Advanced Topics in Software Engineering
CSE - 6324 - Section 001

TEAM 4

FINAL ITERATION
(Written Deliverable)

<u>Team Members</u>

Suvarna, Arjun - 1002024437
Sinari, Navina - 1002072310
Palnati, Netra - 1002030626
Waje, Sanjana - 1002069940

I. Project Objective

Slither is a static analysis framework that provides rich information about Ethereum smart contracts [1]. Slither's plugin architecture lets you integrate new detectors from the command line [2].

We propose to add three new detectors using the Python API [3].

The first detector will be targeted toward the improper usage of the ecrecover() function in solidity smart contracts [4]. The ecrecover function can be used to process meta transactions in smart contracts and verify signed data coming from someone other than the transaction signer [5], but due to the nature of public key private key signatures, this function is vulnerable to replay attacks [11]. The proposed detector will detect usage of the ecrecover function without nonce or chain id and flag it as a security vulnerability. The detector will also check the validity of the ECDSA signature based on the conditions defined in the Ethereum yellow paper [16]. This detector is based on the pull request (#2015) [17] which checks for nonce protection. Chain ID and ECDSA signature verification are added to this detector to expand its use cases.

The second detector will detect API Keys stored as plain text within the solidity code (SWC 136) [6]. Hardcoded credentials present a grave security risk and compromise the security setup associated with the keys [6]. Since the inherent nature of smart contracts is public, any hard-coded secret is not safe irrespective of whether it is stored in a public or private variable [8]. The second detector will target common API key types.

The third detector will detect passwords hardcoded into the solidity code (SWC 136) [6]. Hardcoded passwords in solidity smart contracts are not private and can be seen by third parties due to the public nature of smart contracts [7] [8]. The third detector will specifically target passwords.

II. Project Plan

Task	Start	End	Current Status	Risk Associated
Setup Development Environment	Inception	Inception	Completed	Changing Solidity Standards
Define rules for improper use of ecrecover() detector	Iteration 1	Iteration 1	Completed	Incomplete rule definition
Build improper use of ecrecover() detector (SWC-121)	Iteration 1	Iteration 1	Completed	1. Incomplete rule definition 2. False Positives 3. False Negatives 4. Change in security standards
Test and iterate improper use of ecrecover() detector	Iteration 1	Iteration 2	Completed	Change in security standards
Define Rules for hardcoded API Key detector	Iteration 1	Iteration 2	Completed	Change in security standards
Define Rules for hardcoded password detector	Iteration 1	Iteration 2	Completed	Change in security standards
Review and implement feedback from Iteration 1	Iteration 2	Iteration 2	Completed	
Build hardcoded API Key detector	Iteration 2	Iteration 2	Completed	1. False Positives 2. False Negatives
Build hardcoded Password Detector	Iteration 2	Final Iteration	Completed	1. False Positives 2. False Negatives
Review and implement feedback from Iteration 2	Final Iteration	Final Iteration	Completed	

III. Risk Mitigation Plan

Risk	Risk Description	Mitigation Plan	Risk Exposure
Incomplete rule definition	Rules for detectors could miss edge cases.	Iteratively build and test the detectors with rules defined.	There is a 30% probability that this issue might occur, and we would need 10 hours to fix it. Hence, risk exposure would be 3 extra hours.
Changing Solidity Standards	Changing standards in solidity and Ethereum could make the detector and its recommendations obsolete.	Fix Solidity version 0.8.18 as the target version.	There is a 20% probability that this issue might occur, and we would need 10 hours to fix it. Hence, risk exposure would be 2 extra hours.
Missing Slither User's needs from the detector's	Rules defined in the detector may not meet the slither user's needs for analyzing smart contracts.	Thoroughly research the issue and interact with slither users to understand detector needs better.	There is a 10% probability that this issue might occur, and we would need 10 hours to fix it. Hence, risk exposure would be 1 extra hour.
False Positive	The detector might detect that there is a data leak that does not exist.	Test with multiple negative scenarios.	We need 4 hours to resolve this problem, which has a 20% chance of happening. Thus, 0.8 hours of risk exposure.
False Negatives	The detector may not detect that there is a data leak that exists.	Test with multiple positive scenarios.	There is a 10% chance that this problem would arise, and it would take 4 hours to fix. Hence, risk exposure would be 0.4 extra hours.

