decurity

Smart Contract Security Audit Report

Cryptolegacy Audit

1. Contents

2. General Information

This report contains information about the results of the security audit of the Cryptolegacy (hereafter referred to as “Customer”) smart contracts, conducted by Decurity in the period from 2025-04-21 to 2025-05-05.

2.1. Introduction

Tasks solved during the work are:

- Review the protocol design and the usage of 3rd party dependencies,
- Audit the contracts implementation,
- Develop the recommendations and suggestions to improve the security of the contracts.

2.2. Scope of Work

The audit scope included the contracts in the following repository: https://github.com/CryptoCust/cryptolegacy-contracts. Initial review was done for the commit e409929501d5b78941bc78411f1fa3da36249a94 and the re-testing was done for the commit 137d597231a13c1d4951dfb847581d7ba577d369.

The following contracts have been included in the scope by the Customer:

- contracts/BeneficiaryRegistry.sol
- contracts/BuildManagerOwnable.sol
- contracts/CryptoLegacy.sol
- contracts/CryptoLegacyBuildManager.sol
- contracts/CryptoLegacyDiamondBase.sol
- contracts/CryptoLegacyFactory.sol
- contracts/CryptoLegacyOwnable.sol
- contracts/FeeRegistry.sol
- contracts/LegacyMessenger.sol

- contracts/LifetimeNft.sol
- contracts/LockChainGate.sol
- contracts/PluginsRegistry.sol
- contracts/SignatureRoleTimelock.sol
- contracts/libraries/LibCreate2Deploy.sol
- contracts/libraries/LibCryptoLegacy.sol
- contracts/libraries/LibCryptoLegacyPlugins.sol
- contracts/libraries/LibSafeMinimalMultisig.sol
- contracts/libraries/LibTrustedGuardiansPlugin.sol
- contracts/plugins/CryptoLegacyBasePlugin.sol
- contracts/plugins/LegacyRecoveryPlugin.sol
- contracts/plugins/TrustedGuardiansPlugin.sol

2.3. Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, token owner, token service owner, a contract). The centralization risks have not been considered upon the request of the Customer.

The main possible threat actors are:

- User,
- Protocol owner,
- Liquidity Token owner/contract.

2.4. Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

2.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided “as is” and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer’s project, nor is it an investment advice.

That being said, Decurity exercises best effort to perform their contractual obligations and follow the industry methodologies to discover as many weaknesses as possible and maximize the audit coverage using the limited resources.

3. Summary

As a result of this work, we have discovered high-severity security issues which has been fixed and re-tested in the course of the work.

The other suggestions included fixing the low-risk issues and some best practices.

The team has given the feedback for the suggested changes and explanation for the underlying code.

3.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of July 15, 2025.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
originalBeneficiaryHash collision may happen	contracts/plugins/CryptoLegacyBasePlugin.sol	**High**	Fixed
Rebase tokens may become stuck on the contract during distribution	contracts/libraries/LibCryptoLegacy.sol	**High**	Fixed
checkBuildManagerValid() Validation Bypass	contracts/BuildManagerOwnable.sol	**Medium**	Fixed
If a voted guardian was removed, its vote is still counted	contracts/plugins/TrustedGuardiansPlugin.sol	**Medium**	Fixed
Invalid votes of beneficiaries are not deleted	contracts/plugins/TrustedGuardiansPlugin.sol	**Medium**	Fixed
Anyone can lock LifetimeNft for user	contracts/CryptoLegacyBuildManager.sol	**Medium**	Fixed

Issue	Contract	Risk Level	Status
deployByCreate2 could return unexpected address with salt = 0	cryptolegacy-contracts/contracts/libraries/LibCreate2Deploy.sol	Medium	Fixed
Use of deprecated transfer and send functions	contracts/CryptoLegacyBuildManager.sol contracts/FeeRegistry.sol contracts/libraries/LibCryptoLegacy.sol contracts/plugins/CryptoLegacyBasePlugin.sol	Medium	Fixed
tokenPrepareToDistribute can be simplified	contracts/libraries/LibCryptoLegacy.sol	Low	Fixed
Incorrect event emission	contracts/LockChainGate.sol	Low	Fixed
DoS in takeFee	contracts/FeeRegistry.sol	Low	Fixed
Lack of the beneficiary registry update in renounceOwnership()	cryptolegacy-contracts/contracts/plugins/CryptoLegacyBasePlugin.sol	Low	Fixed
Lack of destination chain check in the referral codes functionality	cryptolegacy-contracts/contracts/FeeRegistry.sol	Low	Fixed
Some of the contract functions can be called even if the initial fee wasn't paid	contracts/plugins/TrustedGuardiansPlugin.sol contracts/plugins/LegacyRecoveryPlugin.sol	Low	Fixed
Valid proposal can be not executed	contracts/plugins/LegacyRecoveryPlugin.sol	Low	Fixed

Issue	Contract	Risk Level	Status
Fees bypass using locked NFT transfer	contracts/LockChainGate.sol	Low	Fixed
Missing reentrancy protection in fee handling functions	contracts/FeeRegistry.sol	Low	Fixed
Unsafe ERC721 minting operation	contracts/LifetimeNft.sol contracts/mocks/MockLifetimeNft.sol	Low	Fixed
Hardcoded gas limits	contracts/CryptoLegacyDiamondBase.sol contracts/libraries/LibCryptoLegacy.sol	Low	Fixed
Excess msg.value not refunded to users	contracts/CryptoLegacyBuildManager.sol	Low	Fixed
Wrong comment	contracts/CryptoLegacyBuildManager.sol	Info	Fixed
Double event emitting	cryptolegacy-contracts/contracts/CryptoLegacyOwnable.sol	Info	Fixed
Gas optimization in SignatureRole	/cryptolegacy-contracts/contracts/SignatureRoleTimelock.sol	Info	Fixed
Beneficiaries can set their own referral address and share during the claim	contracts/plugins/CryptoLegacyBasePlugin.sol	Info	Acknowledged
Wrong supported chains configuration in the deployment script	script/LibDeploy.sol	Info	Fixed

Issue	Contract	Risk Level	Status
Lifetime NFTs takeover in case of incorrect deployement configuration	script/CryptoLegacyFactory.s.sol	**Info**	Fixed
updateInterval and challengeTimeout can be constants	contracts/CryptoLegacyBuildManager.sol	**Info**	Acknowledged

4. General Recommendations

This section contains general recommendations on how to improve overall security level.

