



AtomPad

Smart Contract Security Audit

Prepared by ShellBoxes

April 10th, 2023 - April 13th, 2023

[Shellboxes.com](https://shellboxes.com)

contact@shellboxes.com

Document Properties

Client	AtomPad
Version	1.0
Classification	Public

Scope

Contract Name	Contract Address
PresaleInternal	0x97363b2c94552246912209CDf77241D0b1AFfFdB

Re-Audit

Contract Name	Contract Address
PresaleInternal	0xe5C2A47545F1f7B2e588B66d9E7cEA33883e01fA

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1	Introduction	5
1.1	About AtomPad	5
1.2	Approach & Methodology	5
1.2.1	Risk Methodology	6
2	Findings Overview	7
2.1	Summary	7
2.2	Key Findings	7
3	Finding Details	9
SHB.1	Rounding Error In The Swapped Token Amount	9
SHB.2	Lost Precision Due To A Division Before Multiplication	12
SHB.3	Mismatch In Allocation Calculation Between <code>getUserAllocated</code> And <code>_swap</code> Functions	14
SHB.4	The Contract Is Not Guaranteed To Have Funds For Vesting Payments	16
SHB.5	Potential Vesting Disruption In <code>returnWantTokens</code> Function	18
SHB.6	Potential Vesting Disruption With Setter Functions	19
SHB.7	Centralization Risk	21
SHB.8	Unchecked Transfer Calls	22
SHB.9	Missing Value and Address Verification	25
SHB.10	Renounce Ownership Risk	28
4	Best Practices	30
BP.1	Remove Unnecessary Checks	30
BP.2	Use Pre-increment (i.e., <code>++i</code>) in for Loops	31
BP.3	Use Custom Solidity Errors with <code>if</code> and <code>revert</code> Instead of <code>require</code> Statements	32
5	Tests	33
6	Conclusion	34
7	Scope Files	35
7.1	Audit	35
7.2	Re-Audit	35

1 Introduction

AtomPad engaged ShellBoxes to conduct a security assessment on the AtomPad beginning on April 10th, 2023 and ending April 13th, 2023. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About AtomPad

AtomPad is a multichain launchpad, focused on secure and faultless project launches, which grants token stakers exclusive access to pre-sales of projects which have been carefully selected for their launchpad. AtomPad is deployed on Binance Smart Chain and provides a platform with multichain support.

Issuer	AtomPad
Website	https://www.atompad.io
Type	Solidity Smart Contract
Documentation	AtomPad Gitbook docs
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the AtomPad implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include , 2 high-severity, 5 medium-severity, 3 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Rounding Error In The Swapped Token Amount	HIGH	Fixed
SHB.2. Lost Precision Due To A Division Before Multiplication	HIGH	Fixed
SHB.3. Mismatch In Allocation Calculation Between <code>getUserAllocated</code> And <code>_swap</code> Functions	MEDIUM	Fixed
SHB.4. The Contract Is Not Guaranteed To Have Funds For Vesting Payments	MEDIUM	Acknowledged
SHB.5. Potential Vesting Disruption In <code>returnWantTokens</code> Function	MEDIUM	Acknowledged
SHB.6. Potential Vesting Disruption With Setter Functions	MEDIUM	Acknowledged
SHB.7. Centralization Risk	MEDIUM	Acknowledged
SHB.8. Unchecked Transfer Calls	LOW	Fixed

SHB.9. Missing Value and Address Verification	LOW	Partially Fixed
SHB.10. Renounce Ownership Risk	LOW	Acknowledged

3 Finding Details

SHB.1 Rounding Error In The Swapped Token Amount

- Severity: **HIGH**
- Likelihood: 3
- Status: Fixed
- Impact: 2

Description:

The `getTokenAmount` function calculates the token amount based on the `_amount`, `iDecimals`, and `wDecimals` values. However, there is a rounding error in the returned value that may cause a loss of tokens for the user.

The `getTokenAmount` function has a rounding error in its calculation, which can lead to a loss of tokens for the user, potentially up to $10^{18-wDecimals}$ tokens whenever `_amount * rate` is lower than 10^{18} or `_amount * rate % 1018` is different from zero.

