InterFi NETWORK

@INTERFINETWORK

SMART CONTRACT SECURITY AUDIT OF

PANDASALE

SMART CONTRACT AUDIT | SOLIDITY DEVELOPMENT & TESTING | PROJECT EVALUATION

RELENTLESSLY SECURING THE PUBLIC BLOCKCHAIN

Audit Introduction

Auditing Firm	InterFi Network
Audit Architecture	InterFi Echelon Auditing Standard
Language	Solidity
Client Firm	PandaSale
Website	https://pandasale.finance/
Telegram	https://t.me/PandaEcosystem
Twitter	https://twitter.com/Panda_ecosystem
Report Date	June 13, 2022

About PandaSale

- A diverse, user-orientated project ecosystem featuring the most accessible, efficient and user friendly AIO Launchpad Protocol for both investors and project owners.
- As a user-orientated platform, PandaSale focuses on facilitating a balanced and mutually beneficial environment for both investors and project owners.
- PandaSale not only provides the next generation, all in one DeFi Launchpad for all EVM powered networks but also integrates many other complimentary services (Marketing, KYC audit, Contests, etc.) under the same platform.
- Having many partners in the crypto space, PandaSale provides access to many discount services which raise project success rate.
- PandaSale is financially accessible and simple to use for both investors and project owners.

Audit Summary

InterFi team has performed a line-by-line manual analysis and automated review of smart contracts. Smart contracts were analyzed mainly for common contract vulnerabilities, exploits, and manipulation hacks. According to the audit:

- PandaSale's solidity source codes have **LOW RISK SEVERITY**
- PandaSale's smart contracts have **ACTIVE OWNERSHIP**
- Centralization risk correlated to the active owner is **MEDIUM**
- Important owner privileges – **WHITELIST, UPDATE LAUNCHPAD PARAMETERS, SET FEES, WITHDRAW, CLAIM LIQUIDITY**

Be aware that smart contracts deployed on the blockchain aren't resistant to internal exploit, external vulnerability, or hack. For a detailed understanding of risk severity, source code vulnerability, exploitability, and audit disclaimer, kindly refer to the audit.

🚨 Contract addresses: **Not deployed**

⛓️ Blockchain: **Not chained**

✅ Verify the authenticity of this report on InterFi's GitHub: **https://github.com/interfinetwork**

Table Of Contents

Audit Scope

InterFi was consulted by PandaSale to conduct the smart contract security audit of their solidity source codes. The audit scope of work is strictly limited to the mentioned solidity file(s) only:

- BaseSale.sol
- EventEmitter.sol
- Factory.sol
- Fairlaunch.sol
- Presale.sol
- WhiteList.sol

Solidity Source Code On GitHub

//Private Repository//

SHA-1 Hash

Solidity source codes are audited at hash #c56916d9e94dd986930df1512c3bfeeff7f6f77e

Audit Methodology

The scope of this report is to audit the smart contract source code of PandaSale. InterFi has scanned contracts and reviewed codes for common vulnerabilities, exploits, hacks, and back-doors. Due to being out of scope, InterFi has not tested contracts on testnet to assess any functional flaws. Below is the list of commonly known smart contract vulnerabilities, exploits, and hacks:

Category	
Smart Contract Vulnerabilities	❖ Re-entrancy ❖ Unhandled Exceptions ❖ Transaction Order Dependency ❖ Integer Overflow ❖ Unrestricted Action ❖ Incorrect Inheritance Order ❖ Typographical Errors ❖ Requirement Violation ❖ Gas Limit and Loops
Source Code Review	❖ Deployment Consistency ❖ Repository Consistency ❖ Data Consistency ❖ Token Supply Manipulation ❖ Access Control and Authorization ❖ Operations Trail and Event Generation ❖ Assets Manipulation ❖ Ownership Control ❖ Liquidity Access

InterFi's Echelon Audit Standard

The aim of InterFi's "Echelon" standard is to analyze smart contracts and identify the vulnerabilities and the hacks. Kindly note, InterFi does not test smart contracts on testnet. It is recommended that smart contracts are thoroughly tested prior to the audit submission. Mentioned are the steps used by InterFi to audit smart contracts:

