Resultados del Análisis

Información del Contrato

Dirección: `0x698b1d54e936b9f772b8f58447194bbc82ec1933`

ContractName: PEEZY

CompilerVersion: v0.8.23+commit.f704f362

CompilerType: solc

OptimizationUsed: 0

Runs: 200

EVMVersion: Default

LicenseType: Unlicense

Proxy: 0

SwarmSource:
ipfs://9f842dab930e316f4eadb31a7ec3a00daa2dec8d2770c1ff2518abfddadef21c

balance: 0.012227541702609092

txCount: 2566

contractAddress: 0x698b1d54e936b9f772b8f58447194bbc82ec1933

contractCreator: 0xe8c129d6349cd9347f100f5c789b4476147f5449

txHash:
0xcc106634c3372bb984ece989e1f59fb1bae3ba8536dcbd7e7ba033c9f3f27cf0

blockNumber: 21066647

timestamp: 1730148863

Análisis de Vulnerabilidades

Okay, let's start by looking at the code provided. The contract is an ERC20 token called PEEZY. The first thing I notice is that it's using Solidity 0.8.23, which is a recent version and has security patches, so that's good. The contract imports

SafeMath, but since it's 0.8.23, SafeMath might not be necessary because the compiler handles overflow checks. However, the code still uses SafeMath functions, which is redundant but not a vulnerability.

Next, looking at the Ownable contract. The owner can renounce ownership, which is a standard function. The constructor sets the owner correctly. There's a tax system in place with initial and final buy/sell taxes. The taxes decrease over time based on buy counts. The contract also includes a Uniswap router for trading and liquidity.

Now, checking for common vulnerabilities. Let's look at the _transfer function. There's a tax calculation based on whether the transaction is a buy or sell. The code checks if the sender or receiver is a bot. The owner can add or remove bots, which is a centralization risk but not a vulnerability per se. However, the addBots and delBots functions are public and onlyOwner, so that's okay.

Looking at the swapTokensForEth function, it uses lockTheSwap modifier to prevent reentrancy. The inSwap flag is set, which is a good practice. The swap is done via Uniswap's swapExactTokensForETHSupportingFeeOnTransferTokens, which is correct for fee-on-transfer tokens.

One potential issue is the tax calculation. In the _transfer function, when from is the Uniswap pair (a buy), the tax is applied. However, the code has a condition where if _buyCount is 0, it uses a different tax calculation. Wait, let me check that again. The code says:

```
if(_buyCount==0){
taxAmount = amount.mul((_buyCount>_reduceBuyTaxAt)?
_finalBuyTax:_initialBuyTax).div(100);
}
```

But if _buyCount is 0, then (_buyCount>_reduceBuyTaxAt) would be false, so it uses _initialBuyTax. Then, if the buy is from the pair, it again calculates taxAmount. Wait, there's a possible double taxation here. Let me see:

In the _transfer function, after the initial tax calculation, there's another check:

```
if (from == uniswapV2Pair && to != address(uniswapV2Router) && !
_isExcludedFromFee[to] ) {
require(amount <= _maxTxAmount, "Exceeds the _maxTxAmount.");
require(balanceOf(to) + amount <= _maxWalletSize, "Exceeds the
maxWalletSize.");
taxAmount = amount.mul((_buyCount>_reduceBuyTaxAt)?
_finalBuyTax:_initialBuyTax).div(100);
_buyCount++;
}
```

So, if _buyCount is 0, the first taxAmount is set using _initialBuyTax (23%), then in this block, taxAmount is recalculated again, overwriting the previous value. Then

_buyCount is increased. So, for the first time, this would set taxAmount to 23% again, but then _buyCount becomes 1. Wait, but the initial code outside this block may have already set taxAmount. Let me trace the flow.

If it's a buy (from == uniswapV2Pair), then the code first checks if _buyCount is 0. If yes, taxAmount is set to 23%. Then, since from is the pair, the code enters the if block and recalculates taxAmount again, which would be 23% again (since _buyCount is 0 before increment). Then _buyCount is incremented. So, the tax is applied twice? No, because the second assignment overwrites the first. Wait, no. Let's see:

Suppose it's the first buy. The code first checks if _buyCount ==0. Then taxAmount is set to 23% (since _buyCount is 0, which is not > _reduceBuyTaxAt (23), so uses _initialBuyTax). Then, since from is the pair, the code enters the if block, recalculates taxAmount again as 23% (since _buyCount is still 0 at this point), then increments _buyCount. So the taxAmount is set twice, but the second one overwrites the first. So the tax is 23% for the first buy. Then, for subsequent buys, _buyCount is >0, so the first part (if _buyCount==0) is skipped, and the tax is calculated as _transferTax (50%)? Wait, no. Let me check:

The code has:

```
if (from != owner() && to != owner()) {
require(!bots[from] && !bots[to]);

if(_buyCount==0){
taxAmount = amount.mul((_buyCount>_reduceBuyTaxAt)?
_finalBuyTax:_initialBuyTax).div(100);
}
if(_buyCount>0){
taxAmount = amount.mul(_transferTax).div(100);
}

if (from == uniswapV2Pair && to != address(uniswapV2Router) && !
_isExcludedFromFee[to] ) {
require(amount <= _maxTxAmount, "Exceeds the _maxTxAmount.");
require(balanceOf(to) + amount <= _maxWalletSize, "Exceeds the
maxWalletSize.");
taxAmount = amount.mul((_buyCount>_reduceBuyTaxAt)?
_finalBuyTax:_initialBuyTax).div(100);
_buyCount++;
}

if(to == uniswapV2Pair && from!= address(this) ){
taxAmount = amount.mul((_buyCount>_reduceSellTaxAt)?
_finalSellTax:_initialSellTax).div(100);
}

// rest of the code
```