Files Affected:

SHB.1.1: PresaleBase.sol

```
225 function _swap(  
226     address _from,  
227     uint256 _amount,  
228     uint256 _perc,  
229     uint256 _iDecimals  
230 ) private {  
231     uint256 _allocation = (hardCap * _perc);  
232  
233     if (_iDecimals > 6)  
234         _allocation = _allocation * (10 ** (_iDecimals - 6));  
235     if (_iDecimals < 6) _allocation = _allocation / (10 ** (_iDecimals))  
        ↪ ;  
236
```

```

237     uint256 _swapped = swaps[_from];
238
239     uint256 _remaining = _allocation - _swapped;
240
241     require(_remaining >= _amount, "Presale: Insufficient allocation");
242
243     swaps[_from] += _amount;
244
245     claims[_from].reserved += getTokenAmount(_amount);
246
247     swapTotal += _amount;
248
249     emit Swapped(msg.sender, _amount);
250 }

```

SHB.1.2: PresaleBase.sol

```

157 function getTokenAmount(uint256 _amount) public view returns (uint256) {
158     uint256 iDecimals = investToken.decimals;
159     uint256 wDecimals = wantToken.decimals;
160
161     if (iDecimals != wDecimals) {
162         _amount = _amount / 10 ** (iDecimals);
163         _amount = _amount * 10 ** (wDecimals);
164     }
165
166     return (_amount * rate) / (10 ** 18);
167 }

```

Recommendation:

To address this issue, consider requiring the `_amount * rate % 1018` to be equal to zero before performing the swap.

Updates

The AtomPad team resolved the issue by adding a `require` statement to prevent rounding errors.

SHB.1.3: PresaleBase.sol

```
44 function swap(  
45     uint256 _amount  
46 )  
47     external  
48     nonReentrant  
49     whenNotPaused  
50     swapEnabled  
51     onProgress  
52     returns (bool)  
53 {  
54     uint256 _perc = allocPercentageOf(msg.sender);  
55  
56     uint256 _swapTotalAfter = swapTotal + _amount;  
57  
58     Token memory _investToken = investToken;  
59  
60     require(_perc > 0, "Presale: No allocation");  
61  
62     require(  
63         _swapTotalAfter <= hardCap * (10 ** (_investToken.decimals)),  
64         "Presale: Hard cap reached"  
65     );  
66  
67     require(  
68         (_amount * rate) % (10 ** 18) == 0,  
69         "Presale: Swap not allowed due to potential rounding errors."  
70     );
```

SHB.2 Lost Precision Due To A Division Before Multiplication

- Severity: **HIGH**
- Likelihood: 3
- Status: Fixed
- Impact: 2

Description:

The `getTokenAmount` function performs a division operation before multiplication, which may result in significant precision loss, leading to inaccuracies in the calculated token amounts.

In the `getTokenAmount` function, the division operation is performed prior to multiplication when adjusting the `_amount` value based on `iDecimals` and `wDecimals`. This ordering can cause significant precision loss, negatively affecting the accuracy of the calculated token amounts.

Files Affected:

SHB.2.1: PresaleBase.sol

```
157 function getTokenAmount(uint256 _amount) public view returns (uint256) {
158     uint256 iDecimals = investToken.decimals;
159     uint256 wDecimals = wantToken.decimals;
160
161     if (iDecimals != wDecimals) {
162         _amount = _amount / 10 ** (iDecimals);
163         _amount = _amount * 10 ** (wDecimals);
164     }
165
166     return (_amount * rate) / (10 ** 18);
167 }
```

Recommendation:

To resolve this issue, consider reordering the operations by performing the multiplication first, followed by the division. This approach will help to minimize precision loss and maintain the accuracy of the token amount calculations. The issue can be resolved by calculating the `_amount` using the following code:

SHB.2.2: PresaleBase.sol

```
_amount = _amount * 10 ** (wDecimals-iDecimals);
```

Updates

The AtomPad team resolved the issue by performing the multiplication operation before the division.

