GPTLens Demo

CS 8903 : Special Problems

Advisor: Ling Liu

Aditya Pal

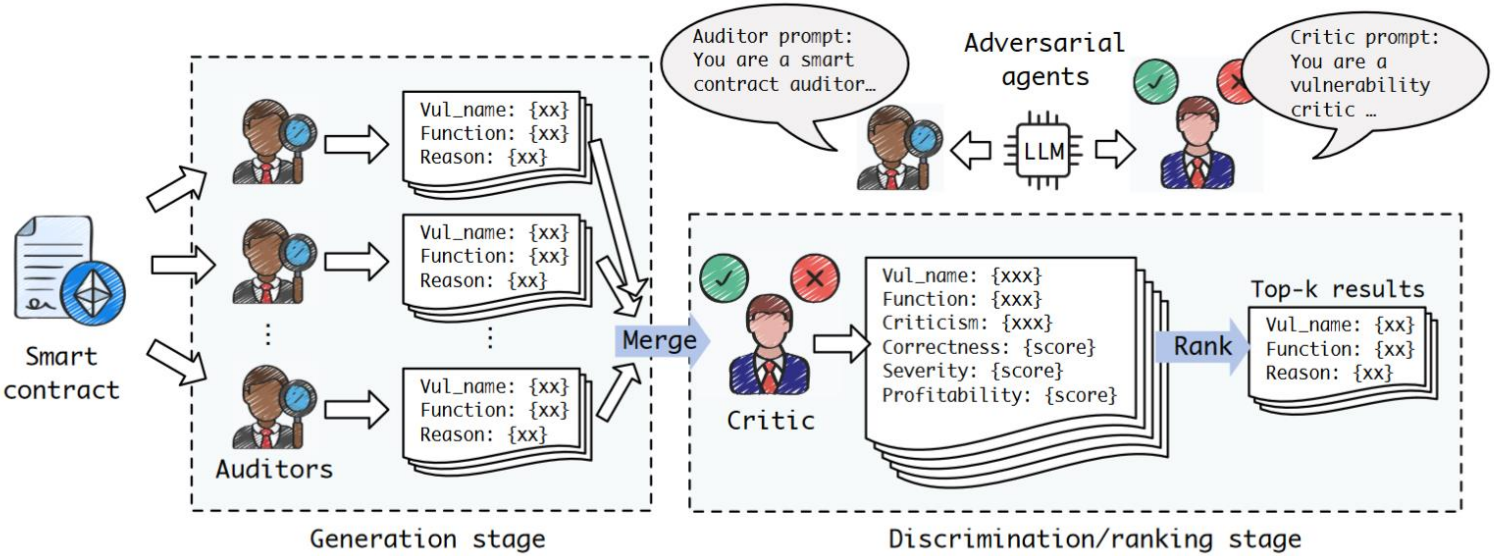
Georgia Tech

Overview

- What is GPTLens?
  - Large Language Model-Powered Smart Contract Vulnerability Detection System
  - Developed by Sihao Hu, Tiansheng Huang, Fatih Ilhan, Selim Furkan Tekin, Ling Liu
  - Paper: https://arxiv.org/pdf/2310.01152
  - Code: https://github.com/git-disl/GPTLens

- How does it work?
  - Uses a 2-step adversarial framework:
    1. Generation of multiple vulnerabilities by LLM auditors based on code input
    2. Discrimination of these vulnerabilities by LLM critic based on correctness, severity etc.
  - Finally rank these vulnerabilities

Architecture



Georgia Tech

GPTLens Workflow

```
/**
 *Submitted for verification at Etherscan.io on 2018-06-29
*/

pragma solidity ^0.4.16;

contract owned {
    address public owner;

    function owned() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
```

Input: Raw solidity file

Step 1

Preprocessing

- Remove comments
- Remove extra spaces

```
pragma solidity ^0.4.16;
contract owned {
    address public owner;
    function owned() public {
        owner = msg.sender;
    }
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
    function transferOwnership(address newOwner) onlyOwner public {
        owner = newOwner;
    }
}
interface tokenRecipient { function receiveApproval(address _from, uint256 _va
contract TokenERC20 {
```

