SALUS SECURITY

OCT 2023

CODE SECURITY ASSESSMENT

TWITTERSCANXBT

Overview

Project Summary

- Name: TwitterscanXBT (XBT)
- Address:
  - Proxy: 0x5697117A06A1129Ae0E1e24A8017287EC8d22134
  - Implementation:0x14B66F2062cC91f64f1E72D2DDcc06fB95c0c9d5
- Platform: BNB Smart Chain
- Language: Solidity
- Audit Scope: See Appendix - 1

Project Dashboard

Application Summary

Name	TwitterscanXBT (XBT)
Version	v2
Type	Solidity
Dates	Oct 25 2023
Logs	Oct 18 2023; Oct 25 2023

Vulnerability Summary

Total High-Severity issues	3
Total Medium-Severity issues	0
Total Low-Severity issues	1
Total informational issues	4
Total	8

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Lack of access control leads to relayer tampering	High	Access Control	Resolved
2	Signature replay leads to data tampering and free upgrades	High	Business Logic	Resolved
3	The payment logic can be bypassed	High	Business Logic	Resolved
4	Missing events	Low	Logging	Resolved
5	Missing zero address check	Informational	Data Validation	Resolved
6	Use sendValue instead of transfer	Informational	Code Quality	Acknowledged
7	Gas optimization	Informational	Gas Optimization	Resolved
8	Redundant code	Informational	Redundancy	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Lack of access control leads to relayer tampering

Severity: High	Category: Access Control
Target: - InfluenceXBT.sol	

Description

According to the logic of the contract, `_setRelayer()` function can only be called through the constructor or a function restricted by onlyOwner, but this function is decorated with public and has no other permission restrictions. In addition, functions using this naming method cannot be called directly by users under normal circumstances.

And if this function can be called by anyone, it will affect the correct execution of `_sigValidation()`. The attacker can set his controllable contracts as relayers to pass the signature verification in `_sigValidation()` function.

InfluenceXBT.sol

```
function _setRelayer(address[] memory _relayer) public {
        for(uint i=0; i<relayer.length; i++)
            isRelayer[relayer[i]] = false;
        relayer = _relayer;
        for(uint i=0; i<relayer.length; i++)
            isRelayer[relayer[i]] = true;
        emit SetRelayer(relayer);
}
```

Recommendation

Consider using private/internal instead of public restrictions.

Status

The team has resolved this issue.

2. Signature replay leads to data tampering and free upgrades

Severity: High	Category: Business Logic
Target: - InfluenceXBT.sol	

Description

In InfluenceXBT, there is no verification for the reuse of signatures. This can lead to the reuse of signatures, enabling attackers to tamper with contract data again using old signatures.

InfluenceXBT.sol

```
function updateInfluenceData(UpdateParams memory updateParams) public {
    bool isValid = _checkUpdateSig(updateParams);
    require(isValid, 'Invalid signature');
    ...
}
```

In the above function, the absence of duplicate validation for the updateParams.signature parameter allows attackers to maliciously tamper with the values in the influenceList mapping using old signatures.

- **Proof of Concept**

We simulated a scenario where the system, while operating normally, had its data maliciously tampered with.

```
function test_Replay() public {
    InfluenceXBT.UpdateParams memory param;
    bytes[] memory signatures = new bytes[](1);

    param = InfluenceXBT.UpdateParams(10000, keccak256("hashedId"), 10000, 10000, 10000,
10000, 10000, signatures);

    bytes32 hash = _checkUpdateSig(param);
    (uint8 v,bytes32 r,bytes32 s) = vm.sign(pk, hash);
    signatures[0] = abi.encodePacked(r, s, v);
    param.signature = signatures;

    InfluenceXBT(XBT).updateInfluenceData(param);
    assertEq(InfluenceXBT(XBT).getInfluenceScore(10000), 10000);

    //Data Changes Resulting from Simulated Normal System Operation
    changeData();
    assertEq(InfluenceXBT(XBT).getInfluenceScore(10000), 1);

    //Malicious Activity: Data Restoration Using Old Signatures
    InfluenceXBT(XBT).updateInfluenceData(param);
    assertEq(InfluenceXBT(XBT).getInfluenceScore(10000), 10000);
}
```

Recommendation

Consider recording signatures that have already been used and preventing them from being verified again.

