

# Lecture 10

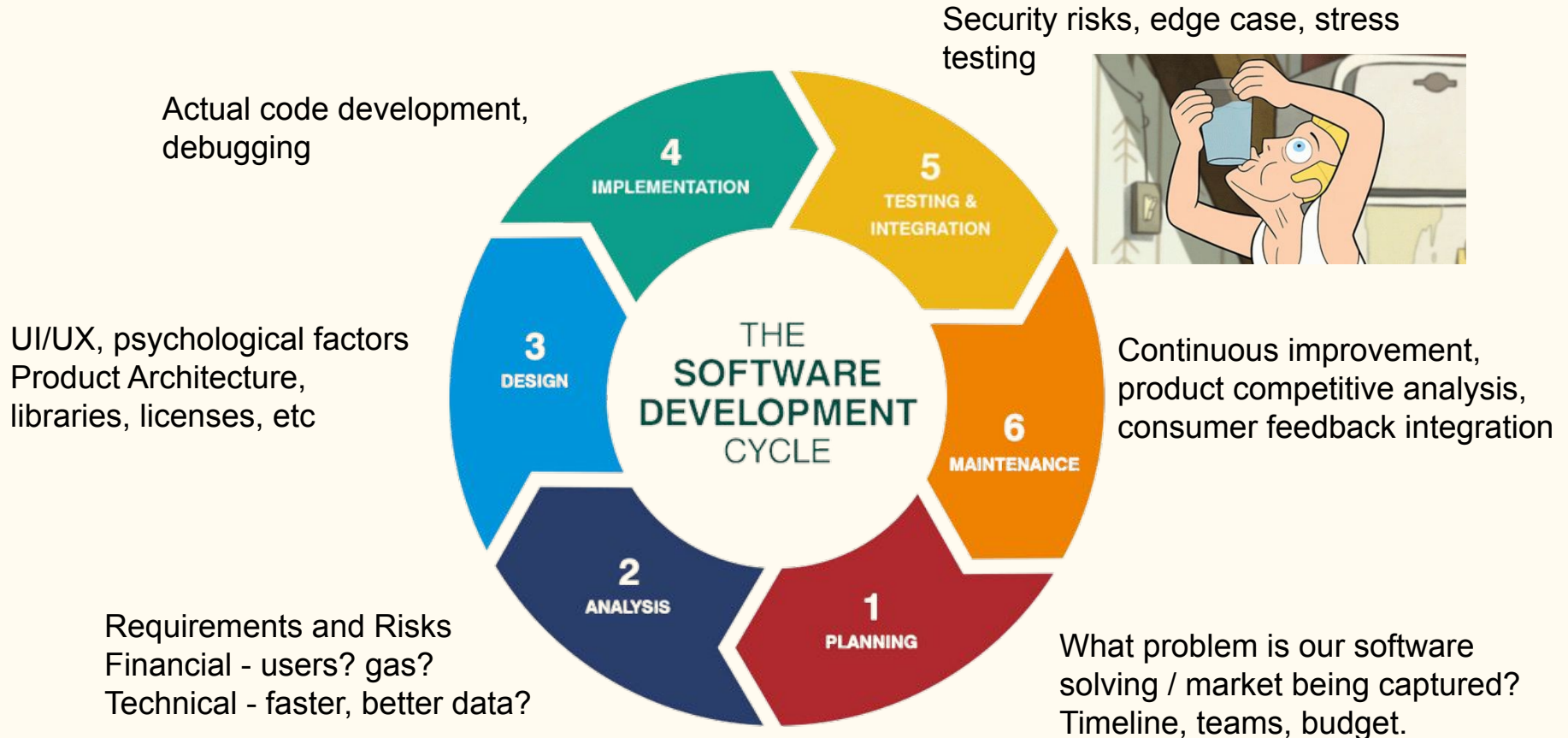
—

Software Testing

# Software Development Lifecycle

Idea to prototype

# 6 Stages of SDLC



# From DevOps to DevSecOps



## Plan and Develop

- ☐ Threat modelling
- ☐ IDE Security plugins
- ☐ Pre-commit hooks
- ☐ Secure coding standards
- ☐ Peer review

