Competitive Security Assessment

xPET

Dec 20th, 2023

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

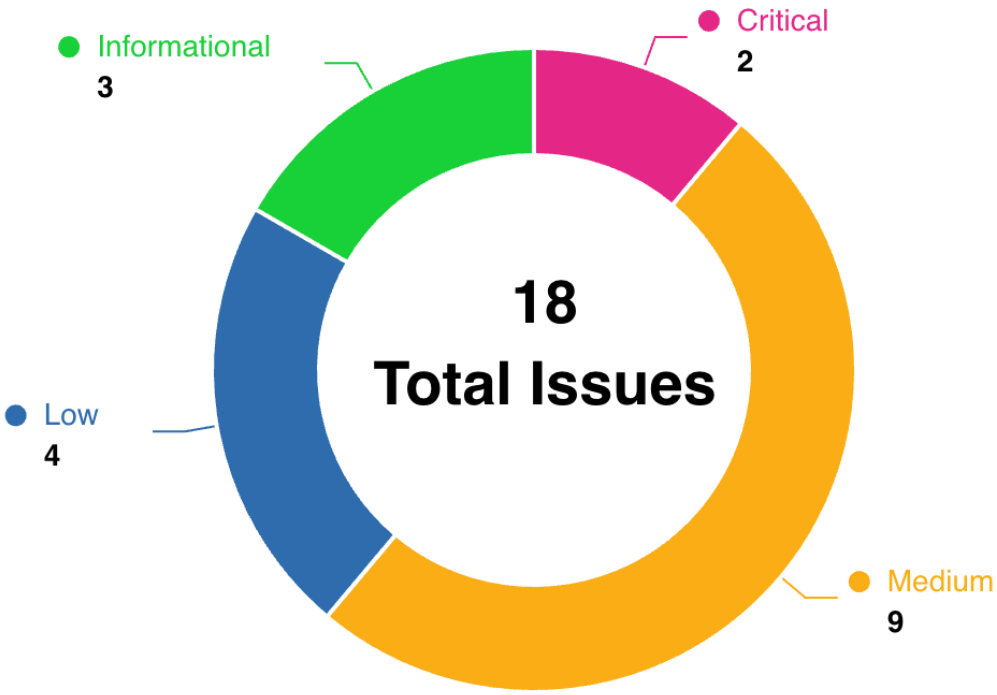
Project Detail

Project Name	xPET
Platform & Language	Solidity
Codebase	• https://github.com/xpet-tech/xpet-core • audit commit - b54eb8637f90b141b3498ef0677ca5dbdcfa91e2 • final commit - 0e1ea742c3edee3df980b1b3cfe7cbc25ea2cce7
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Audit Scope

File	SHA256 Hash
./contracts/XpetMarket.sol	56f8832cf38b057acdbda1ec9e750df2debb3df14a1c07179e3fe08f357a697f
./contracts/XpetNFT.sol	a6d7a0dc7a50136c6113ceb655b606c1100f1ce5baf46cd447b1c86c94ce5570
./contracts/XpetBorrow.sol	34f584d022fb056d4491852f182547fe803a7d4242c4c3db806600e54baa644f
./contracts/XpetWithdraw.sol	414c23bb585597d1fe4f65f83678d448f3b72d3621cac514a598b441c3fb883f
./contracts/XpetFarm.sol	ba05709f3e9fc3e155d56f967645aac7ecb15da956b2e339940db92be84e0ced
./contracts/BpetPrice.sol	8f2cb0f3bd0a2f7892b75fc7a34c6cd6b311dec384efb0a1f180886d3b517845
./contracts/XpetPrice.sol	ab452e31b3e454a83a9abf71e1e44ef08838fb6a542e01232e8789ccdd1cbcdd
./contracts/XpetLiquidatedFund.sol	7823f6bb064645e2995685c72674b591575f077fc6d7c487eb92b1f85491b71b
./contracts/XPET.sol	ac52a5d9c4f9a33e4222ed53d1a8255bed86a24c7a7c6c4650b1c9111e7f8c8e
./contracts/BPET.sol	e83ba7c115ed46c0917a27364ac36dc447f040d676e410d0ca01f59ee0b86a73
./contracts/interfaces/IBPET.sol	0ecf743748d6e6cfedf07b89a621894c6bc3b40a8aeb1c1ee049c6ac4383ee58
./contracts/interfaces/IXPET.sol	b802c74287d9f125a568f0798928111bc4b498e82b54547862a164973ad7850e
./contracts/interfaces/IBpetPrice.sol	0b85d35528bfb55e8e25be02c92e793d0d72646903cf4476fc4a732dd26616b7
./contracts/interfaces/IXpetNFT.sol	ba64a8736c5268aef386e0eea27e0d57590a460410adcc707795ae3400cb33e3
./contracts/interfaces/IXpetPrice.sol	e06c82a142c88658a6bdb34e9e658faffe04178df5052c0870520acd2c5319c1

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
XPT-1	Users will have to wait twice the expected time for claiming XPET in XpetWithdraw::requestConvert()	Logical	Critical	Fixed	grep-er
XPT-2	Risk of over-privilege	Logical	Critical	Mitigated	n16h7m4r3, Kong7ych3
XPT-3	Merkle Tree signatures replay attack possible.	Signature Forgery or Replay	Medium	Fixed	n16h7m4r3, Kong7ych3

XPT-4	Return values not checked when sending ETH	Logical	Medium	Fixed	0xzoobi, Meliclit, grep-er
XPT-5	Incorrect ways to get prices from Chainlink	Logical	Medium	Fixed	Meliclit
XPT-6	call() should be used instead of `transfer()` on an `address payable`	DOS	Medium	Fixed	0xzoobi, Meliclit, n16h7m4r3, Kong7ych3
XPT-7	`OwnableUpgradeable`: Does not implement 2-Step-Process for transferring ownership	Logical	Medium	Mitigated	0xzoobi
XPT-8	Using deprecated ChainLink API -- `latestAnswer`	Oracle Manipulation	Medium	Fixed	0xzoobi, Kong7ych3, n16h7m4r3, grep-er
XPT-9	Reentrancy possible in `mint()` and `safeMint()` functions.	Reentrancy	Medium	Fixed	Meliclit, Kong7ych3, n16h7m4r3
XPT-10	Attacker can froze funds of other users	DOS	Medium	Fixed	Meliclit, Kong7ych3
XPT-11	State variable `revenue` and `fund` not updated when owner withdraws `XPET` tokens from the contract.	Logical	Medium	Fixed	n16h7m4r3
XPT-12	Missing Check for Duplicate Minting in the `XpetNFT::safeMint()`	Logical	Low	Fixed	0xzoobi
XPT-13	Missing `address(0)` checks when minting the tokens	Logical	Low	Fixed	0xzoobi
XPT-14	Use basis points instead of % across the project	Logical	Low	Acknowledged	grep-er
XPT-15	Add checks while upgrading MIN_CONVERT and MAX_CONVERT to avoid braking invariant in XpetWithdraw.sol	DOS	Low	Fixed	grep-er

XPT-16	Follow CEI pattern	Reentrancy	Informational	Fixed	0xzoobi, Meliclit, Kong7ych3
XPT-17	Redundant receiveETH function	Code Style	Informational	Fixed	Kong7ych3
XPT-18	Risk that any user can use the same price in the market	Logical	Informational	Acknowledged	Kong7ych3

XPT-1:Users will have to wait twice the expected time for claiming XPET in XpetWithdraw::requestConvert()

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	grep-er

Code Reference

- code/contracts/XpetWithdraw.sol#L116-L118

```
116:users[msg.sender].timeClaim = lastTimeRequest < block.timestamp ? block.timestamp : lastTimeRequ
est + TIME_WAIT;
117:
118:        lastTimeRequest = users[msg.sender].timeClaim + TIME_WAIT;
```

Description

grep-er : Summary: In function requestConvert() which is need to be called before claiming to set a appropriate claim time available.