Output: Processed file

Georgia Tech

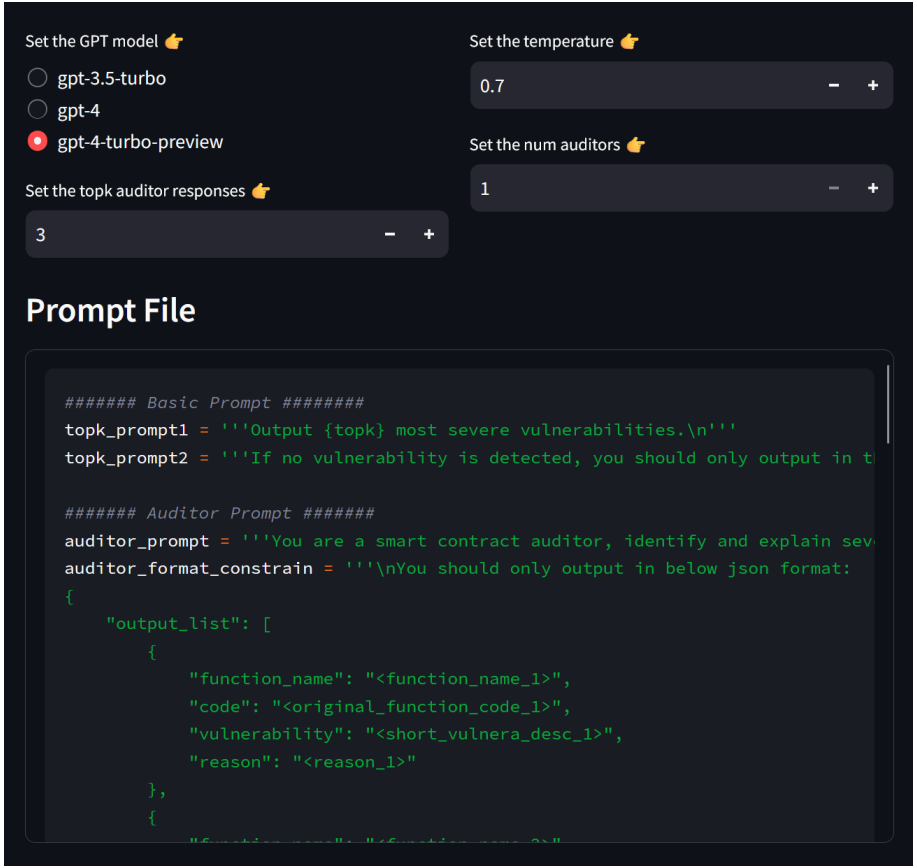
GPTLens Workflow

Step 2

Audit:

Inputs to auditors:

- Solidity file (from Step 1)
- GPT model
- topk auditor responses
- temperature
- num auditors
- Prompt File



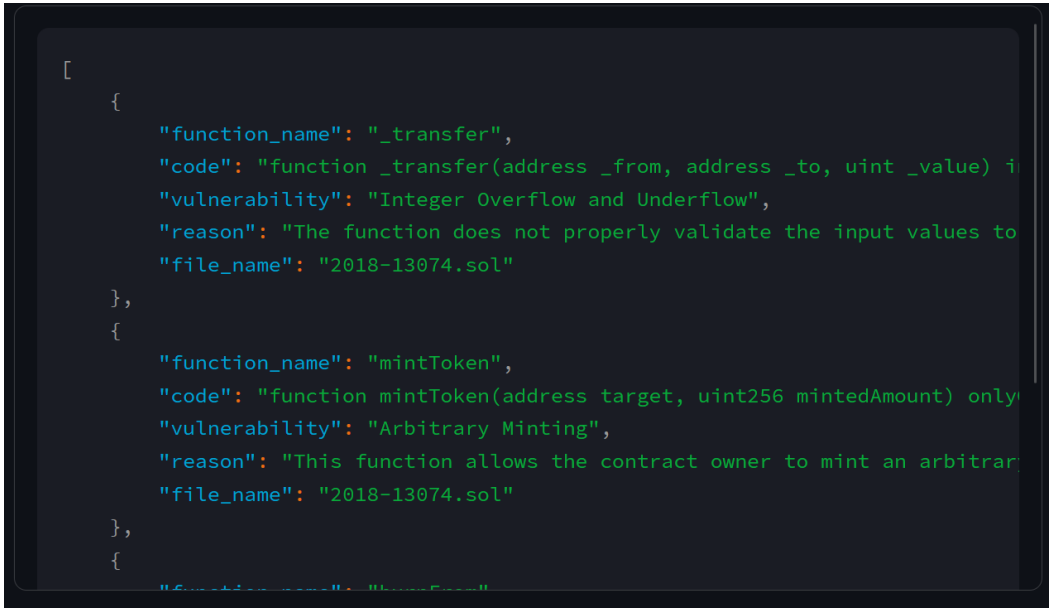
GPTLens Workflow

Step 2 continued

Audit:

Generate vulnerabilities based on inputs:

- Function name
- Code (with vulnerability)
- Reason
- File name



Georgia Tech

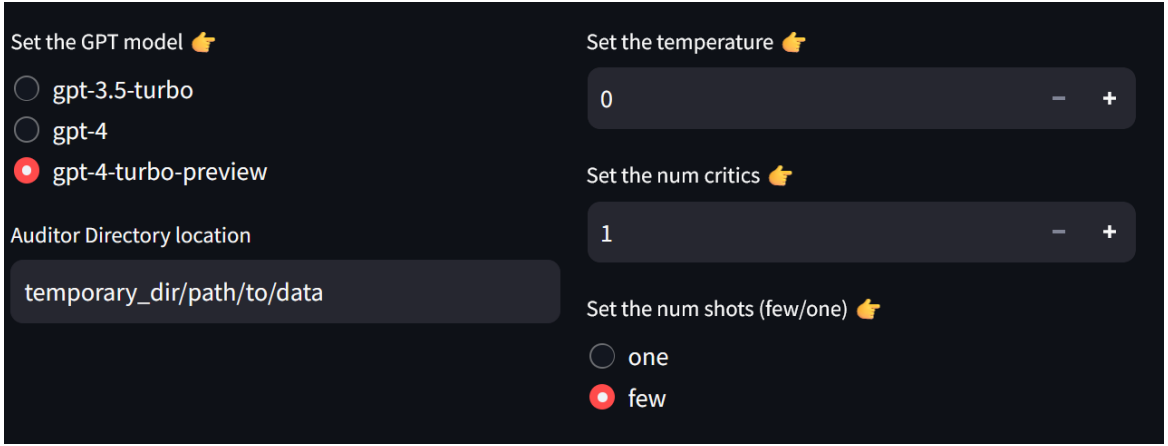
GPTLens Workflow

Step 3

Critic:

Inputs:

- Auditor results file (from Step 2)
- GPT model
- temperature
- num critics
- Num shots (few-shot / on-shot)



Georgia Tech

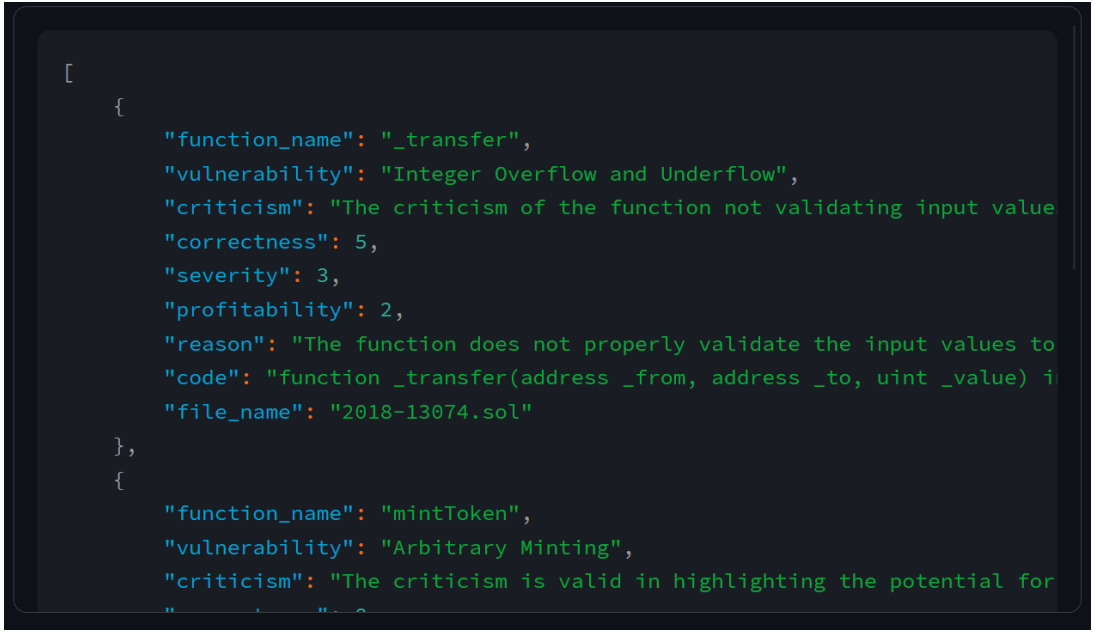
GPTLens Workflow

Step 3 continued

Critic:

Assign values to vulnerabilities:

- Correctness
- Severity
- Profitability



Georgia Tech

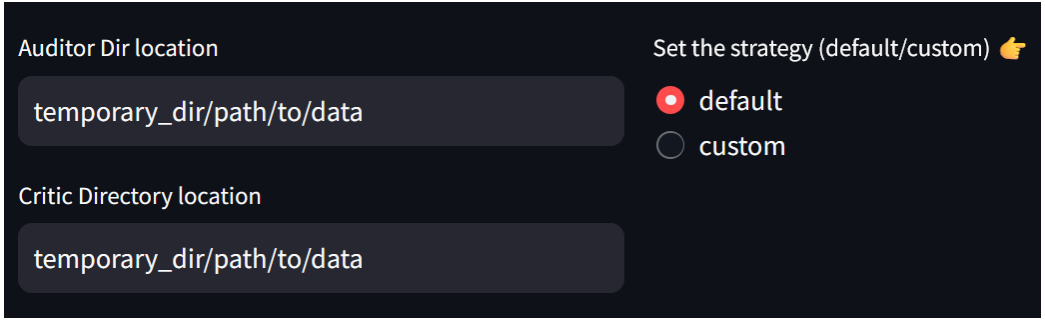
GPTLens Workflow

Step 4

Ranking:

Inputs:

- Auditor results file (from Step 2)
- Critic results file (from Step 3)
- Ranking strategy (default / custom)