IV. Specification and Design

1. Ecrecover Detector

In iteration 1, we checked for the improper usage of the ecrecover() function, which checked for certain conditions. The conditions are:

i. ECDSA check: checks the validity of the signature based on the conditions defined in the Ethereum yellow paper [16] and checks implemented in openzeppelin [20][22] implementation of ECDSA signature validation. This check prevents signature malleability by enforcing the bounds for the keys in the ECDSA signature. Without this check, the ecrecover function would allow non-unique signatures [20].

ii. ChainId check: By adding chain ID into the payload, replay attacks can be prevented by sending a transaction from a different network as a valid transaction [18].
Chain ID is unique for every chain so a transaction with chain ID would prevent transactions of different chains being replayed on another chain [18].

We are using a solidity smart contract as input to our detector. For iteration 1, we wrote the code into ‘ecrecover.sol’ created in the pull request #2015 [17]. To analyze and detect the improper usages we have defined various functions that satisfy either of these conditions, none of them, or both in our file.

In iteration 2 we added additional conditions for the ECDSA key validity check to prevent signature malleability described in the yellow paper [16], this is based on the open zeppelin implementation [20] for the ECDSA signature check.

A few code snippets of the ‘ecrecover.sol’ are below.

```
//Missing Chain ID or Nonce and valid ECDSA signature validation
function bad_missingChainId_missingNonce_withECDSACheck(
    Info calldata info,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
    bytes32 data = keccak256(
        abi.encodePacked(
            "\x19Ethereum Signed Message:\n32",
            keccak256(abi.encode(info))
        )
    );
    if((r>0 &&
    uint256(s) <= 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0
    && s>0 && (v==0 || v==1))){
        address receiver = ecrecover(data, v, r, s);
        require(receiver != address(0), "ECDSA: invalid signature");
        mint(info.tokenId, receiver);
    }
}
```

```
//Valid with Chain Id and ECDSA singature validation
function good_withChainId_withECDSACheck(
    Info calldata info,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
    bytes32 hash = keccak256(abi.encode(info,chainId));
    bytes32 data = keccak256(
        abi.encodePacked("\x19Ethereum Signed Message:\n32", hash, chainId)
    );
    if((r > 0 && uint256(s) <= 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0 &&
    s>0 && (v == 0 || v == 1))){
        address receiver = ecrecover(data, v, r, s);
        require(receiver != address(0), "ECDSA: invalid signature");
        mint(info.tokenId, receiver);
    }
}
```

```
//Valid Nonce and  missing ECDSA singature validation
function bad_withNonce_missingECDSACheck(
    Info calldata info,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
    bytes32 data = keccak256(
        abi.encodePacked(
            "\x19Ethereum Signed Message:\n32",
            keccak256(abi.encode(info,nonces[msg.sender]++))
        )
    );
        address receiver = ecrecover(data, v, r, s);
        require(receiver != address(0), "ECDSA: invalid signature");
        mint(info.tokenId, receiver);

}
```

```
    //Valid with Nonce and ECDSA singature validation
    function good_withNonce_withECDSACheck(
        Info calldata info,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
        bytes32 hash = keccak256(abi.encode(info,nonces[msg.sender]++));
        bytes32 data = keccak256(
            abi.encodePacked("\x19Ethereum Signed Message:\n32", hash, nonces[msg.sender]++)
        );
        if((r > 0 && s>0 &&
        uint256(s) <= 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0 &&
        (v == 0 || v == 1))){
            address receiver = ecrecover(data, v, r, s);
            require(receiver != address(0), "ECDSA: invalid signature");
            mint(info.tokenId, receiver);
        }
    }
```