1. When first user calls this function they are assigned(`users[msg.sender].timeClaim`) block.timestamp(latest) appointment to claim.
2. But on next line the `lastTimeRequest` get updated and shows time of `TIME_WAIT` more then `block.timestamp` but it should be `block.timestamp`.
3. Now if second person calls this function to set appointment to claim xpet they are scheduled at `users[msg.sender].timeClaim = lastTimeRequest + TIME_WAIT;` as lastTimeRequest > block.timestamp till 2XTIME_WAIT of first request.

```
function requestConvert(uint256 amount) public {
.
.
users[msg.sender].timeClaim = lastTimeRequest < block.timestamp ? block.timestamp : lastTimeRequest
+ TIME_WAIT;
lastTimeRequest = users[msg.sender].timeClaim + TIME_WAIT;// @audit adding TIME_WAIT again despite o
f TIME_WAIT included in users[msg.sender].timeClaim
.
.
}
```

This also cause problems in getTimeQueue view function

```
    function getTimeQueue() public view returns(uint256) {
        return lastTimeRequest < block.timestamp ? block.timestamp : lastTimeRequest + TIME_WAIT;
    }
```

Recommendation

grep-er : Don't add `TIME_WAIT` twice

```
    function requestConvert(uint256 amount) public {
        require(amount <= MAX_CONVERT, "XpetWithdraw: Amount too big");
        require(amount >= MIN_CONVERT, "XpetWithdraw: Amount too small");
        require(fund >= amount, "XpetWithdraw: Fund not enough");
        require(users[msg.sender].amount == 0, "XpetWithdraw: In queue");

        // take BPET from user
        SafeERC20.safeTransferFrom(BPET, msg.sender, address(this), amount);

        fund = fund.sub(amount);

        users[msg.sender].amount = amount;
        users[msg.sender].timeClaim = lastTimeRequest < block.timestamp ? block.timestamp : lastTime
Request + TIME_WAIT;

--          lastTimeRequest = users[msg.sender].timeClaim + TIME_WAIT;
++          lastTimeRequest = users[msg.sender].timeClaim;


        emit RequestEvent(msg.sender, amount, users[msg.sender].timeClaim);
    }
```

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/8298cb60fdff5c8ee81d54f0b5e254b5a37439a2

XPT-2:Risk of over-privilege

Category	Severity	Client Response	Contributor
Logical	Critical	Mitigated	n16h7m4r3, Kong7ych3

Code Reference

- code/contracts/BpetPrice.sol#L23-L25
- code/contracts/XpetPrice.sol#L23-L25
- code/contracts/XpetPrice.sol#L24
- code/contracts/XpetBorrow.sol#L54-L60

```
23:function updatePriceByUpdater(uint256 _price) public onlyRole(UPDATER_ROLE) {
24:         price = _price;
25:     }

23:function updatePriceByUpdater(uint256 _price) public onlyRole(UPDATER_ROLE) {
24:         price = _price;
25:     }

24:price = _price;

54:function updateDataFeed(address _dataFeed) public onlyOwner() {
55:         dataFeed = AggregatorInterface(_dataFeed);
56:     }
57:
58:     function updateXpetPrice(address _XpetPrice) public onlyOwner {
59:         XpetPrice = IXpetPrice(_XpetPrice);
60:     }
```

Description

n16h7m4r3 : The price of `BPET` and `XPET` tokens in the contracts `BpetPrice` and `XpetPrice` respectively can be set to arbitrary value without any restriction by a wallet with `UPDATER_ROLE` privilege. The current demand and market value of the tokens are not taken into account when determining the price of the tokens. This would also lead to use of outdated token value for the respective tokens.

Kong7ych3 : In the protocol, the owner role can arbitrarily modify token prices as well as arbitrarily modify sensitive parameters in the contract. Arbitrarily modifying prices may lead to users' collateral in XpetBorrow being liquidated. This will lead to the risk of the owner having excessive privileges.

Recommendation

n16h7m4r3 : Consider determining and updating value of tokens regularly based on current market value and demands.
Kong7ych3 : In the short term, the project team may need to frequently modify protocol parameters, so controlling the owner rights by the project team can ensure the protocol runs as expected. And the owner role ownership should be transferred to a multi-sig contract to avoid single point risk. But this still cannot alleviate the risk of excessive privileges.
In the long run, it is necessary to use on-chain oracles, and the owner ownership can be given to a timelock contract and governed by the community. This can alleviate the risk of excessive owner privileges and increase the trust of community users.

Client Response

Mitigated,There are many features related to price, we don't want to have problems such as manipulating prices by flashloan and then borrowing, buying items, buying pets, upgrading... at extremely cheap prices.
We will stop the borrow feature when the xpet balance in the xpet contract token runs out (Currently 1.9M XPET left). We always ensure updated prices are completely accurate.
Long-term: We will connect with reputable oracle companies to get prices on oracle.

XPT-3:Merkle Tree signatures replay attack possible.

Category	Severity	Client Response	Contributor
Signature Forgery or Replay	**Medium**	**Fixed**	**n16h7m4r3, Kong7ych3**

Code Reference

- code/contracts/XpetFarm.sol#L40
- code/contracts/XpetFarm.sol#L44-L54
- code/contracts/XpetFarm.sol#L47
- code/contracts/XpetFarm.sol#L50-L51
- code/contracts/XpetMarket.sol#L91
- code/contracts/XpetMarket.sol#L107
- code/contracts/XpetMarket.sol#L122
- code/contracts/XpetMarket.sol#L134

```
40:root = _root;

44:function claim(bytes32[] memory proof, uint256 amount) public {
45:         require(claimId[msg.sender] < lastUpdateId, "XpetFarm: Claimed");
46:
47:         bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(msg.sender, amount))));
48:         require(MerkleProof.verify(proof, root, leaf), "XpetFarm: Invalid proof");
49:
50:         BPET.convert(msg.sender, amount);
51:         claimId[msg.sender] = lastUpdateId;
52:
53:         emit ClaimEvent(msg.sender, amount);
54:     }

47:bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(msg.sender, amount))));

50:BPET.convert(msg.sender, amount);
51:         claimId[msg.sender] = lastUpdateId;

91:bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(id, price))));

107:bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(id, price, level))));

122:bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(level, price))));

134:bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(level, price))));
```

Description

n16h7m4r3 : The `keccak256` hash computed `leaf` does not have an unique attribute to differentiate the Merkle leaf hash. If the contracts are deployed on another blockchain with same root hash, then the wallet would be able to replay the signature multiple times by providing the same Merkle proof across different blockchain.