SHB.2.3: PresaleBase.sol

```
125 function getTokenAmount(  
126     uint256 investAmount  
127 ) public view returns (uint256) {  
128     uint256 iDecimals = investToken.decimals;  
129     uint256 wDecimals = wantToken.decimals;  
130  
131     //rate: 18 decimals  
132     uint256 wantAmount = investAmount * rate;  
133  
134     if (iDecimals < wDecimals) {  
135         uint256 decimalDifference = 10 ** (wDecimals - iDecimals);  
136         wantAmount = wantAmount * decimalDifference;  
137     }  
138  
139     if (iDecimals > wDecimals) {  
140         uint256 decimalDifference = 10 ** (iDecimals - wDecimals);  
141         wantAmount = wantAmount / decimalDifference;  
142     }  
143  
144     return wantAmount / (10 ** 18);
```

SHB.3 Mismatch In Allocation Calculation Between `getUserAllocated` And `_swap` Functions

- Severity: **MEDIUM**
- Likelihood: 2
- Status: Fixed
- Impact: 2

Description:

There is an inconsistency in the allocation calculation logic between the `getUserAllocated` function and the `_swap` function. The `getUserAllocated` function calculates the basic allocation using `hardCap * _perc`, and then it adjusts the allocation based on `_iDecimals`. However, in the `_swap` function, the allocation adjustment logic differs and includes a case for `_iDecimals < 6`.

Files Affected:

SHB.3.1: PresaleBase.sol

```

194 function getUserAllocated(address _wallet) external view returns (
    ↪ uint256) {
195     uint256 _iDecimals = investToken.decimals;
196     /// retrieve absolute amount of remaining allocation for this;
197     uint256 _perc = allocPercentageOf(_wallet);
198
199     /// retrieve basic allocation
200     uint256 _allocate = (hardCap * _perc);
201
202     if (_iDecimals > 6) _allocate = _allocate * (10 ** (_iDecimals - 6))
    ↪ ;
203

```

```

204     /// check to avoid < 0 error
205     if (_allocate <= swaps[_wallet]) return 0;
206
207     /// returns remaining allocation
208     return (_allocate - swaps[_wallet]);
209 }

```

SHB.3.2: PresaleBase.sol

```

225 function _swap(
226     address _from,
227     uint256 _amount,
228     uint256 _perc,
229     uint256 _iDecimals
230 ) private {
231     uint256 _allocation = (hardCap * _perc);
232
233     if (_iDecimals > 6)
234         _allocation = _allocation * (10 ** (_iDecimals - 6));
235     if (_iDecimals < 6) _allocation = _allocation / (10 ** (_iDecimals))
236         ↔ ;
237
238     uint256 _swapped = swaps[_from];
239
240     uint256 _remaining = _allocation - _swapped;
241
242     require(_remaining >= _amount, "Presale: Insufficient allocation");
243
244     swaps[_from] += _amount;
245
246     claims[_from].reserved += getTokenAmount(_amount);
247
248     swapTotal += _amount;
249
250     emit Swapped(msg.sender, _amount);

```

```
250 }
```

Recommendation:

To fix this issue, ensure that both the `getUserAllocated` and `_swap` functions have consistent calculation logic for determining the allocation. This will prevent discrepancies in the allocated amounts and ensure accurate allocation values across the smart contract.

Updates

The AtomPad team resolved the issue by removing the `_iDecimals < 6` case in the `_swap` function.

SHB.3.3: PresaleBase.sol

```
192 function _swap(  
193     address _from,  
194     uint256 _amount,  
195     uint256 _perc,  
196     uint256 _iDecimals  
197 ) private {  
198     uint256 _allocation = (hardCap * _perc);  
199  
200     if (_iDecimals > 6)  
201         _allocation = _allocation * (10 ** (_iDecimals - 6));  
202  
203     uint256 _swapped = swaps[_from];
```

SHB.4 The Contract Is Not Guaranteed To Have Funds For Vesting Payments

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Acknowledged
- Impact: 3

Description:

The smart contract does not guarantee the availability of sufficient funds in the `wantToken` to fulfill vested amounts when users claim their tokens.

The current implementation of the smart contract does not ensure that there are enough funds in the `wantToken` balance to cover the vested amounts when users claim their tokens. This may result in users being unable to receive their full vested token amounts upon claiming.

Files Affected:

SHB.4.1: PresaleBase.sol

```
282  /// extra check2 to avoid overspending
283  if (claims[_from].claimed > claims[_from].reserved) {
284      // we are overspending here!!! revert
285      claims[_from].claimed -= _amount;
286  } else {
287      // transfer tokens to the investor
288      IERC20(wantToken.token).safeTransfer(_from, _amount);
289  }
```

Recommendation:

To address this issue, consider implementing safeguards within the smart contract to guarantee that the `wantToken` balance is sufficient to cover all vested amounts. This may include checks or restrictions during the token allocation process, ensuring that tokens are reserved for vesting payouts and preventing any withdrawals that would cause an insufficient balance for vested claims.

