Static Taint Analysis for Ethereum Contracts

Program Analysis for System Security and Reliability
Spring 2022

In this project, you will implement a static taint analyzer for Ethereum smart contracts. Your analyzer takes as input an Ethereum smart contract written in Solidity and outputs whether an untrusted user can benefit from the contract by exploiting `selfdestruct`.

Context

`selfdestruct` is a security-sensitive operation in Solidity, used to 1) remove a contract from the blockchain, including the contract's code and storage, and 2) transfer the remaining contract funds to the address provided as the argument to `selfdestruct`. The argument of `selfdestruct` should be a trusted address so that only authorized users (e.g., the owner) can receive the funds when the contract gets removed. We say that a contract is *vulnerable* if an untrusted address can be used as the argument of `selfdestruct`. The goal of your analyzer is to identify whether the input contract is vulnerable or not.

Example

To understand the intended behavior of your analyzer, see the following example (Example 1):

```
pragma solidity ^0.5.0;

contract Contract {
 address payable user;
 address payable owner;

 function registerUser() public {
   user = msg.sender;
 }

 function changeOwner(address payable newOwner) public {
   require(msg.sender == user);
   owner = newOwner;
 }

 function kill() public {
   selfdestruct(owner);
 }
}
```

The contract has a function named `kill` which calls `selfdestruct` and sends the contract's funds to `owner`. The contract is vulnerable because an untrusted attacker `A` can become `owner` and exploit the contract with the following three transactions:

1) `A` calls `registerUser()`.
2) `A` calls `changeOwner(A)`.
3) `A` (or any other user) calls `kill()`.

To fix the vulnerability, we can change the `require` statement in `changeOwner` to `require(msg.sender == owner)`. Then, only the current trusted `owner` (instead of the untrusted `user` before the fix) can set a new `owner` and thus `owner` is always a trusted address.

Definitions

In this section, we formally define the vulnerability to detect in this project.

Explicit and Implicit Dependencies. Now we discuss two forms of dependency between two values `v1` and `v2`. Explicit dependency results from data dependency. We say `v1` explicitly depends on `v2` if `v1` is computed from `v2` with assignments and arithmetic operations. One simple example of explicit dependency is `v1 = v2 + 1`. Implicit dependency results from control-flow dependency. `v1` implicitly depends on `v2` if `v2` controls the result of a branch condition that will affect the *value* of `v1` after the branch. For instance, `if (v2 == 1) v1 = 42; else v1 = 0;` is such an implicit dependency. Note that we only track implicit dependencies between values, not between values and the execution of program statements.

In the rest of this document, unless specified, the words "depend(s)", "dependency" and etc. means that *either* explicit *or* implicit dependency must exist.

Trusted and Untrusted Values. We classify values (user addresses, constants, local values, storage values, and transaction parameters) as either *trusted* or *untrusted*. The contract is constructed by a trusted user and therefore all values that appear 1) in the contract's code and 2) in the initial storage of the contract are trusted. All trusted addresses (e.g., the address stored in the field `owner`) define the set of *trusted users U*. Users not contained in *U* are considered *untrusted*.

Users can send transactions to the contract and modify the values stored in the contract's storage. Values depending on inputs from untrusted users are untrusted. Otherwise, they are trusted. Note that the set of trusted users *U* can therefore be expanded by a trusted user. Further, a storage value may become untrusted. For example, the address stored at field `user` in Example 1 becomes untrusted after the untrusted attacker `A` calls function `registerUser()`.

Guard. To distinguish whether a transaction's sender (`msg.sender`) is trusted or not, contracts employ *guard* conditions that check if the value of `msg.sender` meets certain requirements. In this project, a condition is a *guard* if it depends on the value of `msg.sender` and all other values that it depends on are trusted (or there is no other value). For example, the statement `require(msg.sender == user)` is not a guard in Example 1 because, although its condition explicitly depends on `msg.sender`, the value of `user` is untrusted after the attacker calls `registerUser()`. The statement becomes a guard if `user` is replaced by `owner`, because its condition explicitly depends on `msg.sender` and the value of `owner` is always trusted.

If the execution of a transaction checks a guard condition, the transaction's sender (`msg.sender`) and all function parameters are considered to be trusted for that execution.

Otherwise, those transaction input values are considered to be untrusted. The trustworthiness of the input values will affect other values involved in the current transaction and even future transactions, as storage values may depend on the input values of the current transaction.

Vulnerability. A contract is *vulnerable* if there exists a successfully invoked `selfdestruct` in the contract whose argument is an untrusted address. Otherwise, the contract is safe. Next, we provide more examples to illustrate the property we want to enforce.

Example 2: `require` statements that do not depend on `msg.sender` are not guards.
In the following contract, the `require` statement is not a guard because it does not depend on `msg.sender`. Therefore, `msg.sender` in `foo` is untrusted and the contract is vulnerable.

```
contract Contract {
 address owner;
 function foo(address x) public {
   require(x == owner);         // not a guard
   selfdestruct(msg.sender);    // vulnerable
 }
}
```

Example 3: constants and initial storage values are trusted.
In the following contract, `owner` is trusted initially and can only be assigned to trusted values (`0xDEADBEEF` and `admin`). Therefore, the `require` statement is a guard and the contract is safe.

```
contract Contract {
 address owner;
 address admin;
 function changeOnwer1() public {
   owner = admin;
 }
 function changeOnwer2() public {
   owner = address(0xDEADBEEF);
 }
 function kill() public {
   require(msg.sender == owner); // guard
   selfdestruct(msg.sender);     // safe
 }
}
```

Example 4: in this project, we only track information flows but do not consider the semantics of expressions precisely (e.g., the types of the operators, arithmetics, etc.).
As a result, the `require` statements in `func1`, `func2`, and `func3` are guards, and the following `selfdestruct` statements are safe. However, the `require` statement in `func4` is not a guard because the untrusted value of `x` affects the value of `y` based on information flow, even though the value of `y` does not change in the end. Therefore, the `selfdestruct` statement in `func4`, as well as the whole contract, must be considered vulnerable.

```
contract Contract {
 address owner;
 int y;
 function func1() public {
   require(msg.sender != owner); // guard
```

```
    selfdestruct(msg.sender);     // safe
  }
  function func2() public {
    require(msg.sender == owner || true); // guard
    selfdestruct(msg.sender);             // safe
  }
  function func3() public {
    require(msg.sender == msg.sender); // guard
    selfdestruct(msg.sender);          // safe
  }
  function func4(int x) public {
    y = y + x;  // y is untrusted
    y = y - x;  // y is untrusted
    require(msg.sender == owner && y > 1); // not a guard
    selfdestruct(msg.sender);              // vulnerable
  }
}
```

Example 5: control flow.

`if` conditions can be seen as guards. Both `selfdestruct` statements in `func1` are safe, because the if branch is guarded by `msg.sender == owner` and the else branch is guarded by `msg.sender != owner`. The `selfdestruct` in `func2` is safe because both execution paths to the `selfdestruct` are guarded. The same applies to `func3`, even though the else guard is implicit. In `func4`, the if branch is not guarded and the else branch is guarded.

```
contract Contract {
  address payable owner;
  function func1() public {
    if(msg.sender == owner)       // guard
      selfdestruct(msg.sender);   // safe
    else                          // guard
      selfdestruct(msg.sender);   // safe
  }
  function func2() public {
    if(msg.sender == owner) {}    // guard
    else {}                       // guard
    selfdestruct(msg.sender);     // safe
  }
  function func3() public {
    if(msg.sender == owner) {}    // guard
    selfdestruct(msg.sender);     // safe
  }
  function func4(uint x) public {
    if(x < 5) {                   // not a guard
      selfdestruct(msg.sender);   // vulnerable
    } else {
      require(msg.sender == address(0xDEADBEEF)); // guard
      selfdestruct(msg.sender);                   // safe
    }
  }
}
```

Example 6: ordering of guards.

Guards can affect values used in statements that appear before the guards, as long as the values are guarded in all possible execution paths. In the following contract, `user` and `x` are trusted due to the guards after the assignments. Therefore, the `selfdestruct` in `func3` is safe.

```
contract Contract {
 address payable user;
 address owner;
 function func1(address payable x) public {
   user = x;  // x becomes trusted after seeing the guard
   require(msg.sender == owner);  // guard
 }
 function func2(address payable x) public {
   user = x;  // x becomes trusted after seeing the guard
   if (msg.sender == owner) {}  // guard
 }
 function func3() public {
   selfdestruct(user);  // safe
 }
}
```

Example 7: explicit and implicit dependency.
In function `foo`, `addr` implicitly depends on the value of `c`, which can be controlled by an attacker through function `setc` and therefore is untrusted. As a consequence, the `require` statement is not a guard and the `selfdestruct` statement is vulnerable. This demonstrates that tracking only explicit dependencies is not enough for `foo`, as it would treat `addr` as trusted and thus mark the `selfdestruct` statement as safe.

In function `bar`, `addr` implicitly depends on `msg.sender`. Moreover, all other values that the `require` statement depends on are trusted. Therefore, the `require` statement is a guard. Only modeling explicit dependency would miss the dependency of `addr` on `msg.sender` and the `require` being a guard. Nonetheless, the `selfdestruct` statement would be still marked as safe because of the guard in the if condition.

```
contract Contract {
 address payable owner;
 address payable admin;
 int c;
 function setc(int newc) public {
   c = newc;
 }
 function foo() public {
   address addr = owner;
   if (c > 0) addr = admin;
   require(msg.sender == addr);  // not a guard
   selfdestruct(msg.sender);     // vulnerable
 }
 function bar() public {
   address addr = admin;
   if (msg.sender == admin) addr = owner;
   require(addr == address(0xDEADBEEF));  // guard
   selfdestruct(msg.sender);              // safe
 }
}
```

Example 8: function calls.

The contract has two `require` statements, each followed by a `selfdestruct` statement. The first `require` is a guard checking if `msg.sender` equals a trusted address stored at field `owner`. Therefore, the first `selfdestruct` is safe. The second is not a guard because `z` is an untrusted address. Therefore, the second `selfdestruct` is vulnerable.

```
contract Contract {
 address owner;
 function check(address x) public returns(bool) {
   return (msg.sender == x);
 }
 function foo() public {
   require(check(owner));          // guard
   selfdestruct(msg.sender);       // safe
 }
 function bar(address z) public {
   require(check(z));              // not a guard
   selfdestruct(msg.sender);       // vulnerable
 }
}
```

Example 9: more on implicit dependencies.

In `func`, `i` controls the value of `x` (i.e., the value of `x` differs after the if and else branches) and `x` implicitly depends on `i`. Therefore, the `require` is not a guard and the `selfdestruct` is vulnerable. In `foo`, `selfdestruct` terminates the execution. As a result, after the branch, `x`'s value always comes from the else branch and thus does not implicitly depend on `i`. In `bar`, `i` does not control any value. Therefore, for foo and bar, the `require` is a guard and the `selfdestruct` is safe. Note that, for foo and bar, `i` affects the execution of the statements inside the if branch, which is not considered as implicit dependency in this project.

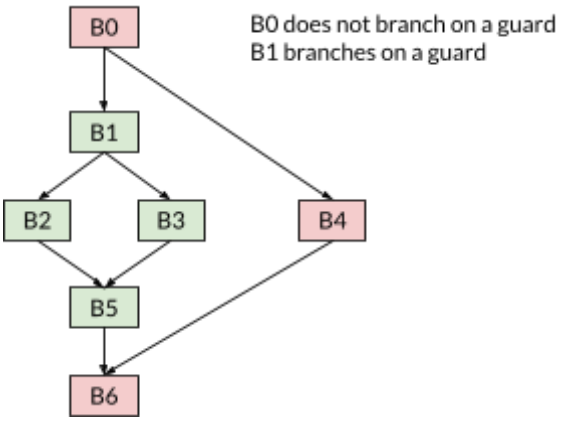
```
contract Contract {
 function func(int i) public {
   address x;
   if (i > 5) {
     x = address(0xDEADBEEF);
   }
   require(msg.sender == x); // not guard
   selfdestruct(msg.sender); // vulnerable
 }
 function foo(int i) public {
   address x;
   if (i > 5) {
     x = address(0xDEADBEEF);
     require(msg.sender == x); // guard
     selfdestruct(msg.sender); // safe
   }
 }
 function bar(int i) public {
   if (i > 5) {
     require(msg.sender == address(0xDEADBEEF)); // guard
     selfdestruct(msg.sender); // safe
   }
 }
}
```

Taint analysis

In this project, your task is to detect vulnerable contracts using static taint analysis. The idea is to assign *taint tags* (or, *taints*) to untrusted values. A contract is then deemed vulnerable if there exists a successfully invoked `selfdestruct` in the contract whose argument is tainted. More formally, to conduct taint analysis we define the following three components:

1) *Sources*: Taints are assigned to function arguments to `public` functions and `msg.sender`.
2) *Sinks*: The argument of successfully invoked `selfdestruct` statements.
3) *Sanitizers*: Guard conditions that restrict which users may execute operations. We consider as a sanitizer any condition that (a) depends on `msg.sender`, and (b) all other values that it depends on are not tainted.

Taints are propagated along the control-flow graph of the contract. Guards sanitize the blocks of the control-flow graph for which all executions pass through the sanitizer. For example, consider the following control-flow graph:



B1 branches on a guard and all executions passing through B1 also pass through blocks B2, B3, and B5. Therefore we sanitize blocks B1, B2, B3, and B5, depicted in green above. Blocks B0, B4, and B6 are *not* sanitized because there are executions that do not execute the guard in B1.

To sanitize a block, we clean the taints for `msg.sender` and function parameters. Therefore, all values that depend only on these values are sanitized. Note that values may also depend on storage values tainted in previous transactions, and these values are *not* sanitized.

Scope

Taint Analysis Features to Support. Your analyzer must be able to handle explicit and implicit dependencies. Moreover, call-site sensitivity is expected (see slide 67 or lecture 2). Make sure the history is reasonably bounded so that the analysis finishes in a reasonable amount of time.

Solidity Features to Support. Your analyzer must support the following Solidity features:

- Solidity pragma 0.5.0 (as in the first example above).
- Solidity files with a single contract.
- Primitive types: `uint`, `int`, `address`, etc.
- Unary and binary operations (addition, negation, etc.).

- Reads and writes to storage variables.
- Reading `msg.sender`.
- Control flow with `if`.
- `require` and `if` sanitizers.
- Public functions with multiple parameters, and zero or multiple return values.
- Function calls within contract.
- Recursive function calls.
- Multiple transactions.

Solidity Features to Ignore. You can ignore the following Solidity features (they will not appear in the test contracts):

- Solidity file imports, multiple contracts per file, and inheritance.
- Library declarations, such as `SafeMath`.
- Inlined EVM bytecode.
- Complex types, such as structs, mappings, arrays.
- Message values except `msg.sender`.
- `while`, and `for` loops.
- Internal, external, and private functions (you can assume all functions are public).
- Custom modifiers.
- Function calls to other contracts.
- Constructors (all storage variables take default values).
- Non-reachable code. The test cases will not have dead code.

Code Repository and Implementation

Your task is to implement the static taint analyzer to detect the vulnerability defined above. Your analyzer must be **sound**. That is, if the contract is indeed vulnerable, your analyzer must report the contract as vulnerable (`Tainted`). Note that it is trivial to build a sound analyzer by reporting all contracts as `Tainted`. Therefore, your analyzer should be **precise**, i.e., reporting as few safe contracts as `Tainted` as possible.