Kong7ych3 : In the XpetFarm contract, users can claim BPET tokens through the claim function. When a user claims, it first checks whether the claimId is less than lastUpdateId and verifies the user's merkle proof. Finally, the user's claimId state is updated to lastUpdateId. This can effectively prevent users from claiming repeatedly.

However, it should be noted that the UPDATER_ROLE can update the merkle root through the updateRootHash function. At the same time as the update, lastUpdateId will increase by 1. This will lead to some risks of duplicate claims.

If the UPDATER_ROLE incorrectly updates the merkle root to be the same as the previous one, then the lastUpdateId will increase by 1 even though the root is unchanged. This allows users to claim again.

In another case, the project team may use a new merkle root for unknown reasons and regenerate new proofs for all users. This will cause users who have already claimed to be able to claim again, while users who have not claimed can

only claim once. In other words, when the project team updates the merkle root to allow users who meet the conditions to claim again, they have not considered that users who have not claimed before may be missed.

Recommendation

n16h7m4r3 : Consider adding a unique blockchain specific signature or validating the blockchain ID when computing the leaf node as follow:

```
bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(block.chainid, msg.sender, amount))));
```

Kong7ych3 : To avoid updating the same merkle root, it is recommended to check in the updateRootHash function that the new root cannot be equal to the old root.

As for the other case, on the one hand, the user's claim eligibility and the amount of tokens that can be claimed should be checked when the proof is generated for the user off-chain. On the other hand, a round-based approach can be used to record the claimId (for example, `claimId[rounds][users]`). The round can be added to the leaf for checking.

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/9ee148d7a0aab7d310c251eb34a596f86670ecb1

XPT-4:Return values not checked when sending ETH

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	0xzoobi, Meliclit, grep-er

Code Reference

- code/contracts/XpetBorrow.sol#L115-L120
- code/contracts/XpetBorrow.sol#L117-L119
- code/contracts/XpetBorrow.sol#L117

```
115:// transfer ETH from contract to liquidatedFund
116:        if(fundContract != address(0)) {
117:            (bool success, bytes memory data) = fundContract.call{value: myBorrow.amountETH}(
118:                abi.encodeWithSignature("liquidated()")
119:            );
120:        }

117:(bool success, bytes memory data) = fundContract.call{value: myBorrow.amountETH}(
118:                abi.encodeWithSignature("liquidated()")
119:            );

117:(bool success, bytes memory data) = fundContract.call{value: myBorrow.amountETH}(
```

Description

0xzoobi : The return value of low level call is not checked in `XpetBorrow::liquidate()` function.

If the `fundContract` address is a contract and its `receive()` function has the potential to revert, the code `fundContract.call{value: myBorrow.amountETH}(abi.encodeWithSignature("liquidated()"))` could potentially return a false result, which is not being verified.

As a result, the calling functions may exit without transferring native Ether to `fundContract` post liquidation.

Meliclit : On lines 117-119, ETH is transferred using a .call to an fundContract, but there is no verification that the call call succeeded. This can result in a call to fundContract appearing successful but in reality it failed.

```
if(fundContract != address(0)) {
            (bool success, bytes memory data) = fundContract.call{value: myBorrow.amountETH}(
                abi.encodeWithSignature("liquidated()")
            );
        }
```

grep-er : Summary: When a contract calls another contract, the called contract can fail silently without throwing an exception. If the calling contract doesn't check the outcome of the call, it might assume that the call was successful, even if it wasn't.

Unchecked external calls can lead to failed transactions, lost funds, or incorrect contract state.

```
        if(fundContract != address(0)) {
            (bool success, bytes memory data) = fundContract.call{value: myBorrow.amountETH}(
                abi.encodeWithSignature("liquidated()")
            );//@audit success not checked and data not used
        }
```

Recommendation

0xzoobi : It's recommended to check the return value to be true for low level calls.

Sample Fix:

```
(bool success, bytes memory data) = fundContract.call{value: myBorrow.amountETH}(
                abi.encodeWithSignature("liquidated()")
            );
require(success, "Transfer of Ether failed!!");
```

Meliclit : Modify code in a following way

```
if(fundContract != address(0)) {
            (bool success, bytes memory data) = fundContract.call{value: myBorrow.amountETH}(
                abi.encodeWithSignature("liquidated()")
            );
            require(success, "call failed");
}
```

grep-er : Check return value of success and recommended to not load data in memory as it is not used

```
        if(fundContract != address(0)) {
            (bool success, bytes memory data) = fundContract.call{value: myBorrow.amountETH}(
                abi.encodeWithSignature("liquidated()")
            );
++              require(success);
        }
```

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/0e1ea742c3edee3df980b1b3cfe7cbc25ea2cce7

XPT-5:Incorrect ways to get prices from Chainlink

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	Meliclit

Code Reference

- code/contracts/XpetBorrow.sol#L70
- code/contracts/XpetBorrow.sol#L108
- code/contracts/XpetBorrow.sol#L132

```
70:uint256 priceETH = uint256(dataFeed.latestAnswer());

108:uint256 priceETH = uint256(dataFeed.latestAnswer());

132:return dataFeed.latestAnswer();
```

Description

Meliclit : Using Chainlink in L2 chains such as Arbitrum requires to check if the sequencer is down to avoid prices from looking like they are fresh although they are not. The bug could be leveraged by malicious actors to take advantage of the sequencer downtime.
Meliclit : According to Chainlink's documentation (API Reference), the latestAnswer function is deprecated. This function does not throw an error if no answer has been reached, but instead returns 0, possibly causing an incorrect price to be fed.
The impact is the same as with the L2 sequencer report. When users borrow, they may receive fewer or more XPET tokens. Additionally, if the sequencer is down and the wrong price is reported, users may be liquidated.

Recommendation

Meliclit : It is recommended to follow the code example of Chainlink: https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-code
Meliclit : It is recommended to use Chainlink's latestRoundData() function to get the price instead. It is also recommended to add checks on the return data with proper revert messages if the price is stale or the round is incomplete

Client Response

Fixed
Commit URL: https://github.com/xpet-tech/xpet-core/commit/e35cbcd41298eb82e166e2f8d004ff27016f227d

XPT-6:call() should be used instead of `transfer()` on an `address payable`

Category	Severity	Client Response	Contributor
DOS	Medium	Fixed	0xzoobi, Meliclit, n16h7m4r3, Kong7ych3

Code Reference

- code/contracts/XpetLiquidatedFund.sol#L21
- code/contracts/XpetBorrow.sol#L97
- code/contracts/XpetNFT.sol#L122

```
21:payable(user).transfer(amount);

97:payable(msg.sender).transfer(_amountETH);

97:payable(msg.sender).transfer(_amountETH);

122:payable(user).transfer(amount);
```

Description

0xzoobi : The use of the deprecated `transfer()` function for an address will inevitably make the transaction fail when:

1. The claimer smart contract does not implement a payable function.
2. The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit.
3. The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.
4. Additionally, using higher than 2300 gas might be mandatory for some MultiSig wallets.

