Project 02: AI Powered Smart Contract Auditor

Proposal

Ilan Klimberg, Nick Johnson, Stephan Volynets, Hunter Kline

1. Problem Statement

Ethereum smart contracts often contain vulnerabilities that can lead to significant real-world consequences, including theft of millions or billions in funds. Even skilled developers frequently overlook the top ten most common attack types:

- Reentrancy
- Access control mistakes
- Oracle manipulation
- Arithmetic errors
- Flash loan abuse
- MEV and front running risks
- Upgradeability issues
- Faulty ERC-20 logic
- Signature and ECDSA misuse
- Broken business logic

Security audits today are expensive, time-consuming, and impossible to scale to the number of contracts deployed on-chain. Many production contracts remain unaudited or insecure. Individual developers are not able to get there smart contracts audited.

Central Question

Can we build a Large Language Model powered auditor that detects common smart contract exploits and produces structured, actionable vulnerability reports for Solidity contracts?

2. Objective

The objective of this project is to construct an LLM based smart contract auditor that can:

- Classify whether a contract is safe or malicious
- Identify the specific exploit type
- Highlight the function and line numbers containing the vulnerability
- Explain why the issue is dangerous in clear language
- Produce a structured JSON vulnerability report

To train and evaluate the auditor, we will construct a balanced dataset of approximately 200 short Solidity contracts:

- 100 synthetic malicious contracts
- 100 corrected safe versions

Each synthetic contract will have a metadata JSON file using this schema:

```
{
  "id": "{contract_id}",
  "solidity": "^0.8.20",
  "attacks": [
    {
      "type": "<attack_type>",
      "severity": "<low|medium|high>",
      "lines": [<line_numbers>],
      "description": "<1 sentence description>",
      "refs": [
        "https://swcregistry.io/docs/<SWC-ID>"
      ]
    }
  ],
  "paths": {
    "malicious": "data/synthetic/malicious/{contract_id}.sol",
    "safe": "data/synthetic/safe/{contract_id}_safe.sol",
    "analysis": "data/synthetic/slither/{contract_id}.json"
  }
}
```

This format ensures consistent structure for training and evaluation.

3. Why This Topic Is Important

Smart contract exploits have caused billions of dollars in cumulative losses across decentralized finance, NFT platforms, and DAOs. Many developers:

- Lack formal security training
- Cannot afford professional audits
- Rely on tools that are noisy or difficult to interpret

An accessible AI assisted auditor could:

- Improve developer safety
- Reduce catastrophic vulnerabilities
- Provide educational explanations of risks
- Offer an automated first-pass security review

This project also provides a controlled research environment to evaluate how LLMs reason about security-critical code.

4. How LLMs Will Be Used

We designed three tiers of increasing difficulty to ensure feasibility and meaningful results.

Tier 1: Baseline Auditor (Easy)

A prompt engineered model that analyzes contracts and reports vulnerabilities.

Input

- Solidity contract
- Instructions

- Top 10 exploit checklist

Output

- JSON or Markdown vulnerability report

No fine-tuning or retrieval. Establishes a baseline for comparison.

Tier 2: RAG Enhanced Auditor (Medium)

A retrieval augmented system that enhances reasoning with external security knowledge.

Knowledge Sources

- SWC Registry documentation
- OpenZeppelin security guidelines
- Curated exploit descriptions
- Real exploit examples

Mechanism

- Store materials in a vector database
- Retrieve the most relevant passages
- Inject them into the prompt

Tier 3: Fine Tuned Auditor (Hard)

A specialized LLM fine tuned on our dataset.

Models

- GPT 4.1 mini
- GPT 4.1
- GPT 5

Training Data

- All 200 synthetic contracts
- Slither analysis outputs
- Almanax.ai labels
- Consistent JSON schemas

Expected outcome: improved consistency, reduced hallucinations, and more stable structured outputs.

5. Data Sources

Verified Solidity Contracts

Source: Verifier Alliance Public Dataset

Original Collection Method: Contract deployers verify source code using services like Etherscan or Sourcify. The Verifier Alliance aggregates these verified files into Parquet datasets.

Observations: These are real, publicly verified contracts suitable as templates for safe baseline examples.