## Commit the code

- ☐ Static application security testing
- ☐ Security unit and functional tests
- ☐ Dependency management
- ☐ Secure pipelines

## Build and test

- ☐ Dynamic application security testing
- ☐ Cloud configuration validation
- ☐ Infrastructure scanning
- ☐ Security acceptance testing

## Go to production

- ☐ Security smoke tests
- ☐ Configuration checks
- ☐ Live Site Penetration testing

## Operate

- ☐ Continuous monitoring
- ☐ Threat intelligence
- ☐ Penetration testing
- ☐ Blameless postmortems

# Functional Tests

## Unit Test

Test small chunks - units. Can be a function, interface, module.

Write tests as you code!

Test expected failures and event emissions.

Test edge cases.

**Problem - cannot test private functions**

## Integration Test

Different modules tested as an entity.

Bottom-up or top-to-bottom approach.

Checks conditions unit testing cannot detect - code interaction

Either logical test suites or black box testing.

# Integrated Foundry Test Suite

## Fuzz Testing

Automatic input of extreme, malformed data into a function.

```
function  
testFuzz_name(uint x){}  
  
assume();
```

**runs:** # attempts  
 **$\mu$**  - mean gas used  
 **$\sim$**  - median gas used

## Invariant Testing

Project wide coverage instead of unit testing. Check for absolute conditions.

```
function invariant_name(){}  
  
config - runs, depth
```

**runs:** # attempts  
**calls:** # func called  
**revert:** # revert triggered

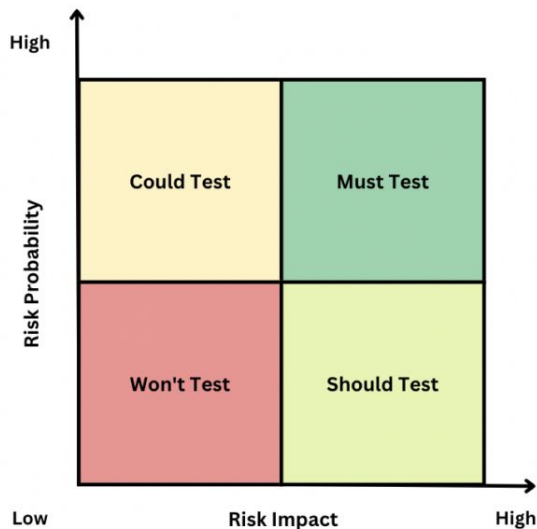
## Differential Testing

Cross reference your implementation with someone else's.

This requires execution of non-Solidity code using the ffi command to compare against Solidity code

# Test Coverage

Lines of code reached by test / total # of lines in project



Software Activities Prioritization

- "debug" flag prints out which functions were not tested.
- Discover which functions are called the most frequent - increase security, decrease gas cost
- Remove redundant tests
- Refactor unused functions

Code Coverage vs Test Coverage  
Quality vs Quantity  
100% is NOT the ideal

1. View summarized coverage:

```
forge coverage
```

2. Create lcov file with coverage data:

```
forge coverage --report lcov
```

3. Output uncovered code locations:

```
forge coverage --report debug
```

# Common Attacks



# Reentrancy Attack

Contract Bank {

```
function withdraw(uint amount){  
    msg.sender.call(value: amount);  
}  
}
```

1. Request  
withdraw



2. Bank sends,  
goes to fallback  
function



3. Fallback function  
calls withdraw  
again! Re-enters the  
Bank



Contract Attacker {

```
function attack() external payable{  
    withdraw();  
}  
fallback() external payable {  
    Bank.call("withdraw()");  
}  
}
```

- Uniswap/Lendf.Me hacks (April 2020) – \$25 mln, attacked by a hacker using a reentrancy.
- The BurgerSwap hack (May 2021) – \$7.2 million because of a fake token contract and a reentrancy exploit.
- The SURGEBNB hack (August 2021) – \$4 million seems to be a reentrancy-based price manipulation attack.
- CREAM FINANCE hack (August 2021) – \$18.8 million, reentrancy vulnerability allowed the exploiter for the second borrow.
- Siren protocol hack (September 2021) – \$3.5 million, AMM pools were exploited through reentrancy attack.

