BLOCKSEC

Security Audit Report for Puffer Protocol

Date: November 27, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction **1**

1.1 About Target Contracts . . . 1
1.2 Disclaimer . . . 1
1.3 Procedure of Auditing . . . 2
1.3.1 Software Security . . . 2
1.3.2 DeFi Security . . . 2
1.3.3 NFT Security . . . 3
1.3.4 Additional Recommendation . . . 3
1.4 Security Model . . . 3

Chapter 2 Findings **5**

2.1 Software Security . . . 5
2.1.1 Potential precision loss in function `getPendingDistributionAmount()` . . . 5
2.2 Additional Recommendation . . . 6
2.2.1 Remove unused state variables . . . 6
2.2.2 Add checks for constructor parameters . . . 7
2.2.3 Add checks for parameters in function `callTargets()` . . . 8
2.2.4 Refactor for gas optimizations . . . 8
2.3 Note . . . 9
2.3.1 Potential centralization risks . . . 9
2.3.2 Discrepancies in exchange rates when purchasing validator tokens . . . 10
2.3.3 Inconsistency in return values of contract functions . . . 11
2.3.4 Potential DoS due to Aera Vault configuration changes . . . 12

Report Manifest

Item	Description
Client	Puffer Finance
Target	Puffer Protocol

Version History

Version	Date	Description
1.0	November 27, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-11-27 06:27:37+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The focus of this audit is on the Puffer Protocol [1] of Puffer Finance. The Puffer Protocol includes two feature upgrades:

1. The validators can now purchase Validator Tickets (VTs) with the `pufETH` token (previously only `ETH` is supported).
2. The contract `PufferRevenueDepositor` is added to manage and distribute revenue within the Puffer Protocol ecosystem. It handles the gradual deposit of revenue into the contract `PufferVault` while also distributing rewards to various stakeholders.

Specifically, only the following contracts in the repository are included in the scope of this audit. Other files are not within the scope of this audit.

- `mainnet-contracts/src/PufferRevenueDepositor.sol`
- `mainnet-contracts/src/PufferVaultV4.sol`
- `mainnet-contracts/script/DeployRevenueDepositor.s.sol`
- `mainnet-contracts/src/ValidatorTicket.sol`
- `mainnet-contracts/script/UpgradeValidatorTicket.s.sol`

Project	Version	Commit Hash
Puffer Protocol	Version 1	e0047f15d950526cc0241b364ea5efd36e27b3d5
	Version 2	209c176d987d95b13b02fdcaa71063f572d55dd1

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

[1] https://github.com/PufferFinance/puffer-contracts

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **one** potential security issue. Besides, we have **four** recommendations and **four** notes.

- Low Risk: 1
- Recommendation: 4
- Note: 4

ID	Severity	Description	Category	Status
1	Low	Potential precision loss in function `getPendingDistributionAmount()`	Software Security	Fixed
2	-	Remove unused state variables	Recommendation	Fixed
3	-	Add checks for constructor parameters	Recommendation	Fixed
4	-	Add checks for parameters in function `callTargets()`	Recommendation	Fixed
5	-	Refactor for gas optimizations	Recommendation	Fixed
6	-	Potential centralization risks	Note	-
7	-	Discrepancies in exchange rates when purchasing validator tokens	Note	-
8	-	Inconsistency in return values of contract functions	Note	-
9	-	Potential DoS due to Aera Vault configuration changes	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential precision loss in function `getPendingDistributionAmount()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `PufferRevenueDepositor`, the function `getPendingDistributionAmount()` calculates the pending distribution amount to be subtracted from the `totalAssets` of the contract `PufferVault`. This calculation uses the proportion of the remaining time within the reward distribution window to determine the appropriate amount.

