Competitive Security Assessment

XCarnival - UPenn CIS 7000

Nov 28th, 2022

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

Project Detail

Project Name	XCarnival - UPenn CIS 7000
Platform & Language	Solidity
Codebase	• https://github.com/Secure3Audit/XCarnival_UPenn_CIS7000 • audit commit - 983fb4ec6d12c8e3e6453d2f4e06586a6cd340ed • final commit - 6a7ffdb107c30ea7e54ce83afa9976f6a86a3286
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

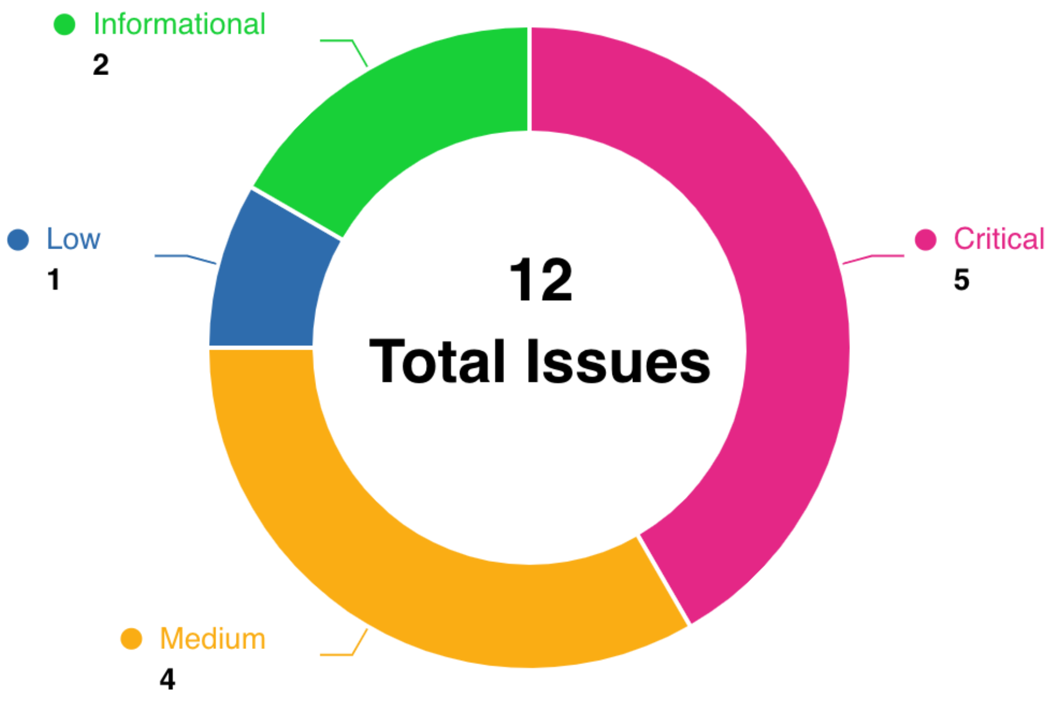
Code Vulnerability Review Summary

Vulnerability Level	Total	Reported	Acknowledged	Fixed	Mitigated	Declined
Critical	5	0	0	5	0	0
Medium	4	0	0	4	0	0
Low	1	0	0	1	0	0
Informational	2	0	0	2	0	0

Audit Scope

File	Commit Hash
code/contracts/XCVToken.sol	983fb4ec6d12c8e3e6453d2f4e06586a6cd340ed
code/contracts/XController.sol	983fb4ec6d12c8e3e6453d2f4e06586a6cd340ed
code/contracts/XNFT.sol	983fb4ec6d12c8e3e6453d2f4e06586a6cd340ed

Code Assessment Findings



ID	Name	Category	Severity	Status	Contributor
CIS-1	Attackers can "sell" the same NFT multiple times	Logical	Critical	Fixed	david-cao, lohpaul9, seanwjcho, Secure3
CIS-2	Integer overflow risk in `XController::mint`	Integer Overflow and Underflow	Low	Fixed	david-cao, lohpaul9, seanwjcho, taran317, Secure3
CIS-3	Literals with too many digits	Code Style	Informational	Fixed	seanwjcho

CIS-4	NFT locking can be bypassed to be transferred	Logical	Medium	Fixed	Secure3
CIS-5	Password paradigm is insecure in `XController` contract	Logical	Critical	Fixed	david-cao, lohpaul9, seanwjcho, taran317, Secure3
CIS-6	`XController::confirmDistribute` Reentrancy risk	Reentrancy	Medium	Fixed	david-cao, lohpaul9, taran317, Secure3
CIS-7	`XController::confirmDistribute` function can be stuck by malicious contracts	DOS	Critical	Fixed	david-cao, lohpaul9, seanwjcho, Secure3
CIS-8	`XController::getRandomNumber` the random number can be predicted	Weak Sources of Randomness	Medium	Fixed	david-cao, lohpaul9, seanwjcho, taran317, Secure3
CIS-9	`XController::mint` Incorrect `msg.value` check	Logical	Critical	Fixed	lohpaul9, seanwjcho, taran317, Secure3
CIS-10	`XController` locks Ether	Logical	Medium	Fixed	david-cao, Secure3
CIS-11	`XNFT::lock` Missing permission control check	Logical	Critical	Fixed	david-cao, lohpaul9, seanwjcho, taran317, Secure3
CIS-12	`XVCToken` has no apparent utility	Logical	Informational	Fixed	david-cao

CIS-1:Attackers can "sell" the same NFT multiple times

Category	Severity	Code Reference	Status	Contributor
Logical	Critical	• code/contracts/XController.sol#L103-L110	Fixed	david-cao, lohpaul9, seanwjcho, Secure3

Code

```
103:    function swapNFTsforETH(uint256[] calldata _ids) external {
104:        for (uint256 i = 0; i < _ids.length; ++i) {
105:            uint256 id = _ids[i];
106:            require(msg.sender == xnft.ownerOf(id));
107:            xnft.lock(id);
108:        }
109:        toDistribute.push(Distribute(msg.sender, _ids));
110:    }
```

Description

david-cao : A user is able to sell an NFT multiple times. Specifically, consider the scenario where an attacker owns an NFT already and they call `swapNFTsforETH` multiple times. Since no accounting is done on the NFT, multiple entries for the same address and NFT id will exist in the `toDistribute` array. Thus, when `confirmDistribute` is called, the contract send `x*payout` to the attacker, where `x` is the number of times they called `swapNFTsforETH`.
However, since an attacker can easily determine what the password is (see password issue), they can effectively drain the contract in a single transaction by constructing a malicious contract similar to below:

```
contract Attack {
    ...
    // x calculated beforehand
    function attack(uint256 x) {
        for(uint256 i = 0; i < x; i++) {
            controller.swapNFTsforETH([nftID]);
        }

        // password determined by looking at deployment bytecode
        controller.confirmDistribute(password);
    }
}
```

The attacker of course can always do this in multiple transactions without a contract as well.

lohpaul9 : In the `swapNFTforETH()` function, a malicious exteral party can pass in an **array of repeated token ids which they own**.

Suppose the malicious party owns token id 1. The malicious array can be [1,1,1,1]. In each of the iterations of the for loop, token 1 will be checked for ownership, then locked (of couse the locked token can be relocked). Lastly, the entire array is pushed to `Distribute[]`.

Once in `Distribute[]`, `confirmDistribute()` when called will transfer the appropriate price to each of the token in the array, in this case [1,1,1,1]. Note that the malicious party essentially gets to receive the price of 4 tokens rather than 1. This can potentially drain the smart contract with enough repeated token ids.

Consider below functions in XController.sol

```
/// @notice Swap NFTs for ETH
    ///
    /// @dev data will be stored in the `toDistribute`
    /// ETH will be distributed through the confirmDistribute function
    ///
    /// Requirements:
    /// - Check NFT id ownership
    /// - Swapped NFTs will be locked and cannot be transferred
    ///
    /// @param _ids The array of NFT ID the msg.sender would sell
    function swapNFTsforETH(uint256[] calldata _ids) external {
        for (uint256 i = 0; i < _ids.length; ++i) {
            uint256 id = _ids[i];
            require(msg.sender == xnft.ownerOf(id));
            xnft.lock(id);
        }
        toDistribute.push(Distribute(msg.sender, _ids));
    }

    /// @notice Admin confirms distribution of ETH
    ///
    /// @dev Only admins who know the password can confirm the swap and distribute ETH
    ///
    /// @param _password Throws if called without correct password
    function confirmDistribute(string calldata _password)
        external
        requirePassword(_password)
    {
        for(uint256 i = lastDistribute; i < toDistribute.length; ++i) {
            Distribute memory dis = toDistribute[i];
            uint256 sell_total = dis.ids.length * NFT_PRICE_ETH * (BASE - SELL_FEE) / BASE;
            (bool success, ) = dis.to.call{value: sell_total}("");
            require(success, "Distribute Failed");
        }

        lastDistribute = toDistribute.length;
    }
```

seanwjcho : The contract doesn't check if an NFT is already locked when you swap it for eth, so you can swap one multiple times. The way confirmDistribute is set up, once these swaps are queued they have to be processed before other valid swaps.

Secure3 : User-controlled parameter _ids is not validated. The `swapNFTsforETH` function does not check whether an NFT is already locked, resulting in an NFT that can be sold multiple times for additional ETH.

Consider below POC contract

```
contract Attack {
    function hack() external {
        uint256[] memory ids = new uint256[](3);
        ids[0] = 1;
        ids[1] = 1;
        ids[2] = 1;
        controller.swapNFTsforETH(ids);
    }
    ...
}
```

Recommendation

david-cao : Our recommendation is to remove the `confirmDistribute` functionality altogether. See issue #3 more information.

However, if the intent is still to have a `confirmDistribute` method, a few changes are required.

1. Add an additional require in `swapNFTsforETH` to represent that an NFT is held for a sell.

```
require(!xnft.locked(id));
```

2. Add the ability to unlock NFTs, which is only allowed by the controller.
3. In `confirmDistribute`, unlock the nft and either burn the NFT or transfer it to the controller (this requires an approve transaction to be done by the user).

lohpaul9 : There are a few ways to fix this. We want to guarantee the uniqueness of the tokens that are locked and pending distribution.

We can do this by adding a mapping in the function where repeated entry is overwritten.

However, the easiest way to do this is to revert any repeated locking of the same token id. This can be done by adding a `require()` in the `lock()` function in `XNFT.sol` to check if the token is already locked.

Reference the fix for Issue 10

seanwjcho : check if an not is already locked in swapNFTsforETH

Consider below fix in the function

```
    function swapNFTsforETH(uint256[] calldata _ids) external {
          for (uint256 i = 0; i < _ids.length; ++i) {
              uint256 id = _ids[i];
              require(msg.sender == xnft.ownerOf(id));
              require(!xnft.locked()[id]);
              xnft.lock(id);
          }
          toDistribute.push(Distribute(msg.sender, _ids));
      }
```

Alternatively perform this check in the xnft contract's lock function

```
  function lock(uint256 tokenId) external {
          require(!locked[tokenId]);
          locked[tokenId] = true;
      }
```

Secure3 : Check if NFTs are locked in the for loop. Consider below fix in the for loop:

```
   require(!xnft.locked(id), "already locked");
```

Client Response

Fixed. Adding a check on whether NFT is locked.

CIS-2:Integer overflow risk in `XController::mint`

Category	Severity	Code Reference	Status	Contributor
Integer Overflow and Underflow	Low	• code/contracts/XController.sol#L80-L82	Fixed	david-cao, lohpaul9, seanwjcho, taran317, Secure3

Code

```
80:         unchecked {
81:             require(amount + xnft.totalSupply() <= xnft.MAX_SUPPLY());
82:         }
```

Description

david-cao : The `unchecked` block is used improperly in `mint`, which allows an integer overflow. Specifically, if `amount + xnft.totalSupply()` is larger than the max uint256, it will overflow. Thus, the impact is that someone will be able to mint more NFTs than the max supply. However, the user will still need that much ETH to pay for the mint, so the cost of the attack is almost infeasible since the minimum value required to overflow is (MAX_UINT256 - MAX_SUPPLY) * NFT_PRICE_ETH.

In general, the `unchecked` block should only be used if the developer is *certain* the code is not able to overflow.

lohpaul9 : Because the `require(amount + xnft.totalSupply() <= xnft.MAX_SUPPLY())` is unchecked in the function below, it is possible for integer overflow to happen, meaning that the require function can be bypassed with a large enough `uint256 amount` value.

The impact is that a malicious contract could mint a large amount of NFTs that is way above the limit of `xnft.MAX_SUPPLY()`.

Consider below function in XController.sol

```
function mint(uint256 amount) external payable {
    // unchecked to save gas
    unchecked {
        require(amount + xnft.totalSupply() <= xnft.MAX_SUPPLY());
    }

    require(amount > 0, "amount should gt 0");

    for (uint256 i = 0; i < amount; ++i) {
        require(msg.value > NFT_PRICE_ETH, "Insufficient ETH");
        uint256 id = xnft.mint(msg.sender);
        xnft.setRarity(id, getRandomNumber());
    }
}
```

seanwjcho : The mint function in XController.sol encompasses the require statement in an unchecked mark. This means Solidity should not worry about Integer Overflow or Underflow. However, any contract can submit their own amount variable, which could be set near the Integer Max Value which would cause integer overflow. This also means a user would be able to mint more NFTs than the total supply, potentially breaking the entire use case of this NFT contract.

The impact is that the malicious contract can call the mint function with an amount=Integer.MAX_VALUE, which would cause integer overflow when the total supply is greater than zero.

taran317 : Although this contract is written in Solidity 0.8.17 where overflow issues generally shouldn't be a problem, line 81 where addition occurs is wrapped in `unchecked` which means that this arithmetic isn't checked for underflow/overflow. `unchecked` is not appropriate here because there is a real chance `amount + xnft.totalSupply()` overflows which causes `require(amount + xnft.totalSupply() <= xnft.MAX_SUPPLY())` to pass even if the condition does not hold.

This would have serious repercussions because it would allow someone to mint more NFTs than the MAX_SUPPLY which is clearly undesired.

Secure3 : Although the solidity version used by the contract is 0.8, which means it has built-in overflow/underflow checking, the `unchecked` block still has overflow/underflow risk.

Solidity does not check overflow problem in the `unchecked` block. The attacker can pass a malicious amount parameter to make `amount + xnft.totalSupply()` is larger than the `type(uint256).max` and overflow, so it is possible that the attacker mint a large `amount` of NFT.

However, it's difficult to exploit because the attacker needs a large amount of ETH to pay for minting.

Recommendation

david-cao : Remove the `unchecked` block. If the intent is to save gas, one can instead add the block around the for loop, since it is guaranteed to not overflow. Moreover, consider checking the supplied ETH is sufficient before the for loop for code cleanliness and gas savings.

```
    function mint(uint256 amount) external payable {
        require(amount + xnft.totalSupply() <= xnft.MAX_SUPPLY());
        require(msg.value >= amount * NFT_PRICE_ETH, "Insufficient ETH");

        for (uint256 i = 0; i < amount; ++i) {
            uint256 id = xnft.mint(msg.sender);
            xnft.setRarity(id, getRandomNumber());
        }
    }
```

lohpaul9 : Remove the unchecked block.

Consider below fix.

```
require(amount + xnft.totalSupply() <= xnft.MAX_SUPPLY());
```

seanwjcho : While the use of 'unchecked' can save gas in some cases, it is not worth causing a bug in the supply of the NFT. We recommend removing lines 80 and 82 entirely and leaving line 81.

Consider below fix in the `sample.test()` function

```
// checked to protect against integer overflow
require(amount + xnft.totalSupply() <= xnft.MAX_SUPPLY());
```

taran317 : Put the require statement on line 81 outside the unchecked block (remove unchecked).
Secure3 : Remove the `unchecked` label to enable solidity built-in overflow check.

Client Response

Fixed. Added `require` to prevent integer overflow.

CIS-3:Literals with too many digits

Category	Severity	Code Reference	Status	Contributor
Code Style	**Informational**	• **code/contracts/XCVToken.sol#L10**	**Fixed**	**seanwjcho**

Code

```
10:    uint256 public constant MAX_SUPPLY = 1000000000 * 10**18;
```

Description

seanwjcho : A literal has too many digits, which can lead to human error when using/updating the contract

Recommendation

seanwjcho : Consider below fix

```
MAX_SUPPLY = (10**9) * (10 ** 18)
```

Client Response

Fixed.

CIS-4:NFT locking can be bypassed to be transferred

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	• code/contracts/XNFT.sol#L56-L63	Fixed	Secure3

Code

```
56:    function transferFrom(
57:        address from,
58:        address to,
59:        uint256 tokenId
60:    ) public override {
61:        require(!locked[tokenId], "Can't transfer locked NFT");
62:        super.transferFrom(from, to, tokenId);
63:    }
```

Description

Secure3 : `XNFT` contract overrides `ERC721::transferFrom` and adds a check to ensure that locked NFT will not be transferred. However, `XNFT` does not overrides `ERC721::transfer` and `ERC721::safeTransferFrom` to add the same restriction, so user can still transfer locked NFT by calling `XNFT::transfer` or `XNFT::safeTransferFrom`.

Recommendation

Secure3 : OpenZeppelin ERC721 has official transfer hook function `_beforeTokenTransfer`. It is recommend to override and add restriction to `_beforeTokenTransfer` function.

Consider below fix in the `XNFT` contract:

```
function _beforeTokenTransfer(address from, address to, uint256 tokenId) internal override {
    super._beforeTokenTransfer(from, to, tokenId);
    require(!locked[tokenId], "Can't transfer locked NFT");
}
```

Client Response

Fixed. Override `_beforeTokenTransfer()` function.