Simulate in Remix!

# Reentrancy Guards

## Check-Effect-Interaction coding pattern

```
contract ChecksEffectsInteractions {  
  
    mapping(address => uint) balances;  
  
    function deposit() public payable {  
        balances[msg.sender] = msg.value;  
    }  
  
    function withdraw(uint amount) public {  
        require(balances[msg.sender] >= amount);  
  
        balances[msg.sender] -= amount;  
  
        msg.sender.transfer(amount);  
    }  
}
```

### 1. Reentrancy Modifier

- a. A modifier which contains a bool which turns true the first time a function is call in a transaction
- b. Prevent recursive calls from external contracts and itself.

### 2. Pull payments

- a. Do not allow direct push contract interactions
- b. To withdraw, users/contracts make a payment request and then call a separate withdraw pool themselves instead of the contract sending.

### 3. Pausable Contract

- a. Contract logic of what to do in an emergency

# Denial of Service Attacks - Forced Revert

Very frequent on the internet, not just web3. Cause the service to fail from unexpected behaviour.

```
// INSECURE
contract Auction {
    address currentLeader;
    uint highestBid;

    function bid() payable {
        require(msg.value > highestBid);

        require(currentLeader.send(highestBid)); // Refund the old leader, if it fails then revert

        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```

What if the current leader has a fallback function that rejects any attempt to send money?

# Denial of Service Attacks - Gas Limit

Choke the contract with requests → Remember Black Thursday?

How about a bot contracts which floods the blockchain with useless blocks for a set period of time?

1. On May 1, 2022, a [DDoS attack brought down the Solana network](#) for seven hours due to a non-fungible token (NFT) minting tool, "Candy Machine." The downtime was reportedly caused by bots taking over the tool and flooding the network with over 4,000,000 transactions per second at its peak before it went down.
2. On September 14, 2021, [Arbitrum One experienced a DDoS attack](#), one of Ethereum's largest layer 2 protocols. Sequencer — the entity which receives and reorders transactions on its network — was overwhelmed by a flood of transactions that brought down the network for nearly an hour.

## The dangers of dynamic arrays - a payout for contributors example

```
struct Payee {  
    address addr;  
    uint256 value;  
}
```

↑ Back to top

```
Payee[] payees;  
uint256 nextPayeeIndex;
```

```
function payOut() {  
    uint256 i = nextPayeeIndex;  
    while (i < payees.length && gasleft() > 200000) {  
        payees[i].addr.send(payees[i].value);  
        i++;  
    }  
    nextPayeeIndex = i;  
}
```

# Griefing - Business Logic Attack

1. Malicious but compliant attacks to disrupt business operations.

- A group staking contract which pays dividends 24 hours after last deposit.
- Attacker keeps on putting tiny deposit amount to ensure payouts keeps on getting delayed.

2. Insufficient Gas Griefing - Wallet and Relayers

- A malicious relayer can first pose as legitimate transaction handler, get wallets to send them transactions, then purposefully set a small gas limit, resulting in no transactions ever go through.

3. Self destructing contracts

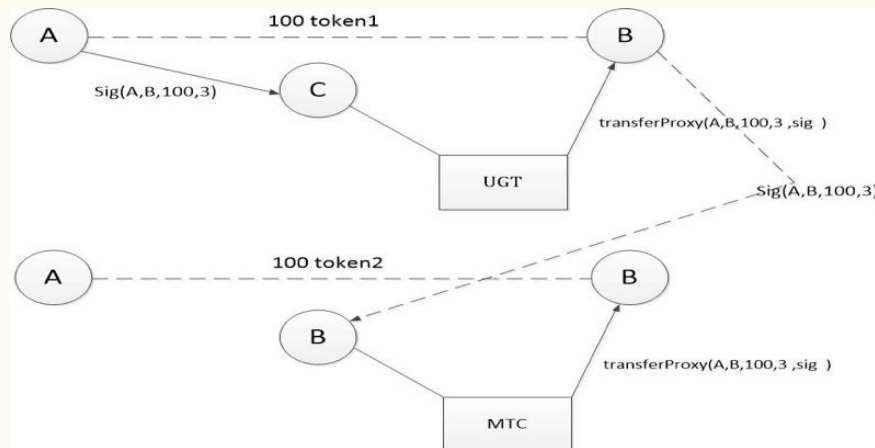
- The self destruct command erases a smart contract and force sends the remaining ether to a target address. Donation Game.

# Replay Attack - Importance of nonces

## Signed Transaction