A loss of precision can occur when the last deposit amount of rewards is small. This can result in a smaller return value of function `getPendingDistributionAmount()`. In this case, the increase of the exchange rate for `pufETH` would be slightly higher than expected.

```
80    function getPendingDistributionAmount() public view returns (uint256) {
81        RevenueDepositorStorage storage $ = _getRevenueDepositorStorage();
82
83        uint256 rewardsDistributionWindow = $.rewardsDistributionWindow;
84
```

```
85        // If the rewards distribution window is not set, return 0 to avoid division by 0
86        // This also means that the deposits are instant
87        if (rewardsDistributionWindow == 0) {
88            return 0;
89        }
90
91        uint256 timePassed = block.timestamp - $.lastDepositTimestamp;
92        uint256 remainingTime = rewardsDistributionWindow - Math.min(timePassed,
              rewardsDistributionWindow);
93
94        return $.lastDepositAmount * remainingTime / rewardsDistributionWindow;
95    }
```

Listing 2.1: mainnet-contracts/src/PufferRevenueDepositor.sol

Impact In the case of precision loss, the exchange rate of `pufETH` may be higher than expected.

Suggestion When calculating the return value in the function `getPendingDistributionAmount()`, perform rounding up to avoid precision loss.

2.2 Additional Recommendation

2.2.1 Remove unused state variables

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `PufferRevenueDepositorStorage`, there are unused state variables in the contract `PufferRevenueDepositor`. Specifically, the fields `restakingOperators`, `rNORewardsBps`, and `treasureRewardsBps` in the struct `RevenueDepositorStorage` are unused.

```
22    struct RevenueDepositorStorage {
23        /**
24         * @notice Restaking operators.
25         */
26        EnumerableSet.AddressSet restakingOperators;
27        /**
28         * @notice RNO rewards in bps.
29         */
30        uint128 rNORewardsBps;
31        /**
32         * @notice Treasury rewards in bps.
33         */
34        uint128 treasuryRewardsBps;
35        /**
36         * @notice Last deposit timestamp.
37         */
38        uint48 lastDepositTimestamp;
39        /**
40         * @notice Rewards distribution window.
41         */
```

```
42      uint104 rewardsDistributionWindow;
43      /**
44       * @notice Last deposit amount.
45       */
46      uint104 lastDepositAmount;
47  }
```

Listing 2.2: mainnet-contracts/src/PufferRevenueDepositorStorage.sol

Suggestion Remove the unused state variables.

2.2.2 Add checks for constructor parameters

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `PufferRevenueDepositor`, parameters are not checked in the constructor. As the variables are immutable, it is recommended to check the parameters in the constructor.

Likewise, in the contract `ValidatorTicket`, there is a missing check for the `operationsMultisig` parameter.

```
57  constructor(address vault, address weth, address aeraVault) {
58      PUFFER_VAULT = PufferVaultV4(payable(vault));
59      AERA_VAULT = IAeraVault(aeraVault);
60      WETH = IWETH(weth);
61      _disableInitializers();
62  }
```

Listing 2.3: mainnet-contracts/src/PufferRevenueDepositor.sol

```
72  constructor(
73      address payable guardianModule,
74      address payable treasury,
75      address payable pufferVault,
76      IPufferOracle pufferOracle,
77      address operationsMultisig
78  ) {
79      if (
80          guardianModule == address(0) || treasury == address(0) || pufferVault == address(0)
81              || address(pufferOracle) == address(0)
82      ) {
83          revert InvalidData();
84      }
85      PUFFER_ORACLE = pufferOracle;
86      GUARDIAN_MODULE = guardianModule;
87      PUFFER_VAULT = pufferVault;
88      TREASURY = treasury;
89      OPERATIONS_MULTISIG = operationsMultisig;
90      _disableInitializers();
91  }
```

Listing 2.4: mainnet-contracts/src/ValidatorTicket.sol

Suggestion Add sufficient checks for constructor parameters.

2.2.3 Add checks for parameters in function `callTargets()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `callTargets()` of the contract `PufferRevenueDepositor`, the lengths of the `targets` and `data` parameter is not checked.

```
155   function callTargets(address[] calldata targets, bytes[] calldata data) external restricted {
156       for (uint256 i = 0; i < targets.length; ++i) {
157           // nosemgrep arbitrary-low-level-call
158           (bool success,) = targets[i].call(data[i]);
159           require(success, TargetCallFailed());
160       }
161   }
```

Listing 2.5: mainnet-contracts/src/PufferRevenueDepositor.sol

Suggestion Add sanity checks to ensure that the lengths of the `targets` and `data` are the same.