The Findings section contains technical recommendations for each discovered issue.

4.1. Security Process Improvement

The following is a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level:

- Keep the whitepaper and documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new contracts and updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign for the contracts.

5. Findings

5.1. originalBeneficiaryHash collision may happen

Risk Level: High

Status: Fixed in the commit.

Contracts:

- contracts/plugins/CryptoLegacyBasePlugin.sol

Description:

The protocol implements a beneficiary switching mechanism to allow the reassignment of vesting rights over time. This is supported via the beneficiarySwitch function, which preserves the original beneficiary hash (originalBeneficiaryHash) to ensure that the cumulative claimed amount (tokenAmountClaimed) is tracked correctly, even after a beneficiary is changed.

However, there is a flaw in how originalBeneficiaryHash is managed and validated. During setup beneficiary configuration (via setBeneficiaries), if a beneficiary is added with an originalBeneficiaryHash that has already been used for another address, a collision may occur. This may happen if beneficiary that was added previously changed his address to to another via beneficiarySwitch function.

Initial setup:

a → a

b → b

c → c

The user switches beneficiaries:

d → a

Later, the original beneficiaries are re-added:

a → a

b → b

c → c

d → a

In this case, both a and d reference the same originalBeneficiaryHash(a). This causes their vesting claims to be tracked using the same underlying accounting entry. As a result, both addresses may experience unexpected behavior or be unable to fully claim their vested tokens, since they share the same claimed amount counter.

If an address is added again via setBeneficiaries with a new BPS value, its originalBeneficiaryHash will be overwritten and the problem will not occur:

a → a

b → b

c → c

d → d

Remediation:

Consider not allowing the switch of beneficiaries until distribution starts.

5.2. Rebase tokens may become stuck on the contract during distribution

Risk Level: **High**

Status: Fixed in the commit.

Contracts:

- contracts/libraries/LibCryptoLegacy.sol

Description:

According to the documentation, the system supports rebase tokens, which can adjust their rebase rate based on factors such as total supply, current block number, and the value of the underlying asset.

During token distribution to beneficiaries, the _tokenPrepareToDistribute() function from the CryptoLegacyBasePlugin is invoked before each claim. This function updates the amountToDistribute based on the current token balance and the amount already claimed.

```
function _tokenPrepareToDistribute(ICryptoLegacy.CryptoLegacyStorage storage
cls, address _token) internal returns(ICryptoLegacy.TokenDistribution
storage td) {
    td = cls.tokenDistribution[_token];
     uint256 bal = IERC20(_token).balanceOf(address(this));
    if (td.amountToDistribute == 0) {
        td.amountToDistribute = bal;
    } else if (bal > td.amountToDistribute - td.totalClaimedAmount) {
        td.amountToDistribute += bal - (td.amountToDistribute -
td.totalClaimedAmount);
    } else if (bal < td.amountToDistribute - td.totalClaimedAmount) {
        td.amountToDistribute = bal + td.totalClaimedAmount; // @audit
rebase tokens can break the logic
    }
}
```

However, if the rebase rate of the token changes after some claims have already been processed — resulting in a reduced balance — the logic may break. This occurs because the amountToDistribute is calculated as the sum of the current balance and the totalClaimedAmount. Since totalClaimedAmount reflects values based on the previous token rate, the resulting amountToDistribute becomes inflated. As a result, later beneficiaries may be unable to claim their share, as the contract attempts to transfer more tokens than it actually holds.

Below is the test that fails due to decreased rebase rate:

```
function testAuditRebaseToken() public {
  bytes8 customRefCode = 0x0123456789abcdef;
  vm.prank(alice);
  (bytes8 refCode, ) = buildManager.createCustomRef(customRefCode,
aliceRecipient, _getRefChains(), _getRefChains());
   vm.startPrank(owner);
  pluginsRegistry.addPlugin(lensPlugin, "123");
  pluginsRegistry.addPluginDescription(lensPlugin, "123");
  vm.stopPrank();
   bytes32[] memory beneficiaryArr;
  ICryptoLegacy.BeneficiaryConfig[] memory beneficiaryConfigArr;
  CryptoLegacyBasePlugin cryptoLegacy;
  ICryptoLegacyLens cryptoLegacyLens;
  uint256 discount = buildFee * refDiscountPct / 10000;
  (cryptoLegacy, cryptoLegacyLens, beneficiaryArr, beneficiaryConfigArr) =
_buildCryptoLegacy(bob, buildFee - discount, customRefCode);
   ICryptoLegacyLens.CryptoLegacyBaseData memory clData =
cryptoLegacyLens.getCryptoLegacyBaseData();
   vm.warp(block.timestamp + clData.updateInterval);
   vm.startPrank(bob);
  cryptoLegacy.update{value: 0.09 ether}(_getEmptyUintList(),
_getEmptyUintList());
  beneficiaryArr = new bytes32[](3);
  beneficiaryArr[0] = keccak256(abi.encode(bobBeneficiary1));
  beneficiaryArr[1] = keccak256(abi.encode(bobBeneficiary2));
  beneficiaryArr[2] = keccak256(abi.encode(bobBeneficiary3));
  beneficiaryConfigArr = new ICryptoLegacy.BeneficiaryConfig[](3);
  beneficiaryConfigArr[0] = ICryptoLegacy.BeneficiaryConfig(0, 0, 4000);
  beneficiaryConfigArr[1] = ICryptoLegacy.BeneficiaryConfig(0, 0, 0);
  beneficiaryConfigArr[2] = ICryptoLegacy.BeneficiaryConfig(0, 0, 6000);
  cryptoLegacy.setBeneficiaries(beneficiaryArr, beneficiaryConfigArr);
  vm.stopPrank();
   vm.startPrank(treasury);
  AuditMockERC20Rebase rebaseToken = new AuditMockERC20Rebase();
  rebaseToken.approve(address(cryptoLegacy), 100 ether);
  rebaseToken.setRebaseRate(1000); // balance = underlying * (100% + 10%)
  vm.stopPrank();
   vm.warp(block.timestamp + clData.updateInterval + 1);
   vm.prank(bobBeneficiary1);
  cryptoLegacy.initiateChallenge();
  address[] memory _treasuries = new address[](1);
  _treasuries[0] = treasury;
  address[] memory _tokens = new address[](1);
  _tokens[0] = address(rebaseToken);
   vm.warp(block.timestamp + clData.challengeTimeout + 1);
   vm.startPrank(bobBeneficiary1);
  cryptoLegacy.transferTreasuryTokensToLegacy(_treasuries, _tokens);
  cryptoLegacy.beneficiaryClaim(_tokens, address(0), 0);
  vm.stopPrank();
   vm.prank(treasury);
  rebaseToken.setRebaseRate(500); // // balance = underlying * (100% + 5%)
```

```
    vm.prank(bobBeneficiary3);
   cryptoLegacy.beneficiaryClaim(_tokens, address(0), 0); // @audit reverts,
beacuse of incorrect balance calculations
}
```