CIS-5:Password paradigm is insecure in `XController` contract

Category	Severity	Code Reference	Status	Contributor
Logical	Critical	• code/contracts/XController.sol#L20-L21 • code/contracts/XController.sol#L49-L53 • code/contracts/XController.sol#L60-L65 • code/contracts/XController.sol#L119-L121 • code/contracts/XController.sol#L140-L142	Fixed	david-cao, lohpaul9, seanwjcho, taran317, Secure3

Code

```solidity
    // Only people who know the password can call certain functions
    string private password;

    constructor(XCVToken _xtoken, string memory _password) {
        xtoken = _xtoken;
        xnft = new XNFT();
        password = _password;
    }

    modifier requirePassword(string calldata _password) {
        if(!equal(_password, password)) {
            revert Unauthorized();
        }
        _;
    }

     function confirmDistribute(string calldata _password)
         external
         requirePassword(_password)

     function withdrawTokens(uint256 _amount, string calldata _password)
         external
         requirePassword(_password)
```

Description

david-cao : The password provided in the constructor can be determined in plaintext and thus does not provide the functionality desired. See a test deployment of the contract on the Goerli network here. The password provided was `mysecretpassword`, which in hex is `6D7973656372657470617373776F7264`. You can clearly see this value is output at the bottom of the "Input Data" section. This effectively allows *anyone* to call any function intended for admins only, e.g. any function with the `requirePassword` modifier. Moreover, if any function that takes in a password as input is called, the password is also visible in the calldata. As an example, see this transaction to `confirmDistribute` on the Goerli testnet. You can again see the hex `6D7973656372657470617373776F7264` is clearly visible.

Specifically, when a contract is deployed, two chunks of bytecode are provided: 1) the bytecode of the contract itself and 2) the bytecode to execute the constructor. Since the password is an argument to the constructor, it is visible in the constructor bytecode. 49-L53 Have `XController` inherit from `Ownable` and use `onlyOwner` modifiers instead of the password. The private key corresponding to the owner can be maintained in a wallet. Defer to using a hardware wallet such as Ledger as it is more secure than a software wallet such as Metamask.

However, without any additional precautions, since this is a single private key, it can easily be stolen by a rogue employee. If you would like to maintain an "M out of N" multisig paradigm, consider using a service like gnosis-safe. This requires M out of N parties to sign any transaction.

lohpaul9 : The contract uses a modifier, requirePassword, to give admin access to certain functions. This modifier requires the caller to input a password, which it then checks against the stored password. However, because the stored password is passed in in the constructor, anyone can go onto a website like etherscan and look at the data passed in. They could also look at recent transactions that succeeded and look at the password called in there. This gives any contract user the ability to access admin functions.

seanwjcho : A password that is supposed to be secret is stored on chain. While it is a private variable in the contract, it is public information that anyone on chain can read - particularly validators need to know it to check if people entered the right password

The impact is that the malicious contract can impersonate an admin (e.g. withdraw all funds, approveEthDistributions, etc)

taran317 : Making the password private in XController does not mean that it cannot be found publicly on the blockchain - thus, every password required function is vulnerable in this contract

The impact is that a malicious user could readily access any restricted functions and perform actions like confirming distributions or withdrawing tokens.

Secure3 : Ethereum privides a JSON-PRC eth_getStorageAt. It returns the value from a storage position at a given contract address. So Anyone can access the Ethereum global storage to read any variable in a contract, the attacker can easily get the value of `password`, allowing the attacker to execute privileged functions.

Recommendation

david-cao : n: The password provided in the constructor can be determined in plaintext and thus does not provide the functionality desired. See a test deployment of the contract on the Goerli network here. The password provided was `mysecretpassword`, which in hex is `6D797365637265747061737373776F7264`. You can clearly see this value is output at the bottom of the "Input Data" section. This effectively allows *anyone* to call any function intended for admins only, e.g. any function with the `requirePassword` modifier. Moreover, if any function that takes in a password as input is called, the password is also visible in the calldata. As an example, see this transaction to `confirmDistribute` on the Goerli testnet. You can again see the hex `6D797365637265747061737373776F7264` is clearly visible.

Specifically, when a contract is deployed, two chunks of bytecode are provided: 1) the bytecode of the contract itself and 2) the bytecode to execute the constructor. Since the password is an argument to the constructor, it is visible in the constructor bytecode. 49-L53 Have `XController` inherit from `Ownable` and use `onlyOwner` modifiers instead of the password. The private key corresponding to the owner can be maintained in a wallet. Defer to using a hardware wallet such as Ledger as it is more secure than a software wallet such as Metamask.

However, without any additional precautions, since this is a single private key, it can easily be stolen by a rogue employee. If you would like to maintain an "M out of N" multisig paradigm, consider using a service like gnosis-safe. This requires M out of N parties to sign any transaction.

lohpaul9 : Instead of the requirePassword modifier, create a private whitelist of users allowed to access admin functions. Additionally, create an onlyOwner modifier using openzeppelin's Ownable interface to allow the contract creator to add/remove users from the whitelist. For example:

```
mapping(address => bool) private whitelistedUsers;

modifier requireWhitelist(address calldata _user) {
  require(whitelistedUsers[_user] = true);
}

function addUser(address _user) public onlyOwner {
  whitelistedUsers[_user] = true;
}

function removeUser(address _user) public onlyOwner {
  whitelistedUsers[_user] = false;
}
```

seanwjcho : import ownable and make the function onlyOwner instead. Then you can share the owner account private key among the admins

taran317 : One solution is to store a hashed password (hashed with keccak256()) and comparing the hashed input a user gives to this hashed password. However, the best method is instead to introduce a mapping of approved users and compare msg.sender to these users when verifying identity. Introduce an instance mapping from address to boolean called "approved." Then, for each privileged function, require that msg.sender is mapped to true in the mapping before executing any other code. There should also be another function introduced that can only be called by the owner of XCVToken that allows for a new address to be mapped to true - giving them "password" privilege.

Secure3 : It is recommended to use OpenZeppelin Ownable.sol contract for permission control.

Client Response

Fixed as suggested.

CIS-6: `XController::confirmDistribute` Reentrancy risk

Category	Severity	Code Reference	Status	Contributor
Reentrancy	Medium	• code/contracts/XController.sol#L119-L131	Fixed	david-cao, lohpaul9, taran317, Secure3

Code

```
119:    function confirmDistribute(string calldata _password)
120:        external
121:        requirePassword(_password)
122:    {
123:        for(uint256 i = lastDistribute; i < toDistribute.length; ++i) {
124:            Distribute memory dis = toDistribute[i];
125:            uint256 sell_total = dis.ids.length * NFT_PRICE_ETH * (BASE - SELL_FEE) / BASE;
126:            (bool success, ) = dis.to.call{value: sell_total}("");
127:            require(success, "Distribute Failed");
128:        }
129:
130:        lastDistribute = toDistribute.length;
131:    }
```

Description

david-cao : Since `lastDistribute` is updated *after* an external call is made, reentrancy is possible. Specifically, a malicious contract is able to completely drain the `XController` contract of ETH.

```
contract Attack {

    XController controller;
    uint256 entries;
    uint256 receiveCount;

    constructor() {
        controller = XController(0xcontrolleraddress);
        entries = 0;
    }

    function attack() external payable {
        controller.mint{value: msg.value}(1);
        // attacker determines nft_id just minted
        controller.swapNFTsforETH(nft_ids);
        // attacker calculates receiveCount to drain contract
    }

    receive() external payable {
        if (entries <= receiveCount) {
            entries++;
            controller.confirmDistribute(password);
        }
    }
}
```

lohpaul9 : In the confirmDistribute function, the contract sends eth to the 'to' address of each item in the toDistribute list. As such, each item in toDistribute should only perform one eth sending operation to the 'to' address. This is done using to.call()

Note that because we only alter state (by setting lastDistribute) after all the calls to to.call(), it is possible to perform a re-entrancy attack where to.call for any one user is called multiple times per item in toDistribute when it should only have been called once. This can be done in the following way:

Assuming a user knows the password string (which can be done using etherscan, as documented in other bugs, or assuming its a malicious administrator):

1. The attack contract's address is one of the items that have yet to be distributed to
2. The attack contract call confirmDistribute again in their fallback function
3. When confirmDistribute is called (either by the user or the admin), when (attack_address).call is called, the fallback function will call confirmDistribute again and because state has not yet been changed (lastDistribute is still the same), then the exact same call to send ether to the attack contract will be made, making potentially multiples calls to any one address for a single item in the toDistribute list.

In this way, an attacker would receive potentially much more eth than it should receive, potentially draining the entire contract.

taran317 : Because the lastDistribute is set to toDistribute.length (in line 130) after the call on line 126 (`(bool success, ) = dis.to.call{value: sell_total}("");`).

The impact is that the malicious contract can repeatedly make calls to `confirmDistribute` after line 126 is run but before line 130 sets lastDistribute. As a result, more ETH than intended can be distributed because at the time of each subsequent call, it is possible that lastDistribute did not change (if line 130 isn't executed by then). If lastDistribute doesn't change, the for-loop iterates for the same number of iterations each time; i.e. more ETH is distributed than is intended.

Consider below POC contract

```
contract XControllerHack {
    function reenter() external {
        confirmDistribute("password");
    }
    ...
}
```

Secure3 : `lastDistribute` is updated after external calls, hence there is a risk of reentrancy attack.

Consider below situation:

`dis.to` is a hacker-controlled contract. `XController` sends ETH and calls that malicious contract and the malicious contract calls back into `XController.confirmDistribute`. Because `lastDistribute` has not been updated, Ether distribution will be processed again. And hacker can get more Ether.

Because only the owner can execute `confirmDistribute` function(need password), hacker must be authorized before performing reentrancy attack.

Recommendation

david-cao : Our recommendation is to completely remove the `confirmDistribute` functionality and instead sell the NFT directly in `swapNFTsforETH`. See issue #3 for complete details.

Alternatively, if `confirmDistribute` is still wanted, update `lastDistribute` before making the call to transfer ETH.

lohpaul9 : 1. Instead of looping using a temp variable i, use a while loop using lastDistribute as an indexer itself. While doing this, ensure to increment lastDistribute BEFORE any ether sending operations are called, such that re-entering the contract would prevent multiple eth sending operations for a single item in the toDistribute.

taran317 : This reentrancy risk can be eliminated by moving line 130 to between the current lines 125 and 126. Then, the length will already be updated and a malicious fallback function will not be able to access previously completed array indices.

Secure3 : Use Checks-Effects-Interactions best practice(Update `lastDistribute` before external call) or use Openzeppelin `nonReentrant` modifier.

Client Response

Fixed as suggested with `nonReentrant` modifier and change the order.

CIS-7: `XController::confirmDistribute` function can be stuck by malicious contracts

Category	Severity	Code Reference	Status	Contributor
DOS	Critical	• code/contracts/XController.sol#L119-L131	Fixed	david-cao, lohpaul9, seanwjcho, Secure3

Code

```
119:    function confirmDistribute(string calldata _password)
120:        external
121:        requirePassword(_password)
122:    {
123:        for(uint256 i = lastDistribute; i < toDistribute.length; ++i) {
124:            Distribute memory dis = toDistribute[i];
125:            uint256 sell_total = dis.ids.length * NFT_PRICE_ETH * (BASE - SELL_FEE) / BASE;
126:            (bool success, ) = dis.to.call{value: sell_total}("");
127:            require(success, "Distribute Failed");
128:        }
129:
130:        lastDistribute = toDistribute.length;
131:    }
```

Description

david-cao : The `confirmDistribute` function is subject to a "griefing contract". Consider an attacker who creates a malicious contract as follows:

```
contract Attack {
    function mintAndSwap() external {
        controller.mint{value: NFT_PRICE_ETH}(1);
        // attacker gets nft id
        controller.swapNFTsforETH(nft_ids);
    }

    // Note, NO receive or fallback implemented
}
```

Note that since the contract does not implement a receive or fallback, it will error when ETH is sent to it. After `mintAndSwap()` is called, any call to `confirmDistribute` on the `XController` contract will fail. The attack is not limited to this specific contract; one could throw any sort of error when receiving ETH from the controller.

This is very similar to the AkuDreams exploit, and we recommend reading about it here or here.

Alternatively, this function provides a main centralization risk. Specifically, if the `confirmDistribute` function is kept, users are dependent on a central party to receive payouts for their sells.

lohpaul9 : In the `confirmDistribute()` function, the logic is supposed to send eth to different members that are currently on the `toDistribute` array. To do this, they send eth using `call()` and revert the entire function if a single attempt to send eth fails. The function that populates this array is `swapNFTsforETH()` which uses the `msg.sender` of the caller of the function to populate the array `toDistribute`.