Date Sourced Verifier Alliance continuously updates the dataset as contracts are verified on the blockchain. We use a random subset sourced on [11/18/25].

Almanax.ai Labels

Source: Almanax.ai vulnerability scoring platform

Original Collection Method: Almanax analyzes deployed contracts using static analysis, expert rules, and semantic heuristics.

Observations: Provides high quality ground-truth labels to validate and compare against our dataset.

Date Sourced We will use Almanax to label data later in the project.

Slither Static Analysis

Source: Crytic Slither

Original Collection Method: Local static analysis that generates JSON vulnerability reports.

Observations: Used to validate synthetic malicious contracts and generate additional labels.

Date Sourced We will run Slither on all synthetic contracts during data generation.

Synthetic Malicious and Safe Contracts

Source: Created by our team from safe templates.

Original Creation Method: Start from a known safe contract and inject a specific exploit pattern. Produce a corrected safe counterpart. Generate metadata and Slither analysis.

Observations: Creates high quality supervised examples with controllable exploit patterns.

Date Sourced We will generate these contracts during Week 1 and 2 of the project.

Knowledge Store Documents

These documents form the core of our Retrieval Augmented Generation (RAG) knowledge base. For each of the ten exploit categories, we include one to three authoritative references describing the vulnerability, example smart contracts, and recommended mitigation patterns.

Note: Many of the data sources listed below originate from the same underlying provider. As a result, they share a common "Last updated in 2020" timestamp, though the specific update date within that year is not available.

Reentrancy

Sources: - SWC 107: Reentrancy
https://swcregistry.io/docs/SWC-107 - OWASP SC05: Reentrancy Attacks
https://owasp.org/www-project-smart-contract-top-10/2025/en/src/SC05-reentrancy-attacks.html

Original Collection Method:
The SWC Registry provides community curated vulnerability specifications and example contracts. OWASP SC05 is collected from the OWASP Top Ten Smart Contract Risks documentation.

Observations:
Includes example contracts demonstrating recursive withdraw patterns, state updates after external calls, and unprotected fallback-triggered withdrawals.

Date Sourced Last updated in 2020 date unknown

Access Control Failures

Sources: - SWC 124: Write to Arbitrary Storage
https://swcregistry.io/docs/SWC-124 - SWC 105: Unprotected Function
https://swcregistry.io/docs/SWC-105

Original Collection Method:
SWC Registry entries are curated by Crytic and Trail of Bits and include examples of missing modifiers, insecure ownership patterns, and storage write vulnerabilities.

Observations:
Documents common mistakes such as missing onlyOwner checks, insecure initializers, and arbitrary storage writes.

Date Sourced Last updated in 2020 exact date unknown.

Oracle Manipulation

Sources: - SWC 128: DoS With Block Gas Limit and Oracle Issues
https://swcregistry.io/docs/SWC-128

Original Collection Method:
SWC 128 is derived from community aggregated examples of oracle manipulation, stale prices, and unbounded loops that depend on external data feeds.

Observations:
Includes examples of AMM price manipulation, TWAP bypass, and gas sensitive oracle reads.

Date Sourced Last updated in 2020 exact date unknown.

Flash Loan Attacks

Sources: - To Be Determined (TBD)
High quality documents will be added describing flash loan exploit patterns such as manipulated collateralization checks or transient price distortions.

Original Collection Method:
Will be sourced from professional audit reports and reputable security writeups.

Observations:
Will cover manipulation of multi-step logic when atomic borrowing is possible.

Date Sourced Last updated in 2020 exact date unknown.

Arithmetic Errors

Sources: - SWC 101: Integer Overflow and Underflow
https://swcregistry.io/docs/SWC-101

Original Collection Method:
Collected from SWC Registry examples showing unsafe arithmetic operations before Solidity 0.8.x built-in checks.

Observations:
Includes demonstrations of wraparound behavior, unchecked arithmetic, and unsafe balance or index calculations.

Date Sourced Last updated in 2020 exact date unknown.

MEV and Front Running

Sources: - SWC 114: Transaction Order Dependence
https://swcregistry.io/docs/SWC-114

Original Collection Method:
Collected from SWC documentation for ordering-dependent vulnerabilities.