Mock for a rebase token used in the test:

```
pragma solidity 0.8.24;
 import "./MockERC20.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
 contract AuditMockERC20Rebase is MockERC20 {
   uint256 public rebaseRate = 1000; // 10%
   uint256 constant BASIS = 10000;
   constructor() MockERC20("Mock Rebase", "MockR") {
     _mint(msg.sender, 1000000 ether);
   }
   function setRebaseRate(uint256 _rebaseRate) external onlyOwner {
     rebaseRate = _rebaseRate;
   }
    function growthFactor() public view returns (uint256) {
     return BASIS + rebaseRate;
   }
      function _transfer(
     address from,
     address to,
     uint256 amount
   ) internal override {
     require(from != address(0), "ERC20: transfer from the zero address");
     require(to != address(0), "ERC20: transfer to the zero address");
      uint256 sharesAmount = (amount * BASIS) / growthFactor();
     _beforeTokenTransfer(from, to, amount);
     uint256 fromBalance = _balances[from];
     require(fromBalance >= sharesAmount, "ERC20: transfer amount exceeds
balance");
     unchecked {
       _balances[from] = fromBalance - sharesAmount;
       _balances[to] += sharesAmount;
     }
     emit Transfer(from, to, sharesAmount);
     _afterTokenTransfer(from, to, amount);
   }
    function balanceOf(address account) public view override returns
(uint256) {
     return (_balances[account] * growthFactor()) / BASIS;
   }
    function sharesOf(address account) public view returns (uint256) {
     return _balances[account];
   }
}
```

Remediation:

Since rebase tokens can alter their balances based on custom implementations, supporting them directly can be complex and sometimes unpredictable. It is recommended to support wrapped versions of these tokens that maintain a fixed balance representation, such as wstETH for stETH.

References:

- https://www.rareskills.io/post/rebase-token
- https://docs.cryptolegacy.app/detailed-beneficiary-claim-flow

5.3. checkBuildManagerValid() Validation Bypass

Risk Level: **Medium**

Status: Fixed

Contracts:

- contracts/BuildManagerOwnable.sol

Description:

The _checkBuildManagerValid() function in BuildManagerOwnable contract is vulnerable to a manipulation attack through a gas-dependent validation mechanism. The function calls cl.buildManager() which can be exploited by a malicious contract that returns different build manager addresses based on the remaining gas.

As demonstrated in the LegacyMessengerTest, an attacker can create a contract that implements the ICryptoLegacy interface with a specially crafted buildManager() function. This function returns a malicious build manager address during the first call (which passes the isCryptoLegacyBuilt() check since the malicious build manager always returns true), but returns the original legitimate build manager address during the subsequent buildManagerAdded.contains() check. Since the original build manager is already in the set, this check passes as well.

This allows bypassing the build manager validation process, enabling unauthorized message sending through the LegacyMessenger and allowing malicious CryptoLegacy contracts to be registered in the BeneficiaryRegistry.

LegacyMessengerTest.t.sol:

```
pragma solidity ^0.8.24;
 import "forge-std/Test.sol";
import "forge-std/console.sol";
import "../contracts/LegacyMessenger.sol";
import "../contracts/interfaces/ILegacyMessenger.sol";
import "../contracts/interfaces/ICryptoLegacy.sol";
import "../contracts/interfaces/ICryptoLegacyBuildManager.sol";
 contract MockBuildManager {
    function isCryptoLegacyBuilt(address _cryptoLegacy) external pure
returns (bool) {
        return false;
    }
}
 contract HackBuildManager {
    function isCryptoLegacyBuilt(address _cryptoLegacy) external pure
returns (bool) {
        return true;
    }
}
 contract HackCryptoLegacy {
    address public owner;
    ICryptoLegacyBuildManager public buildManagerOrig;
    ICryptoLegacyBuildManager public buildManagerHack;
     constructor(address _owner, address orig, address hack) {
        owner = _owner;
        buildManagerOrig = ICryptoLegacyBuildManager(orig);
        buildManagerHack = ICryptoLegacyBuildManager(hack);
    }
     function buildManager() external view returns (address) {
        console.log("[+] Gas Left: ", gasleft());
        if (gasleft() > 8937393460516441179) {
            return address(buildManagerHack);
        } else {
            return address(buildManagerOrig);
        }
    }
}
 contract LegacyMessengerTest is Test {
    LegacyMessenger public legacyMessenger;
    MockBuildManager public buildManager;
    address public owner;
    address public sender;
     function setUp() public {
        owner = address(0x1);
        sender = address(0x2);
         buildManager = new MockBuildManager();
         legacyMessenger = new LegacyMessenger(owner);
         vm.prank(owner);
        legacyMessenger.setBuildManager(address(buildManager), true);
    }
     function testSendMessagesTo() public {
```

```
        bytes32[] memory recipientList = new bytes32[](1);
        recipientList[0] = keccak256(abi.encode("recipient1"));
        bytes32[] memory messageHashList = new bytes32[](1);
        messageHashList[0] = keccak256(abi.encode("message1"));
        bytes[] memory messageList = new bytes[](1);
        messageList[0] = bytes("message1");
        bytes[] memory messageCheckList = new bytes[](1);
        messageCheckList[0] = bytes("check1");
        uint256 messageType = 1;
         vm.roll(100);
         vm.startPrank(sender);
         address hackBuildManager = address(new HackBuildManager());
        HackCryptoLegacy maliciousCryptoLegacy = new
HackCryptoLegacy(sender, address(buildManager), address(hackBuildManager));
         legacyMessenger.sendMessagesTo(
            address(maliciousCryptoLegacy),
            recipientList,
            messageHashList,
            messageList,
            messageCheckList,
            messageType
        );
         uint64[] memory blockNumbers1 =
legacyMessenger.getMessagesBlockNumbersByRecipient(recipientList[0]);
         assertEq(blockNumbers1.length, 1);
        assertEq(blockNumbers1[0], 100);
         vm.stopPrank();
    }
}
```