}

Wait, this is a problem. The code first checks if _buyCount is 0, sets taxAmount to initialBuyTax (23%). Then, if it's a buy (from == pair), it recalculates taxAmount again, which would be again 23% (since _buyCount is 0 at that point). Then, if it's a sell (to == pair), it sets taxAmount to initialSellTax (23%). But the way the code is structured, these conditions can overwrite each other. For example, a transfer that is both a buy and a sell (which is impossible, but maybe a transfer from pair to pair? Unlikely). But more importantly, the code's logic for setting taxAmount is conflicting. For example, if a user is selling (to == pair), then the taxAmount is set to sell tax, but if it's a buy, it's set to buy tax. However, the code allows for multiple conditions to set taxAmount, leading to potential overwrites.

But the main issue here is the initial tax calculation. Let's take a buy transaction. The code first checks if _buyCount is 0, sets taxAmount to 23%. Then, since it's the pair, enters the if block, recalculates taxAmount to 23% again (since _buyCount is 0), increments _buyCount. So for the first buy, tax is 23%. Then, for subsequent buys, since _buyCount is >0, the code sets taxAmount to _transferTax (50%)? Wait, no. Let's see:

If _buyCount is 1 (after first buy), then the first if(_buyCount==0) is false. Then the code checks if(_buyCount>0) and sets taxAmount to _transferTax (50%). Then, if it's a buy (from == pair), it recalculates taxAmount again. Wait, in that case, for the second buy, the code would first set taxAmount to 50% (because _buyCount>0), then if it's a buy, recalculate taxAmount based on _buyCount (which is 1) > _reduceBuyTaxAt (23). Since 1 is not >23, it uses _initialBuyTax (23%). Then increments _buyCount. So the tax would be 23% again. That's conflicting. Because the code first sets taxAmount to 50% (because _buyCount>0), then overwrites it with 23% if it's a buy. So the tax calculation is incorrect here.

This is a logical error. The tax calculation is not working as intended. The _transferTax is supposed to be 50%, but for buys after the first one, the code first sets taxAmount to 50%, then if it's a buy, overwrites it to 23% (if _buyCount <=23). This would result in lower taxes than intended. This is a flaw in the tax logic, leading to incorrect tax application. This is a critical issue because it breaks the intended tax mechanism, potentially allowing users to pay lower taxes than expected.

Another issue is the swapTokensForEth is called with min(amount, min(contractTokenBalance, _maxTaxSwap)). The 'amount' here is the transfer amount, but when swapping, the contract should swap the minimum of the contract's balance and _maxTaxSwap. However, using 'amount' here could lead to swapping less than the threshold. Wait, the code says:

swapTokensForEth(min(amount, min(contractTokenBalance, _maxTaxSwap)));

So, it takes the minimum between the transfer amount and the minimum of contract balance and _maxTaxSwap. This might not swap enough tokens to cover

the accumulated taxes. For example, if the transfer amount is small, but the contract has a large balance, it would only swap the small amount, leading to accumulation of tokens in the contract. This could prevent the contract from properly processing the taxes, leading to stuck funds or inability to pay fees. This is a medium severity issue.

Another point is the sellCount and lastSellBlock variables. The code limits the number of sells per block to 3. However, if multiple sells occur in the same block, the contract will revert after the third, which could be a way to prevent rapid selling but might also cause issues if legitimate users are trying to sell. However, this is a design choice rather than a vulnerability, but it could be a potential DoS vector if not handled properly. However, since it's controlled by the contract, it's more of a medium risk.

The contract also has a _transferTax variable set to 50, which is applied when _buyCount>0. But as per the earlier analysis, the tax calculation is conflicting. This could lead to unexpected tax rates.

Looking at the addBots and delBots functions. The owner can add any address to the bots list, which blocks their transactions. This is a centralization risk, as the owner can blacklist any address, potentially blocking legitimate users. This is a major issue because it gives the owner too much control, which could be abused.

The openTrading function approves the Uniswap router for the total supply, which is a common practice but could be a risk if the router is compromised. However, since it's the standard Uniswap router, this is considered safe. However, the approval is for the maximum uint256 value, which is necessary for adding liquidity but could be a risk if there's a vulnerability in the router. However, this is a low risk as the router is a well-audited contract.

The reduceFee function can be called by _taxWallet (which is set to the owner initially) to reduce the final buy and sell tax. However, there's a check that _newFee must be <= the current _finalBuyTax and _finalSellTax. Wait, the code says:

```
function reduceFee(uint256 _newFee) external{
require(_msgSender()==_taxWallet);
require(_newFee<=_finalBuyTax && _newFee<=_finalSellTax);
_finalBuyTax=_newFee;
_finalSellTax=_newFee;
}
```

This means that the fee can only be reduced further, not increased. However, the initial final taxes are 0. So, the require statement would check if _newFee <=0 and <=0, meaning _newFee must be 0. But the function is called reduceFee, which suggests it's meant to lower the fee. However, since _finalBuyTax and _finalSellTax are initialized to 0, the require statement would only allow setting _newFee to 0, making this function ineffective. Wait, let's check the initial values:

In the contract's variables:

uint256 private _finalBuyTax=0;
uint256 private _finalSellTax=0;

So, when reduceFee is called, the require statement checks that _newFee <=0 and <=0. Therefore, _newFee must be 0. So this function can only set the final taxes to 0, which they already are. This makes the function useless. This is a mistake in the code. The intention was to allow the owner to reduce the final tax from 23% to a lower value, but since the initial final tax is 0, this function cannot be used. This is a bug, leading to the inability to adjust the final tax as intended. This is a medium severity issue because it breaks the contract's functionality.