1. Solidity smart contract source code reviewal:
   - ❖ Review of the specifications, sources, and instructions provided to InterFi to make sure we understand the size, and scope of the smart contract audit.
   - ❖ Manual review of code, which is the process of reading source code line-by-line to identify potential vulnerabilities.
2. Static, Manual, and Software analysis:
   - ❖ Test coverage analysis is the process of determining whether the test cases are covering the code and how much code is exercised when we run those test cases.
   - ❖ Symbolic execution is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts

Automated 3P frameworks used to assess the smart contract vulnerabilities

- ❖ Consensys Tools
- ❖ SWC Registry
- ❖ Solidity Coverage
- ❖ Open Zeppelin Code Analyzer
- ❖ Solidity Code Complier

Risk Classification

Smart contracts are generally designed to manipulate and hold funds denominated in ETH/BNB. This makes them very tempting attack targets, as a successful attack may allow the attacker to directly steal funds from the contract. Below are the typical risk levels of a smart contract:

Vulnerable: A contract is vulnerable if it has been flagged by a static analysis tool as such. As we will see later, this means that some contracts may be vulnerable because of a false positive.

Exploitable: A contract is exploitable if it is vulnerable and the vulnerability could be exploited by an external attacker. For example, if the "vulnerability" flagged by a tool is in a function that requires owning the contract, it would be vulnerable but not exploitable.

Exploited: A contract is exploited if it received a transaction on the main network which triggered one of its vulnerabilities. Therefore, a contract can be vulnerable or even exploitable without having been exploited.

Risk severity	Meaning
! High	This level vulnerabilities could be exploited easily and can lead to asset loss, data loss, asset, or data manipulation. They should be fixed right away.
! Medium	This level vulnerabilities are hard to exploit but very important to fix, they carry an elevated risk of smart contract manipulation, which can lead to high-risk severity
! Low	This level vulnerabilities should be fixed, as they carry an inherent risk of future exploits, and hacks which may or may not impact the smart contract execution.
! Informational	This level vulnerabilities can be ignored. They are code style violations and informational statements in the code. They may not affect the smart contract execution

Centralization Risk

Centralization risk is the most common cause of decentralized finance hacks. When a smart contract has an active contract ownership, the risk related to centralization is elevated. There are some well-intended reasons to be an active contract owner, such as:

- Contract owner can be granted the power to `pause()` or `lock()` the contract in case of an external attack.
- Contract owner can use functions like, `include()`, and `exclude()` to add or remove wallets from fees, swap checks, and transaction limits. This is useful to run a presale, and to list on an exchange.

Authorizing a full centralized power to a single body can be dangerous. Unfortunately, centralization related risks are higher than common smart contract vulnerabilities. Centralization of ownership creates a risk of rug pull scams, where owners cash out tokens in such quantities that they become valueless. **Most important question to ask here is, how to mitigate centralization risk?** Here's InterFi's recommendation to lower the risks related to centralization hacks:

- Smart contract owner's private key must be carefully secured to avoid any potential hack.
- Smart contract ownership should be shared by multi-signature (multi-sig) wallets.
- Smart contract ownership can be locked in a contract, user voting, or community DAO can be introduced to unlock the ownership.

Centralization Status

- PandaSale's smart contracts have **active ownership.**
- Smart contracts are **not deployed** on blockchain at the time of the audit.

Static Analysis

Symbol	Meaning
🛑	Function can modify state
💵	Function is payable
🔒	Function is locked
🔐	Function can be accessed
!	Important functionality

BaseSale.sol

BaseSale	Implementation	Initializable, ICronosPadBaseSale, Whitelist		
└	<Constructor>	Public !	🛑	NO!
└	_initialize	Public !	🛑	initializer
└	_setFees	Public !	🛑	onlyOwner
└	updateSocialIpfsHash	External !	🛑	onlyOwner
└	contribute	External !	💵	checkWhitelist
└	getContribution	Public !		NO!
└	emergencyWithdraw	External !	🛑	NO!
└	withdrawContribution	External !	🛑	NO!
└	claimTokens	External !	🛑	NO!
└	getClaimed	Public !		NO!
└	getClaimableTokens	Public !		NO!
└	cancel	External !	🛑	NO!
└	withdrawCancelledTokens	External !	🛑	onlyOwner
└	_devClaim	Internal 🔒	🛑	
└	finalize	Public !	🛑	NO!
└	detectFeeExcluded	External !	🛑	NO!
└	withdrawVestedTokens	Public !	🛑	onlyOwner
└	getTeamClaimableTokens	Public !		NO!
└	claimLiquidity	External !	🛑	onlyOwner
└	isFinished	Public !		NO!
└	_inLimits	Internal 🔒		
└	_beforeFinalize	Internal 🔒	🛑	
└	_afterFinalize	Internal 🔒	🛑	
└	_getTokensSold	Public !		NO!
└	_getListingRate	Public !		NO!
└	_getLiquidityTokenAmount	Internal 🔒		
└	_getTokenFeeAtCap	Internal 🔒		
└	setPublicWithDelay	External !	🛑	onlyOwner