Remediation:

To remediate this vulnerability, implement a pattern that obtains the build manager address once and reuses it for all subsequent checks:

```
function _checkBuildManagerValid(address _cryptoLegacy, address _clOwner)
internal view {
    ICryptoLegacy cl = ICryptoLegacy(_cryptoLegacy);
     if (_clOwner != address(0) && cl.owner() != _clOwner) {
        revert NotTheOwnerOfCryptoLegacy();
    }
     // Get the build manager address once and store it
    address buildManagerAddr = address(cl.buildManager());
     // Use the stored address for all subsequent calls
    if
(!ICryptoLegacyBuildManager(buildManagerAddr).isCryptoLegacyBuilt(_cryptoLeg
acy)) {
        revert CryptoLegacyNotRegistered();
    }
     if (!buildManagerAdded.contains(buildManagerAddr)) {
        revert BuildManagerNotAdded();
    }
}
```

5.4. If a voted guardian was removed, its vote is still counted

Risk Level: Medium

Status: Fixed in the commit.

Contracts:

- contracts/plugins/TrustedGuardiansPlugin.sol

Description:

The _isGuardianVoted() function includes logic to automatically remove voters when beneficiaries act as guardians and a beneficiary who has voted is subsequently removed. However, this mechanism does not apply to trusted guardians — if a trusted guardian is removed after casting a vote, their vote remains counted, as the removal logic is not implemented for this case.

Remediation:

Implement automatic non-guardian voters removal for the following condition: isInitialized && !_pluginStorage.guardians.contains(voted).

5.5. Invalid votes of beneficiaries are not deleted

Risk Level: Medium

Status: Fixed in the commit.

Contracts:

- contracts/plugins/TrustedGuardiansPlugin.sol

Description:

The _isGuardianVoted function fails to remove votes from beneficiaries who were removed if the guardians have already been initialized (i.e., guardians.length() != 0). As a result, stale or unauthorized votes may persist in the guardiansVoted list, potentially affecting the integrity of voting-based logic.

```
function _isGuardianVoted(ICryptoLegacy.CryptoLegacyStorage storage cls,
ITrustedGuardiansPlugin.PluginStorage storage _pluginStorage, bytes32 _hash)
internal returns(bool isVoted) {
        bool isInitialized = _pluginStorage.guardians.length() != 0;
        uint256 i = 0;
        while(i < _pluginStorage.guardiansVoted.length) {
            bytes32 voted = _pluginStorage.guardiansVoted[i];
            if (!isInitialized && !cls.beneficiaries.contains(voted)) {
 // Remove vote if it's from a non-beneficiary AND guardians not yet
initialized
 ...
 _pluginStorage.guardiansVoted.pop();
 continue;
 }
            if (voted == _hash) {
                isVoted = true;
            }
            i++;
        }
        return isVoted;
    }
```

Remediation:

Consider removing beneficiary votes when initializing guardians.

5.6. Anyone can lock LifetimeNft for user

Risk Level: Medium

Status: Fixed

Contracts:

- contracts/CryptoLegacyBuildManager.sol

Description:

isLifetimeNftLockedAndUpdate function allows any authorized lock operator to unintentionally or maliciously extend the lock period of a user's Lifetime NFT, effectively preventing the NFT holder from withdrawing or transferring it even after the original lock period has expired.

```
function isLifetimeNftLockedAndUpdate(address _owner) public returns(bool) {
return ILockChainGate(address(feeRegistry)).isNftLockedAndUpdate(_owner); //
@audit-issue public function
}
```

Remediation:

Consider introducing access control check to restrict updates of the lock timer.

5.7. deployByCreate2 could return unexpected address with salt = 0

Risk Level: Medium

Status: Fixed in the commit.

Contracts:

- cryptolegacy-contracts/contracts/libraries/LibCreate2Deploy.sol

Description:

The _deployByCreate2 function always returns a random contract address when the salt is equal to 0.

```
File: cryptolegacy-contracts/contracts/libraries/LibCreate2Deploy.sol
function _deployByCreate2(
address _contractAddress,
bytes32 _factorySalt,
bytes memory _contractBytecode
) internal returns (address) {
if (_contractBytecode.length == 0) {
revert BytecodeEmpty();
}

// Compute the expected contract address
bytes32 bytecodeHash = keccak256(_contractBytecode);
address predictedAddress = _computeAddress(_factorySalt, bytecodeHash);

// Check if the contract already exists at the predicted address
uint256 size;
assembly {
size := extcodesize(predictedAddress)
}
if (size != 0) {
revert AlreadyExists();
}

// Ensure the computed address matches the expected deployed contract address
if (_contractAddress != address(0) && predictedAddress != _contractAddress) {
revert AddressMismatch();
}

// Store bytecode length for gas efficiency
uint256 bytecodeLength = _contractBytecode.length;

bytes32 salt;
if (_factorySalt == 0) {
salt = blockhash(block.number - 1);
} else {
salt = _factorySalt;
}

address addr;
assembly {
// CREATE2 deploys a contract with deterministic address
addr := create2(0, add(_contractBytecode, 0x20), bytecodeLength, salt)
}
if (addr == address(0)) {
revert Create2Failed();
}
emit CryptoLegacyCreation(addr, salt);
return addr;
```

```
62: }
```

This behavior is acceptable when the _contractAddress parameter is also zero (indicating no specific target address is expected). However, when a non-zero _contractAddress is specified along with a zero salt value, the salt used for address checks differs from the actual salt used in the deployment. This inconsistency can lead to deployment failures or interaction errors in scenarios where the deployed contract address is assumed to be deterministically derived using CREATE2.