Status

The team has resolved this issue.

3. The payment logic can be bypassed

Severity: High	Category: Business Logic
Target: - InfluenceXBT.sol	

Description

In InfluenceXBT, users can obtain Pro privileges by calling the mintPro() function, but regular users need to pay the corresponding amount of ether (proPrice).

InfluenceXBT.sol

```
function mintPro(BaseParams memory baseParams) external payable {
    bool isValid = _checkBaseSig(baseParams);
    require(isValid, 'Invalid signature');
    require(balanceOf(baseParams.recipient) == 0, '!0');
    if (!_checkMeta(baseParams.recipient, 4)) {
        require(msg.value == proPrice, '!price');
        payable(vault).transfer(msg.value);
        payedPro[baseParams.recipient] = true;
    }
    require(!twitterIds[baseParams.hashedId], '!twitterId');
    _mintToken(baseParams.recipient, 2);
    influenceList[tokenIdCounter.current()] = InfluenceData({
        hashedId: baseParams.hashedId,
        followerCount: baseParams.followerCount,
        viewCount: baseParams.viewCount,
        registerDays: baseParams.registerDays,
        registerTime: baseParams.registerTime,
        influenceScore: baseParams.influenceScore
    });
    levels[tokenIdCounter.current()] = 2;
    twitterIds[baseParams.hashedId] = true;
}

function _checkMeta(address account, uint256 threshold) internal returns (bool) {
    uint256 tokenCount = IERC721EnumerableUpgradeable(metaToken).balanceOf(
        account
    );
    if (tokenCount > 0) {
        uint256 tokenId;
        for (uint256 i = 0; i < tokenCount; i++) {
            tokenId = IERC721EnumerableUpgradeable(metaToken)
                .tokenOfOwnerByIndex(account, i);
            string memory tokenName = IMETA(metaToken).names(tokenId);
            bytes memory bTokenName = bytes(tokenName);
            if (bTokenName.length <= threshold) {
                return true;
            }
        }
    }
```

```
    return false;
}
```

However, users can become Pro users for free by bypassing the check of the _checkMeta() function, following these steps:

1.Borrow tokens from an address with privileged metaToken.

2.Invoke the mintPro or mintPremium function.

3.Return the privileged metaToken.

This could lead to users with privileged metaTokens providing inexpensive upgrade services, causing a loss to the platform.

- **Proof of Concept**

We simulated a scenario where Alice, an address with privileged metaToken, lent this token to Bob and later reclaimed it. This allowed Bob to become a Pro user without spending any ether.

```
function test_MetaToken() public {
    BaseRegistrarImplementation(metaToken).registerWithName(0, alice, 60 * 60 * 24,
"s");

    //Alice lends legitimate tokens to Bob
    vm.startPrank(alice);
    BaseRegistrarImplementation(metaToken).transferFrom(alice, bob, 0);
    vm.stopPrank();

    bytes[] memory signatures = new bytes[](1);
    InfluenceXBT.BaseParams memory param = InfluenceXBT.BaseParams(10000,
keccak256("hashedId2"), 10000, bob, 10000, 10000, address(this), 10000, signatures);

    bytes32 hash = _checkBaseSig(param);
    (uint8 v,bytes32 r,bytes32 s) = vm.sign(pk, hash);
    param.signature[0] = abi.encodePacked(r, s, v);
    //Bob can become a Pro user without spending ether
    InfluenceXBT(XBT).mintPro(param);

    //Bob returns the tokens to Alice
    vm.startPrank(bob);
    BaseRegistrarImplementation(metaToken).transferFrom(bob, alice, 0);
    vm.stopPrank();

    uint256 tokenId = InfluenceXBT(XBT).tokenOfOwnerByIndex(bob, 0);
    assertEq(InfluenceXBT(XBT).levels(tokenId), 2);
}
```

Recommendation

Consider disabling the transfer of metaTokens.

Status

The team has resolved this issue.

4. Missing events

Severity: Low	Category: Logging
Target: - InfluenceXBT.sol	

Description

Important parameter or configuration changes should trigger an event to enable tracking off-chain, but the function mentioned below changes the important parameters without emitting an event.

InfluenceXBT.sol

```
function setUriDomain(string memory uri, uint256 index) external onlyOwner {
    require(index < 3, '!index');
    uriDomain[index] = uri;
}
```

Recommendation

Design an appropriate event and incorporate it into the aforementioned function.