Observations:
Includes examples of functions where user outcomes depend on transaction order, enabling adversarial reordering.

Date Sourced Last updated in 2020 exact date unknown.

Upgradeability and Initialization Bugs

Sources: - SWC 112: Delegatecall to Untrusted Callee
https://swcregistry.io/docs/SWC-112

Original Collection Method:
Collected from SWC documentation illustrating proxy patterns, unprotected delegatecall usage, and unsafe initialization logic.

Observations:
Shows misuse of delegatecall, untrusted implementation pointers, and initializer function replays.

Date Sourced Last updated in 2020 exact date unknown.

Broken Business Logic

Sources: - SWC 136: Unprotected Self Destruct and Logic Flaws
https://swcregistry.io/docs/SWC-136

Original Collection Method:
Collected from SWC listings where business logic flaws enable unintended state transitions or destructive operations.

Observations:
Includes examples of selfdestruct misuse, incorrect reward distribution logic, and unsafe trust assumptions.

Date Sourced Last updated in 2020 exact date unknown.

ERC 20 Handling Bugs

Sources: - SWC 104: Unchecked Return Value
https://swcregistry.io/docs/SWC-104

Original Collection Method:
Sourced from SWC examples showing failure to check return values for ERC 20 token transfers.

Observations:
Includes cases where token transfers silently fail, leading to stuck funds or incorrect accounting.

Date Sourced Last updated in 2020 exact date unknown.

Signature and ECDSA Validation Bugs

Sources: - SWC 122: Insufficient Signature Validation
https://swcregistry.io/docs/SWC-122

Original Collection Method:
Collected from SWC documentation describing insecure ECDSA recovery, replayable signatures, and improper domain separation.

Observations:
Displays examples of signature reuse, improper hashing, and attacks where signatures validate incorrect senders or scopes.

Date Sourced Last updated in 2020 exact date unknown.

6. Multi Model Benchmarking

We will evaluate multiple LLMs using identical prompts, datasets, and JSON schemas.

Possible Models to Evaluate

- GPT 4.1
- GPT 4.1 mini
- GPT 5.1
- Anthropic Claude models
- Llama and Mixtral baselines

Evaluation Metrics

- Accuracy
- Precision
- Recall (Vulnerability Detection Rate)
- F1 Score
- Per class F1 Score for each exploit category

We will also create a test set of human audited real world contracts with manually written JSON labels.

7. Final Product

Interactive Web Application

- Upload or paste Solidity code
- Select baseline, RAG, or fine tuned auditor
- Receive a structured vulnerability report including:
  - Exploit type
  - Severity
  - Vulnerable lines
  - Explanation and references

Complete Dataset

- 200 contracts
- Safe and malicious versions
- Slither outputs
- Metadata JSONs

Analytical Report

- Model comparisons
- Tier performance analysis
- Error patterns
- Ethical considerations

Optional CLI Tool

Audit an entire directory and export results to CSV.

8. Ethical Concerns

Misuse for Offensive Security

The tool could be used to identify vulnerabilities in third party contracts. **Mitigation:** Only generate defensive analysis. No exploit construction.

Data Privacy and Intellectual Property

Contracts may contain proprietary logic. **Mitigation:** Only use publicly verified, open source contracts.

Over Trust in Model Predictions

LLMs may confidently produce incorrect analyses. **Mitigation:** Treat all outputs as advisory.

Environmental and Cost Impact

Fine tuning requires compute resources. **Mitigation:** Use small models and remain within course budgets.

9. Project Timeline

Week 1

- Generate malicious and safe contract pairs
- Run Slither analysis
- Produce metadata JSONs
- Track all contract files

Week 2

- Implement baseline LLM auditor
- Build minimal UI
- Begin constructing retrieval corpus

Week 3 (Draft Due December 4)

- Complete the dataset
- Implement RAG system
- Begin fine tuning

Week 4

- Benchmark all models
- Compute evaluation metrics
- Complete the analytical report
- Polish UI and CLI

Week 5

- Test and improve model

Feasibility

The project scope is realistic and achievable. It includes a manageable dataset, clear tiers of increasing complexity, and fallback options if fine tuning becomes infeasible. The proposal demonstrates understanding of LLM capabilities, limitations, and responsible use in security analysis.