```
{  
  nonce: "0x0",  
  maxFeePerGas: "0x1234",  
  maxPriorityFeePerGas: "0x1234",  
  gas: "0x55555",  
  to: "0x07a565b7ed7d7a695fe0",  
  value: "0x1234",  
  data: "0xabcd",  
  v: "0x26",  
  r: "0x223a7c9bcf20e",  
  s: "0x28cc7704971491663",  
  hash: "0xeb0d46df3870f8a30e"  
}
```

```
function action(uint256 _param1, bytes32 _param2, bytes memory _sig) external {  
    bytes32 hash = keccak256(abi.encodePacked(_param1, _param2));  
    bytes32 signedHash = hash.toEthSignedMessageHash();  
    address signer = signedHash.recover(_sig);  
  
    require(signer == owner, "Invalid signature");  
  
    // use `param1` and `param2` to perform authorized action  
}
```

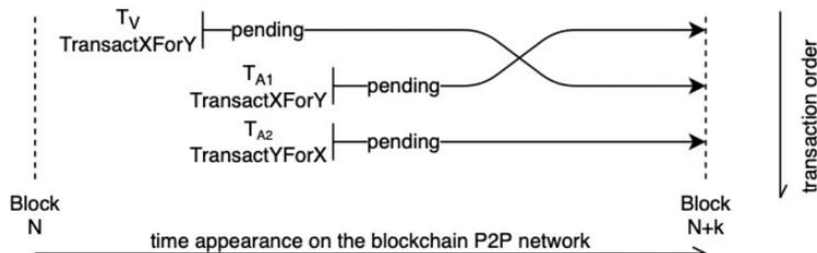


# Toxic MEV - Front Running

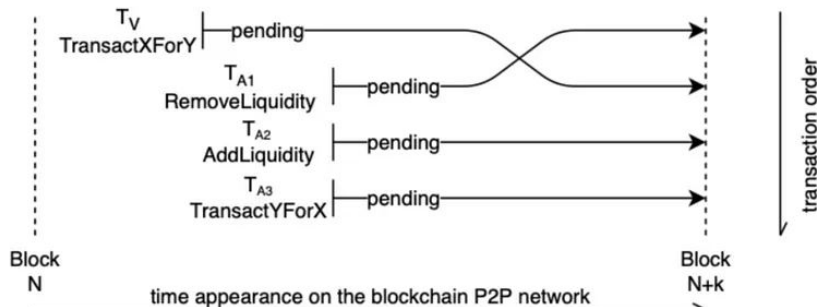
- Scan for a winning transaction
- Simulate the transaction on a fork of the chain
- Submit your transaction with a much higher gas limit