Updates

The AtomPad team acknowledged the risk stating that they support projects even at seed sale stages, and at that stage projects' tokens/coins are still under audit process. Therefore, the process is not automated to maintain flexibility.

SHB.5 Potential Vesting Disruption In `returnWantTokens` Function

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Acknowledged
- Impact: 3

Description:

The `returnWantTokens` function allows the contract owner to withdraw `wantToken` balances, which may disrupt the vesting process if a portion or all of the vested amounts are withdrawn.

The current implementation of the `returnWantTokens` function permits the contract owner to withdraw the entire `wantToken` balance held in the smart contract without considering the vested amounts reserved for users. This withdrawal can potentially disrupt the vesting process, leaving users unable to claim their vested tokens.

Files Affected:

SHB.5.1: PresaleBase.sol

```
319 function returnWantTokens() external onlyOwner {
320     IERC20 _wantToken = IERC20(wantToken.token);
321     //
322     // do some checks
323     require(
324         _wantToken.balanceOf(address(this)) > 0,
325         "Presale: Nothing to return"
326     );
327
328     uint256 _remaining = _wantToken.balanceOf(address(this));
329
330     _wantToken.transfer(msg.sender, _remaining);
331
```

```
332     /// set total supply
333     tokenSupply = 0;
334
335     emit WantTokensReturned(msg.sender, _remaining);
336 }
```

Recommendation:

To mitigate this issue, consider implementing a mechanism within the `returnWantTokens` function to ensure that any withdrawal by the contract owner does not affect the vested amounts reserved for users. This can be achieved by tracking the total vested balance separately and only allowing the contract owner to withdraw amounts in excess of the vested balance. This approach will protect the vested amounts and ensure users can successfully claim their tokens during the vesting period.

Updates

The AtomPad team acknowledged the issue stating that the launchpad is not automated, so they get permission to return want tokens even in the cases where projects try to do malicious activities to give users tokens, and moderate settings.

SHB.6 Potential Vesting Disruption With Setter Functions

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Acknowledged
- Impact: 3

Description:

The `setVest`, `setInvestToken`, and `setWantToken` functions can be called by the contract owner when a vesting is active, potentially disrupting the vesting process.

The current implementation of the `setVest`, `setInvestToken`, and `setWantToken` functions allows the contract owner to modify the vesting parameters, `invest` token, and `want` token,

respectively, without any restrictions during an active vesting period. This can result in the vesting process being disrupted, negatively affecting users participating in the vesting.

Files Affected:

SHB.6.1: PresaleBase.sol

```
395 function setVest(Vest memory _vest) external onlyOwner {
396     vest = _vest;
397     vest.duration.end =
398         _vest.duration.start +
399         (_vest.durationPerVest * _vest.noOfVests);
400
401     emit VestUpdated(msg.sender, _vest);
402 }
```

SHB.6.2: PresaleBase.sol

```
411 function setInvestToken(Token memory _investToken) external onlyOwner {
412     /// check this is a valid address
413     require(
414         _investToken.token != address(0),
415         "Presale: Invalid token address"
416     );
417     require(_investToken.decimals != 0, "Presale: Invalid token decimals
418         ↔ ");
419
420     investToken = _investToken;
421
422     emit InvestTokenUpdated(msg.sender, _investToken.token);
423 }
```

SHB.6.3: PresaleBase.sol

```
424 function setWantToken(Token memory _wantToken) external onlyOwner {
425     /// check this is a valid address
426     require(
```

```

427     _wantToken.token != address(0),
428     "Presale: Invalid token address"
429 );
430 require(_wantToken.decimals != 0, "Presale: Invalid token decimals")
    ↔ ;
431
432 wantToken = _wantToken;
433
434 emit WantTokenUpdated(msg.sender, _wantToken.token);
435 }

```

Recommendation:

To prevent this issue, consider implementing checks within these setter functions to ensure that they can only be called when no active vesting is taking place. By adding such checks, the smart contract can prevent unwanted modifications to the vesting parameters or tokens and protect the vesting process for users.

Updates

The AtomPad team acknowledged the issue, stating that this feature is there to allow projects to issue a new token when they find a security issue in their token.

SHB.7 Centralization Risk

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Acknowledged
- Impact: 3

Description:

The contract has a lot of owner-controlled functions that can modify contract behavior, such as changing the rate, hard cap, and vesting schedule. This introduces a level of centralization that might lead to misuse or abuse of power.