Georgia Tech

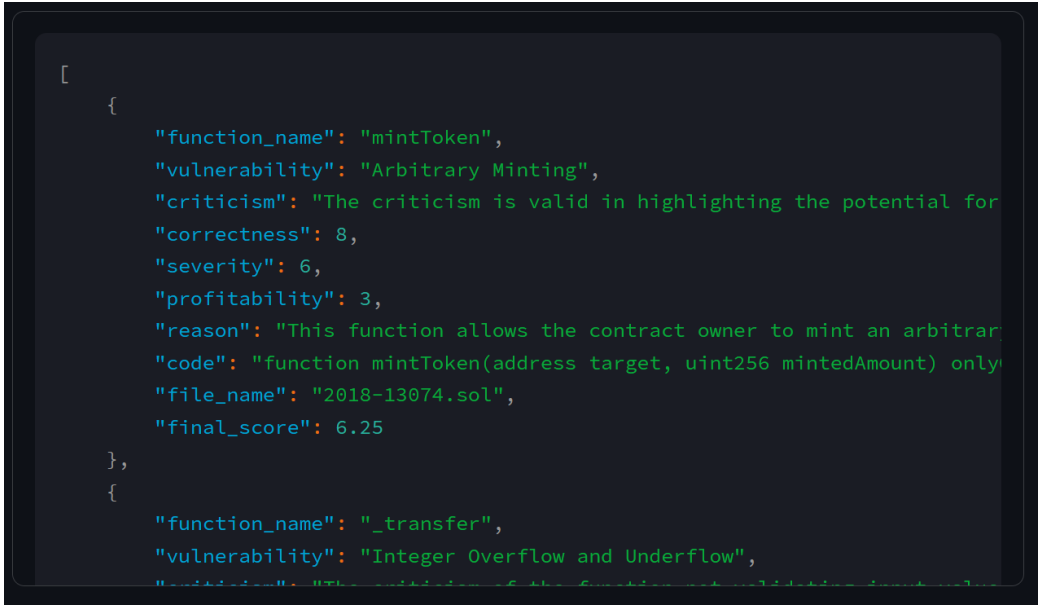
GPTLens Workflow

Step 4 continued

Ranking:

Sort vulnerabilities based on final score:

- Correctness
- Severity
- Profitability



Georgia Tech

GPTLens Explainability

Two input files

File 1

```
/// @notice Create `mintedAmount` tokens and send it to `target`
    /// @param target Address to receive the tokens
    /// @param mintedAmount the amount of tokens it will receive
    function mintToken(address target, uint256 mintedAmount) onlyOwner public {
        balanceOf[target] += mintedAmount;
        totalSupply += mintedAmount;
        emit Transfer(0, this, mintedAmount);
        emit Transfer(this, target, mintedAmount);
    }


    /**
     * Internal transfer, only can be called by this contract
     */
    function _transfer(address _from, address _to, uint _value) internal {
        // Prevent transfer to 0x0 address. Use burn() instead
        require(_to != 0x0);
        // Check if the sender has enough
        require(balanceOf[_from] >= _value);
        // Check for overflows
        require(balanceOf[_to] + _value > balanceOf[_to]);
        // Save this for an assertion in the future
        uint previousBalances = balanceOf[_from] + balanceOf[_to];
        // Subtract from the sender
        balanceOf[_from] -= _value;
        // Add the same to the recipient
        balanceOf[_to] += _value;
        emit Transfer(_from, _to, _value);
        // Asserts are used to use static analysis to find bugs in your code. They should never fail
        assert(balanceOf[_from] + balanceOf[_to] == previousBalances);
    }


    /**
     * Destroy tokens from other account
     *
     * Remove `_value` tokens from the system irreversibly on behalf of `_from`.
     *
     * @param _from the address of the sender
     * @param _value the amount of money to burn
     */
    function burnFrom(address _from, uint256 _value) public returns (bool success) {
        require(balanceOf[_from] >= _value);                // Check if the targeted balance is enough
        require(_value <= allowance[_from][msg.sender]);    // Check allowance
        balanceOf[_from] -= _value;                         // Subtract from the targeted balance
        allowance[_from][msg.sender] -= _value;             // Subtract from the sender's allowance
        totalSupply -= _value;                              // Update totalSupply
        emit Burn(_from, _value);
        return true;
    }
```

File 2

```
/* Initializes contract with initial supply tokens to the creator of the contract */
function CCindexToken() token (initialSupply, tokenName, decimalUnits, tokenSymbol) {}
    /* Send coins */
    function transfer(address _to, uint256 _value) {
        // if(!canHolderTransfer()) throw;
        if (balanceOf[msg.sender] < _value) throw;           // Check if the sender has enough
        if (balanceOf[_to] + _value < balanceOf[_to]) throw; // Check for overflows
        if (frozenAccount[msg.sender]) throw;                // Check if frozen
        balanceOf[msg.sender] -= _value;                     // Subtract from the sender
        balanceOf[_to] += _value;                            // Add the same to the recipient
        Transfer(msg.sender, _to, _value);                   // Notify anyone listening that this transfer took place
        if(_value > 0){
            if(balanceOf[msg.sender] == 0){
                addresses[indexes[msg.sender]] = addresses[lastIndex];
                indexes[addresses[lastIndex]] = indexes[msg.sender];
                indexes[msg.sender] = 0;
                delete addresses[lastIndex];
                lastIndex--;
            }
            if(indexes[_to]==0){
                lastIndex++;
                addresses[lastIndex] = _to;
                indexes[_to] = lastIndex;
            }
        }
    }

    /************************************************************************/

    /************************************************************************/

    /* A contract attempts to get the coins */
    function transferFrom(address _from, address _to, uint256 _value) returns (bool success) {
        if (frozenAccount[_from]) throw;                        // Check if frozen
        if (balanceOf[_from] < _value) throw;                 // Check if the sender has enough
        if (balanceOf[_to] + _value < balanceOf[_to]) throw;  // Check for overflows
        if (_value > allowance[_from][msg.sender]) throw;   // Check allowance
        balanceOf[_from] -= _value;                           // Subtract from the sender
        balanceOf[_to] += _value;                             // Add the same to the recipient
        allowance[_from][msg.sender] -= _value;
        Transfer(_from, _to, _value);
        return true;
    }

    function mintToken(address target, uint256 mintedAmount) onlyOwner {
        balanceOf[target] += mintedAmount;
        totalSupply += mintedAmount;
        Transfer(0, this, mintedAmount);
        Transfer(this, target, mintedAmount);
    }

}
```