Meliclit : The use of the deprecated transfer() function for an address will inevitably make the transaction fail when:

- The claimer smart contract does not implement a payable function.
- The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit.
- The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.

Additionally, using higher than 2300 gas might be mandatory for some multisig wallets.

```
96: // transfer ETH from contract to user
97: payable(msg.sender).transfer(_amountETH);
```

n16h7m4r3 : The use of the deprecated `transfer()` function for an address will inevitably make the transaction fail

when:

1. The receiver smart contract does not implement a payable function.
2. The receiver smart contract does implement a payable fallback which uses more than 2300 gas unit.
3. The receiver smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.

Additionally, using higher than 2300 gas might be mandatory for some multisig wallets.
Kong7ych3 : In the XpetBorrow contract, users can repay XPET to redeem collateral through the pay function. The pay function will transfer ETH tokens to the user through the transfer function. This is obviously to avoid reentrancy risks, but it will have potential compatibility issues. If users borrow through a contract (such as a multi-signature wallet, which is very common), they will be unable to repay through the pay function. This will result in user funds being locked in the contract and eventually being liquidated.

Recommendation

0xzoobi : I recommend using call() instead of transfer()

```
        (bool success, ) = address(user).call{value: msg.value}("");
        require(success, "Call failed");
```

Meliclit : I recommend using call() instead of transfer().
n16h7m4r3 : It is recommend to use `call()` instead of `transfer()`.
Kong7ych3 : It is recommended to use the call method to transfer ETH tokens, or to check in the borrow function that msg.sender must not be a contract.

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/d5ae1d11a23399356308a2280c65c687f7fca49e

XPT-7: `OwnableUpgradeable`: Does not implement 2-Step-Process for transferring ownership

Category	Severity	Client Response	Contributor
Logical	Medium	Mitigated	0xzoobi

Code Reference

- code/contracts/XpetBorrow.sol#L4
- code/contracts/XpetLiquidatedFund.sol#L4
- code/contracts/XpetMarket.sol#L4
- code/contracts/XpetWithdraw.sol#L4

```
4:import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";

4:import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";

4:import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";

4:import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
```

Description

0xzoobi : The contracts `XpetBorrow.sol` , `XpetLiquidatedFund.sol`, `XpetMarket.sol` and `XpetWithdraw.sol` does not implement a 2-Step-Process for transferring ownership. So ownership of the contract can easily be lost when making a mistake when transferring ownership.

Since the privileged roles have critical function roles assigned to them. Assigning the ownership to a wrong user can be disastrous. So Consider using the Ownable2StepUpgradeable.sol contract from OZ (https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v5.0.0/contracts/access/Ownable2StepUpgradeable.sol) instead.

The way it works is there is a `transferOwnership` to transfer the ownership and `acceptOwnership` to accept the ownership. Refer the above Ownable2Step.sol for more details.

Recommendation

0xzoobi : Implement 2-Step-Process for transferring ownership via `Ownable2StepUpgradeable`.

Client Response

Mitigated,Implementing Ownable2StepUpgradeable for the current contracts makes it very difficult to upgrade contracts. We will never transfer ownership to avoid making mistakes.

XPT-8:Using deprecated ChainLink API -- `latestAnswer`

Category	Severity	Client Response	Contributor
Oracle Manipulation	Medium	Fixed	0xzoobi, Kong7ych3, n16h7m4r3, grep-er

Code Reference

- code/contracts/XpetBorrow.sol#L70
- code/contracts/XpetBorrow.sol#L108
- code/contracts/XpetBorrow.sol#L132

```
70:uint256 priceETH = uint256(dataFeed.latestAnswer());

70:uint256 priceETH = uint256(dataFeed.latestAnswer());

108:uint256 priceETH = uint256(dataFeed.latestAnswer());

132:return dataFeed.latestAnswer();
```

Description

0xzoobi : The `XpetBorrow` contract makes use of the `latestAnswer()` method from Chainlink price feeds to get the price of an asset Ex: Ether.

But The NatSpec of latestAnswer says this:

```
@dev #[deprecated] Use latestRoundData instead. This does not error if no answer has been reached, i
t will simply return 0. Either wait to point to an already answered Aggregator or use the recommende
d latestRoundData instead which includes better verification information.```
```

So currently it is possible that `latestAnswer()` returns 0 and the code operates with zero price, leading to incorrect return value of ETH price affecting both borrow and liquidate function as part of the `XpetBorrow.sol` contract.

official API Reference - https://docs.chain.link/data-feeds/api-reference#latestanswer

Kong7ych3 : The xPET protocol is deployed on L2 Arbitrum, have a sequencer that executes and rolls up the L2 transactions by batching multiple transactions into a single transaction. If a sequencer becomes unavailable, it is impossible to access read/write APIs that consumers are using and applications on the L2 network will be down for most users without interacting directly through the L1 contracts. The L2 has not stopped, but it would be unfair to continue providing service on your applications when only a few users can use them. Therefore, the Chainlink oracle provides L2 sequencer feeds service to help users determine whether the L2 sequencer is available. Therefore, when getting the price through the Chainlink oracle, you should first call the L2 sequencer feeds to check whether the sequencer is available.

n16h7m4r3 : According to Chainlink's documentation, the `latestAnswer()` function is deprecated. This function does

not error if no answer has been reached but returns 0. Besides, the `latestAnswer()` is reported with 18 decimals for crypto quotes but 8 decimals for FX quotes (See Chainlink FAQ for more details). A best practice is to get the decimals from the oracles instead of hard-coding them in the contract.

Kong7ych3 : In the XpetBorrow contract, users can borrow XPET tokens by staking ETH through the borrow function. The function uses the Chainlink oracle to get the ETH token price to calculate the value of the user's collateral. However, it is important to note that the latestAnswer interface is used to get the price, which prevents the contract from checking the validity of the price. In fact, the Chainlink documentation states that the latestAnswer interface will be deprecated and that the latestRoundData interface should be used to get the price and check the validity of the price.

grep-er : According to Chainlink's documentation, the `latestAnswer` function is deprecated. This function does not error if no answer has been reached but returns 0. Besides, the `latestAnswer` is reported with 18 decimals for crypto quotes but 8 decimals for FX quotes (See Chainlink FAQ for more details). A best practice is to get the decimals from the oracles instead of hard-coding them in the contract.

```
uint256 priceETH = uint256(dataFeed.latestAnswer());
```

Recommendation

0xzoobi : Use the `latestRoundData()` instead of `latestAnswer()`

Reference - https://docs.chain.link/data-feeds/api-reference/#latestrounddata-1

Kong7ych3 : It is recommended to check whether the L2 sequencer is available before getting the ETH price. Consider the following fixes:

```
    function borrow() payable public {
...
(
/*uint80 roundID*/,
int256 answer,
uint256 startedAt,
/*uint256 updatedAt*/,
/*uint80 answeredInRound*/
) = sequencerUptimeFeed.latestRoundData();

// Answer == 0: Sequencer is up
// Answer == 1: Sequencer is down
bool isSequencerUp = answer == 0;
if (!isSequencerUp) {
revert SequencerDown();
}
...
    }
```

n16h7m4r3 : Use the recommended `latestRoundData()` function to get the price instead. Add checks on the return data with proper revert messages if the price is stale or the round is uncomplete.