Another issue is the lack of a function to update the tax wallet. The _taxWallet is set in the constructor to the owner, but there's no way to change it later. If the owner's wallet is compromised, the funds collected as tax cannot be redirected. This is a major issue because it lacks a recovery mechanism.

The contract also has a manualSwap and manualsend function, which allows the _taxWallet to manually trigger swaps and send ETH. This is a centralization risk, as the tax wallet can manipulate when to swap and collect fees, potentially leading to MEV or front-running. However, since it's controlled by a trusted party, it's more of a major risk due to centralization.

The contract uses the deprecated SafeMath library. While it's not a vulnerability, it's unnecessary in Solidity >=0.8.0, as the compiler handles overflow checks. This is an informational issue.

The maxTxAmount and maxWalletSize are set, but the removeLimits function sets them to _tTotal, effectively removing limits. This is a standard function but could be used to remove restrictions after launch, which might be a centralization aspect.

In terms of reentrancy, the swap function uses a lockTheSwap modifier, which is good. However, the sendETHToFee function uses transfer, which is safe against reentrancy.

Another thing to check is the initial tax rates. The _initialBuyTax and _initialSellTax are 23%, which decreases to 0 after a certain number of buys. However, the _reduceBuyTaxAt is set to 23. So when _buyCount >23, the tax switches to _finalBuyTax (0). But the code increments _buyCount only on buys from the pair. So after 23 buys, the tax drops to 0. This is for the buy tax. The sell tax is reduced when _buyCount>_reduceSellTaxAt (23). So the sell tax also decreases after 23 buys. However, combined with the earlier tax calculation issues, this might not work as intended.

Putting it all together, the main vulnerabilities are:

1. Incorrect Tax Calculation Logic leading to incorrect tax application (Critical)

2. Centralized Control via Bot List (Major)
3. Incorrect Fee Reduction Function (Medium)
4. Swap Amount Calculation Error (Medium)
5. Lack of Tax Wallet Update Mechanism (Major)
6. Use of Deprecated SafeMath (Informational)

Let me verify each one again.

1. The tax calculation in _transfer has conflicting conditions that overwrite taxAmount, leading to incorrect taxes. For example, after the first buy, subsequent buys may apply 50% tax but then overwrite to 23% if the buy condition is met. This breaks the intended tax mechanism. Critical severity because it directly affects the core functionality and economics of the code:

In the _transfer function:

```
if(_buyCount==0){
taxAmount = amount.mul(_transferTax).div(100);
}
```

But then, if it's a buy (from == pair), it recalculates taxAmount again. So for a buy when _buyCount>0, the taxAmount is first set to _transferTax (50%), then overwritten to ((_buyCount>_reduceBuyTaxAt) ? _finalBuyTax : _initialBuyTax). Since _reduceBuyTaxAt is 23, if _buyCount is 23 or less, it uses _initialBuyTax (23%). So the first calculation (50%) is overwritten by 23%. This is a logic error, leading to lower taxes than intended. The intended behavior might have been to apply 50% tax for non-buy/sell transfers, but buys and sells have their own tax rates. However, the code's structure causes the _transferTax to be applied incorrectly.

This is a critical issue because it results in incorrect tax deductions, potentially causing financial loss for the project or users.

2. Centralized Control via Bot List: The owner can add any address to the bots list, which blocks their transactions. This allows the owner to censor any address, which is a major centralization risk and potential abuse. Major severity.

3. The reduceFee function is ineffective because it requires _newFee to be <= the final taxes, which are initialized to 0. Therefore, the function cannot be used to reduce fees as intended. This is a medium severity bug.

4. The swapTokensForEth uses min(amount, ...), which might limit the swapped amount to the transfer amount instead of the contract's balance. This could prevent the contract from swapping enough tokens to cover accumulated fees, leading to stuck funds. Medium severity.

5. No way to update _taxWallet. If the wallet is compromised, funds cannot be recovered. Major severity.

6. SafeMath is redundant. Informational.

Other observations:

- The contract has a _transferTax variable set to 50, but it's unclear when this is applied. It seems that for transfers that are buys or sells, the tax is overwritten, but for other transfers (e.g., between wallets), the _transferTax of 50% is applied. However, the code's logic may not correctly handle this. For example, a normal transfer (not buy/sell) would have taxAmount set to 50% if _buyCount>0. This could be intended, but 50% is very high and could be a trap for users. However, this is part of the tokenomics, not a vulnerability.

- The _preventSwapBefore is set to 26, which means swaps are prevented until _buyCount>26. However, the swap is enabled when swapEnabled is set in openTrading. The swap is only possible after _buyCount exceeds _preventSwapBefore. This is a mechanism to prevent early swaps, which is a common practice.