| └ | setUseWhitelist | Public ❗ | 🛑 | onlyOwner |
| └ | checkValidBps | Internal 🔒 | | |

EventEmitter.sol

EventEmitter	Implementation	Ownable		
└	<Constructor>	Public ❗	🛑	NO❗
└	authorise	Public ❗	🛑	onlyOwner
└	emitSaleCreated	External ❗	🛑	authorized
└	emitContribute	External ❗	🛑	authorized
└	emitEmergencyWithdrawn	External ❗	🛑	authorized
└	emitCancelled	External ❗	🛑	authorized
└	emitFinalized	External ❗	🛑	authorized

Factory.sol

Factory	Implementation	Ownable		
└	<Constructor>	Public ❗	🛑	validBps validBps validBps
└	updateServiceFeeReceiver	Public ❗	🛑	onlyOwner
└	updateAdmin	Public ❗	🛑	NO❗
└	updateLaunchpadRaisedFeeBps	Public ❗	🛑	onlyOwner validBps
└	updateLaunchpadUniformFeeBps	Public ❗	🛑	onlyOwner validBps
└	updateEmergencyWithdrawFineBps	Public ❗	🛑	onlyOwner validBps
└	updatePresaleProxy	Public ❗	🛑	onlyOwner
└	updateFairlaunchProxy	Public ❗	🛑	onlyOwner
└	createPresale	External ❗	💵	NO❗
└	createFairlaunch	External ❗	💵	NO❗
└	deductFixedFee	Internal 🔒	🛑	

Fairlaunch.sol

Fairlaunch	Implementation	BaseSale		
└	<Constructor>	Public ❗	🛑	BaseSale
└	initialize	Public ❗	🛑	NO❗
└	_getTokensSold	Public ❗		NO❗
└	_getListingRate	Public ❗		NO❗
└	_getTokenFeeAtCap	Internal 🔒		
└	_getLiquidityTokenAmount	Internal 🔒		
└	_devTokensRequired	External ❗		NO❗

Presale.sol

| **Presale** | Implementation | BaseSale, ICronosPadPresale |||
| └ | <Constructor> | Public ❗ | 🛑 | BaseSale |

└	initialize	Public ❗	🛑	NO❗
└	registerContest	Public ❗	🛑	onlyOwner
└	removeContest	External ❗	🛑	onlyOwner
└	updateWhitelistFromContest	Public ❗	🛑	NO❗
└	_getLiquidityTokenAmount	Internal 🔒		
└	_devTokensRequired	External ❗		NO❗
└	_getTokensSoldAtCap	Internal 🔒		
└	_getTokenFeeAtCap	Internal 🔒		
└	_getTokensSold	Public ❗		NO❗
└	_getListingRate	Public ❗		NO❗
└	_afterFinalize	Internal 🔒	🛑	

WhiteList.sol

Whitelist	Implementation	Ownable		
└	setWhitelist	Public ❗	🛑	onlyOwner
└	addInWhitelist	Public ❗	🛑	onlyOwner
└	_addInWhitelist	Internal 🔒	🛑	
└	removeFromWhitelist	Public ❗	🛑	onlyOwner
└	_setNonRemovable	Internal 🔒	🛑	
└	getWhitelist	External ❗		NO❗