The detector analyzed the code and detected missing conditions for every function. The output of the ecrecover detector analysis is given below.

```
navina@Navinas-MacBook-Air code % slither ecrecover.sol --detect ecrecover
'solc --version' running
'solc ecrecover.sol --combined-json abi,ast,bin,bin-runtime,srcmap,srcmap-runtime,userdoc,de
vdoc,hashes --allow-paths .,/Users/navina/Downloads/Adv SE/Code' running
Compilation warnings/errors on ecrecover.sol:
Warning: SPDX license identifier not provided in source file. Before publishing, consider ad
ding a comment containing "SPDX-License-Identifier: <SPDX-License>" to each source file. Use
 "SPDX-License-Identifier: UNLICENSED" for non-open-source code. Please see https://spdx.org
 for more information.
--> ecrecover.sol


INFO:Detectors:
A.bad_withChainID_missingECDSACheck(A.Info,uint8,bytes32,bytes32).receiver (ecrecover.sol#35
) lacks a zero-check on :
                - receiver = ecrecover(bytes32,uint8,bytes32,bytes32)(data,v,r,s) (ecrecover
.sol#35)
A.bad_withChainID_missingECDSACheck(A.Info,uint8,bytes32,bytes32).receiver (ecrecover.sol#35
) lacks a r > 0 ecdsa check on :
                - receiver = ecrecover(bytes32,uint8,bytes32,bytes32)(data,v,r,s) (ecrecover
.sol#35)
A.bad_withChainID_missingECDSACheck(A.Info,uint8,bytes32,bytes32).receiver (ecrecover.sol#35
) lacks a s > 0 ecdsa check on :
                - receiver = ecrecover(bytes32,uint8,bytes32,bytes32)(data,v,r,s) (ecrecover
.sol#35)
A.bad_withChainID_missingECDSACheck(A.Info,uint8,bytes32,bytes32).receiver (ecrecover.sol#35
) lacks a v ecdsa check on :
                - receiver = ecrecover(bytes32,uint8,bytes32,bytes32)(data,v,r,s) (ecrecover
.sol#35)
A.bad_missingChainID_missingECDSACheck(A.Info,uint8,bytes32,bytes32).receiver (ecrecover.sol
#49) lacks a zero-check on :
                - receiver = ecrecover(bytes32,uint8,bytes32,bytes32)(data,v,r,s) (ecrecover
.sol#49)
A.bad_missingChainID_missingECDSACheck(A.Info,uint8,bytes32,bytes32) (ecrecover.sol#39-51) l
acks a nonce or chainId on :
                - data = keccak256(bytes)(abi.encodePacked(Ethereum Signed Message:
```

2. API Key Detector

In Iteration 2 we developed an API key detector based on implementing a sonar-secrets API Key flag for Java code [21]. The detector looks for conditions based on the name of the variable storing the API key.
In the final iteration, we added a functionality to detect popular API keys based on the API key value [23].

The conditions are:

i. The variable has names like api_key, api_token, google_key, or any other combination of api|gitlab|github|slack|google|aws|jenkins **with** '_' and key|token|secret|auth.

ii. The variable value matches the regex pattern for AWS secrets [23]:

Amazon AWS: i. AKIA[0-9A-Z]{16} (Client ID)
ii. [0-9a-zA-Z/+]{40} (Secret Key)

```
function useAPI() external {

    //Dummy API Keys generated from : https://api-key.me/index
    //for testing purpose

    api = string("d75441fc38f744439061754630373a63");
    api23 = string("a2ba819c35076f908b0822cd93c233d9");
    google_key = string("9415b6319fdc8f53200e1d6fe1d3d7e3");

    // AWS Client ID and secret key example fetched from
    // https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html
    a = "AKIAIOSFODNN7EXAMPLE"; // AWS Client ID
    b = "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"; // AWS Secret key
}

function doesNotUseAPI(bytes32 t) external {
    tempId = t;
}
```

The output of the API Key detector analysis is given below:

```
INFO:Detectors:
APIData.useAPI() (apikeys.sol#16-29)
 has hardcoded API key :
                - api = string(d75441fc38f744439061754630373a63) (apikeys.sol#21)
                - api23 = string(a2ba819c35076f908b0822cd93c233d9) (apikeys.sol#22)
                - google_key = string(9415b6319fdc8f53200e1d6fe1d3d7e3) (apikeys.sol#23)
                - a = AKIAIOSFODNN7EXAMPLE (apikeys.sol#27)
                - b = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY (apikeys.sol#28)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
INFO:Slither:apikeys.sol analyzed (1 contracts with 1 detectors), 1 result(s) found
```

3. Password Detector

In Iteration 3 we developed a password detector based on the implementation of sonar-secrets for detecting passwords [21]. The detector looks for conditions based on the name of the variable storing the password.