The impact is that a malicious contract could put an automatic `revert()` function call in the fallback function. If the address of this contract is included in the `toDistribute` array, nobody will be able to get any eth through `confirmDistribute()` and will end up being locked, essentially.

Consider below functions in XController.sol

```
function confirmDistribute(string calldata _password)
        external
        requirePassword(_password)
{
    for(uint256 i = lastDistribute; i < toDistribute.length; ++i) {
        Distribute memory dis = toDistribute[i];
        uint256 sell_total = dis.ids.length * NFT_PRICE_ETH * (BASE - SELL_FEE) / BASE;
        (bool success, ) = dis.to.call{value: sell_total}("");
        require(success, "Distribute Failed");
    }

    lastDistribute = toDistribute.length;
}
```

```
function swapNFTsforETH(uint256[] calldata _ids) external {
    for (uint256 i = 0; i < _ids.length; ++i) {
        uint256 id = _ids[i];
        require(msg.sender == xnft.ownerOf(id));
        xnft.lock(id);
    }
    toDistribute.push(Distribute(msg.sender, _ids));
}
```

seanwjcho : An external call is made inside the the for loop to distribute ETHs.

The impact is that the malicious contract can cause the function to always revert midway through, preventing any more eth from being distributed

Consider below POC contract

```
contract Hack {
    XController XC;
    function swapHelper(uint256[] calldata _ids) external {
        XC.swapNFTsforETH(_ids);
    }

    fallback () external payable {
       assert(false);
    }
}
```

Secure3 : The function `confirmDistribute` will send ETH to `dis.to`. If `dis.to` is a contract, `XController` will try to call its `fallback()` or `receive()` function. If `dis.to` does not implements `fallback()` or `receive()`, which means it cannot receive ETH, the whole transaction will fail and revert. An attacker can also deploy a contract that implements a malicious fallback function to consume all remaining gas and make `confirmDistribute` transaction fail and revert.

Recommendation

david-cao : In combination with issues #9 and #11, the recommendation is to completely remove an admin-triggered withdrawal. There does not seem to be any good reason why the admins are required to call `confirmedDistribute` asynchronously; no ETH can be removed from the contract nor can it be added. Thus, we recommend sales to be completely user initiated. This prevents any "griefing contracts" and also removes the centralization risk. For example, the following would allow users to directly sell their NFTs for ETH.

```
    function swapNFTsforETH(uint256[] calldata _ids) external {
        for (uint256 i = 0; i < _ids.length; ++i) {
            uint256 id = _ids[i];
            require(msg.sender == xnft.ownerOf(id), "Must own NFT");
            require(!xnft.locked(id), "NFT cannot already be sold");
            xnft.lock(id);
        }

        uint256 sell_total = _ids.length * NFT_PRICE_ETH * (BASE - SELL_FEE) / BASE;
        (bool success,) = msg.sender.call{value: sell_total}("");
        require(success, "Sale failed");
    }
```

It is worth mentioning here that locking may not be the mechanism you wish to use here. If a user sells their NFT, perhaps another user would be able to buy that same NFT later. If implemented as exactly as above, once an NFT is sold, it can never be unlocked or transferred ever again. If that is not intended design, we encourage the developers to think more deeply about if the sale mechanism is required at all.

lohpaul9 : Allow users to call `confirmDistribute()` individually so that each user can process their own required funds individually. This way, anyone with a malicious fallback function will only be hurting themselves.

Consider below fix (pseudocode)

```
function confirmDistribute(address _to)
        external
{
    //turn toDistribute into a hashmap instead with key(address of sender) - value(Distribute
struct)
    //and in swapNFTsforETH, store values into a toDistribute hashmap instead of array

    Distribute memory dis = toDistribute[_to];
    uint256 sell_total = dis.ids.length * NFT_PRICE_ETH * (BASE - SELL_FEE) / BASE;
    (bool success, ) = dis.to.call{value: sell_total}("");
}
```

seanwjcho : Allow Admins to confirm distributions one at a time based on an index i.

Consider the function below

```
function confirmDistribute(string calldata _password, uint256 i)
        external
        requirePassword(_password)
    {
          Distribute memory dis = toDistribute[I];
          uint256 sell_total = dis.ids.length * NFT_PRICE_ETH * (BASE - SELL_FEE) / BASE;
          (bool success, ) = dis.to.call{value: sell_total}("");
          require(success, "Distribute Failed");
    }
```

Secure3 : Limit the gas consumption, and record when ETH sending fails instead of reverting the entire transaction. Consider below fix in the `XController::confirmDistribute()` function:

```
(bool success, ) = dis.to.call{value: sell_total, gas: gasLimit}("");
if (!success) {
    // process it instead of reverting the whole transaction
}
```

Client Response

Fixed

CIS-8: `XController::getRandomNumber` the random number can be predicted

Category	Severity	Code Reference	Status	Contributor
Weak Sources of Randomness	Medium	• code/contracts/XController.sol#L158-L164	Fixed	david-cao, lohpaul9, seanwjcho, taran317, Secure3

Code

```
158:    function getRandomNumber() internal view returns(uint8) {
159:        return uint8(uint256(keccak256(abi.encode(
160:            msg.sender,
161:            block.difficulty,
162:            gasleft()
163:        )))) % 100;
164:    }
```

Description

david-cao : It is possible for someone who knows `block.difficulty` to manipulate the result of `getRandomNumber()` to be exactly what they want. Note that anyone can fix `gasLeft()` to be exactly what they want by providing any amount of gas. The same is true for `msg.sender`, meaning the only value that is not controlled by the sender is `block.difficulty`. Post merge, this value is now `prevrandao`, which is the previous block's `randao` value. There are two scenarios in which an attacker can manipulate `getRandomNumber()`:

1. Attacker is not a miner In this case, the attacker can pre calculate values based on the last `randao` seen to determine what amount of `gasLeft()` results in a return value of 99 from `getRandomNumber()`. This will only take O(100) executions. However, it's required that the attacker's transaction is included in the next block, something that is not guaranteed but can be circumvented relatively easily by incentivizing miners.
2. Attacker is a miner This is a subset of the case above, except now the attacker deterministically is able to control the value of `getRandomNumber()`. If the miner discovers a new block, they can easily precompute the value of `gasLeft()` needed to.

lohpaul9 : The getRandomNumber() function is not actually random, as it can be predicted by skilled miners. Consider the function code copied below:

```
function getRandomNumber() internal view returns (uint8) {
    return
        uint8(
            uint256(
                keccak256(
                    abi.encode(msg.sender, block.difficulty, gasleft())
                )
            )
        ) % 100;
}
```

Anyone can use the keccak2556 hash. The sender would know the value of msg.sender, and a skilled user could manipulate the block.difficulty and calculate the gasLeft()). In other words, they can optimize the values of block.difficulty and gasleft() to produce a random number that is higher, representing a more rare NFT that has more value. This is an unfair advantage.
seanwjcho : XController has a getRandomNumber() function, which is used to set a rarity score for the NFTs. Solidity is deterministic, thus any malicious party can analyze the function and identify the possible results of the scores for certain NFTs. Thus, if they want a very rare NFT, they can look at their address, and wait for a certain block number and gas combo and mint then to get the rare NFT.
taran317 : When the given code generates randomness, it is using features of the blockchain to create this random effect (like block.timestamp). These are predictable and could thus be exploited by hackers.

The impact is that a malicious individual could manipulate the rarity of NFTs coming from the random function.
Secure3 : The result of `getRandomNumber()` function can be predicted and the attacker can calculate and manipulate the `gasLeft()` to produce a random number that is higher to get a more rare NFT.

For example, the attacker can simulate execution and debug mint function to calculate the remaining gas amount when `gasleft()` is executed. And a possible POC:

```
uint256 gasAmount = calculateGas();
uint8 foreknowledge_rarity = uint8(uint256(keccak256(abi.encode(
    address(this),
    block.difficulty,
    gasAmount
)))) % 100;
require(foreknowledge_rarity > 90);
```

Recommendation

david-cao : If rarity is a critical aspect of the service, consider using Chainlink VRF. This requires funding your contract with LINK to pay for the requests for randomness. This can be done after the contract is deployed.
lohpaul9 : Use Chainlink's Verifiable Random Function, an oracle that generates a random number for the contract. This is much more secure and fair for all users. Docs can be found at this link:

https://docs.chain.link/vrf/v2/introduction.
seanwjcho : Using oracles/external sources is recommended because of Solidity's determinism. Using the Chainlink VRF is an alternative for using a randomness function. This can be done by creating a Chainlink account (some functions will be paid) and calling requestRandomness to import random data:

```
requestRandomness(keyHash, fee, seed);
```

taran317 : Use chainlink VRF instead to generate randomness. This will entail having the current smart contract inherit from VRFConsumerBase and using the requestRandomness() feature using a hard-coded keyHash and fee and a pseudo-random seed that is provided by the user. This returned value can then be manipulated into a uint8 value.
Secure3 : It is recommended to use chainlink VRF to obtain trusted random numbers.

Client Response

Fixed. We will use chainlink VRF to obtain the trusted random numbers

CIS-9: `XController::mint` Incorrect `msg.value` check

Category	Severity	Code Reference	Status	Contributor
Logical	Critical	• code/contracts/XController.sol#L87	Fixed	lohpaul9, seanwjcho, taran317, Secure3

Code

```
87:            require(msg.value > NFT_PRICE_ETH, "Insufficient ETH");
```

Description

lohpaul9 : When users mint NFTs and can specify the amount to mint, the contract keeps minting as long as msg.value is greater than the price of the NFT. However, msg.value is never decremented to account for the minting of a new NFT. That is, as long as msg.value is greater than NFT_PRICE_ETH, the user can mint as many NFTs as they want since the check that msg.value > NFT_PRICE_ETH will always be true in the for loop.
seanwjcho : Inside the mint loop, you have "require(msg.value > NFT_PRICE_ETH, "Insufficient ETH");". This only checks you paid a sufficient amount to mint 1 NFT, not "amount" NFTs.
The impact is that the malicious contract can as many NFTs as they want by just paying for one
taran317 : On line 87 (within the for-loop which mints an NFT `amount` times), the contract requires that `msg.value > NFT_PRICE_ETH`. However, this condition only checks whether `msg.value` is enough for minting one NFT. This same require condition is evaluated in each iteration. This is dangerous because as long as `msg.value` is above `NFT_PRICE_ETH`, `amount` can be set to any value and that many NFTs will be minted since the require will always evaluate to true.
Secure3 : `mint` function checks if `msg.value` is greater than `NFT_PRICE_ETH` in the for loop. It only ensures that the user pays for one NFT. The attacker can only pay `NFT_PRICE_ETH` but mint multiple NFT.
For example, `NFT_PRICE_ETH` is 0.2 ether. The user can send a `mint` transaction that `amount` is 5 but `msg.value` is only 0.2 ether. The transaction will succeed and the user gets 5 NFT.

Recommendation

lohpaul9 : Adjust msg.value accordigly.
Consider below fix in the `mint()` function

```
function mint(uint256 amount) external payable {
    // unchecked to save gas
    unchecked {
        require(amount + xnft.totalSupply() <= xnft.MAX_SUPPLY());
    }

    uint256 fee = msg.value;

    require(amount > 0, "amount should gt 0");

    for (uint256 i = 0; i < amount; ++i) {
        require(fee > NFT_PRICE_ETH, "Insufficient ETH");
        fee = fee - NFT_PRICE_ETH;
        uint256 id = xnft.mint(msg.sender);
        xnft.setRarity(id, getRandomNumber());
    }
}
```

seanwjcho : Consider below fix in the `sample.test()` function

```
require(msg.value * amount > NFT_PRICE_ETH, "Insufficient ETH");
for (uint256 i = 0; i < amount; ++i) {
            uint256 id = xnft.mint(msg.sender);
            xnft.setRarity(id, getRandomNumber());
        }
```