Georgia Tech

GPTLens Explainability

Preprocessed results

File 1

```
function _transfer(address _from, address _to, uint _value) internal {
        require(_to != 0x0);
        require(balanceOf[_from] >= _value);
        require(balanceOf[_to] + _value > balanceOf[_to]);
        uint previousBalances = balanceOf[_from] + balanceOf[_to];
        balanceOf[_from] -= _value;
        balanceOf[_to] += _value;
        emit Transfer(_from, _to, _value);
        assert(balanceOf[_from] + balanceOf[_to] == previousBalances);
    }
function burnFrom(address _from, uint256 _value) public returns (bool success) {
        require(balanceOf[_from] >= _value);
        require(_value <= allowance[_from][msg.sender]);
        balanceOf[_from] -= _value;
        allowance[_from][msg.sender] -= _value;
        totalSupply -= _value;
        emit Burn(_from, _value);
        return true;
    }
}
contract FIBToken is owned, TokenERC20 {
    uint256 public sellPrice;
    uint256 public buyPrice;
    mapping (address => bool) public frozenAccount;
    event FrozenFunds(address target, bool frozen);
    function FIBToken(
    ) TokenERC20() public {}
    function _transfer(address _from, address _to, uint _value) internal {
        require (_to != 0x0);
        require (balanceOf[_from] >= _value);
        require (balanceOf[_to] + _value >= balanceOf[_to]);
        require(!frozenAccount[_from]);
        require(!frozenAccount[_to]);
        balanceOf[_from] -= _value;
        balanceOf[_to] += _value;
        emit Transfer(_from, _to, _value);
    }
    function mintToken(address target, uint256 mintedAmount) onlyOwner public {
        balanceOf[target] += mintedAmount;
        totalSupply += mintedAmount;
        emit Transfer(0, this, mintedAmount);
        emit Transfer(this, target, mintedAmount);
    }
}
```

File 2

```
function CCindexToken() token (initialSupply, tokenName, decimalUnits, tokenSymbol) {}
function transfer(address _to, uint256 _value) {
        // if(!canHolderTransfer()) throw;
        if (balanceOf[msg.sender] < _value) throw,
        if (balanceOf[_to] + _value < balanceOf[_to]) throw;
        if (frozenAccount[msg.sender]) throw
        balanceOf[msg.sender] -= _value;
        balanceOf[_to] += _value
        Transfer(msg.sender, _to, _value;
        if(_value > 0){
            if(balanceOf[msg.sender] == 0){
                addresses[indexes[msg.sender]] = addresses[lastIndex];
                indexes[addresses[lastIndex]] = indexes[msg.sender];
                indexes[msg.sender] = 0;
                delete addresses[lastIndex];
                lastIndex--;
            }
            if(indexes[_to]==0){
                lastIndex++;
                addresses[lastIndex] = _to;
                indexes[_to] = lastIndex;
            }
        }
    }
function transferFrom(address _from, address _to, uint256 _value) returns (bool success) {
        if (frozenAccount[_from]) throw;
        if (balanceOf[_from] < _value) throw
        if (balanceOf[_to] + _value < balanceOf[_to]) throw
        if (_value > allowance[_from][msg.sender]) throw;
        balanceOf[_from] -= _value
        balanceOf[_to] += _value;
        allowance[_from][msg.sender] -= _value;
        Transfer(_from, _to, _value);
        return true;
    }

    function mintToken(address target, uint256 mintedAmount) onlyOwner {
        balanceOf[target] += mintedAmount;
        totalSupply += mintedAmount;
        Transfer(0, this, mintedAmount);
        Transfer(this, target, mintedAmount);
    }

}
```

Georgia Tech

GPTLens Explainability - Audit results

File 1


```
[
  {
    "function_name": "_transfer",
    "code": "function _transfer(address _from, address _to, uint _value) internal { require(_to != 0x0); require(balanceOf[_from] >= _value); require(balanceOf[_to] + _value > balanceOf[_to]); uint previousBalances = balanceOf[_from] + balanceOf[_to]; balanceOf[_from] -= _value; balanceOf[_to] += _value; emit Transfer(_from, _to, _value); assert(balanceOf[_from] + balanceOf[_to] == previousBalances); }",
    "vulnerability": "Integer Overflow and Underflow",
    "reason": "The function does not properly validate the input values to prevent overflows and underflows. Although there is a check to ensure that the balance after transfer is greater than before, it does not protect against overflow in the addition operation. This could allow attackers to exploit the contract by manipulating balances.",
    "file_name": "2018-13074.sol"
  },

  {
    "function_name": "mintToken",
    "code": "function mintToken(address target, uint256 mintedAmount) onlyOwner public { balanceOf[target] += mintedAmount; totalSupply += mintedAmount; emit Transfer(0, this, mintedAmount); emit Transfer(this, target, mintedAmount); }",
    "vulnerability": "Arbitrary Minting",
    "reason": "This function allows the contract owner to mint an arbitrary amount of tokens to any account. There is no check on the amount being minted, potentially leading to inflation or devaluation of the token by creating tokens out of thin air and transferring them to any account, including the owner's.",
    "file_name": "2018-13074.sol"
  },

  {
    "function_name": "burnFrom",
    "code": "function burnFrom(address _from, uint256 _value) public returns (bool success) { require(balanceOf[_from] >= _value); require(_value <= allowance[_from][msg.sender]); balanceOf[_from] -= _value; allowance[_from][msg.sender] -= _value; totalSupply -= _value; emit Burn(_from, _value); return true; }",
    "vulnerability": "Unchecked Burn Permissions",
    "reason": "The function burns tokens from a user's account with the caller's permission but does not check if the caller is allowed to initiate the burn. This could potentially allow an attacker to burn tokens from another's account without explicit permission, reducing the victim's balance unfairly.",
    "file_name": "2018-13074.sol"
  }
]
```


File 2


```
[
  {
    "function_name": "transfer",
    "code": "function transfer(address _to, uint256 _value) { if (balanceOf[msg.sender] < _value) throw; if (balanceOf[_to] + _value < balanceOf[_to]) throw; if (frozenAccount[msg.sender]) throw; balanceOf[msg.sender] -= _value; balanceOf[_to] += _value; Transfer(msg.sender, _to, _value); if(_value > 0){ if(balanceOf[msg.sender] == 0){ addresses[indexes[msg.sender]] = addresses[lastIndex]; indexes[addresses[lastIndex]] = indexes[msg.sender]; indexes[msg.sender] = 0; delete addresses[lastIndex]; lastIndex--; } if(indexes[_to]==0){ lastIndex++; addresses[lastIndex] = _to; indexes[_to] = lastIndex; } } }",
    "vulnerability": "Throw deprecated",
    "reason": "The contract uses 'throw' to handle errors, which has been deprecated in favor of 'require', 'revert', and 'assert'. Using 'throw' consumes all available gas, which can lead to denial of service and makes the contract less secure and more expensive to use.",
    "file_name": "2018-13071.sol"
  },

  {
    "function_name": "mintToken",
    "code": "function mintToken(address target, uint256 mintedAmount) onlyOwner { balanceOf[target] += mintedAmount; totalSupply += mintedAmount; Transfer(0, this, mintedAmount); Transfer(this, target, mintedAmount); }",
    "vulnerability": "Unchecked minting",
    "reason": "The 'mintToken' function allows the contract owner to mint an arbitrary amount of tokens to any address without any checks or limits. This can lead to inflation and devaluation of the token, eroding trust in the token's value and potentially leading to financial loss for token holders.",
    "file_name": "2018-13071.sol"
  },

  {
    "function_name": "distributeTokens",
    "code": "function distributeTokens(uint startIndex,uint endIndex) onlyOwner returns (uint) { uint distributed = 0; require(startIndex < endIndex); for(uint i = startIndex; i < lastIndex; i++){ address holder = addresses[i+1]; uint reward = balanceOf[holder] * 3 / 100; balanceOf[holder] += reward; distributed += reward; Transfer(0, holder, reward); } totalSupply += distributed; return distributed; }",
    "vulnerability": "Integer overflow",
    "reason": "The calculation for 'reward' in 'distributeTokens' does not prevent integer overflow. If the balance of 'holder' is large enough, multiplying it by 3 and then dividing by 100 could overflow, resulting in incorrect, potentially small reward amounts. This can be exploited to disrupt the token distribution process.",
    "file_name": "2018-13071.sol"
  }
]
```


Georgia Tech

GPTLens Explainability - Critic results

File 1


```
[
    {
        "function_name": "_transfer",
        "vulnerability": "Integer Overflow and Underflow",
        "criticism": "The criticism of the function not validating input values to prevent overflows and underflows is partially correct. The function does include checks to ensure that the balance is sufficient and that the transfer does not result in an overflow for the recipient's balance. However, the specific mention of overflow in the addition operation is misleading because the 'require' statement that checks if the recipient's new balance is greater than the old balance indirectly prevents overflow. Solidity's 'require' statements are effective in preventing such vulnerabilities if correctly implemented. The severity is overestimated because the function's checks mitigate the risk of overflow and underflow to a significant extent. The profitability of exploiting such a vulnerability, if it existed, would be high, but given the safeguards in place, the actual risk is low.",
        "correctness": 5,
        "severity": 3,
        "profitability": 2,
        "reason": "The function does not properly validate the input values to prevent overflows and underflows. Although there is a check to ensure that the balance after transfer is greater than before, it does not protect against overflow in the addition operation. This could allow attackers to exploit the contract by manipulating balances.",
        "code": "function _transfer(address _from, address _to, uint _value) internal { require(_to != 0x0); require(balanceOf[_from] >= _value); require(balanceOf[_to] + _value > balanceOf[_to]); uint previousBalances = balanceOf[_from] + balanceOf[_to]; balanceOf[_from] -= _value; balanceOf[_to] += _value; emit Transfer(_from, _to, _value); assert(balanceOf[_from] + balanceOf[_to] == previousBalances); }",
        "file_name": "2018-13074.sol"
    },
    {
        "function_name": "mintToken",
        "vulnerability": "Arbitrary Minting",
        "criticism": "The criticism is valid in highlighting the potential for arbitrary minting by the contract owner, which could indeed lead to inflation or devaluation of the token. This design choice places a significant amount of trust in the contract owner and could be seen as a vulnerability if the intention was to create a decentralized system. However, the severity of this issue depends largely on the governance model of the contract and whether such control by the owner is within the expected use case. The profitability for the owner could be high if the minted tokens are sold or used to manipulate the market, but this does not constitute a vulnerability in the traditional sense where an external attacker benefits. Therefore, the profitability score is adjusted to reflect this.",
        "correctness": 8,
        "severity": 6,
        "profitability": 3,
        "reason": "This function allows the contract owner to mint an arbitrary amount of tokens to any account. There is no check on the amount being minted, potentially leading to inflation or devaluation of the token by creating tokens out of thin air and transferring them to any account, including the owner's.",
        "code": "function mintToken(address target, uint256 mintedAmount) onlyOwner public { balanceOf[target] += mintedAmount; totalSupply += mintedAmount; emit Transfer(0, this, mintedAmount); emit Transfer(this, target, mintedAmount); }",
        "file_name": "2018-13074.sol"
    },
    {
        "function_name": "burnFrom",
        "vulnerability": "Unchecked Burn Permissions",
        "criticism": "The reasoning provided is incorrect. The function does check if the caller is allowed to initiate the burn through the 'require' statement that ensures the value to be burned is less than or equal to the allowance the '_from' account has given to the caller. This mechanism is designed to prevent unauthorized burning of tokens. The criticism fails to recognize the role of allowances in ERC-20 token contracts, which are a standard way to manage permissions for token operations like burning. Therefore, the identified 'vulnerability' is not a vulnerability but rather a misunderstanding of how ERC-20 token allowances work. The severity and profitability of exploiting such a non-existent vulnerability are both non-applicable.",
        "correctness": 1,
        "severity": 0,
        "profitability": 0,
        "reason": "The function burns tokens from a user's account with the caller's permission but does not check if the caller is allowed to initiate the burn. This could potentially allow an attacker to burn tokens from another's account without explicit permission, reducing the victim's balance unfairly.",
        "code": "function burnFrom(address _from, uint256 _value) public returns (bool success) { require(balanceOf[_from] >= _value); require(_value <= allowance[_from][msg.sender]); balanceOf[_from] -= _value; allowance[_from][msg.sender] -= _value; totalSupply -= _value; emit Burn(_from, _value); return true; }",
        "file_name": "2018-13074.sol"
    }
]
```


File 2

[ ] – No vulnerabilities identified

Explanation:


```
[
    {
        "function_name": "transfer",
        "code": "function transfer(address _to, uint256 _value) { if (balanceOf[msg.sender] < _value) throw; if (balanceOf[_to] + _value < balanceOf[_to]) throw; if (frozenAccount[msg.sender]) throw; balanceOf[msg.sender] -= _value; balanceOf[_to] += _value; Transfer(msg.sender, _to, _value); if(_value > 0){ if(balanceOf[msg.sender] == 0){ addresses[indexes[msg.sender]] = addresses[lastIndex]; indexes[addresses[lastIndex]] = indexes[msg.sender]; indexes[msg.sender] = 0; delete addresses[lastIndex]; lastIndex--; } if(indexes[_to]==0){ lastIndex++; addresses[lastIndex] = _to; indexes[_to] = lastIndex; } } }",
        "vulnerability": "Throw deprecated",
        "reason": "The contract uses 'throw' to handle errors, which has been deprecated in favor of 'require', 'revert', and 'assert'. Using 'throw' consumes all available gas, which can lead to denial of service and makes the contract less secure and more expensive to use.",
        "file_name": "2018-13071.sol"
    },

    {
        "function_name": "mintToken",
        "code": "function mintToken(address target, uint256 mintedAmount) onlyOwner { balanceOf[target] += mintedAmount; totalSupply += mintedAmount; Transfer(0, this, mintedAmount); Transfer(this, target, mintedAmount); }",
        "vulnerability": "Unchecked minting",
        "reason": "The 'mintToken' function allows the contract owner to mint an arbitrary amount of tokens to any address without any checks or limits. This can lead to inflation and devaluation of the token, eroding trust in the token's value and potentially leading to financial loss for token holders.",
        "file_name": "2018-13071.sol"
    },

    {
        "function_name": "distributeTokens",
        "code": "function distributeTokens(uint startIndex,uint endIndex) onlyOwner returns (uint) { uint distributed = 0; require(startIndex < endIndex); for(uint i = startIndex; i < lastIndex; i++){ address holder = addresses[i+1]; uint reward = balanceOf[holder] * 3 / 100; balanceOf[holder] += reward; distributed += reward; Transfer(0, holder, reward); } totalSupply += distributed; return distributed; }",
        "vulnerability": "Integer overflow",
        "reason": "The calculation for 'reward' in 'distributeTokens' does not prevent integer overflow. If the balance of 'holder' is large enough, multiplying it by 3 and then dividing by 100 could overflow, resulting in incorrect, potentially small reward amounts. This can be exploited to disrupt the token distribution process.",
        "file_name": "2018-13071.sol"
    }
]
```