The conditions are:

i. The variable has names like password, pass, pwd, passwd, and key

```
// Hardcoded passwords stored in variables 'password', 'pass', 'pwd' etc
function usePassword() external {
    password = "password@123";
    pass = "uta@2023";
    pwd = "abc#765";
    key = "safePass";
}

function doesNotUsePassword(string memory p) external {
    value = p;
}
```

The output of the password detector analysis is given below:

```
INFO:Detectors:
Passwords.usePassword() (passwords.sol#14-19)
 has hardcoded password :
                - password = password@123 (passwords.sol#15)
                - pass = uta@2023 (passwords.sol#16)
                - pwd = abc#765 (passwords.sol#17)
                - key = safePass (passwords.sol#18)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
INFO:Slither:passwords.sol analyzed (1 contracts with 1 detectors), 1 result(s) found
```

V. Code and Tests

i. Setup:

There are various versions available for the solc compiler [19].

```
navina@Navinas-MacBook-Air code % solc-select install
Available versions to install:
0.3.6
0.4.0
0.4.1
0.4.2
```

We are using version 0.8.18, which can be installed using the following command.

```
navina@Navinas-MacBook-Air code % solc-select install 0.8.18
Installing solc '0.8.18'...
Version '0.8.18' installed.
```

To use the specific version, we need to use the following command.

```
navina@Navinas-MacBook-Air code % solc-select use 0.8.18
Switched global version to 0.8.18
```

The next step is to add our ecrecover python file, ‘ecrecover.py’, and API key detector python file, ‘apikey.py’ which has the detector code, in ‘all_detectors.py’ which would be in the local directory where Slither is installed [2]. In our case, the path is: ‘/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-packages/slither/detectors/all_detectors.py‘ and then move our detector code to the local directory path ‘/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-packages/slither/detectors/statements’.

After the installations and setup, we are taking ‘ecrecover.sol’ as input and running slither only for the ecrecover detector using the following command.

```
navina@Navinas-MacBook-Air code % slither ecrecover.sol --detect ecrecover
```

Similarly for the API Key detector, we are taking ‘apikeys.sol’ as input and running slither only for the API Key detector using the following command.

```
navina@Navinas-MacBook-Air Code % slither apikeys.sol --detect apikey
```

For the password detector, we are taking ‘passwords.sol’ as input and running slither only for the password detector using the following command.

```
@Sanjanas-MacBook-Air-2 Adv se % slither passwords.sol --detect passwords
```

ii. Code:

1. Ecrecover Detector

The code is built on ecrecover.py from pull request #2015 [17].

The detector looks for uses of ecrecover() which if returns true will check if the keccak256 function is called and checks for the nonce or chain id conditions as well as the ECDSA signature validation. The ecrecover detector code snippets are as follows:

```python
#Checks the ECDSA validation for the r value and returns true or false
def _r_ecdsa_validation(var: LocalVariable, function: Function) -> bool:
    for node in function.nodes:
        if node.contains_if():
            for ir in node.irs:
                expression = str(ir.expression)
                if isinstance(ir, Binary):
                    if (
                        ir.type == BinaryType.GREATER or ir.type == BinaryType.LESS
                    ) and "r > 0" in expression or "0 < r" in expression:
                        return True
    return False
```

```python
#Checks the ECDSA validation for the s value and returns true or false
def _s_ecdsa_validation(var: LocalVariable, function: Function) -> bool:
    for node in function.nodes:
        if node.contains_if():
            for ir in node.irs:
                expression = str(ir.expression)
                if isinstance(ir, Binary):
                    if (
                        ir.type == BinaryType.GREATER or ir.type == BinaryType.LESS or
                        ir.type == BinaryType.LESS_EQUAL or ir.type == BinaryType.GREATER_EQUAL
                    ) and (("s > 0") in expression or ("0 < s") in expression) or (
                    ("uint256(s) <= 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0"
                    ) in expression):
                        return True
    return False
```

```python
#Checks the ECDSA validation for the v value and returns true or false
def _v_ecdsa_validation(var: LocalVariable, function: Function) -> bool:
    for node in function.nodes:
        if node.contains_if():
            for ir in node.irs:
                expression = str(ir.expression)
                if isinstance(ir, Binary):
                    if (
                        ir.type == BinaryType.EQUAL
                    ) and ("v == 0" in expression) or ("v == 1" in expression):
                        return True
    return False
```

```python
    #If ecrecover function is called
    if SolidityFunction("ecrecover(bytes32,uint8,bytes32,bytes32)") in function.solidity_calls:
        address_list = []
        var_nodes = defaultdict(list)
        for node in function.nodes:
            if SolidityFunction("ecrecover(bytes32,uint8,bytes32,bytes32)") in node.solidity_calls:
                for var in node.local_variables_written:
                    if "ecrecover(bytes32,uint8,bytes32,bytes32)" in str(var.expression):
                        address_list.append(var)
                        var_nodes[var].append(node)

            #If keccak256 hash function is called
            if SolidityFunction("keccak256(bytes)") in node.solidity_calls:
                for ir in node.irs:
                    if isinstance(ir, SolidityCall) and ir.function == SolidityFunction(
                        "keccak256(bytes)"
                    ):
                        # Checking if nonce or chainId is present in the expression
                        if "nonce" not in str(ir.expression) and "chainId" not in str(ir.expression):
                            nonce_results.append(node)
```