Functional Sigs and Graph

BaseSale.sol

```
61588221  =>  claimLiquidity()
94e407f3  =>
_initialize(address[3],uint256[2],uint256[2],uint16,uint256,ICronosPadBaseSale.TokenVestingParams,I
CronosPadBaseSale.TeamVestingParams,string[2])
e6adbc87  =>  _setFees(FeeMode)
533ecc5b  =>  updateSocialIpfsHash(string)
c1cbbca7  =>  contribute(uint256)
21eff7fc  =>  getContribution(address)
db2e21bc  =>  emergencyWithdraw()
0d616d20  =>  withdrawContribution()
48c54b9d  =>  claimTokens()
eb46260e  =>  getClaimed(address)
1b831ead  =>  getClaimableTokens(address)
ea8a1af0  =>  cancel()
fdf467e3  =>  withdrawCancelledTokens()
0c281920  =>  _devClaim()
4bb278f3  =>  finalize()
72860b05  =>  detectFeeExcluded()
39f05521  =>  withdrawVestedTokens()
b6830b4b  =>  getTeamClaimableTokens()
7b352962  =>  isFinished()
8cc0c396  =>  _inLimits(uint256,uint256,uint256)
761fb24a  =>  _beforeFinalize()
d955768e  =>  _afterFinalize()
a9f7bd47  =>  _getTokensSold()
3c18315c  =>  _getListingRate()
b593a042  =>  _getLiquidityTokenAmount()
360ea6c4  =>  _getTokenFeeAtCap(uint256)
4dd2a56f  =>  setPublicWithDelay(uint256)
2b205dc9  =>  setUseWhitelist(bool)
27daa321  =>  checkValidBps(uint16[])
```

EventEmitter.sol

```
66e6c8af  =>  authorise(address)
39b72213  =>  emitSaleCreated(address,address,address,Type,address)
8cf30b86  =>  emitContribute(address,address,uint256)
fd429bfd  =>  emitEmergencyWithdrawn(address,address)
c1c2634c  =>  emitCancelled(address)
9a6c7aaa  =>  emitFinalized(address)
```

Factory.sol

```
50aa5778  =>  updateServiceFeeReceiver(address)
e2f273bd  =>  updateAdmin(address)
5ef1a19e  =>  updateLaunchpadRaisedFeeBps(uint16)
485b8c01  =>  updateLaunchpadUniformFeeBps(uint16)
2e5412b3  =>  updateEmergencyWithdrawFineBps(uint16)
9f2aefe1  =>  updatePresaleProxy(address)
46d0e8a0  =>  updateFairlaunchProxy(address)
10e0f22c  =>
createPresale(address[3],uint256[2],uint256[2],uint256[2],uint256[2],uint256[2],bool,ICronosPadPres
ale.RefundType,ICronosPadBaseSale.TokenVestingParams,ICronosPadBaseSale.TeamVestingParams,string[2]
,ICronosPadBaseSale.FeeMode)
fb90753b  =>
createFairlaunch(address[3],uint256,uint256,uint256,uint256[2],uint256[2],ICronosPadBaseSale.TeamVe
stingParams,string[2],ICronosPadBaseSale.FeeMode)
2662addb  =>  deductFixedFee()
```

Fairlaunch.sol

```
b3881283  =>
initialize(address[3],uint256,uint256,uint256,uint256[2],uint256[2],ICronosPadBaseSale.TeamVestingP
arams,string[2])
a9f7bd47  =>  _getTokensSold()
3c18315c  =>  _getListingRate()
360ea6c4  =>  _getTokenFeeAtCap(uint256)
b593a042  =>  _getLiquidityTokenAmount()
79f9ec75  =>  _devTokensRequired()
```