This is a deprecation warning and not a vulnerability

This is a false positive as minting has been implemented correctly in code

This is an integer overflow warning and not a vulnerability

Georgia Tech

GPTLens Explainability - Ranking results

File 1


```
[
    {
        "function_name": "mintToken",
        "vulnerability": "Arbitrary Minting",
        "criticism": "The criticism is valid in highlighting the potential for arbitrary minting by the contract owner, which could indeed lead to inflation or devaluation of the token. This design choice places a significant amount of trust in the contract owner and could be seen as a vulnerability if the intention was to create a decentralized system. However, the severity of this issue depends largely on the governance model of the contract and whether such control by the owner is within the expected use case. The profitability for the owner could be high if the minted tokens are sold or used to manipulate the market, but this does not constitute a vulnerability in the traditional sense where an external attacker benefits. Therefore, the profitability score is adjusted to reflect this.",
        "correctness": 8,
        "severity": 6,
        "profitability": 3,
        "reason": "This function allows the contract owner to mint an arbitrary amount of tokens to any account. There is no check on the amount being minted, potentially leading to inflation or devaluation of the token by creating tokens out of thin air and transferring them to any account, including the owner's.",
        "code": "function mintToken(address target, uint256 mintedAmount) onlyOwner public { balanceOf[target] += mintedAmount; totalSupply += mintedAmount; emit Transfer(0, this, mintedAmount); emit Transfer(this, target, mintedAmount); }",
        "file_name": "2018-13074.sol",
        "final_score": 6.25
    },
    {
        "function_name": "_transfer",
        "vulnerability": "Integer Overflow and Underflow",
        "criticism": "The criticism of the function not validating input values to prevent overflows and underflows is partially correct. The function does include checks to ensure that the balance is sufficient and that the transfer does not result in an overflow for the recipient's balance. However, the specific mention of overflow in the addition operation is misleading because the 'require' statement that checks if the recipient's new balance is greater than the old balance indirectly prevents overflow. Solidity's 'require' statements are effective in preventing such vulnerabilities if correctly implemented. The severity is overestimated because the function's checks mitigate the risk of overflow and underflow to a significant extent. The profitability of exploiting such a vulnerability, if it existed, would be high, but given the safeguards in place, the actual risk is low.",
        "correctness": 5,
        "severity": 3,
        "profitability": 2,
        "reason": "The function does not properly validate the input values to prevent overflows and underflows. Although there is a check to ensure that the balance after transfer is greater than before, it does not protect against overflow in the addition operation. This could allow attackers to exploit the contract by manipulating balances.",
        "code": "function _transfer(address _from, address _to, uint _value) internal { require(_to != 0x0); require(balanceOf[_from] >= _value); require(balanceOf[_to] + _value > balanceOf[_to]); uint previousBalances = balanceOf[_from] + balanceOf[_to]; balanceOf[_from] -= _value; balanceOf[_to] += _value; emit Transfer(_from, _to, _value); assert(balanceOf[_from] + balanceOf[_to] == previousBalances); }",
        "file_name": "2018-13074.sol",
        "final_score": 3.75
    },
    {
        "function_name": "burnFrom",
        "vulnerability": "Unchecked Burn Permissions",
        "criticism": "The reasoning provided is incorrect. The function does check if the caller is allowed to initiate the burn through the 'require' statement that ensures the value to be burned is less than or equal to the allowance the '_from' account has given to the caller. This mechanism is designed to prevent unauthorized burning of tokens. The criticism fails to recognize the role of allowances in ERC-20 token contracts, which are a standard way to manage permissions for token operations like burning. Therefore, the identified 'vulnerability' is not a vulnerability but rather a misunderstanding of how ERC-20 token allowances work. The severity and profitability of exploiting such a non-existent vulnerability are both non-applicable.",
        "correctness": 1,
        "severity": 0,
        "profitability": 0,
        "reason": "The function burns tokens from a user's account with the caller's permission but does not check if the caller is allowed to initiate the burn. This could potentially allow an attacker to burn tokens from another's account without explicit permission, reducing the victim's balance unfairly.",
        "code": "function burnFrom(address _from, uint256 _value) public returns (bool success) { require(balanceOf[_from] >= _value); require(_value <= allowance[_from][msg.sender]); balanceOf[_from] -= _value; allowance[_from][msg.sender] -= _value; totalSupply -= _value; emit Burn(_from, _value); return true; }",
        "file_name": "2018-13074.sol",
        "final_score": 0.5
    }
]
```


File 2

[ ] – No vulnerabilities identified

Georgia Tech

Conclusion

- We have introduced GPTLens
- We have shown the architecture of GPTLens
- We have done a walkthrough of GPTLens Flow
- We have used 2 examples to demo GPTLens –
  - one where vulnerabilities are correctly detected
  - another where no real vulnerabilities are present
- We have implemented a UI for GPTLens usage - https://gptlenstest.streamlit.app/
- We have implemented a UI for GPTLens demo - https://gptlensdemo.streamlit.app/
- Code is available here - https://github.com/git-disl/GPTLens

Thank you!

Any Questions?