2. API Key Detector

The detector finds the hardcoded API keys in smart contracts. It scans the contract code and looks for the pattern or regular expression with certain keywords.

The detector implementation is based on ecrecover.py from pull request #2015 [17] and the ecrecover.py from ecrecover detector.

The regular expression is built upon the regular expression used by sonar secrets to detect API Keys in Java code [21].

In addition to detecting variable names for possible API keys, AWS API Client ID and Client secrets are detected based on the pattern for these keys as described in “Detecting and Mitigating Secret-Key Leaks in Detecting and Mitigating Secret-Key Leaks in Source Code Repositories” [23] with keys from AWS reference documentation [24].

The detector code snippet is as follows:

```
def _detect_apikey(function: Function,) -> List[Tuple[Function, DefaultDict[LocalVariable, List[Node]]]]:
    results = []

    for node in function.nodes:
        for ir in node.irs:
            expression = str(ir.expression)
            # Regular expressions used:
            # 1. Sonar qube
            # 2. AWS Client ID format
            # 3. AWS Secret key format
            if re.compile('(api|gitlab|github|slack|google|aws|jenkins)_?(key|token|secret|auth)?'
                    ).search(expression) or re.compile(
                    'AKIA[0-9A-Z]{16}').search(expression) or re.compile(
                    '[0-9a-zA-Z/+]{40}').search(expression):
                results.append(node)

    return (results)
```

3. Password Detector

The detector finds the hardcoded passwords in smart contracts. It scans the contract code and looks for the pattern or regular expression with certain keywords.

The regular expression is built upon the regular expression used by sonar secrets to detect passwords in Java code [21].

The detector code snippet is as follows:

```
def _detect_password(function: Function,) -> List[Tuple[Function, DefaultDict[LocalVariable, List[Node]]]]:
    results = []

    password_pattern = re.compile(r'(password|passwd|pass|pwd|key)', re.IGNORECASE)

    for node in function.nodes:
        for ir in node.irs:
            expression = str(ir.expression)

            if password_pattern.search(expression):
                results.append(node)

    return (results)
```

iii. Test Cases

No.	Test Case	Iteration	Expected Output
1.	Running the pull request (#2015) on the solidity code.	Iteration 1	Analyzing the usage of nonce and zero address checks.
2.	Implementing the chainId check on the pull request.	Iteration 1	Analyzing the slither detector with chainId check.
3.	Implementing the ECDSA check on the pull request.	Iteration 1/2	Analyzing the slither detector with ECDSA check.
4.	Implementing API key detector with API key regular expressions based on sonar secrets [21].	Iteration 2	Analyzing the slither detector with API Key check.
5.	Implementing API key detector for API key values from popular APIs [23].	Final Iteration	Analyzing slither deter with AWS [24] Client ID and secrets.
6.	Implementing a hardcoded password detector with password regular expressions based on sonar secrets [21].	Final Iteration	Analyzing the slither detector with hardcoded password check.