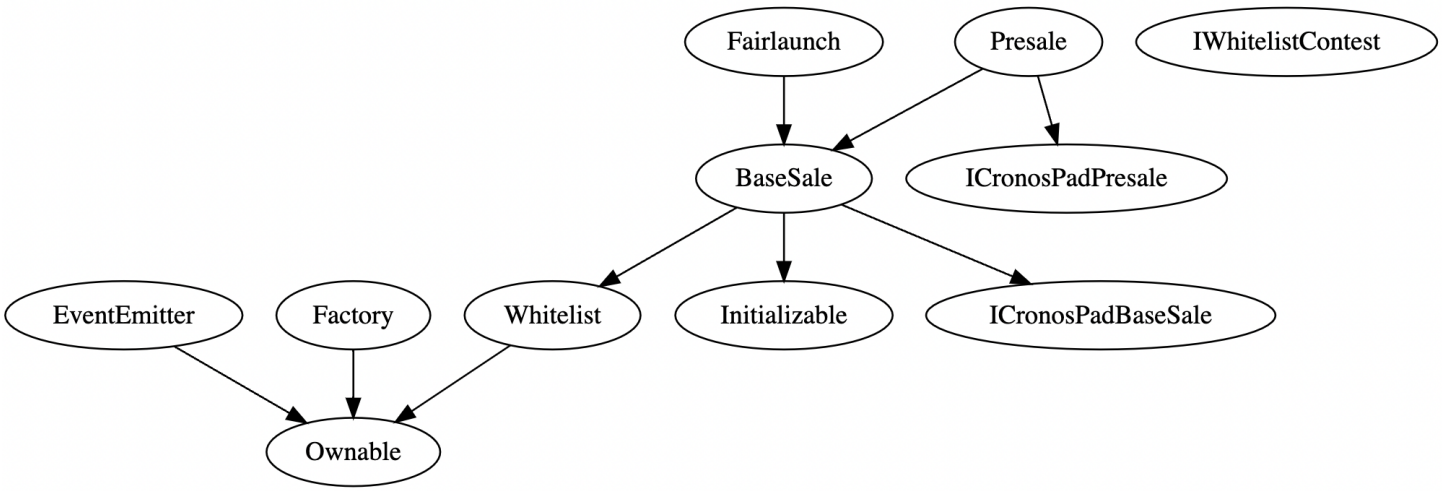
Presale.sol

```
fffe088d  =>  saleAddress()
84f52997  =>
initialize(address[3],uint256[2],uint256[2],uint256[2],uint256[2],uint256[2],ICronosPadBaseSale.Tok
enVestingParams,ICronosPadBaseSale.TeamVestingParams,RefundType,string[2])
820cdce9  =>  registerContest(address)
b434ce0f  =>  removeContest()
fdce3438  =>  updateWhitelistFromContest(address[])
b593a042  =>  _getLiquidityTokenAmount()
79f9ec75  =>  _devTokensRequired()
6f521e8a  =>  _getTokensSoldAtCap(uint256)
360ea6c4  =>  _getTokenFeeAtCap(uint256)
a9f7bd47  =>  _getTokensSold()
3c18315c  =>  _getListingRate()
d955768e  =>  _afterFinalize()
```

WhiteList.sol

```
f958a657  =>  setWhitelist(bool)
e449b436  =>  addInWhitelist(address[])
b8e5987b  =>  _addInWhitelist(address[])
548db174  =>  removeFromWhitelist(address[])
f1466946  =>  _setNonRemovable(address[])
d01f63f5  =>  getWhitelist()
```

Inheritance Graph



Manual Analysis

- WhiteList.sol smart contract owner can **add or remove whitelist** wallets to modify "write contract" parameters. There's an elevated risk of out-of-gas, and potential resource exhaustion errors with multi wallet whitelist.

```
function setWhitelist(bool enabled) public onlyOwner {
    useWhitelist = enabled;
function addInWhitelist(address[] memory _whitelist) public onlyOwner {
    _addInWhitelist(_whitelist);
function removeFromWhitelist(address[] memory _whitelist) public onlyOwner {
    for (uint i = 0; i < _whitelist.length; i++) {
        require(!isNonRemovable[_whitelist[i]], "NON_REMOVABLE");
        whitelist.remove(_whitelist[i]);
```

- Presale.sol smart contract owner can **register and remove contest.**

```
function registerContest(address _contest) public onlyOwner {
    require(
        contest == address(0) &&
            IWhitelistContest(_contest).saleAddress() == address(this),
        "INVALID"
function removeContest() external onlyOwner {
    contest = address(0);
```

- Factory.sol smart contract owner can **update presale and fairlaunch proxies.**

```
function updatePresaleProxy(address _proxy) public onlyOwner {
    presaleProxy = _proxy;
function updateFairlaunchProxy(address _proxy) public onlyOwner {
    fairlaunchProxy = _proxy;
```