Kong7ych3 : It is recommended to use the latestRoundData interface to get the price instead of the latestAnswer

interface. The updatedAt timestamp should also be checked to ensure that the price is fresh. According to the Chainlink design, the ETH price must be updated every 86,400 seconds. The following fix is considered:

```
...
(
  uint80 roundId,
  int256 answer,
  uint256 startedAt,
  uint256 updatedAt,
  uint80 answeredInRound
) = dataFeed.latestRoundData();
require(answer > 0, "")
require(block.timestamp - updatedAt > timeThreshold, "") // timeThreshold is 86400s
...
```

grep-er : Use Use V3 interface functions: https://docs.chain.link/docs/price-feeds-api-reference/

```
(uint80 roundID, int256 price, , uint256 timeStamp, uint80 answeredInRound) = oracle.latestRoundData
();
require(answeredInRound >= roundID, "...");
require(timeStamp != 0, "...");
```

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/e35cbcd41298eb82e166e2f8d004ff27016f227d

XPT-9:Reentrancy possible in `mint()` and `safeMint()` functions.

Category	Severity	Client Response	Contributor
Reentrancy	Medium	Fixed	Meliclit, Kong7ych3, n16h7m4r3

Code Reference

- code/contracts/XpetNFT.sol#L92-L110
- code/contracts/XpetNFT.sol#L106-L107
- code/contracts/XpetNFT.sol#L112-L119

```
92:function mint(address user, uint256 petId, bool payByXPET) public{
93:         require(pets[petId] == 0, "XpetNFT: PetId is minted");
94:
95:         if (payByXPET) {
96:             uint256 priceXPET = XpetPrice.getPrice();
97:             uint256 priceBPET = BpetPrice.getPrice();
98:             uint256 amountXPET = priceBPET.mul(PRICE).div(priceXPET);
99:             SafeERC20.safeTransferFrom(XPET, msg.sender, address(this), amountXPET);
100:         } else {
101:             SafeERC20.safeTransferFrom(BPET, msg.sender, address(this), PRICE);
102:         }
103:
104:         uint256 tokenId = _tokenIdCounter.current();
105:         _tokenIdCounter.increment();
106:         _safeMint(user, tokenId);
107:         pets[petId] = tokenId;
108:
109:         emit MintNFT(msg.sender, user, tokenId, petId);
110:     }
```

```
92:function mint(address user, uint256 petId, bool payByXPET) public{
93:         require(pets[petId] == 0, "XpetNFT: PetId is minted");
94:
95:         if (payByXPET) {
96:             uint256 priceXPET = XpetPrice.getPrice();
97:             uint256 priceBPET = BpetPrice.getPrice();
98:             uint256 amountXPET = priceBPET.mul(PRICE).div(priceXPET);
99:             SafeERC20.safeTransferFrom(XPET, msg.sender, address(this), amountXPET);
100:         } else {
101:             SafeERC20.safeTransferFrom(BPET, msg.sender, address(this), PRICE);
102:         }
103:
104:         uint256 tokenId = _tokenIdCounter.current();
105:         _tokenIdCounter.increment();
106:         _safeMint(user, tokenId);
107:         pets[petId] = tokenId;
108:
109:         emit MintNFT(msg.sender, user, tokenId, petId);
110:     }
```

```
106:_safeMint(user, tokenId);
107:         pets[petId] = tokenId;
```

```
112:function safeMint(address user, uint256 petId) public onlyRole(MINTER_ROLE){
113:        uint256 tokenId = _tokenIdCounter.current();
114:        _tokenIdCounter.increment();
115:        _safeMint(user, tokenId);
116:
117:        pets[petId] = tokenId;
118:        emit MarketMintNFT(user, tokenId);
119:    }
```

Description

Meliclit : In XpetNFT.sol, the mint() function calls _safeMint() which has a callback to the "user" address argument. Functions with callbacks should have reentrancy guards in place for protection against possible malicious actors both from inside and outside the protocol.

```
function mint(address user, uint256 petId, bool payByXPET) public{
        require(pets[petId] == 0, "XpetNFT: PetId is minted");

        if (payByXPET) {
            uint256 priceXPET = XpetPrice.getPrice();
            uint256 priceBPET = BpetPrice.getPrice();
            uint256 amountXPET = priceBPET.mul(PRICE).div(priceXPET);
            SafeERC20.safeTransferFrom(XPET, msg.sender, address(this), amountXPET);
        } else {
            SafeERC20.safeTransferFrom(BPET, msg.sender, address(this), PRICE);
        }

        uint256 tokenId = _tokenIdCounter.current();
        _tokenIdCounter.increment();
        _safeMint(user, tokenId);
        pets[petId] = tokenId;

        emit MintNFT(msg.sender, user, tokenId, petId);
    }
```

Kong7ych3 : In the XpetNFT contract, users can mint XPETNFTs through the mint function. The function first uses the `_safeMint` function to mint the XPETNFT token to the user and then sets `pets[petId]` to tokenId. Unfortunately, if the user is a contract, the `_safeMint` function will call the `onERC721Received` function of to after the minting is completed. Malicious users can re-enter the mint function using the same `petId`, which will overwrite `pets[petId]`. Ultimately, although the contract has minted multiple XPETNFTs, only the last XPETNFT points to `pets[petId]`. If `pets[petId]` or the `MintNFT` event is used for critical game parameters off-chain, it may lead to unexpected consequences.

n16h7m4r3 : In the contract `XpetNFT` the functions `mint()` and `safeMint()` allows wallets to mint `XPETNFT` ERC-

721 tokens. The function uses openzeppelin's `_safeMint()` function which is vulnerable to reentrancy risk. Allowing wallets to mint multiple tokens for same `petId`.

