash) {
        revert InvalidSecret();
    }
    if (!_enoughFunds()) {
        revert InsufficientFunds();
    }

    secret = _secret;
    _transferAsWithdraw();
    status = HTLCStatus.WITHDRAWN;
```

```
        emit Withdrawn();
    }
```

This order of operations can lead to a re-entrancy attack. In a re-entrancy attack, an attacker could exploit the external call within `_transferAsWithdraw` to re-enter the `withdraw` function before the state has been updated. Since the state remains `PENDING` until after the external call, the attacker can repeatedly call `withdraw` with the same valid signature and secret, enabling multiple unauthorized withdrawals. This oversight can lead to significant financial loss as the attacker can drain the contract's funds by executing the attack. Since in the first withdrawal the contract distributes all its funds, the impact is reduced.

Assets:

- HTLC/SignedHTLC_ETH.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]

Status: Fixed

Classification

Impact: 1/5

Likelihood: 4/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

- Implement reentrancy guards.
- Update the contract state before making any external calls to follow the Checks-Effects-Interactions pattern.

Resolution: The `_withdraw` and `_refund` functions now implement CEI pattern [commit ID: c350ca9d70ab4b68e5e2dec5e9178aa652fcac9a].

Observation Details

F-2024-4131 - Missing Zero Address Validation - Info

Description:

In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the “**zero address”**. This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address.

The "**Missing zero address Validation**" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.

For instance, consider a contract that includes a function to change its owner. This function is crucial, as it determines who has administrative access. However, if this function lacks proper validation checks, it might inadvertently permit the setting of the owner to the zero address. Consequently, the administrative functions will become unusable.

Missing checks were observed in the following functions:

- _createSignedHTLC()
- initialize()
- setToken()
- constructor()

Assets:

- HTLC/ChargeableHTLC_ERC.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]
- HTLC/ChargeableHTLC_ETH.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]
- HTLC/HTLC_ERC.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]
- HTLC/SignedHTLC_ERC.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]
- HTLC/SignedHTLC_ETH.sol [https://github.com/archethic-foundation/bridge-contracts/tree/main/evm]

Status: Fixed

Recommendations

Remediation: Implement zero address checks for the aforementioned functions.

Resolution: The zero address checks for the aforementioned functions are now implemented [commit ID: 7ae328681315d3730e85280b5d6d3d189c88c540].

F-2024-4144 - Protocol is not compliant with the EIP-712 - Info

Description:

The protocol uses signature based authorisation in following functions: `provisionHTLC` (`PoolBase`) and `withdraw` (`SignedHTLC_ETH`, `SignedHTLC_ERC`, `ChargeableHTLC_ETH`, `ChargeableHTLC_ERC`).

```
function provisionHTLC(bytes32 _hash, uint256 _amount, uint _lockTime, bytes mem
[...]
        bytes32 _archethicHTLCAddressHash = sha256(_archethicHTLCAddress);
        bytes32 messagePayloadHash = keccak256(abi.encode(_archethicHTLCAddressHa
        bytes32 signedMessageHash = ECDSA.toEthSignedMessageHash(messagePayloadHa

        address signer = ECDSA.recover(signedMessageHash, _v, _r, _s);
        if (signer != archethicPoolSigner) {
            revert InvalidSignature();
        }
[...]
```

```
    function withdraw(bytes32 _secret, bytes32 _r, bytes32 _s, uint8 _v) overrid
        bytes32 sigHash = ECDSA.toEthSignedMessageHash(_secret);
        address signer = ECDSA.recover(sigHash, _v, _r, _s);
[...]
        if (signer != poolSigner) {
            revert InvalidSignature();
        }
[...]
```

In the first instance, for the signature verification a hash of `_archethicHTLCAddressHash`, `block.chainid` and secret is created. In the second instance only the secret is taken into account.

Thus, such implementation is not compliant with the EIP-712: Typed structured data hashing and signing. The EIP-712 assumes that the hash compared with the signature should include a domain separator, created with:

- name
- version
- blochain ID
- contract's address

Lack of compliance may lead to signature replay in other smart contracts and blockchains. It is also considered a deviation from leading security practices.

Status: Accepted

Recommendations

Remediation: It is recommended to implement signature based authorisation compliant with the EIP-712 standard.

Resolution: The Client's team accepted the finding.

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. Severity Definitions

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, do not affect security score but can affect code quality score.

Appendix 2. Scope

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

Scope Details	
Repository	https://github.com/archethic-foundation/bridge-contracts/tree/11cf88221d00c9ea029ae5a4cf08f14705199ce1
Commit	11cf882
Whitepaper	https://wiki.archethic.net/participate/bridge/
Requirements	https://github.com/archethic-foundation/bridge-contracts/blob/11cf88221d00c9ea029ae5a4cf08f14705199ce1/README.md
Technical Requirements	https://github.com/archethic-foundation/bridge-contracts/blob/11cf88221d00c9ea029ae5a4cf08f14705199ce1/README.md

Contracts in Scope
HTLC/ChargeableHTLC_ERC.sol
HTLC/ChargeableHTLC_ETH.sol
HTLC/HTLC_ERC.sol
HTLC/HTLC_ETH.sol
HTLC/HTLCBase.sol
HTLC/SignedHTLC_ERC.sol
HTLC/SignedHTLC_ETH.sol
Pool/ETHPool.sol
Pool/ERCPool.sol
Pool/PoolBase.sol
interfaces/IHTLC.sol
interfaces/IPool.sol

Contract	Address
Pool/ERCPool.sol (Ethereum)	0x346Dba8b51485FfBd4b07B0BCb84F48117751AD9
Pool/ERCPool.sol (Polygon)	0xd5cA9F76495b853a5054814A10b6365ee8ed745B
Pool/ERCPool.sol (BSC)	0xE01F0ee653648192812B2D23CBfe7E147727B672