Smart Contract Vulnerability Injection Guidance

Steps to Inject a Vulnerability

1. Create a Vulnerability File

To begin injecting a vulnerability, you need to create a vulnerability file following the naming convention:

- File Name Format: vul-x-y-z.py
- x-y-z represents the classification number, which can be found at https://openscv.dei.uc.pt/

2. Structure of the Vulnerability File

Each vulnerability file should include two primary functions:

a. Condition Function

- The condition function is responsible for identifying all suitable locations in the target smart contract where the specified vulnerability can be injected.
- It analyzes the Abstract Syntax Tree (AST) of the smart contract to determine potential points for vulnerability injection.

b. Action Function

- The type of vulnerability
    - Adding a vulnerable node,
    - Updating an existing node to introduce a vulnerability,
    - Deleting a node to weaken the security posture.

- The action function is responsible for implementing the identified vulnerability.
- Once the vulnerable location is identified by the condition function, the action function builds the vulnerable node.

3. Reference Existing Vulnerability Files

- Several pre-existing vulnerability files are available in the source code repository at https://github.com/blockchain-dei/injector/
- You can refer to these files to understand how vulnerabilities are structured and injected.

How the Vulnerability Injection Program Works

1. Command-Line Injection Process

To inject a vulnerability into a specific smart contract, follow these steps:

- Run the following command from the terminal:
  - python3 vul-x-y-z.py TargetSmartContract.sol
  - There are a few real vulnerability Python files as well as Solidity source code files in source code repository.

Make sure Python and the Solidity compiler (solc) are installed, and the code is properly cloned from the repository at https://github.com/blockchain-dei/injector/

2. Execution Flow

Once the command is executed, the process follows these steps:

1. Execution of Vulnerability File
   - The first step is executing the vulnerability file (e.g., vul-x-y-z.py).
2. Execution of commonc.py

After the vulnerability file executes, the commonc.py file is invoked to carry out the injection procedure, which involves:

- Parsing Input Arguments: The script checks if the input is a single smart contract file or a directory containing multiple files.
- Inject Function:
  - Smart Contract Version Check: The tool checks the version of the target smart contract.
  - AST Conversion: Using the Solidity compiler (solc), the smart contract's source code is transformed into a compact Abstract Syntax Tree (AST) format.
  - Vulnerability Search: The condition function analyzes the AST to find potential points for vulnerability injection.
  - The action function builds the vulnerable node.
  - AST Update: The update_node_in_ast function is responsible for adding the vulnerable node to the AST structure.

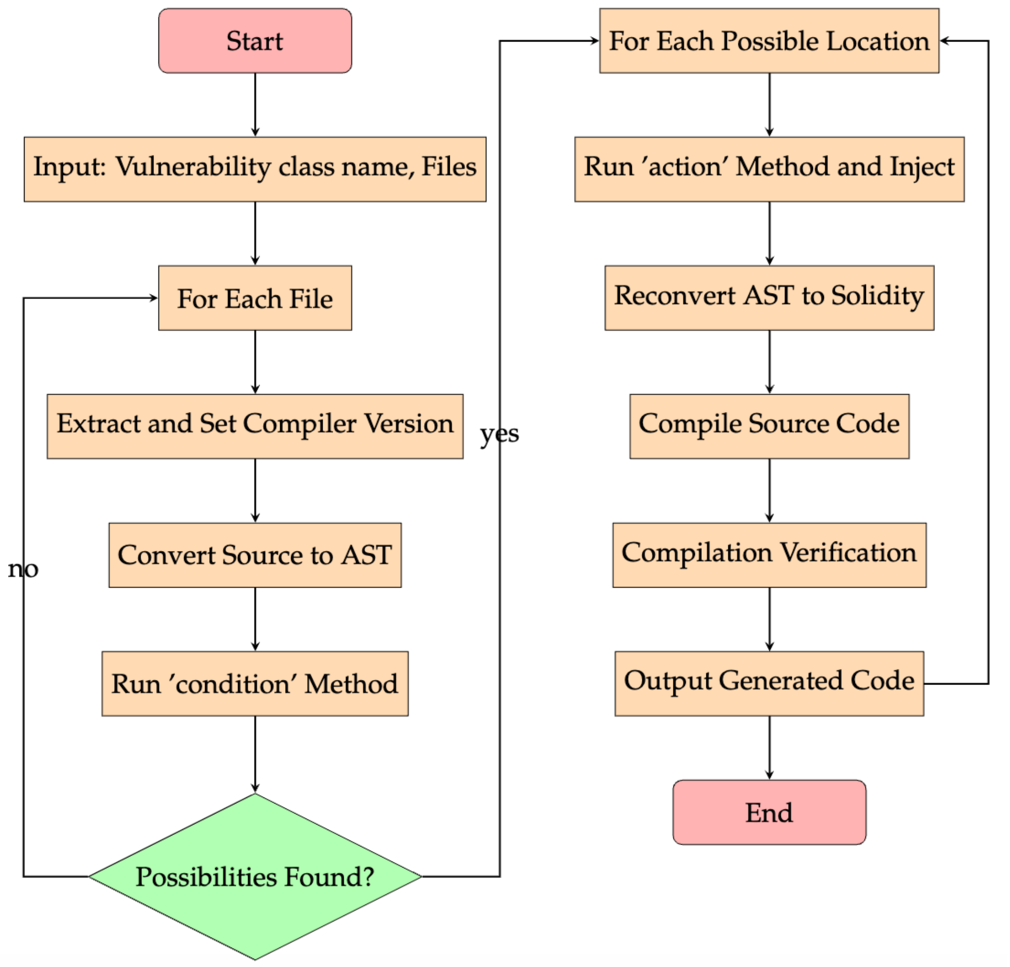
- Solidity Source Code Conversion: The updated vulnerable AST is converted back into Solidity source code using a custom converter function (convert_ast_source). The resulting vulnerable smart contract is saved on disk as a new version of the original contract.
- Compilation Validation: The tool checks if the vulnerable smart contract is compilable using the iscompilable function.

3. Auxiliary Functions

Several auxiliary functions are utilized during the vulnerability injection process. These functions are found in the serializerc.py file:

- getContractSolVersion: Determines the Solidity version of the target contract.
- iscompilable: Ensures that the vulnerable contract is valid and can be compiled.
- convert_ast_source: Converts the AST format back to Solidity source code, supporting various node types, including:
  - ContractDefinition, FunctionDefinition, ModifierDefinition, ExpressionStatement, EnumDefinition, InlineAssembly, ForStatement, IfStatement, FunctionCall, and more.

Vulnerability Injector workflow


Start
Input: Vulnerability class name, Files
For Each File
Extract and Set Compiler Version
Convert Source to AST
Run 'condition' Method
Possibilities Found?
no
yes
For Each Possible Location
Run 'action' Method and Inject
Reconvert AST to Solidity
Compile Source Code
Compilation Verification
Output Generated Code
End