VI. Competitors

i. Slither:- The current stable build of slither (0.9.6) [9], has a robust set of public detectors like Dangerous usage of tx. origin and Unchecked low-level calls [10]. But Slither currently does not have a public detector for API Keys, Passwords, or a public detector for improper usage of the ecrecover function as described in the GitHub Issue #1950 [4].

ii. Mythril:- Mythril is a security analysis tool for Ethereum smart contracts [12]. It has a set of modules to detect vulnerabilities and issues in the Ethereum solidity code like External Calls and Unprotected Ether Withdrawal [13]. But there are no modules to detect unprotected use of the ecrecover function or hardcoded credentials.

iii. MythX:- MythX is a tool that scans for security vulnerabilities in Ethereum and other EVM-based blockchain smart contracts [14]. However, it does not have a detector available for improper use of ecrecover or hardcoded API keys and passwords [15]. MythX is a closed-source tool.

VII. Customers and Users

User	Role	Feedback
Mehul Hivlekar	New User to Smart Contracts and Solidity	All features implemented in the project were useful and interesting.
Shruthaja Patali Rao	CSE-6324-001 Team 5	All 3 detectors will help increase security in smart contracts.
Devyani Singh	CSE-6324-001 Team 8	Enhancement to API key detectors will make it more effective but more popular API key patterns can be added.

VIII. GitHub Repository

https://github.com/arjunsuvarna1/CSE6324_Team4_Fall23

The changes made for this detector have been opened as a pull request to the original slither repository [25] (Pending maintainer review as of 11/27) (https://github.com/crytic/slither/pull/2249)

IX. References

[1] Feist, Josselin & Grieco, Gustavo & Groce, Alex. (2019). Slither
[2] Adding a new detector - https://github.com/crytic/slither/wiki/Adding-a-new-detector
[3] Python API - https://github.com/crytic/slither/wiki/Python-API
[4] Improper usage of ecrecover - https://github.com/crytic/slither/issues/1950
[5] What is ecrecover in Solidity? - https://soliditydeveloper.com/ecrecover
[6] Unencrypted Private Data On-Chain - https://swcregistry.io/docs/SWC-136/
[7] CWE-798: Use of Hard-coded Credentials - https://cwe.mitre.org/data/definitions/798.html
[8] Sensitive Information Disclosure in Smart Contracts - https://knowledge-base.secureflag.com/vulnerabilities/sensitive_information_exposure/sensitive_information_disclosure_smart_contracts.html
[9] Slither - https://github.com/crytic/slither
[10] Slither Detector Documentation - https://github.com/crytic/slither/wiki/Detector-Documentation
[11] SWC-121: Missing Protection against Signature Replay Attacks - https://swcregistry.io/docs/SWC-121/
[12] Mythril - https://github.com/Consensys/mythril
[13] Mythril Analysis Modules - https://mythril-classic.readthedocs.io/en/develop/module-list.html
[14] MythX - https://mythx.io/about/
[15] MythX Detectors - https://mythx.io/detectors/
[16] Ethereum: A Secure Decentralized Generalized Transaction Ledger - https://ethereum.github.io/yellowpaper/paper.pdf
[17] Detector for ecrecover return value validation and use of nonces - https://github.com/crytic/slither/pull/2015
[18] Preventing Replay Attacks Post Ethereum Merge - https://quantstamp.com/blog/preventing-replay-attacks-post-ethereum-merge
[19] Installing the Solidity Compiler - https://docs.soliditylang.org/en/v0.8.9/installing-solidity.html
[20] Openzeppelin ECDSA solidity implementation - https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/ECDSA.sol
[21] Sonar Secrets Github- https://github.com/Skyscanner/sonar-secrets/tree/master
[22] Openzeppelin - https://github.com/OpenZeppelin/openzeppelin-contracts
[23] Detecting and Mitigating Secret-Key Leaks in Source Code Repositories: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7180102
[24] Managing access keys for IAM users - https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html
[25] Improper usage of ecrecover() detector with improvements: https://github.com/crytic/slither/pull/2249