2.2.4 Refactor for gas optimizations

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `purchaseValidatorTicket()` of the contract `ValidatorTicket`, the load of the validator ticket storage can be delayed as the function may return early on Line 129. This can help to save the extra gas cost of storage loads.

```
108   function purchaseValidatorTicket(address recipient)
109       external
110       payable
111       virtual
112       restricted
113       returns (uint256 mintedAmount)
114   {
115       ValidatorTicket storage $ = _getValidatorTicketStorage();
116
117       uint256 mintPrice = PUFFER_ORACLE.getValidatorTicketPrice();
118       mintedAmount = (msg.value * 1 ether) / mintPrice; // * 1 ether is to upscale amount to 18
              decimals
119
120       // slither-disable-next-line divide-before-multiply
121       _mint(recipient, mintedAmount);
122
123       // If we are over the burst threshold, keep everything
124       // That means that pufETH holders are not getting any new rewards until it goes under the
              threshold
125       if (PUFFER_ORACLE.isOverBurstThreshold()) {
```

```
126             // Everything goes to the treasury
127             TREASURY.sendValue(msg.value);
128             emit DispersedETH({ treasury: msg.value, guardians: 0, vault: 0 });
129             return mintedAmount;
130         }
131
132         uint256 treasuryAmount = _sendETH(TREASURY, msg.value, $.protocolFeeRate);
133         uint256 guardiansAmount = _sendETH(GUARDIAN_MODULE, msg.value, $.guardiansFeeRate);
134         uint256 vaultAmount = msg.value - (treasuryAmount + guardiansAmount);
135         // The remainder belongs to PufferVault
136         PUFFER_VAULT.sendValue(vaultAmount);
137         emit DispersedETH({ treasury: treasuryAmount, guardians: guardiansAmount, vault:
                vaultAmount });
138     }
```

Listing 2.6: mainnet-contracts/src/ValidatorTicket.sol

Suggestion Refactor the corresponding code segment to reduce gas costs.

2.3 Note

2.3.1 Potential centralization risks

Introduced by Version 1

Description In this upgrade of the Puffer protocol, all the rewards for the entire protocol would be gradually released through the contract `PufferRevenueDepositor`. The speed of gradual release is controlled by the parameter `rewardDistributionWindow`, therefore, changing this variable can directly affect the exchange rate of the `pufETH`.

```
80      function getPendingDistributionAmount() public view returns (uint256) {
81          RevenueDepositorStorage storage $ = _getRevenueDepositorStorage();
82
83          uint256 rewardsDistributionWindow = $.rewardsDistributionWindow;
84
85          // If the rewards distribution window is not set, return 0 to avoid division by 0
86          // This also means that the deposits are instant
87          if (rewardsDistributionWindow == 0) {
88              return 0;
89          }
90
91          uint256 timePassed = block.timestamp - $.lastDepositTimestamp;
92          uint256 remainingTime = rewardsDistributionWindow - Math.min(timePassed,
                rewardsDistributionWindow);
93
94          return $.lastDepositAmount * remainingTime / rewardsDistributionWindow;
95      }
```

Listing 2.7: mainnet-contracts/src/PufferRevenueDepositor.sol

Feedback from the Project We are in the process of decentralization. Recently, we had our TGE, and we are slowly revising the DAO parameters in our protocol. At the moment, our OPS

Multisig has the power to change this parameter, but eventually, it will be a governance decision.

2.3.2 Discrepancies in exchange rates when purchasing validator tokens

Introduced by Version 1

Description The contract `ValidatorTicket` introduces the functionality that allows users to purchase validator tickets using `pufETH`. When buying with `pufETH` tokens, these tokens are burnt which results in an increase in the `pufETH` exchange rate (Line 295). Although purchasing VTs with either ETH or `pufETH` results in rewards for `pufETH` holders through exchange rate increases, the magnitude of the rate change differs between the two methods. The following is an example to illustrate the discrepancy:

Suppose the current exchange rate is 2, and the contract `PufferVault` holds 200 ETH as the `totalAssets`, and the total supply of`pufETH` is 100. Each VT is priced at 1 ETH.