- The contract uses block.number to track lastSellBlock and allows 3 sells per block. This is a way to prevent rapid sells but could be bypassed by miners. However, it's a minor point.

In conclusion, the main critical issue is the incorrect tax calculation logic, which undermines the contract's intended functionality. The other issues are major or medium, related to centralization and function bugs.

Análisis de Seguridad del Contrato Inteligente PEEZY

---

1. Lógica Incorrecta de Cálculo de Impuestos (Tax Logic Flaw)
- **Severidad:** Critical 🚨
- **Descripción Técnica:**
En la función `_transfer`, la lógica para calcular `taxAmount` contiene condiciones conflictivas que sobrescriben el valor de los impuestos. Inicialmente, se aplica `_transferTax` (50%) cuando `_buyCount > 0`, pero en transacciones de compra (`from == uniswapV2Pair`), se recalcula usando `_initialBuyTax` (23%) si `_buyCount <= _reduceBuyTaxAt` (23). Esto resulta en impuestos más bajos de lo previsto, rompiendo el modelo económico del contrato.
- **Ubicación:**
```solidity
// En _transfer:
if(_buyCount>0){
taxAmount = amount.mul(_transferTax).div(100); // 50%
}
if (from == uniswapV2Pair && ...) {
taxAmount = amount.mul((_buyCount>_reduceBuyTaxAt)?
_finalBuyTax:_initialBuyTax).div(100); // 23%
```

}
```
- **Riesgo:**
Los usuarios podrían pagar menos impuestos de lo diseñado, afectando la recaudación de fondos y la sostenibilidad del proyecto.
- **Mitigación:**
Reestructurar la lógica para evitar sobrescrituras. Separar claramente los casos de compra, venta y transferencias normales, aplicando impuestos específicos en cada caso.

---

### **2. Control Centralizado mediante Lista de Bots**
- **Severidad:** Major 🔥
- **Descripción Técnica:**
Las funciones `addBots` y `delBots` permiten al dueño bloquear cualquier dirección, censurando transacciones arbitrariamente. Esto introduce un riesgo de centralización y abuso.
- **Ubicación:**
```solidity
function addBots(address[] memory bots_) public onlyOwner { ... }
```
- **Riesgo:**
El dueño puede bloquear direcciones legítimas, afectando la descentralización y confianza en el contrato.
- **Mitigación:**
Eliminar la capacidad de bloquear direcciones post-despliegue o implementar un mecanismo descentralizado (e.g., gobernanza).

---

3. Función `reduceFee` Inefectiva
- **Severidad:** Medium ⚠️
- **Descripción Técnica:**
`reduceFee` intenta reducir `_finalBuyTax` y `_finalSellTax`, pero como estas variables se inicializan en 0, el `require(_newFee <= 0)` solo permite `_newFee = 0`, haciendo la función inútil.
- **Ubicación:**
```solidity
function reduceFee(uint256 _newFee) external {
require(_newFee <= _finalBuyTax && _newFee <= _finalSellTax); // _finalBuyTax = 0
...
}
```
- **Riesgo:**
Imposibilidad de ajustar las tasas finales como se pretendía, limitando la flexibilidad del contrato.
- **Mitigación:**
```

Inicializar `_finalBuyTax` y `_finalSellTax` con valores mayores a 0 (e.g., 23%) y corregir la validación en `reduceFee`.

---

4. Error en el Cálculo del Monto de Swap
- **Severidad:** Medium ⚠️
- **Descripción Técnica:**
En `swapTokensForEth`, se usa `min(amount, min(contractTokenBalance, _maxTaxSwap))`, donde `amount` es el monto de la transferencia actual. Esto podría limitar el swap a montos menores al umbral necesario (`_taxSwapThreshold`), acumulando tokens no procesados.
- **Ubicación:**
```solidity
swapTokensForEth(min(amount, min(contractTokenBalance, _maxTaxSwap)));
```
- **Riesgo:**
Acumulación de tokens en el contrato, impidiendo la recaudación de impuestos.
- **Mitigación:**
Usar `min(contractTokenBalance, _maxTaxSwap)` directamente, ignorando `amount`.

---

5. Falta de Mecanismo para Actualizar Tax Wallet
- **Severidad:** Major 🔥
- **Descripción Técnica:**
No existe una función para actualizar `_taxWallet`. Si la dirección actual se compromete, los fondos recolectados se perderían.
- **Ubicación:**
`_taxWallet` se establece en el constructor sin posibilidad de cambio.
- **Riesgo:**
Pérdida irreversible de fondos en caso de compromiso de la billetera.
- **Mitigación:**
Implementar una función `setTaxWallet` con control de acceso (`onlyOwner`).

---

6. Uso de SafeMath Redundante (Informational)
- **Severidad:** Informational ℹ️
- **Descripción Técnica:**
El contrato usa SafeMath aunque Solidity 0.8.23 ya maneja overflows/underflows.
- **Mitigación:**
Eliminar SafeMath y usar operadores nativos (`+`, `-`, etc.) para claridad.

Conclusión
El contrato presenta vulnerabilidades críticas y mayores que afectan su

funcionalidad central y seguridad. La más grave es la **lógica incorrecta de impuestos**, que compromete el modelo económico. Se recomienda priorizar su corrección junto con los riesgos de centralización (bot list y tax wallet). Las demás observaciones (e.g., SafeMath) son secundarias pero relevantes para mejores prácticas.

Información del Contrato

Dirección: `0xb29e475b69f843046a757747943c00dce8a3d982`

ContractName: SPECTRE

CompilerVersion: v0.8.27+commit.40a35a09

CompilerType: solc

OptimizationUsed: 1

Runs: 200

EVMVersion: Default

LicenseType: Unlicense

Proxy: 0

SwarmSource:
ipfs://bc7b07ac82e098a00481f41c78126613ddc04e0c2921656b4b34c380e7b05cdd

balance: 0.00001

txCount: 781

contractAddress: 0xb29e475b69f843046a757747943c00dce8a3d982

contractCreator: 0x1fd1a7b908d271a6683e8ef19a2c602604a9fac1

txHash:
0x797e8709ecb93ba8a0bd914b5cd72ee40ec16800866ed65fb9b82e5753abdac3

blockNumber: 20842902

timestamp: 1727451455

Análisis de Vulnerabilidades

Okay, let's start by looking at the provided Solidity code. The contract is an ERC20 token with some tax mechanisms and features related to Uniswap trading. The first thing I notice is that the contract uses SafeMath, but since Solidity 0.8.x has built-in overflow checks, SafeMath might be redundant here. However, that's more of a code style issue rather than a vulnerability.

Next, looking at the constructor, they allocate 22% of the total supply to _spectreMultiSegWallet and the rest to the contract itself. That's a significant initial allocation, but unless there's a lock-up mechanism, this could be a centralization risk. But the problem is, the owner can call functions like sendTokensToSpectreMultiSeg to transfer any ERC20 tokens held by the contract, which might include the contract's own tokens. Wait, the contract's own balance is part of the total supply, so if the owner can withdraw those, that's a major issue. Let me check the sendTokensToSpectreMultiSeg function.

The function sendTokensToSpectreMultiSeg allows the Spectre wallet to transfer any ERC20 tokens from the contract to the multi-sig wallet. This includes the contract's own tokens. Since the contract holds a large portion of the supply (78% initially), the owner could drain those tokens, leading to a rug pull. That's a critical vulnerability. The function is only callable by the Spectre wallet, which is set during construction. But if the Spectre wallet is controlled by the owner, this is a central point of failure.

Another point is the openTrading function. It approves the Uniswap router to spend the entire total supply of the token. That's risky because if the router is ever compromised, or if there's a bug in the approval, someone could drain the tokens. However, after adding liquidity, the remaining tokens in the contract would be subject to the tax mechanism. But the initial approval is for the total supply, which is excessive. This could be a major issue if not handled properly.

Looking at the tax calculation in the _transfer function. The taxAmount is calculated based on buyCount. However, the logic here might be flawed. For example, when from is the Uniswap pair (a buy), the tax is applied, but the condition checks if _buyCount is greater than _reduceBuyTaxAt. Initially, _buyCount is 0, so the tax would be _initialBuyTax (22%). But after a certain number of buys, the tax reduces. However, the way _buyCount is incremented is only when the transaction is from the pair and not excluded from fee. But if the buyCount is not properly tracked, the tax might not reduce as intended. Not sure if this is a vulnerability, but possible logic errors here could lead to incorrect tax application.

The swapTokensForEth function is called during sells, and it's a check to prevent more than 3 swaps per block. However, combined with the tax mechanism, if the contract accumulates a lot of tokens, the swap could be triggered multiple times, but the limit might prevent effective fee processing. However, this is more of a design choice than a vulnerability.

The contract has a function removeSpectreLimits which allows the owner to remove the max transaction and max wallet limits. This could be used to bypass any anti-whale mechanisms, which might be a centralization risk. If the owner removes these limits, it could lead to large transactions that might manipulate the market.

The addBots and delBots functions allow the owner to blacklist addresses. This

can be abused to block certain users from transacting, which is a centralization risk. However, this is common in some tokens to prevent bots, but it's still a potential issue if the owner acts maliciously.

The _spectreMultiSegWallet is updatable by the owner via updateSpectreTaxWallet. If the owner changes this to a malicious address, funds could be diverted. But this is controlled by the owner, so again a centralization risk.

Another thing to note is the use of block.number in the sellCount and lastSellBlock. Since block numbers can vary between different blockchains and are not a reliable measure of time, but in this case, it's used to limit the number of swaps per block, which is okay. However, if the contract is on a chain with very fast blocks, this might not be effective.

The initial tax rates are 22% for buys and sells, which is quite high. While not a vulnerability per se, it's a red flag for users as it might indicate a scam or high fees.

Looking at the transfer function, when calculating taxAmount for buys, the code checks if from is the uniswap pair. However, if the pair is not set correctly, or if there are multiple pairs, this could lead to incorrect tax application. But the contract only creates one pair in openTrading, so that's probably okay.

The sendETHToMultiSeg function sends ETH to the multi-sig wallet, but if that fails, it sends to _spectreWallet. However, if both addresses are controlled by the same entity, this might not be an issue, but if there's a failure in the multi-sig, the funds go to another wallet. Not a vulnerability, but a possible centralization point.

The contract also has a receive function, so it can accept ETH. But there's no function to withdraw ETH except through swapTaxToETH and sendETHToSpectreMultiSeg, which are controlled by the Spectre wallet. So the contract's ETH balance can be drained by the Spectre wallet, which is a privileged role.

Putting this all together, the critical issues are:

1. The sendTokensToSpectreMultiSeg function allows the Spectre wallet to withdraw any ERC20 tokens from the contract, including the contract's own tokens. Since the contract holds 78% of the supply initially, this is a rug pull risk. Severity: Critical.

2. The openTrading function approves the Uniswap router to spend the entire total supply, which is excessive. If the router is compromised, or if there's a bug, the tokens could be stolen. Severity: Critical.

3. The owner can add/remove bots, which can block certain addresses from transacting. Centralization risk. Severity: Medium.

4. The removeSpectreLimits function allows the owner to disable transaction and wallet limits, which could be abused. Severity: Medium.

5. The initial high tax rates and possible centralization points (updatable tax wallet, privileged functions) are centralization risks but not vulnerabilities per se. However, they are important for users to know.

So the main critical vulnerabilities are the first two points. The others are medium or lower.

Análisis de Seguridad del Contrato SPECTRE

1. Vulnerabilidad: Retiro Ilimitado de Tokens ERC20 por Parte del Spectre Wallet
- **Severidad:** Critical
- **Descripción Técnica:** La función `sendTokensToSpectreMultiSeg` permite al `_spectreWallet` retirar cualquier token ERC20 almacenado en el contrato, incluyendo los tokens SPECTRE propios. Inicialmente, el contrato posee el 78% del suministro total (`_tTotal`), lo que permite al equipo drenar estos fondos.
- **Ubicación del Código:**
```solidity
function sendTokensToSpectreMultiSeg(address tokenAddress)
external
onlySpectreWallet
returns (bool success)
{
emit clearTokens(
tokenAddress,
IERC20(tokenAddress).balanceOf(address(this))
);
return
IERC20(tokenAddress).transfer(
_spectreMultiSegWallet,
IERC20(tokenAddress).balanceOf(address(this))
);
}
```
- **Riesgo:** Posibilidad de un "rug pull", donde los desarrolladores retiren la liquidez o fondos de los usuarios, colapsando el valor del token.
- **Mitigación:** Limitar esta función a tokens específicos (p.ej., ETH) o implementar un mecanismo de timelock/vesting para los tokens SPECTRE del contrato.

---

2. Vulnerabilidad: Aprobación Excesiva al Router de Uniswap
- **Severidad:** Critical
- **Descripción Técnica:** En `openTrading`, se aprueba al router de Uniswap (`uniswapV2Router`) para gastar la totalidad del suministro (`_tTotal`), en lugar de

solo los tokens necesarios para la liquidez inicial.
- **Ubicación del Código:**
```solidity
_approve(address(this), address(uniswapV2Router), _tTotal);
```
- **Riesgo:** Si el router de Uniswap (o una versión maliciosa del mismo) es comprometido, un atacante podría robar todos los tokens del contrato.
- **Mitigación:** Aprobar solo la cantidad necesaria para la liquidez inicial y ajustar las aprobaciones dinámicamente según las necesidades.

---

3. Vulnerabilidad: Centralización en el Control de Transacciones (Bot List)
- **Severidad:** Medium
- **Descripción Técnica:** Las funciones `addBots` y `delBots` permiten al dueño bloquear direcciones arbitrariamente, sin mecanismos de transparencia.
- **Ubicación del Código:**
```solidity
function addBots(address[] memory bots_) public onlyOwner { ... }
function delBots(address[] memory notbot) public onlyOwner { ... }
```
- **Riesgo:** Censura de direcciones legítimas, manipulación del mercado.
- **Mitigación:** Eliminar la funcionalidad o implementar un sistema descentralizado de votación para gestionar la lista.

---

4. Vulnerabilidad: Eliminación de Límites de Transacción y Cartera
- **Severidad:** Medium
- **Descripción Técnica:** La función `removeSpectreLimits` permite al dueño desactivar los límites de `_maxTxAmount` y `_maxWalletSize`, eliminando protecciones contra whales.
- **Ubicación del Código:**
```solidity
function removeSpectreLimits() external onlyOwner {
_maxTxAmount = _tTotal;
_maxWalletSize = _tTotal;
}
```
- **Riesgo:** Concentración de tokens en pocas direcciones, facilitando manipulaciones de precio.
- **Mitigación:** Hacer los límites inmutables después de la activación del trading.

---

5. Vulnerabilidad: Actualización Arbitraria de la Cartera de Impuestos
- **Severidad:** Medium
- **Descripción Técnica:** `updateSpectreTaxWallet` permite al dueño cambiar

`_spectreMultiSegWallet` a cualquier dirección, incluyendo una controlada por un atacante.
- **Ubicación del Código:**
```solidity
function updateSpectreTaxWallet(address payable _newSpectreMultiSegWallet)
external
onlyOwner
{ ... }
```
- **Riesgo:** Desvío de fondos recaudados por impuestos a una dirección maliciosa.
- **Mitigación:** Implementar un mecanismo de múltiples firmas o timelock para cambios en la dirección.

Observaciones Adicionales
- **Tax Inicial Elevado (22%):** Aunque no es una vulnerabilidad técnica, impuestos altos pueden desincentivar a los inversores y ser indicativos de esquemas Ponzi.
- **Uso de SafeMath en Solidity 0.8.27:** Redundante, ya que la versión 0.8.x maneja overflow/underflow por defecto. Recomendación: Eliminar SafeMath para optimizar el gas.

Conclusión: El contrato presenta múltiples vulnerabilidades críticas y de centralización que permiten a los desarrolladores drenar fondos, manipular transacciones y eliminar protecciones clave. Se requieren modificaciones sustanciales para garantizar la seguridad de los usuarios.

Información del Contrato

Dirección: `0x9eaebd7e73d97e78c77fab743e6ffa1b550e224c`

ContractName: REXAS_FINANCE

CompilerVersion: v0.8.26+commit.8a97fa7a

CompilerType: solc

OptimizationUsed: 0

Runs: 200

EVMVersion: Default

LicenseType: MIT

Proxy: 0

SwarmSource:
ipfs://91548d3bbbe8c29c0bfedceb505659a7efd56906d6660c2922826162eaa53a25

balance: 15.45672100445071

txCount: 1385

contractAddress: 0x9eaebd7e73d97e78c77fab743e6ffa1b550e224c

contractCreator: 0x48ee57b01d226d1e1efae1fb15f45d9a11bd018a

txHash:
0x496ab249dcb7f77f2160da1af76438b4b3e82093ff2ce5abcf96b2c4d6ea2ac5

blockNumber: 20700162

timestamp: 1725729431

Análisis de Vulnerabilidades

Okay, let's start by looking at the code provided. The contract is called REXAS_FINANCE, an ERC20 token. The first thing I notice is that it uses Solidity 0.8.20, which is a recent version and has security patches, so that's good.

Looking at the Ownable contract, the renounceOwnership function sets the owner to address(0), which is standard. But wait, in the transferOwnership function, the new owner is assigned directly. That's okay, but I should check if there are any issues with ownership transfers.

Next, the main contract has a whitelist mapping. The _transfer function checks if either the sender or receiver is whitelisted. If not, trading must be enabled. The trading flag is set by enableTrading(), which can only be called once. That's a common pattern to prevent presale sniping.

Now, looking at the transfer functions. The _transfer function uses SafeMath's sub and add. Since they're using Solidity 0.8, which has built-in overflow checks, SafeMath might be redundant here. But including it doesn't hurt, though it's unnecessary.

The removeStuckEth and removeStuckToken functions allow the owner to withdraw ETH or any ERC20 tokens from the contract. These are common in contracts to recover funds, but they could be a risk if the owner is malicious. However, since the owner is supposed to be trusted, this is considered a minor issue, but it's worth mentioning.

Wait, the removeStuckToken function calls IERC20(_token).transfer. What if the token is a contract that doesn't return a boolean? Some old tokens (like USDT) don't return a bool on transfer, which would cause this function to revert. That's a problem. The function should use a low-level call with proper error handling.

Another thing: the whitelist can be modified by the owner via setWhitelist. If the owner can add addresses to the whitelist even after trading is enabled, they

could potentially bypass fees or restrictions. But in this contract, the whitelist is only used to bypass the trading check. If trading is enabled, the whitelist might not matter anymore. However, if the owner can add themselves to the whitelist after trading starts, they could transfer tokens even if trading is paused. Wait, no, because once trading is enabled, the require(trading) passes. The whitelist check is only when trading is not enabled. Wait, the code says: if both from and to are not whitelisted, then check if trading is enabled. So if trading is enabled, the whitelist doesn't affect anything. So the setWhitelist function can only exempt addresses from the trading check before trading starts. Once trading is enabled, the whitelist has no effect. So that's okay.

But the enableTrading function can only be called once. So after that, the whitelist can't be used to bypass anything. So the setWhitelist function is safe in that context.

Looking at the _approve function: it doesn't have a check for allowance race condition. For example, when increasing or decreasing allowance, the standard is to use increaseAllowance and decreaseAllowance to prevent the race condition where a spender might front-run a change in allowance. But in this contract, the approve function is standard, which could be vulnerable to the race condition. However, since the contract uses the OpenZeppelin approach with increase/decreaseAllowance, that's handled. Wait, the contract has increaseAllowance and decreaseAllowance functions, which are implemented. So the approve function is protected against the race condition. So that's okay.

The _transfer function checks for from and to not being zero, amount >0, and uses SafeMath. So no underflow/overflow issues here.

Another thing: the contract has a receive() function to accept ETH. But there's no code that sends ETH to the contract except for possible addLiquidityETH, but the contract itself doesn't handle that. The removeStuckEth function allows the owner to withdraw ETH. But if the contract is supposed to hold ETH for liquidity, that's okay. However, if there's a fee-on-transfer mechanism, the contract might accumulate ETH, which the owner can withdraw. That's a centralization risk, but it's common in such tokens.

Wait, the contract doesn't have any fee mechanism. The _transfer function is a simple transfer without any fees. So the only ETH in the contract would be from someone sending it directly, which is unlikely. So the removeStuckEth is just a way to recover any accidental ETH sent.

Another point: the constructor initializes the total supply to the owner's balance. That's correct.

Potential issues:

1. Use of SafeMath with Solidity 0.8: Since Solidity 0.8 has built-in overflow checks, using SafeMath is redundant. However, it doesn't cause any issues, just unnecessary code. Severity: Informational.

2. removeStuckToken function uses IERC20.transfer which may fail for non-compliant tokens: As mentioned earlier, some tokens (like USDT) don't return a boolean on transfer, causing the function to revert. Mitigation: Use safeTransfer or a low-level call. Severity: Medium.

3. Centralization risks: The owner can withdraw any ERC20 tokens and ETH, and can whitelist addresses. However, this is common in many tokens, but it's a trust issue. Severity: Medium (if the contract is supposed to be decentralized, but for a standard token, this is considered a Medium risk).

4. Missing events for critical functions: Functions like setWhitelist and removeStuckToken don't emit events. Events help in tracking off-chain. Severity: Minor.

5. The whitelist can be modified even after trading is enabled: But once trading is enabled, the whitelist check is irrelevant. So modifying it post-trading has no effect. So no issue here.

6. The enableTrading function can only be called once, which is correct.

So the main issues are the SafeMath redundancy (informational), the removeStuckToken function's handling of non-compliant tokens (medium), and centralization risks (medium). Also, missing events (minor).

Análisis de Seguridad del Contrato REXAS_FINANCE (RXS)

---

1. Vulnerabilidad: Uso Redundante de SafeMath en Solidity 0.8+
- **Severidad**: Informational
- **Descripción Técnica**:
El contrato utiliza la librería `SafeMath` para operaciones aritméticas, pero al compilarse con Solidity 0.8.20, las verificaciones de overflow/underflow están integradas en el lenguaje. Esto hace que `SafeMath` sea redundante y aumente el tamaño del bytecode innecesariamente.
- **Ubicación**:
- Líneas que usan `.sub()`, `.add()` (e.g., `_balances[from] = _balances[from].sub(amount, "...")`).
- **Riesgo**:
Aunque no representa un riesgo de seguridad directo, incrementa el costo de gas y la complejidad del código.
- **Mitigación**:
Eliminar la dependencia de `SafeMath` y usar operadores nativos (`+`, `-`, `*`, `/`) con Solidity 0.8+.

---

2. Vulnerabilidad: Manejo Inseguro de Tokens no ERC20 Compliant en `removeStuckToken`
- **Severidad**: Medium
- **Descripción Técnica**:
La función `removeStuckToken` utiliza `IERC20.transfer`, que puede fallar con tokens que no siguen el estándar ERC20 correctamente (e.g., USDT en Ethereum no retorna `bool`). Esto causaría que la transacción revierta.
- **Ubicación**:
```solidity
function removeStuckToken(address _token, address _receiver, uint256 _amount)
public onlyOwner {
IERC20(_token).transfer(_receiver, _amount);
}
```
- **Riesgo**:
El propietario no podría recuperar tokens no compatibles, lo que podría resultar en pérdida de fondos.
- **Mitigación**:
Usar una llamada de bajo nivel (`call`) con manejo de errores:
```solidity
(bool success, ) =
_token.call(abi.encodeWithSignature("transfer(address,uint256)", _receiver,
_amount));
require(success, "Transfer failed");
```

---

3. Vulnerabilidad: Centralización de Privilegios (Ownership Risks)
- **Severidad**: Medium
- **Descripción Técnica**:
- El propietario (`owner`) puede retirar cualquier cantidad de ETH/ERC20 del contrato (`removeStuckEth`, `removeStuckToken`).
- El `whitelist` puede ser manipulado para eximir direcciones de restricciones antes de habilitar el trading.
- **Ubicación**:
Funciones `removeStuckEth`, `removeStuckToken`, y `setWhitelist`.
- **Riesgo**:
Dependencia excesiva en la confianza hacia el propietario. Un `owner` malicioso o comprometido podría drenar fondos.
- **Mitigación**:
- Limitar los privilegios de `removeStuckEth/Token` a una dirección multisig o timelock.
- Emitir eventos en `setWhitelist` para transparencia (e.g., `WhitelistUpdated(address user, bool status)`).

---

4. Vulnerabilidad: Falta de Eventos en Funciones Críticas

- **Severidad**: Minor
- **Descripción Técnica**:
Las funciones `setWhitelist`, `removeStuckEth`, y `removeStuckToken` no emiten eventos, lo que dificulta el rastreo off-chain de cambios.
- **Ubicación**:
Funciones mencionadas.
- **Riesgo**:
Reduce la auditabilidad y transparencia del contrato.
- **Mitigación**:
Añadir eventos relevantes (e.g., `StuckTokensRemoved`, `WhitelistUpdated`).

---

5. Vulnerabilidad: Whitelist Inefectiva Post-Trading
- **Severidad**: Informational
- **Descripción Técnica**:
Una vez habilitado el trading (`trading = true`), el `whitelist` no tiene efecto, ya que la condición `!whitelist[from] && !whitelist[to]` solo aplica si el trading está deshabilitado.
- **Ubicación**:
```solidity
if (!whitelist[from] && !whitelist[to]) {
require(trading, "Rexas Finance: Trading is disable");
}
```
- **Riesgo**:
No es una vulnerabilidad, pero podría generar confusión si se asume que el `whitelist` tiene efecto después del lanzamiento.
- **Mitigación**:
Documentar claramente el comportamiento del `whitelist`.

Conclusión
El contrato no presenta vulnerabilidades críticas (e.g., reentrancy, balance desactualizado). Las principales observaciones son riesgos de centralización y manejo de tokens no estándar. Se recomienda:
1. Reemplazar `SafeMath` por operadores nativos.
2. Corregir `removeStuckToken` para soportar tokens no compliant.
3. Limitar privilegios de `owner` y añadir eventos.

Verificación Adicional:
- El contrato no implementa fees, liquidity pools, ni mecanismos complejos que suelan introducir vulnerabilidades (e.g., `swapAndLiquify`).
- La función `enableTrading` es irreversible, lo que previene manipulaciones post-lanzamiento.

← Página Anterior Inicio