- Factory.sol smart contract owner can **change token and raised fees**. This function module can be used to impose extraordinary fees.
- BaseSale.sol smart contract owner can **claim liquidity** after locking period has passed.

```
function claimLiquidity() external onlyOwner {
    require(
        finalized &&
```

```
            block.timestamp > finalizedTimestamp + liquidityLockupTime,
        "LOCKED"
    require(!liquidityClaimed, "CLAIMED");
```

- BaseSale.sol smart contract owner can **withdraw cancelled tokens** after failed sale.

```
function withdrawCancelledTokens() external onlyOwner {
    require(cancelled, "!CANCELLED");
```

- BaseSale.sol smart contract owner can **emergency withdraw contribution.** A fee pentalty is levied upon calling this function.

```
function emergencyWithdraw() external {
    require(!isFinished(), "FINISHED");
    uint fine = (amount * emergencyWithdrawFineBps) / 10**4;
```

- BaseSale.sol smart contract owner can **set fees.** This function module can be used to impose extraordinary fees.

```
function _setFees(FeeMode _mode) public override onlyOwner {
    require(!_feesSet, "FEES_SET");
```

- BaseSale.sol smart contract does not utilize **re-entrancy guard** to prevent re-entrant calls.
- Smart contract has a **low severity issue** which may or may not create any functional vulnerability.

  "severity": 8, (! Low Severity)

  “Floating Pragma”

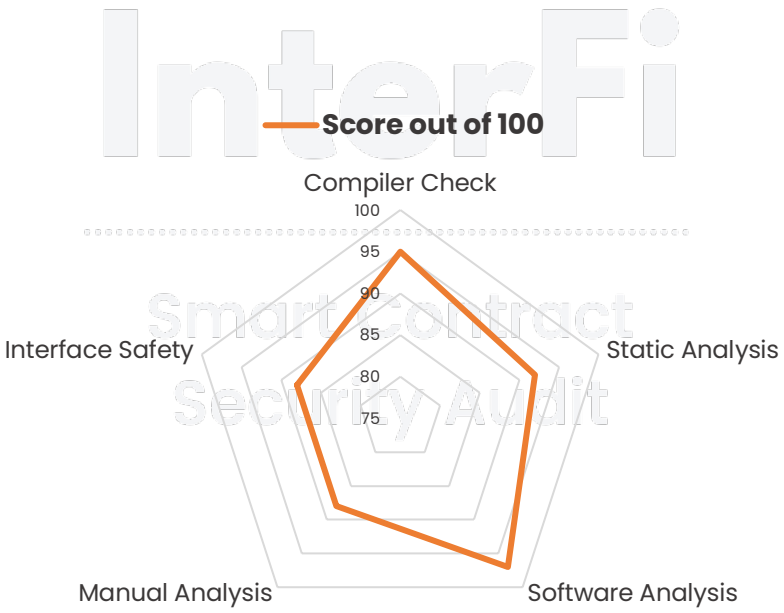
SWC Attacks

SWC ID	Description	Status
SWC-101	Integer Overflow and Underflow	**Passed**
SWC-102	Outdated Compiler Version	**! Informational**
SWC-103	Floating Pragma	**! Low**
SWC-104	Unchecked Call Return Value	**Passed**
SWC-105	Unprotected Ether Withdrawal	**Passed**
SWC-106	Unprotected SELF-DESTRUCT Instruction	**Passed**
SWC-107	Re-entrancy	**! Low**
SWC-108	State Variable Default Visibility	**Passed**
SWC-109	Uninitialized Storage Pointer	**Passed**
SWC-110	Assert Violation	**Passed**
SWC-111	Use of Deprecated Solidity Functions	**Passed**
SWC-112	Delegate Call to Untrusted Callee	**Passed**
SWC-113	DoS with Failed Call	**Passed**
SWC-114	Transaction Order Dependence	**Passed**
SWC-115	Authorization through tx.origin	**Passed**
SWC-116	Block values as a proxy for time	**Passed**
SWC-117	Signature Malleability	**Passed**
SWC-118	Incorrect Constructor Name	**Passed**