A user attempts to buy 10 VTs with the following two options:

- Buy with `ETH`. The rate after the purchase is 210/100 = 2.1
- Buy with `pufETH`. The rate after the purchase is 200/95 = 2.105, higher than the first option.

This discrepancy stems from the difference in the changes of the total supply of `pufETH`. While both options provide the same rewards (10 Ether), the second option decreases the total supply of tokens eligible to split rewards, leading to a higher exchange rate than the first one.

```
266   function _processPurchaseValidatorTicketWithPufETH(address recipient, uint256 vtAmount)
267       internal
268       returns (uint256 pufEthUsed)
269   {
270       require(recipient != address(0), RecipientIsZeroAddress());
271
272       uint256 mintPrice = PUFFER_ORACLE.getValidatorTicketPrice();
273
274       uint256 requiredETH = vtAmount.mulDiv(mintPrice, 1 ether, Math.Rounding.Ceil);
275
276       pufEthUsed = PufferVaultV3(PUFFER_VAULT).convertToSharesUp(requiredETH);
277
278       IERC20(PUFFER_VAULT).transferFrom(msg.sender, address(this), pufEthUsed);
279
280       _mint(recipient, vtAmount);
281
282       // If we are over the burst threshold, send everything to the treasury
283       if (PUFFER_ORACLE.isOverBurstThreshold()) {
284           IERC20(PUFFER_VAULT).transfer(TREASURY, pufEthUsed);
285           emit DispersedPufETH({ treasury: pufEthUsed, guardians: 0, burned: 0 });
286           return pufEthUsed;
287       }
288
289       ValidatorTicket storage $ = _getValidatorTicketStorage();
290
291       uint256 treasuryAmount = _sendPufETH(TREASURY, pufEthUsed, $.protocolFeeRate);
292       uint256 guardiansAmount = _sendPufETH(OPERATIONS_MULTISIG, pufEthUsed, $.guardiansFeeRate);
293       uint256 burnAmount = pufEthUsed - (treasuryAmount + guardiansAmount);
```

```
294
295         PufferVaultV3(PUFFER_VAULT).burn(burnAmount);
296
297         emit DispersedPufETH({ treasury: treasuryAmount, guardians: guardiansAmount, burned:
                burnAmount });
298
299         return pufEthUsed;
300     }
```

Listing 2.8: mainnet-contracts/src/ValidatorTicket.sol

Feedback from the Project The implementation of this feature serves a strategic purpose in mitigating withdrawal pressure, thereby circumventing the necessity for complex validator provisioning and exit procedures typically associated with liquidity management during withdrawal events.

There exists an inverse relationship between validator quantity, Annual Percentage Yield (APY), and Validator Ticket (VT) pricing. Specifically, as the number of validators increases, both the APY and VT price experience a corresponding decrease. While the price discrepancy becomes more pronounced with larger-scale VT acquisitions, the gradual depreciation in VT pricing renders substantial single purchases economically suboptimal.

A more strategic approach involves implementing a "validate-as-you-go" methodology, wherein smaller, incremental purchases significantly reduce pricing disparities. Although the current framework may present certain challenges for Node Operators (NoOps), it ultimately benefits `pufETH` token holders. The protocol's success fundamentally depends on maintaining robust liquidity, which is primarily facilitated by `pufETH` holders. While institutional NoOps typically aim to maximize validator deployment, we have strategically prioritized retail participants and `pufETH` stakeholders

2.3.3 Inconsistency in return values of contract functions

Introduced by `Version 1`

Description In the contract `ValidatorTicket`, the semantics of the return values of the functions `purchaseValidatorTicket()` and `purchaseValidatorTicketWithPufETH()` are inconsistent. Specifically, the return value of the function `purchaseValidatorTicket()` is the **amount of validator tickets purchased**, but the function `purchaseValidatorTicketWithPufETH()` returns the **total amount of pufETH used**.

Feedback from the Project Yes, this behavior is intentional. The goal is to provide a functionality where users can specify the number of Validator Tickets (VT) they want to purchase, and the system calculates the required amount of ETH accordingly. However, the `purchaseValidatorTicket` method is not suitable for this, as it doesn't allow you to determine the exact ETH required at the time of the transaction. Additionally, any off-chain estimations may differ from the actual on-chain requirement, potentially leading to refund scenarios, which add complexity.