Test for test/CryptoLegacyTest.t.sol:

```
  function testBuildWithZeroSaltAndPredictedAddress() public {
    vm.prank(owner);
    pluginsRegistry.addPlugin(lensPlugin, "");
          // salt = blockhash(block.number - 1) will underflow with
block.number == 0 in LibCryptoLegacyDeploy.sol
    vm.roll(1);
     // Define a simple beneficiary array with one beneficiary
    bytes32[] memory beneficiaryArr = new bytes32[](1);
    beneficiaryArr[0] = keccak256(abi.encode(bobBeneficiary1));
    // Define the configuration for the beneficiary (no claim delay, no
vesting, 100% share)
    ICryptoLegacy.BeneficiaryConfig[] memory beneficiaryConfigArr = new
ICryptoLegacy.BeneficiaryConfig[](1);
    beneficiaryConfigArr[0] = ICryptoLegacy.BeneficiaryConfig(0, 0, 10000);
     // Prepare the arguments for building the CryptoLegacy contract
    ICryptoLegacyBuildManager.BuildArgs memory buildArgs =
ICryptoLegacyBuildManager.BuildArgs(
      bytes8(0),
      beneficiaryArr,
      beneficiaryConfigArr,
      _getBasePlugins(),
      updateInterval,
      challengeTimeout
    );
    // Predict the address where the contract will be deployed using create2
with zero salt
    // The owner for the address computation is `bob`
    address predictedAddress = factory.computeAddress(bytes32(0), bob);
     // Switch the context to `bob` who will be the owner of the
CryptoLegacy contract
    vm.prank(bob);
    // Call the buildCryptoLegacy function with the pre-calculated
`predictedAddress` and zero salt
    address payable deployedClAddress =
buildManager.buildCryptoLegacy{value: buildFee}(
      buildArgs,
      _getRefArgsStruct(bob),
      _getCreate2ArgsStruct(predictedAddress, bytes32(0))
    );
     // Assert that the actual deployed address matches the predicted
address
    assertEq(deployedClAddress, predictedAddress, "Deployed contract address
should match the predicted address");
  }
```

Remediation:

The function should check that both _contractAddress and _factorySalt are equal to 0 when returning the random address.

5.8. Use of deprecated transfer and send functions

Risk Level: Medium

Status: Fixed

Contracts:

- contracts/CryptoLegacyBuildManager.solcontracts/FeeRegistry.solcontracts/libraries/LibCryptoLegacy.solcontracts/plugins/CryptoLegacyBasePlugin.sol

Description:

The contract uses .send() and .transfer() functions which forward a fixed amount of 2300 gas. This can cause transactions to fail if the recipient is a contract with a fallback function that requires more gas. Future Ethereum upgrades may change gas costs, making these functions unreliable.

Example:

```
// CryptoLegacyBuildManager.sol
_recipient.transfer(_amount);
 // FeeRegistry.sol
bool isTransferSuccess = payable(shareRecipient).send(share);
payable(fs.feeBeneficiaries[i].recipient).transfer(feeShare);
payable(ref.recipient).transfer(feeToSend);
 // LibCryptoLegacy.sol
payable(_ref).transfer(refValue);
payable(_buildManagerAddress).transfer(value);
 // CryptoLegacyBasePlugin.sol
payable(address(cls.buildManager)).transfer(msg.value);
```

Remediation:

Replace .send() and .transfer() with .call() and implement the checks-effects-interactions pattern to prevent reentrancy attacks. Add proper checks for the success of the call.

5.9. tokenPrepareToDistribute can be simplified

Risk Level: Low

Status: Fixed in the commit.

Contracts:

- contracts/libraries/LibCryptoLegacy.sol

Description:

The _tokenPrepareToDistribute() function can be simplified. Current code has two same ways to calculate the new amountToDistribute value if the balance value was changed:

```
    /**
     * @notice Prepares the token distribution by adjusting the
distributable amount based on the current token balance.
     * @dev Retrieves the TokenDistribution struct for a given token.
     * - If no distribution amount is set (amountToDistribute equals zero),
it is set to the contract's current balance of the token.
     * - Otherwise, if the current balance exceeds the undistributed amount
(amountToDistribute minus totalClaimedAmount),
     * the excess is added to amountToDistribute.
     * - If the balance is lower, the distribution amount is adjusted to be
the sum of the token balance and the already claimed amount.
     * @param cls The CryptoLegacy storage structure.
     * @param _token The address of the token to be prepared for
distribution.
     * @return td The updated TokenDistribution storage reference for the
token.
     */
    function _tokenPrepareToDistribute(ICryptoLegacy.CryptoLegacyStorage
storage cls, address _token) internal
returns(ICryptoLegacy.TokenDistribution storage td) {
        td = cls.tokenDistribution[_token];
         uint256 bal = IERC20(_token).balanceOf(address(this));
        if (td.amountToDistribute == 0) {
            td.amountToDistribute = bal;
        } else if (bal > td.amountToDistribute - td.totalClaimedAmount) {
            td.amountToDistribute += bal - (td.amountToDistribute -
td.totalClaimedAmount);
        } else if (bal < td.amountToDistribute - td.totalClaimedAmount) {
            td.amountToDistribute = bal + td.totalClaimedAmount;
        }
    }
```

Remediation:

Consider refactoring the code:

```
- if (td.amountToDistribute == 0) {
- td.amountToDistribute = bal;
- } else if (bal > td.amountToDistribute - td.totalClaimedAmount) {
- td.amountToDistribute += bal - (td.amountToDistribute -
td.totalClaimedAmount);
- } else if (bal < td.amountToDistribute - td.totalClaimedAmount) {
- td.amountToDistribute = bal + td.totalClaimedAmount;
- }
+ if (td.amountToDistribute == 0) {
+ td.amountToDistribute = bal;
+ } else if (bal != td.amountToDistribute - td.totalClaimedAmount) {
+ td.amountToDistribute = bal + td.totalClaimedAmount;
+ }
```