Files Affected:

All functions with the `onlyOwner` modifier.

Recommendation:

To address this issue, it's important to implement more decentralized and democratic approaches to decision-making, such as multi-signature control or community governance models that distribute power more evenly.

Updates

The AtomPad team acknowledged the risk stating that the launchpad does not claim to be permissionless and should be trusted by its community. However, the user should be aware of the risk associated with trusting a third party that has centralized control over the project.

SHB.8 Unchecked Transfer Calls

- Severity: **LOW**
- Status: Fixed
- Likelihood: 1
- Impact: 2

Description:

The smart contract contains `transfer` calls where the return value is not checked to confirm if the transfer was successful, potentially leading to unexpected behavior or loss of funds.

The current implementation of the smart contract includes `transfer` calls without verifying the return value to ensure that the transfer was successful. Failing to check the return value can result in unexpected behavior if the transfer fails silently without throwing an exception.

Files Affected:

SHB.8.1: PresaleBase.sol

```
301 function depositTokens(uint256 _amount) external onlyOwner {
302     Token memory _token = wantToken;
303     IERC20 _wantToken = IERC20(_token.token);
304     /// @dev set minimum amount of tokens for this presale
305     require(
306         _amount >= (10 * 10 ** _token.decimals),
307         "Presale: Min amount is 10 tokens"
308     );
309     /// transfer x amount of wantToken to presale
310     _wantToken.transferFrom(msg.sender, address(this), _amount);
311
312     /// set total supply
313     tokenSupply += _amount;
314
315     // rate = (tokenSupply / hardCap);
316     emit Deposited(msg.sender, _amount);
317 }
```

SHB.8.2: PresaleBase.sol

```
319 function returnWantTokens() external onlyOwner {
320     IERC20 _wantToken = IERC20(wantToken.token);
321     //
322     // do some checks
323     require(
324         _wantToken.balanceOf(address(this)) > 0,
325         "Presale: Nothing to return"
326     );
327
328     uint256 _remaining = _wantToken.balanceOf(address(this));
329
330     _wantToken.transfer(msg.sender, _remaining);
331
332     /// set total supply
```

```

333     tokenSupply = 0;
334
335     emit WantTokensReturned(msg.sender, _remaining);
336 }

```

SHB.8.3: PresaleBase.sol

```

338 function forwardInvestTokens() external onlyOwner {
339     IERC20 _investToken = IERC20(investToken.token);
340     //
341     /// do some checks
342     require(
343         _investToken.balanceOf(address(this)) > 0,
344         "Presale: Not enough tokens"
345     );
346
347     uint256 _invested = _investToken.balanceOf(address(this));
348
349     _investToken.transfer(msg.sender, _invested);
350
351     emit InvestTokensForwarded(msg.sender, _invested);
352 }

```

Recommendation:

To address this issue, consider updating the transfer calls to include a check for the return value. This can be done by either wrapping the transfer calls in a [require](#) statement or using the [SafeERC20](#) library that ensures the transfer is successful and reverts the transaction if the transfer fails. This approach will help guarantee that transfers are completed successfully and prevent potential issues resulting from unchecked transfer calls.

Updates

The AtomPad team resolved the issue by using the [SafeERC20](#) function to perform transfers.

SHB.9 Missing Value and Address Verification

- Severity: **LOW**
- Likelihood: 1
- Status: Partially Fixed
- Impact: 2

Description:

The **constructor** and the setters for the **PresaleBase** contract are missing value and address verification checks for their arguments, which may lead to unintended consequences or vulnerabilities.

The **constructor** and the setters for the **PresaleBase** contract currently do not include any validation checks for the provided values and addresses of the input arguments, such as **_metadata**, **_rate**, **_hardCap**, **_investToken**, **_wantToken**, **_saleDuration**, and **_vest**. As a result, this lack of validation may lead to unintended consequences or vulnerabilities within the contract.

Files Affected:

SHB.9.1: PresaleBase.sol

```
46 constructor(  
47     Metadata memory _metadata,  
48     uint256 _rate,  
49     uint256 _hardCap,  
50     Token memory _investToken,  
51     Token memory _wantToken,  
52     Duration memory _saleDuration,  
53     Vest memory _vest  
54 ) {  
55     metadata = _metadata;  
56     vest = _vest;  
57     rate = _rate;  
58     hardCap = _hardCap;
```

```

59     duration = _saleDuration;
60     investToken = _investToken;
61     wantToken = _wantToken;
62     swapOn = true;
63     fcfsPercentage = 100;
64     vest.duration.end =
65         _vest.duration.start +
66         (_vest.durationPerVest * _vest.noOfVests);
67 }

```

SHB.9.2: PresaleBase.sol

```

368 function setHardCap(uint256 _cap) external onlyOwner {
369     hardCap = _cap;
370
371     emit HardCapUpdated(msg.sender, _cap);
372 }

```

SHB.9.3: PresaleBase.sol

```

374 function setRate(uint256 _rate) external onlyOwner {
375     rate = _rate;
376
377     emit RateUpdated(msg.sender, _rate);
378 }

```

SHB.9.4: PresaleBase.sol

```

380 function setVestDuration(Duration memory _vestDuration) external
    ↪ onlyOwner {
381     vest.duration = _vestDuration;
382     vest.duration.end =
383         vest.duration.start +
384         (vest.durationPerVest * vest.noOfVests);
385
386     emit VestDurationUpdated(msg.sender, _vestDuration);
387 }

```

SHB.9.5: PresaleBase.sol

```
389 function setSaleTime(Duration memory _saleDuration) external onlyOwner {
390     duration = _saleDuration;
391
392     emit SaleTimeUpdated(msg.sender, _saleDuration);
393 }
```

SHB.9.6: PresaleBase.sol

```
395 function setVest(Vest memory _vest) external onlyOwner {
396     vest = _vest;
397     vest.duration.end =
398         _vest.duration.start +
399         (_vest.durationPerVest * _vest.noOfVests);
400
401     emit VestUpdated(msg.sender, _vest);
402 }
```

SHB.9.7: PresaleBase.sol

```
411 function setInvestToken(Token memory _investToken) external onlyOwner {
412     /// check this is a valid address
413     require(
414         _investToken.token != address(0),
415         "Presale: Invalid token address"
416     );
417     require(_investToken.decimals != 0, "Presale: Invalid token decimals
418         ↔ ");
419
420     investToken = _investToken;
421
422     emit InvestTokenUpdated(msg.sender, _investToken.token);
423 }
```

SHB.9.8: PresaleBase.sol

```
424 function setWantToken(Token memory _wantToken) external onlyOwner {
```

```

425     /// check this is a valid address
426     require(
427         _wantToken.token != address(0),
428         "Presale: Invalid token address"
429     );
430     require(_wantToken.decimals != 0, "Presale: Invalid token decimals")
431         ↪ ;
432
433     wantToken = _wantToken;
434
435     emit WantTokenUpdated(msg.sender, _wantToken.token);
436 }

```

Recommendation:

To address this issue, consider implementing validation checks for these input arguments within the constructor and the setters. This may include checking for non-zero values for parameters like `_rate` and `_hardCap`, ensuring valid token addresses for `_investToken` and `_wantToken`, and verifying that the duration and vesting parameters are within acceptable ranges. Adding these validation checks will enhance the robustness and security of the smart contract.

Updates

The AtomPad team partially resolved the issue by implementing input verification in the `setHardCap`, `setRate`, `setVestDuration`, `setSaleTime` functions.

SHB.10 Renounce Ownership Risk

- Severity: **LOW**
- Likelihood: 1
- Status: Acknowledged
- Impact: 2

Description:

The contract inherits from the `Ownable` pattern, which includes a `renounceOwnership` function. This function, if called, can result in the contract having no owner, causing a Denial of Service (DoS) for the functions with the `onlyOwner` modifier.

In the current implementation, the contract is ownable, and the `renounceOwnership` function allows the contract owner to permanently relinquish ownership. If the ownership is renounced, the contract will not have an owner, and any function with the `onlyOwner` modifier will become unreachable. This scenario could lead to a Denial of Service (DoS) on these functions, as no one would be able to execute them, effectively rendering them useless.

Files Affected:

SHB.10.1: .sol

```
16 contract PresaleBase is PresaleStorage, Ownable, Pausable,  
    ↪ ReentrancyGuard {
```

Recommendation:

To mitigate this risk, consider either removing the `renounceOwnership` function or replacing it with a safer alternative, such as allowing ownership transfer to a predefined address, like a multisig wallet or a timelock contract. This approach will maintain control over the contract and prevent a potential DoS on the functions with the `onlyOwner` modifier.

Updates

The AtomPad team acknowledged the risk stating that the `renounceOwnership` will be used after the sale ends.

4 Best Practices

BP.1 Remove Unnecessary Checks

Description:

The current implementation of the contract contains multiple checks that may not be necessary, as the conditions they validate should always hold true. These additional checks can make the code more complex and harder to read, and they can also increase gas costs for contract interactions.

By removing unnecessary checks, you can make the smart contract code more concise, easier to understand, and more efficient in terms of gas usage. This will contribute to a more maintainable and reliable smart contract.

Files Affected:

The following checks are not necessary, as if the statement is false, the transaction will revert due to the overflow protection.

BP.1.1: PresaleBase.sol

```
264 require(claims[_from].claimed < _released, "Presale: Nothing to claim");
```

BP.1.2: PresaleBase.sol

```
271 require(tokenSupply >= _amount, "Presale: Insufficient token supply");
```

The `if` statement here is unnecessary as `_amount` is equal to `_released - claims[_from].claimed`, therefore by incrementing the `claimed` attribute the `if` statement will never be reached and it will always execute the transfer.

BP.1.3: PresaleBase.sol

```
283 if (claims[_from].claimed > claims[_from].reserved) {  
284     // we are overspending here!!! revert  
285     claims[_from].claimed -= _amount;  
286 } else {  
287     // transfer tokens to the investor
```

```
288     IERC20(wantToken.token).safeTransfer(_from, _amount);
289 }
```

Status - Not Fixed

BP.2 Use Pre-increment (i.e., ++i) in for Loops

Description:

In Solidity, it is generally recommended to use `++i` (pre-increment) instead of `i++` (post-increment) in for loops. The reason for this preference is that `++i` can be slightly more efficient in terms of gas usage.

When using `++i`, the value of `i` is incremented before it is used in the loop. In contrast, when using `i++`, the value of `i` is incremented after it is used. In some programming languages, the post-increment operation may create a temporary variable to store the original value before incrementing it, which can result in additional overhead.

However, it is worth noting that modern Solidity compilers like the one in the solc [0.8.x](#) series are optimized to handle both `++i` and `i++` efficiently, so the difference in gas usage between the two may be negligible. Nonetheless, it is still a good practice to use `++i` in for loops to ensure optimal gas efficiency, especially when working with older compilers or in cases where the optimizations may not be applied.

Files Affected:

BP.2.1: PresaleBase.sol

```
183 for (uint256 i = 0; i < _vest.noOfVests; i++) {
184     if (_time > _startVest + (i * (_vest.durationPerVest)))
185         _perc = (_vest.initialVestPercentage +
186                 (i * (_vest.percPerVest)));
187 }
```

Status - Not Fixed

BP.3 Use Custom Solidity Errors with `if` and `revert` Instead of `require` Statements

Description:

In the current implementation, the contract uses `require` statements for various validation checks. While this approach works, using custom Solidity errors with `if` and `revert` statements can provide more informative and specific error messages. This makes it easier for developers and users to understand the reasons behind failed transactions, and it allows for better error handling.

To implement this best practice, consider replacing the existing `require` statements with `if` and `revert` statements that include custom error messages. Define custom error types using the `error` keyword, and provide descriptive names and parameters to convey the nature of the error. Then, use these custom error types in combination with `revert` statements in your validation checks.

Status - Not Fixed

5 Tests

Because the project lacks unit, integration, and end-to-end tests, we recommend establishing numerous testing methods covering multiple scenarios for all features in order to ensure the correctness of the smart contracts.

6 Conclusion

In this audit, we examined the design and implementation of AtomPad contract and discovered several issues of varying severity. AtomPad team addressed 4 issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised AtomPad Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
PresaleBase.sol	6124a3085b0a95d43245e4793bbeb292
PresaleInternal.sol	de0910092bf79a732501ec8023e374e5
PresaleStorage.sol	892def72c5abdea2e3df20379e1fc93a

7.2 Re-Audit

Files	MD5 Hash
PresaleBase.sol	4e454fee784d3bf473b49fd4669d40cc
PresaleInternal.sol	de0910092bf79a732501ec8023e374e5
PresaleStorage.sol	892def72c5abdea2e3df20379e1fc93a

8 Disclaimer

Shellboxes reports should not be construed as "endorsements" or "disapprovals" of particular teams or projects. These reports do not reflect the economics or value of any "product" or "asset" produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.



For a Contract Audit, contact us at contact@shellboxes.com