On the other hand, the `purchaseValidatorTicketWithPufETH` method addresses this issue. It allows users to approve a larger amount of pufETH and deduct only what is actually needed.

This ensures smoother processing and avoids the complications of handling refunds. Hence this design choice.

2.3.4 Potential DoS due to Aera Vault configuration changes

Introduced by Version 1

Description In the contract `PufferRevenueDepositor`, the function `withdrawAndDeposit()` first withdraws from Aera Vault, with the amount specified as the balance of the Aera Vault. Then, the function deposits the withdrawn tokens to the contract `PufferVaultV4`.

However, in the contract `AeraVaultV2`, the function `withdraw()` would check if the asset holdings of the vault exceeds the amount specified in the parameter. On calculating the holdings of the assets for the vault, if the token is specified as fee token, the fees would be calculated and deducted from the holdings.

Currently, the Aera Vault used by the contract `PufferRevenueDepositor` is configured with zero fee. If the vault is to be changed or the configuration is changed, there is a potential issue that the funds in the Aera Vault cannot be withdrawn with the function `withdrawAndDeposit()`.

```
138   function withdrawAndDeposit() external restricted {
139       AssetValue[] memory assets = new AssetValue[](1);
140
141       assets[0] = AssetValue({ asset: IERC20(address(WETH)), value: WETH.balanceOf(address(
              AERA_VAULT)) });
142
143       // Withdraw WETH to this contract
144       AERA_VAULT.withdraw(assets);
145
146       _depositRevenue();
147   }
```

Listing 2.9: mainnet-contracts/src/PufferRevenueDepositor.sol

```
742   function _checkWithdrawRequest(
743       IAssetRegistry.AssetInformation[] memory assets,
744       AssetValue[] memory amounts
745   ) internal view {
746       uint256 numAmounts = amounts.length;
747
748       AssetValue[] memory assetAmounts = _getHoldings(assets);
749
750       bool isRegistered;
751       AssetValue memory assetValue;
752       uint256 assetIndex;
753
754       for (uint256 i = 0; i < numAmounts;) {
755           assetValue = amounts[i];
756           (isRegistered, assetIndex) =
757               _isAssetRegistered(assetValue.asset, assets);
758
759           if (!isRegistered) {
760               revert Aera__AssetIsNotRegistered(assetValue.asset);
```

```
761             }
762
763             if (assetAmounts[assetIndex].value < assetValue.value) {
764                 revert Aera__AmountExceedsAvailable(
765                     assetValue.asset,
766                     assetValue.value,
767                     assetAmounts[assetIndex].value
768                 );
769             }
770             unchecked {
771                 i++; // gas savings
772             }
773         }
774     }
```

Listing 2.10: The function from Aera Vault:aera-contracts-public/v2/AeraVaultV2.sol

```
828     function _getHoldings(IAssetRegistry.AssetInformation[] memory assets)
829         internal
830         view
831         returns (AssetValue[] memory assetAmounts)
832     {
833         uint256 numAssets = assets.length;
834
835         assetAmounts = new AssetValue[](numAssets);
836         IAssetRegistry.AssetInformation memory assetInfo;
837
838         for (uint256 i = 0; i < numAssets;) {
839             assetInfo = assets[i];
840             assetAmounts[i] = AssetValue({
841                 asset: assetInfo.asset,
842                 value: assetInfo.asset.balanceOf(address(this))
843             });
844
845             if (assetInfo.asset == _feeToken) {
846                 assetAmounts[i].value -=
847                     Math.min(feeTotal, assetAmounts[i].value);
848             }
849
850             unchecked {
851                 i++; //gas savings
852             }
853         }
854     }
```

Listing 2.11: The function from Aera Vault:aera-contracts-public/v2/AeraVaultV2.sol

Feedback from the Project We can DDoS ourselves by changing the fee to > 0, but we can also change it back to 0, since we are the contract owners (Multisig through the contract `PufferRevenuDepositor`). Also, by design, our system is not taking anything from the rewards. We are taking a cut on VT sales.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS