Audit with Xaudit

Welcome to our beta smart contract auditing tool! This tool has been designed to assist developers in enhancing the security and reliability of their smart contracts. However, it's important to note that this is not a substitute for a comprehensive, human-led audit. While our system aims to provide valuable insights and identify potential vulnerabilities, it may not catch all issues or assess the full scope of risks associated with your smart contract. Therefore, we strongly recommend complementing our tool with thorough manual audits conducted by experienced professionals before deploying your smart contract in a production environment. Remember, the security of your smart contract is paramount, and multiple layers of scrutiny are crucial to ensure its integrity and protect both you and your users.

Introduction

Our auditing process harnesses the power of three cutting-edge tools: Slither, Mythril, and Mythril. Slither and Mythril are renowned for their capability to analyze smart contracts for vulnerabilities, providing invaluable insights into potential security risks and code quality issues. Additionally, our system integrates ChatGPT, a sophisticated language model trained to comprehend and analyze code, enabling it to offer nuanced suggestions and identify subtle vulnerabilities that traditional tools might overlook. By combining these tools, our auditing process aims to provide a comprehensive assessment of your smart contract's security and functionality. However, it's important to recognize that while these tools are powerful aids, they are not infallible, and human oversight remains essential for a thorough evaluation of your smart contract's safety and reliability.

chatGPT

ChatGPT is a tool developed by OpenAI based on the GPT (Generative Pre-trained Transformer) architecture. It is a language model that utilizes machine learning to generate coherent and relevant text in response to user inputs. ChatGPT is capable of understanding and generating responses in natural language across a variety of topics and contexts, making it useful for tasks such as text generation, answering questions, translation, and casual conversation. It employs a machine learning approach called "pre-training" to capture and learn complex patterns in the training data, allowing it to generate responses that are often indistinguishable from human-written text. In summary, ChatGPT is an advanced natural language processing tool that can comprehend and generate text with high fluency and accuracy.

GPTLens

GPTLENS is an innovative framework that leverages large language models (LLMs) for intelligent detection of vulnerabilities in smart contracts. This framework operates in two stages: Auditor and Critic.

In the Auditor stage, an LLM scans the smart contract code to identify potential vulnerabilities. It utilizes contextual capabilities to detect patterns or constructs associated with vulnerabilities.

In the Critic stage, another LLM evaluates the Auditor's findings to determine whether the identified vulnerabilities are genuine or false positives. This involves a deeper analysis of the code's context and logic.

The uniqueness of GPTLENS lies in its two-stage approach, which reduces false positives and maximizes the detection of real vulnerabilities. This is achieved by filtering the Auditor's results through the Critic.

GPTLENS offers several advantages, such as more precise and reliable detection of vulnerabilities, efficiency in processing complex code, adaptability to new types of vulnerabilities and contract structures, and the ability to understand the context and semantics of smart contract code.

Slither

Slither is an open-source static code analysis framework for Solidity and Vyper, written in Python3. Its aim is to provide valuable insights into Ethereum smart contracts. It offers automated vulnerability detection, code optimization suggestions, improved code understanding, and assistance in code review. Slither works by converting Solidity contracts into an intermediate representation called SlithIR, which facilitates analysis implementation while preserving semantic information. Slither runs a set of predefined analyses in various stages, from information retrieval to vulnerability detection and code optimizations. Additionally, it provides an API for writing custom analyses in Python. It is easy to integrate into development workflows, has a low false positive rate, and supports Solidity and Vyper, offering fast runtime and high accuracy in its analyses.

Mythril

Mythril is an open-source smart contract security analyzer that utilizes symbolic execution to detect a wide range of vulnerabilities. It is compatible with various blockchains such as Ethereum, Hedera, Quorum, Vechain, Roostock, Tron, among others. This analyzer combines static analysis and symbolic execution to detect vulnerabilities such as timestamp dependency, arbitrary address writing, and integer overflow. It integrates with the MythX security analysis platform, which offers static analysis, symbolic execution, and input fuzzing to detect security flaws and verify the correctness of smart contract code. MythX uses tools like Maru for static analysis and Harvey for greybox fuzzing. Additionally, Mythril provides symbolic execution and SMT resolution to detect vulnerabilities by representing program states and control flow. Finally, Mythril undergoes a correlation and communication process between tools to ensure accurate and understandable results for the user.

Analysis Findings and Recommendations

Based on the analysis of the contract using various tools, the following vulnerabilities were identified:

1.

Reentrancy vulnerability: The `launch_attack()` function can be exploited to repeatedly withdraw funds from the `vulnerable_contract` until it runs out of gas or the contract balance is depleted.

2.

Unchecked external calls: The `deposit()` function does not check for success or failure when calling the `addToBalance()` function of the `vulnerable_contract`, which could lead to unexpected behavior.

3.

Lack of access control: The `get_money()` function allows anyone to call it and transfer the contract balance to the `owner` address, which could be exploited to drain the contract balance.

4.

Lack of input validation: The `deposit()` function does not validate the input address for the `vulnerable_contract` parameter, allowing an attacker to deposit funds into a malicious contract.

To address these vulnerabilities, the following recommendations are suggested:

1.

Implement reentrancy guards in the `launch_attack()` function to prevent unauthorized fund withdrawal.

2.

Check for success or failure when making external calls in the `deposit()` function to ensure expected behavior.

3.

Implement access control in the `get_money()` function to restrict who can trigger the transfer of contract balance.

4.

Validate

Suggested Unit Tests for Validation

Here are the suggested function names for the test cases along with explanations:

1.

`test_deposit`: This test case will verify the functionality of the `deposit` function in the ReentranceExploit contract. It will simulate a deposit and check if the `addToBalance` function is called on the vulnerable contract with the correct value.

2.

`test_launchAttack`: This test case will verify the behavior of the `launch_attack` function. It will check if the `withdrawBalance` function is called on the vulnerable contract when the attack mode is turned on.

3.

`test_reentrantFallback`: This test case will simulate a reentrant attack by triggering the fallback function of the ReentranceExploit contract and check if it successfully calls the `withdrawBalance` function on the vulnerable contract.

4.

`test_getMoney`: This test case will verify the `get_money` function, ensuring that it transfers the remaining funds to the owner of the contract.

These test cases will cover the deposit, attack launch, reentrant fallback, and fund retrieval functionalities of the ReentranceExploit contract.

Appendix

The following appendix contains the comprehensive output code generated by the tools utilized in the detection of vulnerabilities within the smart contracts under scrutiny. This compilation encompasses the detailed findings and analyses provided by ('rawchatGPT', 'GPTLens', 'Slither', 'Mythril'), meticulously documenting potential security risks, code quality assessments, and nuanced suggestions. The inclusion of this exhaustive output serves to offer stakeholders a transparent and thorough overview of the auditing process, facilitating a deeper understanding of the identified vulnerabilities and recommended remedial actions.

rawchatGPT

```
The following are some potential vulnerabilities in the given contract:

    1.

Reentrancy vulnerability: The launch_attack() function sets the
attackModeIsOn flag to true and then calls the withdrawBalance() function
of the vulnerable_contract. This function then calls back the fallback
function of the ReentranceExploit contract, which in turn calls the
withdrawBalance() function again. This creates a loop, allowing an
attacker to repeatedly withdraw funds from the vulnerable_contract until
it runs out of gas or the contract balance is depleted.

    2.

Unchecked external calls: The deposit() function uses the call() function
to call the addToBalance() function of the vulnerable_contract. This
external call is not checked for success or failure, which could lead to
unexpected behavior if the addToBalance() function fails.

    3.

Lack of access control: The get_money() function allows anyone to call it
and transfer the contract balance to the owner address. This could be
exploited by an attacker to drain the contract balance.

    4.

Lack of input validation: The deposit() function does not validate the
input address for the vulnerable_contract parameter. This could allow an
attacker to deposit funds into a malicious
```

GPTLens

```
[{'function_name': 'deposit', 'vulnerability': 'Reentrancy', 'criticism':
"The reasoning is correct. The deposit function is vulnerable to
reentrancy attack by allowing an attacker to call the addToBalance
function of the vulnerable_contract before updating the state variable
vulnerable_contract. This allows an attacker to repeatedly call the
addToBalance function and drain the contract's balance. The severity is
high because it can lead to unauthorized fund withdrawal. The
profitability is moderate because an attacker can profit from draining the
contract's balance.", 'correctness': 9, 'severity': 8, 'profitability': 6,
'final_score': 8.0}, {'function_name': 'launch_attack', 'vulnerability':
'Unrestricted Access', 'criticism': 'The reasoning is correct. The
launch_attack function sets the attackModeIsOn variable to true without
any access control, allowing anyone to trigger the attack. This can lead
to unauthorized withdrawal of funds from the vulnerable_contract. The
severity is high because it allows unauthorized fund withdrawal. The
profitability is moderate because an attacker can profit from unauthorized
fund withdrawal.', 'correctness': 9, 'severity': 8, 'profitability': 6,
```

'final_score': 8.0}, {'function_name': 'fallback', 'vulnerability': 'Reentrancy', 'criticism': 'The reasoning is correct. The fallback function can be exploited for reentrancy attack as it allows an attacker to repeatedly call the withdrawBalance function of the vulnerable_contract by setting the attackModeIsOn variable to true. This can result in unauthorized fund withdrawal. The severity is high because it allows unauthorized fund withdrawal. The profitability is moderate because an attacker can profit from unauthorized fund withdrawal.', 'correctness': 9, 'severity': 8, 'profitability': 6, 'final_score': 8.0}, {'function_name': 'get_money', 'vulnerability': 'Unprotected Self-Destruct', 'criticism': 'The reasoning is correct. The get_money function uses the suicide (self-destruct) operation without any access control, allowing anyone to destroy the contract and transfer all remaining funds to the owner. This can lead to loss of funds if triggered by an unauthorized party. The severity is high because it allows unauthorized destruction of the contract and fund transfer. The profitability is low because an external attacker cannot profit from this vulnerability.', 'correctness': 9, 'severity': 8, 'profitability': 2, 'final_score': 7.0}]

Slither

```
'solc --version' running 'solc out/flat_contract.sol --combined-json abi,ast,bin,bin-runtime,srcmap,srcmap-runtime,userdoc,devdoc,hashes,compact-format --allow-paths .,/home/farduh/xcapit/xaudit/out' running

ReentranceExploit.get_money() (out/flat_contract.sol#35-37) allows anyone to destruct the contract Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#suicidal

ReentranceExploit.deposit(address)._vulnerable_contract (out/flat_contract.sol#12) lacks a zero-check on : - vulnerable_contract = _vulnerable_contract (out/flat_contract.sol#13) Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

Deprecated standard detected require(bool)(vulnerable_contract.call.value(msg.value)(bytes4(sha3()(addToBalance())))) (out/flat_contract.sol#15): - Usage of "sha3()" should be replaced with "keccak256()" Deprecated standard detected require(bool)(vulnerable_contract.call(bytes4(sha3()(withdrawBalance())))) (out/flat_contract.sol#22): - Usage of "sha3()" should be replaced with "keccak256()" Deprecated standard detected require(bool)(vulnerable_contract.call(bytes4(sha3()(withdrawBalance())))) (out/flat_contract.sol#31): - Usage of "sha3()" should be replaced with "keccak256()" Deprecated standard detected suicide(address)(owner) (out/flat_contract.sol#36): - Usage of "suicide()" should be replaced with "selfdestruct()" Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#deprecated-standards

Pragma version^0.4.15 (out/flat_contract.sol#1) allows old versions solc-0.4.15 is not recommended for deployment Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Low level call in ReentranceExploit.deposit(address) (out/flat_contract.sol#12-16): - require(bool)(vulnerable_contract.call.value(msg.value)(bytes4(sha3()(addToBalance())))) (out/flat_contract.sol#15) Low level call in ReentranceExploit.launch_attack() (out/flat_contract.sol#18-23): - require(bool)(vulnerable_contract.call(bytes4(sha3()(withdrawBalance())))) (out/flat_contract.sol#22) Low level call in ReentranceExploit.fallback()
```

(out/flat_contract.sol#26-33): -
require(bool)(vulnerable_contract.call(bytes4(sha3()(withdrawBalance())))
(out/flat_contract.sol#31) Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-ca
lls

Parameter ReentranceExploit.deposit(address)._vulnerable_contract
(out/flat_contract.sol#12) is not in mixedCase Function
ReentranceExploit.launch_attack() (out/flat_contract.sol#18-23) is not in
mixedCase Function ReentranceExploit.get_money()
(out/flat_contract.sol#35-37) is not in mixedCase Variable
ReentranceExploit.vulnerable_contract (out/flat_contract.sol#5) is not in
mixedCase Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-
to-solidity-naming-conventions out/flat_contract.sol analyzed (1 contracts
with 84 detectors), 15 result(s) found

Mythril

==== External Call To User-Supplied Address ==== SWC ID: 107 Severity: Low
Contract: ReentranceExploit Function name: launch_attack() PC address: 965
Estimated Gas Usage: 7761 - 63078 A call to a user-supplied address is
executed. An external message call to an address specified by the caller
is executed. Note that the callee account might contain arbitrary code and
could re-enter any function within this contract. Reentering the contract
in an intermediate state may lead to unexpected behaviour. Make sure that
no state modifications are executed after this call and/or reentrancy
guards are in place.

```
In file: out/flat_contract.sol:22

vulnerable_contract.call(bytes4(sha3("withdrawBalance()")))
```

Initial State:

Account: [CREATOR], balance: 0x1, nonce:0, storage:{} Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0 Caller: [CREATOR], function: deposit(address), txdata: 0xf340fa010000000000000000000000deadbeefdeadbeefdeadbeefdeadbeefdeadbeef, decoded_data: ('0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef',), value: 0x0 Caller: [ATTACKER], function: launch_attack(), txdata: 0x4eb864ce, value: 0x0

==== Unprotected Ether Withdrawal ==== SWC ID: 105 Severity: High Contract: ReentranceExploit Function name: launch_attack() PC address: 965 Estimated Gas Usage: 7761 - 63078 Any sender can withdraw Ether from the contract account. Arbitrary senders other than the contract creator can profitably extract Ether from the contract account. Verify the business logic carefully and make sure that appropriate security controls are in place to prevent unexpected loss of funds.

```
In file: out/flat_contract.sol:22

vulnerable_contract.call(bytes4(sha3("withdrawBalance()")))
```

Initial State:

Account: [CREATOR], balance: 0x0, nonce:0, storage:{} Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0 Caller: [SOMEGUY], function: deposit(address), txdata: 0xf340fa010000000000000000000000deadbeefdeadbeefdeadbeefdeadbeefdeadbeef, decoded_data: ('0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef',), value: 0x1 Caller: [ATTACKER], function: launch_attack(), txdata: 0x4eb864ce, value: 0x0

==== Unprotected Selfdestruct ==== SWC ID: 106 Severity: High Contract: ReentranceExploit Function name: get_money() PC address: 1097 Estimated Gas Usage: 1049 - 1474 Any sender can cause the contract to self-destruct. Any sender can trigger execution of the SELFDESTRUCT instruction to destroy this contract account. Review the transaction trace generated for this issue and make sure that appropriate security controls are in place to prevent unrestricted access.

```
In file: out/flat_contract.sol:36

suicide(owner)
```

Initial State:

Account: [CREATOR], balance: 0x0, nonce:0, storage:{} Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0 Caller: [ATTACKER], function: get_money(), txdata: 0xb8029269, value: 0x0

==== External Call To User-Supplied Address ==== SWC ID: 107 Severity: Low Contract: ReentranceExploit Function name: deposit(address) PC address: 1373 Estimated Gas Usage: 7879 - 63196 A call to a user-supplied address is executed. An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.

```
In file: out/flat_contract.sol:15

vulnerable_contract.call.value(msg.value)(bytes4(sha3("addToBalance()")))
```

Initial State:

Account: [CREATOR], balance: 0x0, nonce:0, storage:{} Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0 Caller: [SOMEGUY], function: deposit(address), txdata: 0xf340fa010000000000000000000000deadbeefdeadbeefdeadbeefdeadbeefdeadbeef, decoded_data: ('0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef',), value: 0x0

==== Unprotected Ether Withdrawal ==== SWC ID: 105 Severity: High Contract: ReentranceExploit Function name: deposit(address) PC address: 1373 Estimated Gas Usage: 7879 - 63196 Any sender can withdraw Ether from the contract account. Arbitrary senders other than the contract creator can profitably extract Ether from the contract account. Verify the business logic carefully and make sure that appropriate security controls are in place to prevent unexpected loss of funds.

```
In file: out/flat_contract.sol:15

vulnerable_contract.call.value(msg.value)(bytes4(sha3("addToBalance()")))
```

Initial State:

Account: [CREATOR], balance: 0x0, nonce:0, storage:{} Account: [ATTACKER], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0 Caller: [SOMEGUY], function: deposit(address), txdata: 0xf340fa0100000000000000000000000000deadbeefdeadbeefdeadbeefdeadbeefdeadbeef, decoded_data: ('0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef',), value: 0x1 Caller: [ATTACKER], function: deposit(address), txdata: 0xf340fa01, value: 0x0