```
function guess(uint256 number) public payable {  
    require(msg.value == 1 ether, "Submission fee required");  
    uint256 balance = address(this).balance;  
    require(balance != 0, "Game has ended");  
  
    bytes32 userChallenge = keccak256(abi.encode(number));  
    if (userChallenge == challenge) {  
        (bool success, ) = msg.sender.call{value: balance}("");  
        require(success, "Transfer failed");  
    }  
}
```

# Toxic MEV - Sandwich Attack



Sandwich attack strategy 1, when liquidity taker attacks taker



Sandwich attack strategy 2, when liquidity provider attacks taker

1. Victim identifies arbitrage opportunity
2. Copy the transaction and pay higher gas
3. victim suffers slippage
4. Two transactions lowers price of Y against X, trade all Y for X.

1. Provider sees large order for Y.
2. Remove all liquidity for Y in that pool.
3. Victim suddenly pays high X for little Y.
4. Provider refills pool with Y, normalizing price again.
5. Buys X with remaining Y.



# Code level vulnerabilities

- ❖ Integer under and overflow, patched in Solidity 0.8.0 onwards
- ❖ Pseudo-random number generation
  - block level parameters can be deduced/calculated.
    - `block.timestamp`
    - `block.number`
- ❖ Incorrect parameter ordering
  - Solidity do not name parameter inputs. Make sure the ordering is correct for structs and functions with many inputs.

# Defensive Programming

- Aim for simplicity
  - Less lines of code is cheaper
  - Less code = less room for mistakes
- Be as private as possible
  - Public functions and variables are expensive and potentially open to manipulation. Always try to keep things private.
  - Use external / internal above public.
- Reuse secure code
  - Copy code from the internet. But copy secure code. Openzeppelin, Safe, Eth, etc.
  - Refactor as much as possible. Aggregate similar features into a library.
- Write readable code
  - Natspec commenting as much as possible
  - Clear variable names, follow conventions
- Test! Test! Test!

# Cyber Security & Smart Contract Auditing

# Vulnerability Registries

Cyber Security is a never ending race of discovering vulnerabilities. All vulnerabilities are tracked on registries.

- CWE, CVE, OWASP Top 10, NVD, [SWC](#), [SCSVS](#)

Registries differ by area (OWASP is web), vulnerability approach (CWE - flaws/weakness pattern vs CVE - named vulnerability)

Last 20 Scored Vulnerability IDs & Summaries	CVSS Severity
<b>CVE-2024-21346</b> - Win32k Elevation of Privilege Vulnerability <b>Published:</b> February 13, 2024; 1:15:50 pm -0500	V3.1: <b>7.8 HIGH</b>