Status

The team has resolved this issue.

2.3 Informational Findings

5. Missing zero address check

Severity: Informational	Category: Data Validation
Target: - InfluenceXBT.sol	

Description

It is considered a security best practice to verify addresses against the zero address in the constructor or setting. However, this precautionary step is absent for the variables highlighted below.

InfluenceXBT.sol

```
function __InfluenceXBT_init(
        address[] memory _relayer,
        address _vault,
        address _owner
    ) external initializer {
        ...
        __Base_init(_owner);
        _setRelayer(_relayer);
        vault = _vault;
        ...
}
```

Recommendation

Consider adding zero-address checks.

Status

The team has resolved this issue.

6. Use sendValue instead of transfer

Severity: Informational	Category: Code Quality
Target: - InfluenceXBT.sol	

Description

In the mintPro, mintPremium, upgradePro and upgradePremium functions of the InfluenceXBT contract, the transfer of Ether is executed with Solidity's transfer function, which forwards a limited amount of gas to the receiver. Should the receiver be a contract with a fallback function that needs more than 2300 units of gas to execute, the transfer of Ether would inevitably fail.

After the Istanbul hard fork, it has become a recommended practice not to use transfer to avoid hard dependencies on specific gas costs.

Recommendation

Consider replacing transfer with the sendValue function available in the OpenZeppelin Contracts library.

Status

This issue has been acknowledged by the team.

7. Gas optimization

Severity: Informational	Category: Gas Optimization
Target: - InfluenceXBT.sol	

Description

1. For storage variables that are read frequently in a function, we can first temporarily store its contents in a temporary variable to reduce the gas consumption of SLOAD.

InfluenceXBT.sol

```
function mintBase(BaseParams memory baseParams) external returns(uint256) {
        require(balanceOf(baseParams.recipient) == 0, '!0');
        bool isValid = _checkBaseSig(baseParams);
        require(isValid, 'Invalid signature');
        require(!twitterIds[baseParams.hashedId], '!twitterId');
        _mintToken(baseParams.recipient, 1);
        influenceList[tokenIdCounter.current()] = InfluenceData({
            hashedId: baseParams.hashedId,
            followerCount: baseParams.followerCount,
            viewCount: baseParams.viewCount,
            registerDays: baseParams.registerDays,
            registerTime: baseParams.registerTime,
            influenceScore: baseParams.influenceScore
        });
        levels[tokenIdCounter.current()] = 1;
        twitterIds[baseParams.hashedId] = true;
        return tokenIdCounter.current();
}
```

2. Functions without modifying the state of the contract can be marked as “view” which will reduce gas costs.

InfluenceXBT.sol

```
function checkMetaPro(address account) public returns (bool) {
        return _checkMeta(account, 4);
}

function checkMetaPremium(address account) public returns (bool) {
        return _checkMeta(account, 3);
}

function _checkMeta(address account, uint256 threshold) internal returns (bool) {
        ...
}
```

Recommendation

Consider applying the gas optimizations where needed.

Status

The team has resolved this issue.

8. Redundant code

Severity: Informational	Category: Redundancy
Target: - InfluenceXBT.sol	

Description

The mintPremiumWithPass() function implements the upgrade functionality for NFTs without incurring any fees, and, therefore, does not require the payable modifier.
This could result in incorrect funds entering the contract and being permanently locked within the contract.

InfluenceXBT.sol

```
function mintPremiumWithPass(PremiumParams memory premiumParams) external payable {
    require(balanceOf(premiumParams.recipient) == 0, '!0');
    bool isValid = _checkPremiumSig(premiumParams);
    ...
}
```

Recommendation

Consider removing the payable modifier.

Status

The team has resolved this issue.

Appendix

Appendix 1 - Files in Scope

This audit covered the following file from the implementation address of 0xdCDdB807aF076d43e1715F1179712110A7b10fe3:

File	SHA-1 hash
InfluenceXBT.sol	a0ad13b602496a353ecbaee36a5236bb0ab476ff

The revised code has been deployed to the following addresses:

- Proxy: 0x5697117A06A1129Ae0E1e24A8017287EC8d22134
- Implementation:0x14B66F2062cC91f64f1E72D2DDcc06fB95c0c9d5