Recommendation

Meliclit : Add a reentrancy guard modifier on the mint() function in XpetNFT.sol
Kong7ych3 : It is recommended to add the nonReentrant modifier to the mint function, or to assign `pets[petId]` first, and then perform the `_safeMint` operation.
n16h7m4r3 : Consider updating `pets[petId]` state before minting ERC-721 token in the functions `mint()` and `safeMint()` following checks-effect-interaction pattern as follow:

```
    function mint(address user, uint256 petId, bool payByXPET) public{
        require(pets[petId] == 0, "XpetNFT: PetId is minted");

        if (payByXPET) {
            uint256 priceXPET = XpetPrice.getPrice();
            uint256 priceBPET = BpetPrice.getPrice();
            uint256 amountXPET = priceBPET.mul(PRICE).div(priceXPET);
            SafeERC20.safeTransferFrom(XPET, msg.sender, address(this), amountXPET);
        } else {
            SafeERC20.safeTransferFrom(BPET, msg.sender, address(this), PRICE);
        }

        pets[petId] = tokenId;

        uint256 tokenId = _tokenIdCounter.current();
        _tokenIdCounter.increment();
        _safeMint(user, tokenId);

        emit MintNFT(msg.sender, user, tokenId, petId);
    }

    function safeMint(address user, uint256 petId) public onlyRole(MINTER_ROLE){
        pets[petId] = tokenId;
        uint256 tokenId = _tokenIdCounter.current();
        _tokenIdCounter.increment();
        _safeMint(user, tokenId);

        emit MarketMintNFT(user, tokenId);
    }
```

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/21a2df8695f02e66740a12cfb77d61c7f814f140

XPT-10:Attacker can froze funds of other users

Category	Severity	Client Response	Contributor
DOS	Medium	Fixed	Meliclit, Kong7ych3

Code Reference

- code/contracts/XpetWithdraw.sol#L39-L46
- code/contracts/XpetWithdraw.sol#L116
- code/contracts/XpetWithdraw.sol#L116-L118
- code/contracts/XpetWithdraw.sol#L124

```
39:function initialize(address _XPET, address _BPET) initializer public {
40:         __Ownable_init();
41:         XPET = IXPET(_XPET);
42:         BPET = IBPET(_BPET);
43:         lastTimeRequest = block.timestamp;
44:         MAX_CONVERT = 100 * 1e18;
45:         TIME_WAIT = 5 * 60;
46:     }

116:users[msg.sender].timeClaim = lastTimeRequest < block.timestamp ? block.timestamp : lastTimeRequ
est + TIME_WAIT;

116:users[msg.sender].timeClaim = lastTimeRequest < block.timestamp ? block.timestamp : lastTimeRequ
est + TIME_WAIT;
117:
118:         lastTimeRequest = users[msg.sender].timeClaim + TIME_WAIT;

124:require(block.timestamp >= users[msg.sender].timeClaim, "XpetWithdraw: Not enough wait time");
```

Description

Meliclit : An attacker can call the requestConvert() function multiple times with a value of 0 for the amount parameter, causing the lastTimeRequest value to increase significantly.

```
function requestConvert(uint256 amount) public {
        require(amount <= MAX_CONVERT, "XpetWithdraw: Amount too big");
        require(amount >= MIN_CONVERT, "XpetWithdraw: Amount too small");
        require(fund >= amount, "XpetWithdraw: Fund not enough");
        require(users[msg.sender].amount == 0, "XpetWithdraw: In queue");

        // take BPET from user
        SafeERC20.safeTransferFrom(BPET, msg.sender, address(this), amount);

        fund = fund.sub(amount);

        users[msg.sender].amount = amount;
        users[msg.sender].timeClaim = lastTimeRequest < block.timestamp ? block.timestamp : lastTime
Request + TIME_WAIT;

        lastTimeRequest = users[msg.sender].timeClaim + TIME_WAIT;

        emit RequestEvent(msg.sender, amount, users[msg.sender].timeClaim);
    }
```

All subsequent users invoking requestConvert() will transfer their BPET but won't be able to call the claim() function due to this condition:

```
require(block.timestamp >= users[msg.sender].timeClaim, "XpetWithdraw: Not enough wait time");
```

Notably, only the MAX_CONVERT value is set in the initialize() function. This means that MIN_CONVERT is set to 0:

```
function initialize(address _XPET, address _BPET) initializer public {
        __Ownable_init();
        XPET = IXPET(_XPET);
        BPET = IBPET(_BPET);
        lastTimeRequest = block.timestamp;
        MAX_CONVERT = 100 * 1e18;
        TIME_WAIT = 5 * 60;
    }
```

Kong7ych3 : In the XpetWithdraw contract, users can request to convert BPET to XPET through the requestConvert function. When requesting, the contract will record the user's timeClaim status. Once the current time is greater than timeClaim, the user can claim XPET tokens through the claim function.

When a user performs a requestConvert operation, if the current time is less than lastTimeRequest, the user's timeClaim will be set to `lastTimeRequest + TIME_WAIT`. This means that if the first user performs a requestConvert operation, and the next user performs a requestConvert operation within a time interval of TIME_WAIT, the XPET token claim time for that user will be delayed to `lastTimeRequest + TIME_WAI`. And lastTimeRequest is calculated by adding the timeClaim of the previous user to TIME_WAIT. By analogy, if each user performs a requestConvert operation when block.timestamp is less than lastTimeRequest, the timeClaim time for users who perform operations later will be longer.

Malicious users can use this method to perform requestConvert with 1wei BPET. After a large number of operations, the timeClaim time for ordinary users will be infinitely extended. Ultimately, ordinary users will need to wait a very long time to perform the claim operation, which indirectly locks the user's tokens.

Here is a simple exploit description:

1. Malicious users use addresses A, B, C, D, E, etc. to perform requestConvert spam, using 1wei BPET each time, and the time interval between each operation is less than TIME_WAIT.
2. Assuming that the current time is 1, and TIME_WAIT is 10. Then, after the malicious user's first operation, lastTimeRequest will be set to `11(1+10)`, `31(11+10+10)` after the second operation, `51(31+10+10)` after the third operation, and `1+10*(2n-1)` after the nth operation.
3. After the malicious user performs n operations in a short period of time, lastTimeRequest will be set to `1+10*(2n-1)`. This means that the next normal user who performs a requestConvert operation will have their timeClaim set to `1+10*(2n-1) + 10`.
4. As long as the malicious user performs enough operations, it will lead to the normal user's funds being locked in for an extended period of time.

Recommendation

Meliclit : Set MIN_CONVERT in initialize function

Kong7ych3 : It is recommended to maintain a separate timeClaim state for each different user. When a user performs a requestConvert operation, their timeClaim will be directly set to `block.timestamp + TIME_WAIT`. If the user requests again within TIME_WAIT, their new timeClaim state will be set to `users[msg.sender].timeClaim + TIME_WAIT`. This avoids the risk caused by all users sharing lastTimeRequest.

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/ae3d19b34ae06e03cf59f07b6611f3579497794e

XPT-11:State variable `revenue` and `fund` not updated when owner withdraws `XPET` tokens from the contract.

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	n16h7m4r3

Code Reference

- code/contracts/XpetWithdraw.sol#L70-L81
- code/contracts/XpetWithdraw.sol#L96-L98
- code/contracts/XpetWithdraw.sol#L104-L133

```
70:function convert(uint256 amount) public {
71:         require(revenue >= amount, "XpetWithdraw: Revenue not enough");
72:         revenue = revenue.sub(amount);
73:
74:         // take BPET from user
75:         SafeERC20.safeTransferFrom(BPET, msg.sender, address(this), amount);
76:
77:         // transfer XPET for user
78:         SafeERC20.safeTransfer(XPET, msg.sender, amount);
79:
80:         emit ConvertEvent(msg.sender, amount, revenue);
81:     }

96:function withdrawXPET(address user, uint256 amount) public onlyOwner() {
97:         SafeERC20.safeTransfer(XPET, user, amount);
98:     }

104:function requestConvert(uint256 amount) public {
105:         require(amount <= MAX_CONVERT, "XpetWithdraw: Amount too big");
106:         require(amount >= MIN_CONVERT, "XpetWithdraw: Amount too small");
107:         require(fund >= amount, "XpetWithdraw: Fund not enough");
108:         require(users[msg.sender].amount == 0, "XpetWithdraw: In queue");
109:
110:         // take BPET from user
111:         SafeERC20.safeTransferFrom(BPET, msg.sender, address(this), amount);
112:
113:         fund = fund.sub(amount);
114:
115:         users[msg.sender].amount = amount;
116:         users[msg.sender].timeClaim = lastTimeRequest < block.timestamp ? block.timestamp : last
TimeRequest + TIME_WAIT;
117:
118:         lastTimeRequest = users[msg.sender].timeClaim + TIME_WAIT;
119:
120:         emit RequestEvent(msg.sender, amount, users[msg.sender].timeClaim);
121:     }
122:
123:     function claim() public{
124:         require(block.timestamp >= users[msg.sender].timeClaim, "XpetWithdraw: Not enough wait t
ime");
125:         require(users[msg.sender].amount > 0, "XpetWithdraw: No request");
126:
```

```
127:         // transfer XPET for user
128:         SafeERC20.safeTransfer(XPET, msg.sender, users[msg.sender].amount);
129:
130:         emit ClaimEvent(msg.sender, users[msg.sender].amount);
131:
132:         users[msg.sender].amount = 0;
133:     }
```

Description

n16h7m4r3 : The state variables `revenue` and `fund` allows the contract to keep track of the amount of `XPET` tokens available for the users to convert or claim `XPET` tokens in exchange for `BPET` tokens. The function `withdrawXPET()` allows owner to withdraw `XPET` tokens from the contract, the state variables `revenue` and `funds` are not updated by `withdrawXPET()` function. Which could lead to revert in the function `convert()` and `claim()` due to lack of `XPET` tokens in the contract.

Recommendation

n16h7m4r3 : Consider updating the state variables `revenue` and `fund` when owner transfer `XPET` tokens from the contract using the function `withdrawXPET()`. Also consider having a state variable to track pending `XPET` token claims and not allow owner to withdraw if the amount of `XPET` tokens available after withdrawing is less than amount of pending claims.

Client Response

Fixed Commit URL: https://github.com/xpet-tech/xpet-core/commit/ba81ceb0dedf8ebd8effff49995b93fe750a5ab0

XPT-12:Missing Check for Duplicate Minting in the `XpetNFT::safeMint()`

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	0xzoobi

Code Reference

- code/contracts/XpetNFT.sol#L112

```
112:function safeMint(address user, uint256 petId) public onlyRole(MINTER_ROLE){
```

Description

0xzoobi : The `safeMint()` function, while introducing access control with the `onlyRole(MINTER_ROLE)` modifier, lacks a crucial check to prevent the minting of tokens with duplicate pet IDs. The absence of the `require(pets[petId] == 0, "XpetNFT: PetId is minted");` validation creates an issue where the `MINTER_ROLE` could overwrite existing mappings in the `pets` variable.

This oversight poses a risk of unintentional token overwritesof other users, potentially leading to data inconsistency and unexpected behavior.

Recommendation

0xzoobi : To mitigate this issue, it is crucial to incorporate the check for duplicate minting within the safeMint function. The modified code snippet below includes the necessary require statement: By adding the require statement similar to the `mint()` function , ensures that a pet with the same ID hasn't been minted previously, thereby preventing unintentional overwriting of existing tokens.

```
function safeMint(address user, uint256 petId) public onlyRole(MINTER_ROLE){
    require(pets[petId] == 0, "XpetNFT: PetId is minted");

  //REST OF THE CODE
}
```

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/2284fc1d1731ab680b3ae6700d99cd42d1ffceb7

XPT-13:Missing `address(0)` checks when minting the tokens

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	0xzoobi

Code Reference

- code/contracts/BPET.sol#L17
- code/contracts/XPET.sol#L19

```
17:_mint(user, amount);

19:SafeERC20.safeTransfer(IERC20(address(this)), user, amount);
```

Description

0xzoobi : The contract lacks essential sanity checks during the token minting process, as it fails to validate whether the provided user address is a zero address or not. This oversight could lead to the unintended consequence of minting tokens to `address(0)`, resulting in permanent loss of the tokens after minting.

Recommendation

0xzoobi : To address this issue, it is strongly recommended to implement robust sanity checks in the contract's token minting logic. Specifically, ensure that the provided user address is validated to prevent any attempt to mint tokens to `address(0)`.

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/5017e42981e9f8b77f15065500c7fc5cf5397b68

XPT-14:Use basis points instead of % across the project

Category	Severity	Client Response	Contributor
Logical	Low	Acknowledged	grep-er

Code Reference

- code/contracts/XpetBorrow.sol#L43

```
43:BORROW_PERCENT = 85;
```

Description

grep-er : Since we only have integer math operations in Solidity, the basis points notation facilitates calculations allowing for 2 decimal places of precision, during calculations before truncation. Produces less deviations due to precision errors.

```
BORROW_PERCENT = 85;// @audit user basis points

XpetMarket.sol:160:         uint256 toDevAmount = amountXPET.mul(5).div(100);
XpetMarket.sol:164:         uint256 toWithdrawContractAmount = amountXPET.mul(50).div(100);
XpetMarket.sol:170:         uint256 BPEToFarmContractAmount = amount.mul(45).div(100);
XpetMarket.sol:176:         uint256 toWithdrawContractAmount = amount.mul(50).div(100);
XpetBorrow.sol:74:        uint256 amountXPETReceive = amountXPET.mul(BORROW_PERCENT).div(100);
```

Recommendation

grep-er :

```
-- BORROW_PERCENT = 85;
++ BORROW_PERCENT = 8500;//in bps
```

Client Response

Acknowledged,BORROW_PERCENT we will not update. And the number will not be less than 70%.

XPT-15:Add checks while upgrading MIN_CONVERT and MAX_CONVERT to avoid braking invariant in XpetWithdraw.sol

Category	Severity	Client Response	Contributor
DOS	Low	Fixed	grep-er

Code Reference

- code/contracts/XpetWithdraw.sol#L52
- code/contracts/XpetWithdraw.sol#L56

```
52:function updateBPET(address _BPET) public onlyOwner() {

56 :function updateMaxConvert(uint256 _MAX_CONVERT) public onlyOwner() {
```

Description

grep-er : Add checks instead of directly updating to avoid breaking invariant that max should be equal or more then min logic which will cause unexpected behaviour.