<b>CVE-2024-20679</b> - Azure Stack Hub Spoofing Vulnerability <b>Published:</b> February 13, 2024; 1:15:47 pm -0500	V3.1: <b>6.5 MEDIUM</b>
<b>CVE-2024-20684</b> - Windows Hyper-V Denial of Service Vulnerability <b>Published:</b> February 13, 2024; 1:15:47 pm -0500	V3.1: <b>6.5 MEDIUM</b>

# CVSS - Common Vulnerability Scoring System

## CVSS SCORE METRICS

A CVSS score is composed of three sets of metrics (**Base**, **Temporal**, **Environmental**), each of which have an underlying scoring component.



# Static / Dynamic Scanners

## Static Scanning

- In depth code analysis
- Auto detection of vulnerable coding practices
- Building of intermediate execution flows and recognise attacks
- Taint analysis - detect injection attacks which corrupt data and memory

## Dynamic Scanning

- User POV testing
- Send malicious requests
- Network attacks and stress testing
- User behaviour modelling. In the real world, perhaps your user fell victim to social engineering?

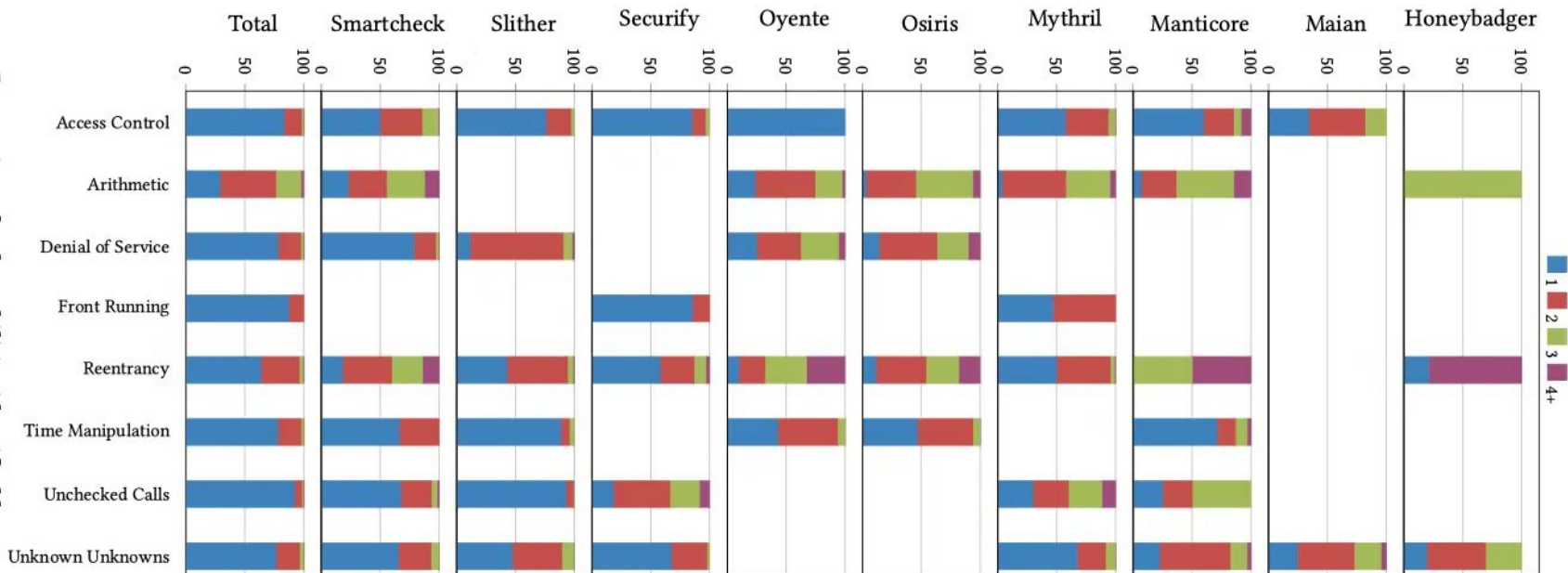


[List of scanners](#)



# No solution is perfect

Figure 1: Proportion of vulnerabilities identified by exactly one (1), two (2) or three (3) tools, and by four tools or more (4+).



Accuracy, speed, vulnerability type & age

Source

# Audit Lifecycle

## 1. Documentation

Code freeze and scope agreement.

Provide all specifications, system architecture and intended execution of system.

## 2. Automated Testing

Dynamic and Static scanning.

Formal Verification.

Custom test suites.

## 3. Manual Review

Humans are better at detecting logic errors, poor coding patterns and business manipulations like gas attacks.

Analysis of code with consultation to original goals and specifications.

## 4. Vulnerability Classification

A list of vulnerabilities with their risk levels.

Provide recommended resolution techniques.

### **\*\* Note:**

From a business perspective, each vulnerability is a cost. Weigh it against the cost of the exploit.

## 5. Reporting

Provide initial report with findings.

Provide further consulting for the client.

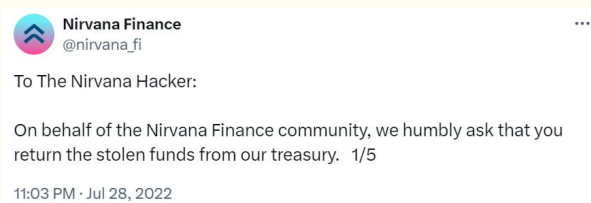
Most Web3 projects release their audits publicly to gain confidence from the public.

Cost: \$5000 - \$15000



# Social Bug Finding

## The "please give our money back" method



## First account of Indictment



### Payouts

There is a 2X bounty multiplier until September 30th, \$50,000. The payouts vary according to severity. For an additional client through an on-chain data request, an additional

#### Severity Payment in LINK

Low	\$2,000
Medium	\$4,000
High	\$8,000
Critical	\$16,000



Sign up now!