5.10. Incorrect event emission

Risk Level: Low

Status: Fixed in the commit 8fc6ec

Contracts:

- contracts/LockChainGate.sol

Description:

The unlockLifetimeNft() function emits an incorrect event when msg.sender is not the owner of the NFT, but the approved address.

```
File: cryptolegacy-contracts/contracts/LockChainGate.sol
302: emit UnlockNft(ls.lockedNft[msg.sender].lockedAt, _tokenId, holder,
msg.sender);
```

Remediation:

Consider fixing the event emission:

```
emit UnlockNft(ls.lockedNft[holder].lockedAt, _tokenId, holder, msg.sender);
```

5.11. DoS in takeFee

Risk Level: Low

Status: Fixed in the commit 996cfa

Contracts:

- contracts/FeeRegistry.sol

Description:

The takeFee() function is susceptible to a DoS attack when the sum of the discount percentage and the share percentage exceeds 10000. This situation can occur because:

- the _calculateFee() function does not check if the sum of the discount percentage and the share percentage exceeds 10000
- the _calculateFee() function calculates the share without subtracting the discount from the fee.

```
File: cryptolegacy-contracts/contracts/FeeRegistry.sol
554: function _calculateFee(FRStorage storage fs, bytes8 _code, uint256
_fee) internal view returns(uint256 discount, uint256 share, uint256 fee) {
555: (uint32 discountPct, uint32 sharePct) = _getCodePct(fs, _code);
556: discount = (_fee * uint256(discountPct)) / uint256(PCT_BASE);
557: share = (_fee * uint256(sharePct)) / uint256(PCT_BASE);
558: fee = _fee - discount;
559: }
```

In a result function takeFee() will revert because of the underflow on line 180, leading to a DoS of the contract.

```
File: cryptolegacy-contracts/contracts/FeeRegistry.sol
167: function takeFee(address _contract, uint8 _case, bytes8 _code, uint256
_mul) external payable {
168: FRStorage storage fs = lockFeeRegistryStorage();
169: uint256 contractCaseFee =
uint256(fs.feeByContractCase[_contract][_case]) * _mul;
170: (uint256 discount, uint256 share, uint256 fee) = _calculateFee(fs,
_code, contractCaseFee);
171: _checkFee(fee);
172: address shareRecipient = fs.refererByCode[_code].recipient;
173: bool isTransferSuccess = payable(shareRecipient).send(share);
174: if (isTransferSuccess) {
175: emit SentFee(fs.refererByCode[_code].owner, _code, shareRecipient,
share);
176: } else {
177: fs.refererByCode[_code].accumulatedFee += uint128(share);
178: emit AccumulateFee(fs.refererByCode[_code].owner, _code,
shareRecipient, share);
179: }
180: fs.accumulatedFee += uint128(fee - share); // underflow when
discountPct + sharePct > 10000
181: emit TakeFee(_contract, _case, _code, discount, share, fee, msg.value);
182: }
```

Remediation:

Consider adding a check to ensure that the sum of the discount percentage and the share percentage does not exceed 10000.

5.12. Lack of the beneficiary registry update in renounceOwnership()

Risk Level: **Low**

Status: Fixed

Contracts:

- cryptolegacy-contracts/contracts/plugins/CryptoLegacyBasePlugin.sol

Description:

The renounceOwnership() in the CryptoLegacyBasePlugin contract doesn't update the owner in the beneficiary registry.

Remediation:

Call the _updateOwnerInBeneficiaryRegistry() function in the same way as it is implemented in the transferOwnership() function.

5.13. Lack of destination chain check in the referral codes functionality

Risk Level: **Low**

Status: Fixed in the commit.

Contracts:

- cryptolegacy-contracts/contracts/FeeRegistry.sol

Description:

The function setCrossChainsRef() doesn't check whether the destination chain is supported or not. If an attempt to set a referral code on unsupported chain would be made, the transaction will be reverted and additional fee will be charged.

Remediation:

Check that the corresponding contract is defined in the destinationChainContracts mapping during cross-chain calls.

5.14. Some of the contract functions can be called even if the initial fee wasn't paid

Risk Level: **Low**

Status: Fixed in the commit.

Contracts:

- contracts/plugins/TrustedGuardiansPlugin.solcontracts/plugins/LegacyRecoveryPlugin.sol

Description:

The CryptoLegacy contract is designed to remain locked (paused) until the initial activation fee is paid. This restriction is enforced via the onlyOwner modifier, which blocks access to most functions until activation. However, several functions in the TrustedGuardiansPlugin and LegacyRecoveryPlugin contracts lack both the onlyOwner modifier and pause state checks. As a result, they can be called even before the initial fee is paid — for example: guardiansVoteForDistribution(), guardiansTransferTreasuryTokensToLegacy(), and resetGuardianVoting().

Remediation:

Make sure that no functions in the CryptoLegacy contract can be called until the initial fee is paid.

5.15. Valid proposal can be not executed

Risk Level: Low

Status: Fixed in the commit.

Contracts:

- contracts/plugins/LegacyRecoveryPlugin.sol

Description:

If the contract owner lowers the requiredConfirmations threshold after voters have already confirmed a proposal, that proposal may become unexecutable, because users who have already confirmed it cannot reconfirm, and their existing confirmations do not re-trigger execution.

```
function _confirm(ISafeMinimalMultisig.Storage storage s, bytes32[] memory
_allVoters, uint256 _proposalId) internal {
    bytes32 voter = LibCryptoLegacy._addressToHash(msg.sender);
    ISafeMinimalMultisig.Proposal storage p = s.proposals[_proposalId];
    if (p.executed) {
        revert ISafeMinimalMultisig.MultisigAlreadyExecuted();
    }
    if (s.confirmedBy[_proposalId][voter]) { // @audit-issue redundant
check, blocks reconfirmation
        revert ISafeMinimalMultisig.MultisigAlreadyConfirmed();
    }
    _checkIsVoterAllowed(_allVoters, voter);
     s.confirmedBy[_proposalId][voter] = true;
    p.confirms = _getConfirmedCount(s, _allVoters, _proposalId);
     emit
ISafeMinimalMultisig.ConfirmSafeMinimalMultisigProposal(_proposalId, voter,
s.requiredConfirmations, p.confirms);
     if (p.confirms >= s.requiredConfirmations) {
        _execute(s, voter, _proposalId);
    }
}
```

There is also no mechanism for canceling votes in the contract.