Recommendation

grep-er :

```
    function updateMaxConvert(uint256 _MAX_CONVERT) public onlyOwner() {
++          require(_MAX_CONVERT >= MIN_CONVERT);
        MAX_CONVERT = _MAX_CONVERT;
    }

    function updateMinConvert(uint256 _MIN_CONVERT) public onlyOwner() {
++        require(_MIN_CONVERT <= MAX_CONVERT);
        MIN_CONVERT = _MIN_CONVERT;
    }
```

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/8370a7469e4e61aefe4b72a3651c5b9709a6bc6a

XPT-16:Follow CEI pattern

Category	Severity	Client Response	Contributor
Reentrancy	Informational	Fixed	0xzoobi, Meliclit, Kong7ych3

Code Reference

- code/contracts/XpetFarm.sol#L50-L51
- code/contracts/XpetBorrow.sol#L97-L100
- code/contracts/XpetNFT.sol#L106-L107
- code/contracts/XpetNFT.sol#L115-L117
- code/contracts/XpetBorrow.sol#L116-L124
- code/contracts/XpetWithdraw.sol#L123-L134
- code/contracts/XpetWithdraw.sol#L128-L132

```
50:BPET.convert(msg.sender, amount);
51:         claimId[msg.sender] = lastUpdateId;

97:payable(msg.sender).transfer(_amountETH);
98:
99:         myBorrow.amountETH = myBorrow.amountETH.sub(_amountETH);
100:         myBorrow.amountXPET = myBorrow.amountXPET.sub(_amountXPET);

106:_safeMint(user, tokenId);
107:         pets[petId] = tokenId;

115:_safeMint(user, tokenId);
116:
117:         pets[petId] = tokenId;

116:if(fundContract != address(0)) {
117:             (bool success, bytes memory data) = fundContract.call{value: myBorrow.amountETH}(
118:                 abi.encodeWithSignature("liquidated()")
119:             );
120:         }
121:
122:         emit LiquidateEvent(user, myBorrow.amountETH, myBorrow.amountXPET, priceETH, priceXPET);
123:         myBorrow.amountXPET = 0;
124:         myBorrow.amountETH = 0;

123:function claim() public{
124:         require(block.timestamp >= users[msg.sender].timeClaim, "XpetWithdraw: Not enough wait t
ime");
125:         require(users[msg.sender].amount > 0, "XpetWithdraw: No request");
126:
127:         // transfer XPET for user
128:         SafeERC20.safeTransfer(XPET, msg.sender, users[msg.sender].amount);
129:
130:         emit ClaimEvent(msg.sender, users[msg.sender].amount);
131:
132:         users[msg.sender].amount = 0;
133:     }
134:}

128:SafeERC20.safeTransfer(XPET, msg.sender, users[msg.sender].amount);
```

```
129:
130:        emit ClaimEvent(msg.sender, users[msg.sender].amount);
131:
132:        users[msg.sender].amount = 0;
```

Description

0xzoobi : The `XpetNFT::mint()` function in the provided solidity code appears to lack adherence to the Checks Effects Interaction (CEI) pattern. The CEI pattern is a best practice in smart contract development that involves separating external interactions into distinct functions to enhance security and maintainability.

Meliclit : claim() function in XpetWithdraw.sol doesn't follow CEI(Checks Effects Interactions pattern)

```
function claim() public{
        require(block.timestamp >= users[msg.sender].timeClaim, "XpetWithdraw: Not enough wait tim
e");
        require(users[msg.sender].amount > 0, "XpetWithdraw: No request");

        // transfer XPET for user
        SafeERC20.safeTransfer(XPET, msg.sender, users[msg.sender].amount);

        emit ClaimEvent(msg.sender, users[msg.sender].amount);

        users[msg.sender].amount = 0;
    }
```

Kong7ych3 : In the XpetBorrow contract, the owner can liquidate a specified user through the liquidate function. First, it will transfer the user's ETH collateral to the fundContract using the call method. It is best to set the user's amountXPET/amountETH state to 0 before doing so.

Theoretically, there is a reentrancy risk at this point, but this function can only be called by a privileged role, which will eliminate the risk of abuse. However, this is still not a good practice, as it causes users to have concerns.

Similarly, in the pay function of the XpetBorrow contract, the contract first transfers funds to the user through the transfer function, and then updates the user's amountETH/amountXPET state. This also does not comply with the CEI principle.

The claim function in the XpetFarm contract is also the same. It first transfers BPET tokens and then updates the user's claimId state. This does not comply with the CEI principle.

The claim function in the XpetWithdraw contract is also the same. It first transfers XPET tokens and then updates the user's `users[msg.sender].amount` state. This does not comply with the CEI principle.

Recommendation

0xzoobi : Follow the CEI Pattern. The Code Fix should look like the sample code

```
    function mint(address user, uint256 petId, bool payByXPET) public{
        ///////

         uint256 tokenId = _tokenIdCounter.current();
        _tokenIdCounter.increment();
        pets[petId] = tokenId;
        _safeMint(user, tokenId);

        ///////
}
```

Meliclit : Modify claim function in a following way

```
function claim() public{
        require(block.timestamp >= users[msg.sender].timeClaim, "XpetWithdraw: Not enough wait tim
e");
        require(users[msg.sender].amount > 0, "XpetWithdraw: No request");

        users[msg.sender].amount = 0;

        // transfer XPET for user
        SafeERC20.safeTransfer(XPET, msg.sender, users[msg.sender].amount);

        emit ClaimEvent(msg.sender, users[msg.sender].amount);
    }
```

Kong7ych3 : It is recommended to follow the CEI principle. When a user deposits funds, the transfer should be made before the contract variable is updated. However, when the contract makes an external transfer, the contract variable should be updated before the transfer is made.

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/21a2df8695f02e66740a12cfb77d61c7f814f140

XPT-17:Redundant receiveETH function

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	Kong7ych3

Code Reference

- code/contracts/XpetBorrow.sol#L48

```
48:function receiveETH() external payable {}
```

Description

Kong7ych3 : In the XpetBorrow contract, there is a receive function, which means that the contract can receive native tokens. However, the contract also implements the receiveETH function, which does not implement any logic. This is redundant. In fact, only the receive function needs to be implemented to receive ETH.

Recommendation

Kong7ych3 : It is recommended to remove the redundant receiveETH function.

Client Response

Fixed

Commit URL: https://github.com/xpet-tech/xpet-core/commit/2bb3f91986da8c4fd85afcc32d8ec60d194d844d

XPT-18:Risk that any user can use the same price in the market

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	Kong7ych3

Code Reference

- code/contracts/XpetMarket.sol#L91
- code/contracts/XpetMarket.sol#L107
- code/contracts/XpetMarket.sol#L122
- code/contracts/XpetMarket.sol#L134

```
91:bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(id, price))));

107:bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(id, price, level))));

122:bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(level, price))));

134:bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(level, price))));
```

Description

Kong7ych3 : In the XpetMarket contract, when users call the buyItem, buyPet, upPetLevel, and upFactoryLevel functions, they need to perform a Merkle verification to prove the validity of their price. However, it is important to note that the leaf does not contain any identity information associated with the user. This means that any user can use the publicly available proof to call the functions, meaning that any user can interact with the market using the same price. If this is not the intended design, it will lead to market chaos, with all users inevitably using high-level and low-cost proofs to interact with the market.

Recommendation

Kong7ych3 : If this is not the intended design, it is recommended to add user information such as `msg.sender` to the leaf to ensure that only trusted users can use this price and level.

Client Response

Acknowledged,The price on the market will be the same price for all users. We use Merkle Tree only to reduce gas each time we update the price of pets and items.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.