Alternatively, store a user balance and decrement it inside the loop every time an NFT is minted
taran317 : Remove the require statement on line 87. Instead, add `require(msg.value >= amount * NFT_PRICE_ETH, "Insufficient ETH");`. In doing so, we check whether msg.value is sufficient for all `amount` NFTs, rather than only one.
Secure3 : Consider below fix in the `XController::mint()` function:

```
require(msg.value == NFT_PRICE_ETH * amount, "Insufficient ETH");
...
```

Client Response

Fixed

CIS-10: `XController` locks Ether

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	• code/contracts/XController.sol#L77	Fixed	david-cao, Secure3

Code

```
77:    function mint(uint256 amount) external payable {
```

Description

david-cao : As `XController` mints NFTs, it will gather ETH sent by users to pay for the mints. However, there does not exist a way for that ETH to be withdrawn. As a result, `XController` acts as a way to burn ETH. Even if users sell NFTs back to the contract, some ratio will be left over. We assume the developers would like to collect the ETH sent to `XController` for mints as a payout.
Secure3 : Because user will send Ether to the `XController` contract when minting NFT, `XController` will accumulate Ether during mint. But there is no function for owner to take out Ether, resulting in Ether being locked in the contract forever.

Recommendation

david-cao : Implement a method to withdraw funds to an owner. See issue #2 regarding ownership. Assuming this is fixed, one could implement a simple method such as

```
    function withdraw() public onlyOwner {
        uint balance = address(this).balance;
        msg.sender.transfer(balance);
    }
```

to ensure payouts can be collected.

Moreover, since the contract requires the ability to payout users, we recommend additionally adding a receive function to enable the ability to fund the contract with ETH in the case of accounting mishaps.
Secure3 : Add a function that allows contract owner to withdraw Ether in the contract.

For example:

```
function withdraw() onlyOwner external {
    msg.sender.call{value:address(this).balance}("");
}
```

Client Response

Fixed. Added `withdraw` function

CIS-11: `XNFT::lock` Missing permission control check

Category	Severity	Code Reference	Status	Contributor
Logical	Critical	• code/contracts/XNFT.sol#L83-L85	Fixed	david-cao, lohpaul9, seanwjcho, taran317, Secure3

Code

```
83:    function lock(uint256 tokenId) external {
84:        locked[tokenId] = true;
85:    }
```

Description

david-cao : The `lock()` function is callable by the public, allowing anyone to prevent NFTs from being transferred. Effectively, this renders the entire contract worthless as someone could lock every NFT as soon as they are minted.
lohpaul9 : In the `lock()` function, the dev comment clearly asks for the function to be called only by the Controller. Currently, the contract is callable by any external parties.

The impact of this negligence is that any arbitrary party can lock tokens that they may or may not own, thus affecting the core logic of the contracts. The most immediate effect would be that no one is able to transfer their token anymore.

Consider below functions in XNFT.sol

```
/// @notice Controller locks an NFT. Locked NFT can no longer be transferred.
    ///
    /// @dev Only controller contract can call this function
    ///
    /// @param tokenId uint256 ID of the NFT to be locked
    function lock(uint256 tokenId) external {
        locked[tokenId] = true;
    }
```

seanwjcho : The lock() function in XNFT.sol has no onlyController modifier to ensure that only the controller contract can call this function. This missing in the header allows any external malicious contract to call the lock function and thus lock any NFT to prevent it from being transferred.

A general malicious contract example would be:

```
contract LockHack {
    function hackLock() external {
        this.lock(uint256 tokenId);
    }
    ...
}
```

taran317 : The lock function in XNFT should only be able to be called by the controller contract (as specified by the documentation for this function). There is a modifier for onlyController, but it's not applied on this lock function. This is an issue because this condition is not checked for when lock is called.
`function lock(uint256 tokenId) external { locked[tokenId] = true; }`
Secure3 : The `XNFT.lock` function can be called by any address, but according to the annotation, this function can only be called by the controller contract.

Recommendation

david-cao : Simply add the `controllerOnly` modifier to the function.
lohpaul9 : Adding modifier `onlyController()` to the function.
Consider below fix

```
function lock(uint256 tokenId) external onlyController {
        locked[tokenId] = true;
    }
```

seanwjcho : Add an onlyController modifier in the header.
Consider below the fix in the `lock()` function with the correct header:

```
function lock(uint256 tokenId) onlyController external {
        locked[tokenId] = true;
    }
```

taran317 : Add onlyController modifier to lock function.
Secure3 : Add missing `onlyController` modifier. Consider below fix:

```
function lock(uint256 tokenId) external onlyController {
    locked[tokenId] = true;
}
```

Client Response

Fixed. Added `onlyController` modifier for access control.

CIS-12: `XVCToken` has no apparent utility

Category	Severity	Code Reference	Status	Contributor
Logical	Informational	code/contracts/XController.sol#L140-L148	Fixed	david-cao

Code

```
    function withdrawTokens(uint256 _amount, string calldata _password)
        external
        requirePassword(_password)
    {
        uint256 amount = _amount == 0 ? xtoken.balanceOf(address(this)) : _amount;
        if (amount > 0) {
            xtoken.transfer(msg.sender, amount);
        }
    }
```

Description

david-cao : Overall, it's unclear what utility the ERC20 are adding to the ecosystem. Currently, only the owner is able to mint tokens. Thus, if the goal is to provide them to users, we assume some amount will be provided as liquidity into DEXes to allow users to purchase them. However, since an external user has no use case for tokens in the code given, a user has no incentive to purchase them besides speculation. Moreover, if a user were to examine the contract code and see the `withdrawTokens()` function, they may be even more disincentivized to purchase and hold tokens. See more in the centralization risk issue.

Furthermore, the token does not appear to provide any utility to the `XController` contract either. The only method is `withdrawTokens()`, which means the owner of the `XCVToken` contract must transfer tokens to `XController`. From there, the tokens will simply sit until they are withdrawn.

Recommendation

david-cao : Provide utility for the ERC20 or remove it. Some potential uses could be allowing the token to be used on the platform to pay for fees, requiring the NFT to be purchased using the token, or allow staking of the NFT that yields the token.

Client Response

Fixed

Disclaimer