Remediation:

Consider removing redundant check.

5.16. Fees bypass using locked NFT transfer

Risk Level: Low

Status: Fixed in the commit 9a865b

Contracts:

- contracts/LockChainGate.sol

Description:

Single NFT could be used to bypass the fees from building and updating multiple cryptology contracts by just transferring the locked NFT to the new address and calling buildCryptoLegacy().

Remediation:

Consider adding a check to ensure that locked NFT could not be used for multiple cryptology contracts.

5.17. Missing reentrancy protection in fee handling functions

Risk Level: Low

Status: Fixed

Contracts:

- contracts/FeeRegistry.sol

Description:

The contract's fee handling functions do not follow the checks-effects-interactions pattern and lack reentrancy protection. The functions modify state after making external calls.

```
function takeFee(address _contract, uint8 _case, bytes8 _code, uint256 _mul)
external payable {
    FRStorage storage fs = lockFeeRegistryStorage();
    // ... calculations ...
    bool isTransferSuccess = payable(shareRecipient).send(share); //
External call before state changes
    if (isTransferSuccess) {
        emit SentFee(fs.refererByCode[_code].owner, _code, shareRecipient,
share);
    } else {
        fs.refererByCode[_code].accumulatedFee += uint128(share); // State
change after external call
    }
    fs.accumulatedFee += uint128(fee - share); // State change after
external call
}
 function withdrawAccumulatedFee() external {
    FRStorage storage fs = lockFeeRegistryStorage();
    for (uint256 i = 0; i < len; i++) {
        uint256 feeShare = fs.accumulatedFee *
fs.feeBeneficiaries[i].sharePct / PCT_BASE;
        payable(fs.feeBeneficiaries[i].recipient).transfer(feeShare); //
External calls before state reset
    }
    fs.accumulatedFee = 0; // State change after external calls
}
```

Remediation:

Add nonReentrant modifier and implement checks-effects-interactions pattern by updating state variables before making external calls. Consider using pull-over-push pattern for fee withdrawals.

5.18. Unsafe ERC721 minting operation

Risk Level: **Low**

Status: Fixed

Contracts:

- contracts/LifetimeNft.solcontracts/mocks/MockLifetimeNft.sol

Description:

The contract uses _mint() instead of _safeMint() for ERC721 token minting. The _mint() function does not verify if the recipient is capable of receiving NFTs, which could lead to tokens being locked if sent to a contract that doesn't support ERC721 tokens.

Example:

```
// LifetimeNft.sol
_mint(_tokenOwner, tokenId);
```

Remediation:

Replace _mint() with _safeMint() to ensure the recipient can handle ERC721 tokens properly.

5.19. Hardcoded gas limits

Risk Level: **Low**

Status: Fixed in the commits 5a6a6a, de78d3 and 0d8cd9.

Contracts:

- contracts/CryptoLegacyDiamondBase.solcontracts/libraries/LibCryptoLegacy.sol

Description:

The contract uses hardcoded gas limits in try-catch blocks which can cause function execution failures if the required gas exceeds the limit. Most try-catch blocks silently fail, masking potential errors.

Example:

```
contracts/libraries/LibCryptoLegacy.sol: try
cls.buildManager.isLifetimeNftLockedAndUpdate{gas: 6e5}(_owner) returns
(bool _isNftLocked) {
contracts/libraries/LibCryptoLegacy.sol: try
cls.buildManager.getUpdateFee{gas: 6e5}(cls.invitedByRefCode) returns
(uint256 fee) {
contracts/libraries/LibCryptoLegacy.sol: try
cls.buildManager.beneficiaryRegistry{gas: 2e5}() returns
(IBeneficiaryRegistry _br) {
contracts/libraries/LibCryptoLegacy.sol: try br.setCryptoLegacyOwner{gas:
4e5}(_hash, _isAdd) {} catch {}
contracts/libraries/LibCryptoLegacy.sol: try
br.setCryptoLegacyBeneficiary{gas: 4e5}(_hash, _isAdd) {} catch {}
contracts/libraries/LibCryptoLegacy.sol: try br.setCryptoLegacyGuardian{gas:
4e5}(_hash, _isAdd) {} catch {}
contracts/libraries/LibCryptoLegacy.sol: try
cls.buildManager.beneficiaryRegistry{gas: 2e5}() returns
(IBeneficiaryRegistry _br) {
contracts/libraries/LibCryptoLegacy.sol: try
br.setCryptoLegacyRecoveryAddresses{gas: gasLimit}(_oldHashes, _newHashes)
{} catch {}
```

Remediation:

Remove hardcoded gas limits and implement proper error handling in try-catch blocks. If gas limits are necessary, make them configurable parameters.

5.20. Excess msg.value not refunded to users

Risk Level: **Low**

Status: Fixed in the commit.

Contracts:

- contracts/CryptoLegacyBuildManager.sol

Description:

The contract accepts msg.value from users but does not refund excess amounts when the required fee is less than the sent value. This can lead to users losing funds if they accidentally send too much ETH or if fee amounts change between transaction submission and execution.

Remediation:

Calculate the exact fee required and return excess msg.value to the sender.

5.21. Wrong comment

Risk Level: Info

Status: Fixed in the commit.

Contracts:

- contracts/CryptoLegacyBuildManager.sol

Description:

The _payFee() has a _feeCase param used to calculate the feeToTake amount. Description of the _feeCase param says that it can be REGISTRY_UPDATE_CASE or REGISTRY_LIFETIME_CASE:

```
contracts/CryptoLegacyBuildManager.sol:
  163: * @param _feeCase The fee case (REGISTRY_UPDATE_CASE or
REGISTRY_LIFETIME_CASE).
```

But it doesn't mention that the function may be called with REGISTRY_BUILD_CASE.

Remediation:

Consider changing the description to mention REGISTRY_BUILD_CASE.