Code Repository. We provide a GitLab repository containing the code template with a README file explaining the repository structure, setup instructions, and example usages. The analysis should be implemented in Datalog by modifying the Datalog program in **project/analyze.dl**. Here you can find a basic introduction to program analysis with Datalog: https://souffle-lang.github.io/pdf/SoufflePLDITutorial.pdf. If you want, you can also modify **project/analyze.py**.

Input and Output Format. Your analyzer takes a smart contract as input and outputs whether the input contract is vulnerable or not. Specifically, your analyzer will be executed with the following command under the directory **project**:

```
$ python analyze.py <contract.sol>
```

where `<contract.sol>` is the file containing the input contract source code. The analyzer should

only print one of the following outputs (**case sensitive**):

- `Tainted`, if the analyzer identifies the contract as vulnerable.
- `Safe`, if the analyzer proves that the contract is not vulnerable.

If your analyzer prints multiple lines, we only take the last line as your output.

Grading

The analyzer will be evaluated on a set of test contracts (with a time and memory limit) and graded based on the **precision** and **soundness**:

- You start with 0 points.
- **You receive 1 point**, if your analyzer correctly outputs `Safe` for the provided test contract. That is, the analyzer is **precise** enough for the test contract.
- **You lose 2 points**, if your analyzer outputs `Safe` but the test contract is actually vulnerable. This means the analyzer is **unsound** for the test contract.
- You get 0 point, if your analyzer outputs `Tainted`.
- If your analyzer exceeds the time limit or memory limit on a test contract, then the result will be considered as `Tainted`.

Evaluation Conditions. Your analyzer will be evaluated on a machine with Intel Xeon E5-2690 v4 CPUs and 512GB RAM. We will use one process to run your analyzer on each test contract. The time limit per test contract is **2 minutes**. The memory limit per test contract is **5 GB**.

Submission

The project organization consists of the following events and deadlines:

Event	Deadline
Group registration	6PM CEST, March 29, 2022
Project announcement	6PM CEST, March 31, 2022
Preliminary submission (optional)	6PM CEST, May 9, 2022
Preliminary feedback	6PM CEST, May 13, 2022
Final submission	6PM CEST, June 10, 2022

After the group registration deadline, each group will receive an invitation to a GitLab repository named **COURSE-PASS-2022/team-{dd}** where **{dd}** is the group number that will be assigned to you. This repository contains the code template and example test contracts.

You can make two submissions. The first is a preliminary submission. It is optional and meant to provide feedback for you to improve the analyzer. It does not affect your project grade. The final submission will be used for grading. The GitLab repository will be used for both submissions. Note that we are **not going to accept** any submission by email. Below are detailed

steps for submission and evaluation. You should obey the rules in the instructions otherwise you may be **deducted points**:

- Before the submission deadlines, you should prepare your code in the **master** branch of the repository. For the final submission, you need to prepare a **2-page write-up PDF** that explains the design of your analyzer, possibly including demonstrations on examples. Note that we will not consider commits after the submission deadlines.
- At the deadline time, we will pull your solution and put it into newly created branches (**preliminary** branch and **final** branch, respectively). You **should not** 1) create branches with the same names yourself, and 2) modify the two branches after we create them.
- After the evaluation, we will put the results into the new branches.

Miscellaneous

TA for Contact. Jingxuan He (jingxuan.he@inf.ethz.ch).

Additional Libraries. Only libraries used by the code template are allowed. Using other libraries may make your analyzer crash in the evaluation environment. If you would like to use other libraries, you need to request explicit permission from the TA.

Questions. You can email your questions to the project TA. We encourage you to attach concrete examples when you ask questions, which would significantly increase the efficiency of understanding the questions and providing answers.

Changes to This Document. We will add changelog items with dates at the end of this document if we apply any changes.

Changelog

We will add log items below for the changes introduced to this document.

10.05.2022 and 11.05.2022

- Refine the definition of implicit dependency and add Example 9.

30.05.2022

- The final deadline is extended for one week.