SWC-119	Shadowing State Variables	**Passed**
SWC-120	Weak Sources of Randomness from Chain Attributes	**Passed**
SWC-121	Missing Protection against Signature Replay Attacks	**Passed**
SWC-122	Lack of Proper Signature Verification	**Passed**
SWC-123	Requirement Violation	**Passed**
SWC-124	Write to Arbitrary Storage Location	**Passed**
SWC-125	Incorrect Inheritance Order	**Passed**
SWC-126	Insufficient Gas Griefing	**Passed**
SWC-127	Arbitrary Jump with Function Type Variable	**Passed**
SWC-128	DoS With Block Gas Limit	**Passed**
SWC-129	Typographical Error	**Passed**
SWC-130	Right-To-Left-Override control character (U+202E)	**Passed**
SWC-131	Presence of unused variables	**Passed**
SWC-132	Unexpected Ether balance	**Passed**
SWC-133	Hash Collisions With Multiple Variable Length Arguments	**Passed**
SWC-134	Message call with the hardcoded gas amount	**Passed**
SWC-135	Code With No Effects (Irrelevant/Dead Code)	**Passed**
SWC-136	Unencrypted Private Data On-Chain	**Passed**

Risk Status & Radar Chart

Risk Severity	Status
High	No high severity issues identified
Medium	No medium severity issues identified
Low	2 low severity issues identified
Informational	1 informational severity issue identified
Centralization Risk	Active contract ownership identified



Auditor's Verdict

InterFi team has performed a line-by-line manual analysis and automated review of smart contracts. Smart contracts were analyzed mainly for common contract vulnerabilities, exploits, and manipulation hacks. According to the audit:

- PandaSale's solidity source codes have **LOW RISK SEVERITY**
- PandaSale's smart contracts have **ACTIVE OWNERSHIP**
- Centralization risk correlated to the active owner is **MEDIUM**

Note for stakeholders

- Be aware that active smart contract owner privileges constitute an elevated impact on smart contract safety and security.
- If the smart contract is not deployed on any blockchain at the time of the audit, the contract can be modified or altered before blockchain development. Verify contract's deployment status in the audit report.
- Make sure that the project team's KYC/identity is verified by an independent firm.
- Always check if the contract's liquidity is locked. A longer liquidity lock plays an important role in the project's longevity. It is recommended to have multiple liquidity providers.
- Examine the unlocked token supply in the owner, developer, or team's private wallets. Understand the project's tokenomics, and make sure the tokens outside of the LP Pair are vested or locked for a longer period.

Important Disclaimer

InterFi Network provides contract development, testing, auditing and project evaluation services for blockchain projects. The purpose of the audit is to analyze the on-chain smart contract source code and to provide a basic overview of the project. **This report should not be transmitted, disclosed, referred to, or relied upon by any person for any purpose without InterFi's prior written consent.**

InterFi provides the easy-to-understand assessment of the project, and the smart contract (otherwise known as the source code). The audit makes no statements or warranties on the security of the code. It also cannot be considered as enough assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have used all the data at our disposal to provide the transparent analysis, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. **Be aware that smart contracts deployed on a blockchain aren't resistant to external vulnerability, or a hack. Be aware that active smart contract owner privileges constitute an elevated impact on smart contract safety and security. Therefore, InterFi does not guarantee the explicit security of the audited smart contract.**

The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed.

This report should not be considered as an endorsement or disapproval of any project or team. The information provided in this report does not constitute investment advice, financial advice, trading advice, or any other sort of advice and you should not treat any of the report's content as such. Do conduct your due diligence and consult your financial advisor before making any investment decisions.

About InterFi Network

InterFi Network provides intelligent blockchain solutions. InterFi is developing an ecosystem that is seamless and responsive. Some of our services: Blockchain Security, Token Launchpad, NFT Marketplace, etc. **InterFi's mission is to interconnect multiple services like Blockchain Security, DeFi, Gaming, and Marketplace under one ecosystem that is seamless, multi-chain compatible, scalable, secure, fast, responsive, and easy to use.**

InterFi is built by a decentralized team of UI experts, contributors, engineers, and enthusiasts from all over the world. Our team currently consists of 6+ core team members, and 10+ casual contributors. **InterFi provides manual, static, and automatic smart contract analysis, to ensure that project is checked against known attacks and potential vulnerabilities.**

To learn more, visit **https://interfi.network**

To view our audit portfolio, visit **https://github.com/interfinetwork**

To book an audit, message **https://t.me/interfiaudits**

@INTERFINETWORK

RELENTLESSLY SECURING THE PUBLIC BLOCKCHAIN | MADE IN CANADA