5.22. Double event emitting

Risk Level: Info

Status: Fixed in the commit 7a45b4

Contracts:

- cryptolegacy-contracts/contracts/CryptoLegacyOwnable.sol

Description:

The _transferOwnership() function emits the OwnershipTransferred event twice.

First emit in the _transferOwnership() function:

```
File: cryptolegacy-contracts/contracts/CryptoLegacyOwnable.sol
function _transferOwnership(address _owner) internal virtual {
address oldOwner = LibDiamond.contractOwner();
LibDiamond.setContractOwner(_owner);
emit OwnershipTransferred(oldOwner, _owner);
}
```

Second emit in the setContractOwner() function:

```
File: cryptolegacy-contracts/contracts/libraries/LibDiamond.sol
function setContractOwner(address _newOwner) internal {
DiamondStorage storage ds = diamondStorage();
address previousOwner = ds.contractOwner;
ds.contractOwner = _newOwner;
emit OwnershipTransferred(previousOwner, _newOwner);
}
```

Remediation:

Consider removing one of the events.

5.23. Gas optimization in SignatureRole

Risk Level: Info

Status: Fixed in the commit b0f0cc

Contracts:

- /cryptolegacy-contracts/contracts/SignatureRoleTimelock.sol

Description:

The functions _getAddressIndex() and _getBytes4Index() are used to get the index of an address or a bytes4 value in an array. They could do it more efficiently in terms of gas costs by breaking the loop when the index is found.

```
File: cryptolegacy-contracts/contracts/SignatureRoleTimelock.sol
function _getAddressIndex(address[] memory _list, address _addr) internal pure returns(uint index){
index = type(uint256).max;
for (uint256 i = 0; i < _list.length; i++) {
if (_list[i] == _addr) {
index = i;
}
}
}

/**
* @notice Internal helper that returns the index of a bytes4 value in an array.
* @dev Returns type(uint256).max if the value is not found.
* @param _list The array of bytes4 values.
* @param _hash The bytes4 value to search for.
* @return index The index of _hash in _list.
*/
function _getBytes4Index(bytes4[] memory _list, bytes4 _hash) internal pure returns(uint index){
index = type(uint256).max;
for (uint256 i = 0; i < _list.length; i++) {
if (_list[i] == _hash) {
index = i;
}
}
}
```

Remediation:

Consider adding a break statement to the loops to avoid unnecessary iterations.

5.24. Beneficiaries can set their own referral address and share during the claim

Risk Level: Info

Status: Referral data is intentionally not stored in the contract for beneficiary claims to ensure fault tolerance. The referral logic is handled off-chain via the frontend, which pre-fills donation amounts and referrer info when a user initiates a claim. This ensures the process remains functional even if the FeeRegistry changes or becomes unavailable.For owners, referral codes are embedded during contract creation and tied to the FeeRegistry prior to distribution. For beneficiaries, direct contract interaction is unlikely, as it requires knowledge of the contract address and manual tooling. Using the web interface is significantly easier and more accessible, especially for non-technical users.Preserving claim availability in inheritance scenarios takes precedence over enforcing on-chain referral validation.

Contracts:

- contracts/plugins/CryptoLegacyBasePlugin.sol

Description:

During the claim, beneficiaries can specify their own address as the referral and set _refShare to 100% to pay fee but receive it back.

Remediation:

Define the referral address and share directly in the CryptoLegacy contract and enforce their use during beneficiary claims.

5.25. Wrong supported chains configuration in the deployment script

Risk Level: **Info**

Status: Fixed in the commit.

Contracts:

- script/LibDeploy.sol

Description:

In the deployment script, the if statement within the _setFeeRegistryCrossChains function incorrectly calls _getNftMainnetId() twice instead of using _getRefMainnetId() for the second comparison:

```
function _setFeeRegistryCrossChains(FeeRegistry _feeRegistry) internal {
    ...
     if (block.chainid == _getNftMainnetId() || block.chainid ==
_getNftMainnetId()) { // @audit _getNftMainnetId => _getRefMainnetId
        ...
        crossChainIds[0] = block.chainid == _getNftMainnetId() ?
_getRefMainnetId() : _getNftMainnetId();
        crossChainIds[1] = 59144;
        crossChainIds[2] = 8453;
        crossChainIds[3] = 10;
    } else {
        ...
    }
}
```

Remediation:

Replace the second _getNftMainnetId() call with _getRefMainnetId()

5.26. Lifetime NFTs takeover in case of incorrect deployement configuration

Risk Level: Info

Status: Fixed in the commit.

Contracts:

- script/CryptoLegacyFactory.s.sol

Description:

By design, a lifetime NFT can be minted on the Ethereum chain only. However, currently there is only one deployment script CryptoLegacyFactory.s.sol , which deploys all contracts including LifetimeNFT on the chain, where the script is executed.

If the LifetimeNFT contract will be defined not only on Ethereum but on other chains also, it will be possible to mint NFTs with inconsistent tokenIDs on different chains. In addition, the attacker may mint the NFT with same tokenId as the victim on another chain and call the cross-chain token owner transfer that will transfer the actual NFT from the victim to the attacker on all chains.

Remediation:

Ensure that the lifetime NFT is deployed on the Ethereum mainnet only and cannot be minted on other chains.

5.27. updateInterval and challengeTimeout can be constants

Risk Level: Info

Status: We intentionally preserve _checkBuildArgs() to support forward compatibility with future build manager contracts. Although the current implementation uses fixed values for updateInterval and challengeTimeout, accepting these parameters explicitly allows seamless integration with alternative configurations that may arise as the system evolves.This design maintains clear separation of concerns: the factory logic remains decoupled from build-time constants, and the frontend can continue operating uniformly across different contract versions. The minimal overhead introduced by this check is justified by the long-term flexibility it provides without requiring protocol-level refactoring.

Contracts:

- contracts/CryptoLegacyBuildManager.sol

Description:

The _checkBuildArgs() function checks that updateInterval equals to 180 days and challengeTimeout equals to 90 days. These values can be constant and the function, as well as the arguments, can be removed.

Remediation:

Remove redundant checks and arguments.

6. Appendix

6.1. About us

The Decurity team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.