Descrição

O Solidity suporta tipos de função. Ou seja, uma variável de tipo função pode ser atribuída a uma referência de uma função com uma assinatura correspondente. A função armazenada nessa variável pode ser chamada como uma função comum. O problema surge quando um usuário tem a capacidade de alterar arbitrariamente a variável do tipo função e, assim, executar instruções de código aleatórias. Como o Solidity não suporta aritmética de ponteiros, não é possível alterar essa variável para um valor arbitrário. No entanto, se o desenvolvedor utilizar instruções em assembly, como `mstore` ou o operador de atribuição, no pior cenário um atacante pode fazer com que a variável de tipo função aponte para qualquer instrução de código, violando as validações necessárias e as mudanças de estado exigidas.

Mitigação

O uso de assembly deve ser minimizado. O desenvolvedor não deve permitir que um usuário atribua valores arbitrários a variáveis de tipo função.

Contrato de exemplo da vulnerabilidade do SWC

FunctionTypes.sol

```
/*
 * @source: https://gist.github.com/wadeAlexC/7a18de852693b3f890560ab6a211a2b8
 * @author: Alexander Wade
 */

pragma solidity ^0.4.25;

contract FunctionTypes {

    constructor() public payable { require(msg.value != 0); }

    function withdraw() private {
        require(msg.value == 0, 'dont send funds!');
        address(msg.sender).transfer(address(this).balance);
    }

    function frwd() internal
        { withdraw(); }

    struct Func { function () internal f; }

    function breakIt() public payable {
        require(msg.value != 0, 'send funds!');
        Func memory func;
        func.f = frwd;
        assembly { mstore(func, add(mload(func), callvalue)) }
        func.f();
    }
}
```

---

1. Sobre a Vulnerabilidade no Contrato `FunctionTypes.sol`

O que o Código Faz:

- **Estrutura `Func`**: Armazena um ponteiro para uma função interna (`function () internal f`).
- **Função `breakIt()`**:
  1. Inicializa `func.f` com `frwd()` (endereço fixo na memória).
  2. Usa assembly para modificar `func.f`:

     ```
     assembly { mstore(func, add(mload(func), callvalue)) }
     ```

     - `mload(func)` lê o endereço atual de `func.f` (ex: `0x1000`).
     - `add(..., callvalue)` soma `msg.value` (controlado pelo usuário) ao endereço.
     - `mstore(func, ...)` atualiza `func.f` com o novo endereço.
  3. Chama `func.f()`, que agora pode apontar para um local arbitrário.

Por Que Isso é uma Vulnerabilidade (SWC-127):

- **Controle do `callvalue`**: O usuário pode enviar um `msg.value` que desloca `func.f` para qualquer endereço na memória do contrato.
- **Consequências**:
  - Chamar funções não autorizadas (ex: `selfdestruct` se existir no contrato).
  - Causar `revert` intencional (se o endereço não for um `JUMPDEST` válido).
  - Executar código arbitrário (se o deslocamento apontar para instruções EVM manipuláveis).

Objetivo do Programador (Provavelmente):

- O contrato é um **exemplo didático** para demonstrar a vulnerabilidade. Não há uma lógica prática legítima para modificar um ponteiro de função com base em `msg.value`.
- Em cenários reais, esse tipo de código seria um erro grave, mas aqui serve para ilustrar como o uso de `assembly` pode contornar as proteções do Solidity.

Código da mythril

```
def _analyze_state(self, state):
        """

        :param state:
        :return:
        """

        jump_dest = state.mstate.stack[-1]

        if jump_dest.symbolic is False:
            return []

        if is_unique_jumpdest(jump_dest, state) is True:
            return []
```

```
        try:
            transaction_sequence = get_transaction_sequence(
                state, state.world_state.constraints
            )
        except UnsatError:
            return []
```

```
def is_unique_jumpdest(jump_dest: BitVec, state: GlobalState) -> bool:
    """
    Handles cases where jump_dest evaluates to a single concrete value
    """

    try:
        model = get_model(state.world_state.constraints)
    except UnsatError:
        return True
    concrete_jump_dest = model.eval(jump_dest.raw, model_completion=True)
    try:
        model = get_model(
            state.world_state.constraints
            + [symbol_factory.BitVecVal(concrete_jump_dest.as_long(), 256) !=
jump_dest]
        )
    except UnsatError:
        return True
    return False
```

2. Sobre o Funcionamento do Mythril

Análise Simbólica e `jump_dest`:

- **`jump_dest = state.mstate.stack[-1]`**:
  - Em operações `JUMP`/`JUMPI`, o destino do salto é o valor no topo da stack.
  - Se esse valor é **simbólico** (depende de entrada do usuário, `mstore`, etc.), o Mythril inicia a análise de vulnerabilidade.

Verificação de Múltiplos Valores (`is_unique_jumpdest`):

1. **Primeiro Modelo**:

   ```
   model = get_model(state.world_state.constraints)
   concrete_jump_dest = model.eval(jump_dest.raw, model_completion=True)
   ```

   - Obtém um valor concreto possível para `jump_dest`.

2. **Nova Constraint**:

   ```
   new_constraint = symbol_factory.BitVecVal(concrete_jump_dest.as_long(), 256) !=
   jump_dest
   ```

   - Cria uma restrição: "`jump_dest` deve ser diferente do valor já encontrado".

3. **Teste de Viabilidade**:

- Se o solver encontrar um novo modelo satisfazendo `state.world_state.constraints + [new_constraint]`, há **múltiplos valores possíveis** para `jump_dest` → vulnerabilidade.

Por Que `mstore` Pode Tornar `jump_dest` Simbólico?

- **Dependência de Dados Externos**: Se o valor armazenado via `mstore` vem de `calldata`, `msg.value`, ou outra entrada do usuário, o Mythril rastreia essa dependência e trata `jump_dest` como uma variável simbólica.
- Exemplo:

```
assembly {
    let input := calldataload(0x00)  // Dado controlado pelo usuário
    mstore(0x00, input)
    let dest := mload(0x00)
    jump(dest)
}
```

Aqui, `dest` (e, portanto, `jump_dest`) é simbólico porque depende de `calldata`.

---

3. Pontos de Atenção Adicionais

Symbolic Execution vs. Concreto:

- **Variáveis Simbólicas**: Representam valores desconhecidos que podem assumir múltiplas formas. O Mythril as usa para explorar todos os caminhos de execução possíveis.
- **Concretização**: Quando o solver encontra um modelo (ex: `jump_dest = 0x1000`), ele "concretiza" a variável simbólica, mas continua verificando se há outros valores possíveis.

Mitigação no Exemplo:

- **Remover `assembly`**: A única maneira segura de evitar essa vulnerabilidade é não usar operações de baixo nível que permitam manipulação de ponteiros de função.
- **Validar Entradas**: Se o uso de `assembly` for inevitável, validar estritamente os valores que podem alterar ponteiros (ex: restringir `msg.value` a um intervalo conhecido).
